PROGRAM REPRESENTATION AND EXECUTION IN REAL-TIME MULTIPROCESSOR SYSTEMS

Douglas Niehaus

CMPSCI Technical Report 94-22

February 1994

PROGRAM REPRESENTATION AND EXECUTION IN REAL-TIME MULTIPROCESSOR SYSTEMS

A Dissertation Presented

by

DOUGLAS NIEHAUS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1994

Department of Computer Science

PROGRAM REPRESENTATION AND EXECUTION IN REAL-TIME MULTIPROCESSOR SYSTEMS

A Dissertation Presented

by

DOUGLAS NIEHAUS

Approved as to style and content by:

John A. Stankovic, Chair

Krithi Ramamritham, Member

J. Eliot B. Moss, Member

C. M. Krishna, Member

W. Richards Adrion, Department Head
Department of Computer Science

Dedicated to

My Mother who always did her best

and

My Friends who did the rest

ACKNOWLEDGEMENTS

Obtaining a Ph.D. is a lengthy task that cannot be completed without incurring many debts.

I would like to express sincere gratitude to my advisor Prof. Jack Stankovic who always believed me (I think) when I said it would work. His patience, guidance, and good humor were always welcome and frequently invaluable. Through years of working together I have learned how to do research, I hope, and found a standard of integrity and excellence which will always be a challenge to emulate. I would like to express gratitude equally sincere to Prof. Krithi Ramamritham, my "other advisor" who was always willing to talk things over, and has been the source of innumerable important lessons. That some of them were less pleasant than I would have liked only serves to demonstrate their importance.

I would also like to offer sincere thanks to the other two people who read the dissertation and offered their comments. Prof. Eliot Moss provided well taken and important specific feedback as well as more general advice. Some of it will help guide me through much of my future career. Prof. Krishna exhibited the dedication required of all the readers of a an expository endeavor this thick.

The Spring project, directed by my inestimable advisors, was an excellent venue within which to consider the issues that formed this dissertation. As a project developing an entire system implementation, it is the product of many people's effort. I would like to thank them all, but must particularly acknowledge the help given to me

by the individuals with whom I worked directly. Michael Pasieka spent a year working with me on the implementation of the compiler related ideas presented here, and they would not have come to fruition without him. His dedication to doing a good job, his interest in knowing how different pieces fit together, and his cheerful approach to life were all invaluable. He has also provided invaluable help in managing the logistics of finishing this dissertation after I left Massachusetts. Quite simply, without his aid I would never have finished. Gary Wallace played an equally important role in implementing the system side of Spring, and in making many of the system level ideas in this dissertation concrete realities. While we did not work quite as closely, it is equally true that without his dedication to doing a good job and careful attention to detail I would never have finished.

The collaboration with several students over the years was equally important in my success. Chung-Huei Kuan was involved at the very beginning, when it was most important to demonstrate that the approach I advocated would work. His work included aspects of the communication between the host and target systems, support for the system debugger, and predictable memory management methods. Mao Decao provided invaluable contributions to the implementation of memory management, and an amazing set of stories about his life. Eric and Tim expended much of their youthful enthusiasm and energy on the SDL implementation. Leela helped with resource management, and Carlton provided feedback from the application level about using the system.

Many others worked on their own portions of the Spring system, which were equally vital to my success. I would particularly like to thank Eric Nahum, Chia Shen, Goran Zlokapa, Fuxing Wang, and Lory Molesky. Others unmentioned deserve

thanks, I am sure, but the omission of their names is due only to a faulty memory, not to a lack of gratitude.

Finally, I should like to particularly thank the set of friends with which I am blessed. Dave, Becky, Steve, Myra, Ellen, David, Edward, Rhonda, Cindy, Panos, and Rebecca. All have, over the years listened to my thoughts, fears, ambitions, and sorrows. Some over a longer period, some for only a little while, but each has offered the advice, objectivity, and reassurance without which I would not be where I am.

This has been a lengthy enumeration of my good fortune. I have had excellent academic opportunities, excellent advisors, excellent colleagues, and wonderful friends. I will try to give them all the best thanks I can, by doing as well as I can with the opportunities they have made possible.

ABSTRACT

PROGRAM REPRESENTATION AND EXECUTION
IN REAL-TIME MULTIPROCESSOR SYSTEMS

FERUARY 1994

DOUGLAS NIEHAUS,

B.S., NORTHWESTERN UNIVERSITY

M.S., UNIVERSITY OF MICHIGAN

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor John A. Stankovic

In real-time systems the correctness of the computation depends not only on the logical correctness of the answer, but also on *when* it is produced. Run-time management in a real-time system must therefore exercise detailed control of how and when computations execute. As a result, most approaches to real-time scheduling require substantially more information about a computation's run-time behavior than those used by conventional (non-real-time) systems. The computation behaviors of concern vary with each system, but include: deadline, worst case execution time (WCET), resource use, communication behavior, and precedence relations. One of the ways in which real-time systems are significantly different from conventional systems is that much of of this information must be *predictions* about worst case

computation behavior, since it is required *before the computation executes.* Conventional systems are, however, design to exhibit good average case behavior, not to make worst case behavior predictable. Adopting predictable worst case behavior as a primary design criterion challenges a wide range of conventional system designs and design principles. This dissertation addresses many of the problems associated with building a *predictable real-time system.* It presents contributions, of both theoretical and practical interest, at every level of system design and implementation, including: the Spring-C programming language, program translation and behavioral prediction methods, a predictable operating system implementation, and real-time hardware design principles and suggestions.

The Spring-C programming language provides a process based programming interface and syntactic features supporting the creation of predictable programs. The compiler analyzes the program by creating a *time graph* representing its behavior, and then simplifying it using a technique called subgraph reduction. The simplified time graph is used to build a representation of the program's behavior as a group of tasks with known WCET and resource use, whose execution order is constrained by precedence relations. This representation is used by the system to manage the computation's execution. Predictable execution is also affected by hardware support. Our experience in implementing the Spring system generated a number of principles and examples addressing the design of hardware for real-time system support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As real-time computer systems and applications are created to interact with ever more complex and dynamic environments, the sophistication of their design must increase to meet new demands. Examples of such dynamic and demanding application environments include command-and-control systems, space shuttle and aircraft avionics, automated factories, sophisticated and autonomous robots, nuclear power plants, and process control systems. In such systems, the correctness of the computation depends not only on the logical correctness of the answer, but also on *when* it is produced. Run-time management in a real-time system must therefore include control of when computations are completed, and thus control of when they execute. Such decisions are generally made by the system scheduler.

Different real-time systems employ a wide range of methods for scheduling computations, which have an equally wide range of properties and requirements. The most common way to temporally constrain a computation is to assign it a *deadline.* The impact of each computation on the performance of the system as a whole is often described by assigning it a *value* which is credited to the system when the computation completes. How the computation's value varies with time, before and after its deadline, is specified by its *value function.* There are many possible value functions, but a computation commonly produces its full value when completed on or before its deadline. When a computation's value falls sharply after its deadline, it is often

termed a *hard* deadline. When a computation's value falls gradually after the deadline it is often termed a *soft* deadline.

The terms hard and soft are also commonly used to characterize the overall orientation of a system. A hard real-time system is thus one which is primarily oriented toward computations with hard deadlines, while a soft real-time system is primarily oriented toward computations with soft deadlines. Hard and soft real-time systems, owing to the significantly different properties of their computations, often employ significantly different scheduling algorithms. Schedulers in both types of systems, however, seek to maximize the value produced by the computations they manage.

Whether hard or soft, most approaches to scheduling in real-time systems require substantially more information about a computation's run-time behavior than those used by conventional (non-real-time) systems. While the specific information varies, the computation behaviors of most common concern are the deadline and the worst case execution time (WCET). Other information of interest includes: resource use, communication behavior, and precedence relations. An important aspect of this information is that much of it must be *predictions* about computation behavior, since it is required *before the computation executes.* In contrast, most scheduling algorithms in conventional systems make decisions based on descriptions of a computation's average case behavior, or infer future behavior from that of the recent past. Most real-time systems assume the ability to predict aspects of a computation's behavior, although the form of the predictions vary with the specific needs of each system. This need for a significant level of prediction is one of the ways in which real-time systems, particularly hard real-time systems, are significantly different from conventional systems [84, 85].

The Spring project has, for many years, been investigating the design and implementation of hard real-time systems [87]. The project began by developing scheduling algorithms for systems operating in complex dynamic environments [97, 70, 7, 76]. The algorithms assume that computations are represented by one or more *tasks* with known deadlines, WCET, precedence relations, and resource use. The scheduler *guarantees* each task will meet its deadline by dynamically constructing an explicit execution plan, called a *feasible schedule*, satisfying every task's deadline, resource use, precedence, and WCET constraints. Note that the worst case execution time and resource use is used when building a feasible schedule to ensure that the computations will complete by their deadline under every possible circumstance, excluding faults. Tolerance of specific types and severity of faults is also possible, but is not the focus of this dissertation.

Each time a new computation requires a guarantee, the scheduler attempts to construct a feasible schedule containing those tasks in the current feasible schedule as well as the tasks representing the new computation. The Spring scheduling algorithms construct the feasible schedule task by task, guided by one of several heuristic functions, backtracking as necessary and appropriate. There are several algorithms representing variations on the basic theme, each with individual properties and performance [93].

Having developed and validated these scheduling algorithms through simulation, the next logical step was to consider how a system using them, and capable of satisfying both their implicit and explicit assumptions, could be implemented [52]. The most attractive property of the algorithms is their ability to dynamically guarantee tasks as they arrive, but this comes at the price of providing the detailed behavioral information the algorithms require. A *practical* implementation of Spring, and of

most real-time systems, must thus be *predictable* in the sense that they must support the creation and execution of programs whose worst case behaviors can be predicted as required by the scheduler. Note that for Spring this does *not* mean that *every* aspect of program behavior must be predicted, only those behaviors required to support the guarantee algorithm(s) used [57]. Behaviors that are *not* predicted include all those associated with the semantics of the program such as division by zero and other logical errors.

This dissertation describes novel designs and implementations addressing many of the problems associated with building Spring; a *predictable real-time system* capable of supporting and using the Spring scheduling algorithms. Adopting predictable worst case behavior as a primary design criterion challenges a wide range of conventional system designs and design principles [85]. The average case pervades all levels of conventional computer system design and implementation as a performance metric, design criterion, and implicit or explicit assumption. Adopting the new criterion thus motivates a significant reconsideration of conventional approaches at every level; familiar approaches providing good average case performance may or may not provide adequate, and adequately predictable, worst case performance [86].

Finally, it is important to note that adopting predictable worst case performance as a primary design criterion strongly motivates a highly *integrated* approach to system design. Design and implementation decisions at *every* level of the system affect the predictability of WCET, and decisions at one level frequently interact with those at other levels. As a result, it is difficult to construct an adequately predictable system when treating the several traditional levels of system design as nearly autonomous. On the contrary, as each design decision is taken, any interactions with every level of the system must be considered. Leaving analysis of such interactions to a later design

phase is unlikely to produce a sufficiently coordinated, and thus predictable, design. An analogy to a familiar proverb is appropriate here. Just as a chain is only as strong as its weakest link a real-time system, to a significant degree, is only as predictable as its least predictable aspect.

This dissertation presents the results of our attempt to discover how a predictable real-time system can be built, what aspects of its design affect its predictability, how difficult it is to alter the design of specific components for greater predictability, and whether the advantages afforded by increased predictability justifies the effort such alterations require. The degree of predictability achieved and the effort required to achieve it indicate how radically the design of next generation real-time systems are likely to depart from conventional practice. The rest of this chapter will first give an overview of the research discussed in the dissertation, and then describe what portion is addressed by each chapter.

1.1 Research Overview and Contributions

This dissertation considers several problems of conceptual interest, but has a significant practical component as well since it addresses the design and implementation of an entire, albeit still small, computer system. This point is emphasized by two questions ultimately motivating the work described here:

1. How can we design and build a real-time system which represents a computation's behavior as a set of non-preemptable segments with known characteristics and which is capable of supporting complex real-time applications?

2. Will its design differ from that of conventional systems?

The Spring system is an example of a system with the relevant characteristics. It is a distributed real-time system which can support the predictable execution of complex real-time computations. The Spring approach to scheduling assumes that a computation is represented as a set of tasks with known WCET, resource use, and execution precedence constraints. The importance of Spring's approach to scheduling and its advantages *vis a vis* other popular scheduling methods have been discussed in detail elsewhere [97, 70, 77, 76, 98]. This previous work, extensively testing the various Spring scheduling algorithms via simulation, did not reduce the need for extensions to make them effective and efficient in the context of the Spring system implementation. We also felt it was important to design the system to support a wide range of scheduling algorithms, and to make choosing an algorithm at boot and/or run time as flexible as possible. Many of the results presented are as applicable to real-time systems using other scheduling approaches as they are to Spring, since most approaches to scheduling require the kind of information about the WCET, resource use, and other characteristics of the computations being scheduled whose prediction is discussed in this dissertation.

The work addressed in this dissertation falls into four broad categories:

1. the Spring-C programming language and Spring's programming environment,
2. integrated program translation and behavior prediction,
3. operating system features and ensuring predictable program execution, and
4. suggestions for new hardware designs improving the predictable WCET and other properties of the system.

We wanted, if possible, to use a process based programming model. The advantage provided by its familiarity are obvious, and we decided to adopt it, subject

to modification as experience revealed problems. The Spring programming environment consists of the familiar elements of most Software Generation Systems (SGS): compiler, assembler, linker, debugger, and related analysis and inspection tools. An advantage of adopting a conventional programming model is that we could start with existing tools available from the Free Software Foundation [81], thus reducing the task of implementing Spring's SGS to an ambitious but achievable scale.

The source language developed, Spring-C, extends the C language in two significant ways, and is discussed in Chapter 4. First, several syntactic changes and use restrictions guarantee that every Spring-C program is *finite*, and that reliable worst case behavioral predictions can be made for every program. Every Spring program is finite in the sense that it will either terminate, or the violation of its behavioral constraints will be detected and its execution aborted. Second, the System Description Language (SDL), discussed in Chapter 4, was developed to support specifying all information required to describe, compile, and predictably execute Spring applications. The SDL is fully supported by the Spring-C compiler, permitting descriptive statements to be inserted into application source files as appropriate.

The other tools in the SGS use, modify, and produce information supported by various parts of the SDL as Spring-C programs are compiled, linked, downloaded, and executed. The SDL information originating in each source file is collected during compilation, and stored in a compiled form making it available to tools performing subsequent processing steps. The compiled SDL information from earlier steps is used during subsequent compilation of other source files, is used when linking application executables, and is used by the Spring debugger when downloading system and application code for execution as well as during debugging. The SDL thus provides the

mechanism for preserving and exchanging information by which the various stages of SGS operation are coordinated and controlled.

As the Spring-C compiler processes a source file, it predicts the worst case behavior of each procedure. These predictions obviously depend on, among other data, the behavioral predictions for procedures called by the procedure being processed. When the behavioral prediction for a procedure is generated, it is preserved using the SDL and is thus available for use during later compilation of other procedures calling it. The behavioral prediction for the process as a whole is used to build the task group representation of the process's behavior required by the Spring scheduler. The behavioral prediction for the process is given by the prediction for its entry point procedure (e.g., *main* in conventional C). In addition to producing behavioral predictions, the Spring-C compiler performs all the normal functions of a conventional compiler.

Enabling the compiler to perform the behavioral prediction and build its task group representation required significant changes in several areas. The parser was changed to accept the unique features of Spring-C syntax, which are primarily oriented toward supplying information required for behavioral prediction. The intermediate representation used by the compiler was modified to represent this and other important information, thus making it available to the new compilation phase producing the behavioral prediction and its task group representation. The design and implementation of this new compiler phase represents the single most significant contribution presented in this dissertation.

At the operating system level, this dissertation presents contributions in two areas: extensions to the scheduler implementation, and the design and implementation

of several other aspects of system support required for predictable run-time management. The extensions to the scheduler implementation enable it to handle computations represented as groups of tasks, with precedence relations constraining task execution order, correctly and efficiently. Previous schedulers only addressed computations represented by single independent tasks. Contributions to other aspects of system support include, but are not limited to, *predictable* implementations of: the memory management routines supporting independent logical address spaces for each process, simple and efficient support for system calls, and context switching adapted to Spring's specific needs.

A feature of practical interest is Spring's implementation as a *distributed multiprocessor microkernel.* Spring's OS is distributed across a set of processor boards, each with on-board memory that is both locally and globally accessible, on a VME backplane. One board is designated the system processor (SP), while the others are application (AP) processors, and I/O processors. Different versions of the system code run on the SP, AP, and I/O boards. System and application code execution on each board is kept as independent and asynchronous as possible, but necessary synchronization is supported by the scheduler, semaphores, and a simple handshaking protocol as appropriate.

The SP version of the OS is implemented as a microkernel, with the scheduler and the code handling process activation running as independent processes. The process activation code includes any preprocessing required by the scheduler to initialize specific task attribute values [98, 95]. This design makes selecting a scheduler at boot time trivial, since the user specifies which set of scheduler and process activation processes to run, as described in Section 4.2.6. By extending the current microkernel it would also be possible to have several schedulers active *simultaneously*, enabling the

system to choose the most appropriate algorithm each time a scheduling operation is required, subject to the compatibility of their run-time data structures. Switching among schedulers using different data structures to represent computations would be more difficult, but might still be worthwhile to support changing schedulers during mode changes [71].

Finally, our experiences during the implementation of a predictable system using the current target hardware revealed a number of inconvenient and inappropriate constraints arising from the hardware's average case orientation. We have developed a number of principles and suggestions which we believe should guide the design of hardware intended to support real-time systems. The desire for new designs arises since we have now changed the primary design criterion from optimizing average case performance to optimizing the predictable worst case time. We have also taken a first step toward hardware specifically designed to support real-time systems with the Spring Scheduling Co-Processor (SSCoP). SSCoP is a custom VLSI chip implementing the Spring scheduling algorithm. It is designed as a daughter board plugging into the existing PROM sockets of the SP, and serves to accelerate the execution of the scheduling algorithm [9, 60].

In summary, this thesis presents a number of contributions, of both theoretical and practical interest, at every level of system design and implementation, including:

- the Spring-C programming language which is process based and thus familiar, yet contains the new features required for a predictable real-time system;
- program translation methods which predict the worst case run-time behavior of any Spring-C program and builds a task group representing that behavior to the scheduler;

- an integrated software generation system including tools for compiling, linking, loading, analyzing, and debugging Spring system and application code;
- a predictable operating system implementation featuring an enhanced scheduler handling precedence constraints, memory management support of independent address spaces and shared segments, as well as other essential aspects of program execution;
- hardware design principles and suggestions adopting optimization of the predictable worst case execution time as a primary design criterion;
- the SSCoP, a collaborative design effort of several researchers [9], as a concrete example of hardware specifically designed to support real-time systems.

Finally, an important aspect of the Spring system is the high degree of interaction among issues in the programming interface design, program translation methods, program execution support, and hardware properties, which had to be considered and coordinated to produce a system with the desired properties. We believe that this clearly demonstrates the need for highly *integrated* design and implementation methods for real-time systems, and is itself an important contribution of this thesis.

The practicality and correctness of the working system has been demonstrated in a number of ways, including:

- the programming interface was used to implement extensive test cases for behavioral prediction and scheduling, as well as a simulated robotics applications modeled after a real robotic manufacturing examples [6, 94];
- behavioral prediction methods were verified using a series of test cases focusing on specific source code structures as well as a number of test programs whose behavior was measured and compared to the predictions;

- the practicality of the SGS was demonstrated by its use in compiling, linking, downloading, and debugging the Spring operating system, subroutine libraries, independent shared segments, and application programs;

- the operating system's predictable run-time management of computations was demonstrated by testing the logical address space support, predictable context switching time, and its successful execution of the test programs verifying the behavioral predictions, as well as application code;

- and finally, the need for, and practical utility of, hardware designed specifically for real-time systems was demonstrated by our experiences in implementing Spring on conventional hardware, and by the design and implementation of the Spring scheduling co-processor, respectively [9, 60].

1.2 Dissertation Organization

The organization of the dissertation is simple, addressing each major category of the contributions described with a separate chapter. Additional chapters discuss state of the art in relevant areas, provide an overview of the Spring system, and present our conclusions and future work. Chapter 2 describes related work in several categories, helping establish a context within which to consider the Spring system and the results presented by this dissertation. Chapter 3 provides an overview of the Spring system, and gives an idea of how the elements described in subsequent chapters fit together.

Chapter 4 describes Spring's programming environment, the Spring-C programming language, and its use for specifying, preserving, and presenting all the information required to coordinate the compilation, linking, loading, execution, and debugging of the Spring system and its applications. The chapter concludes by discussing how Spring-C was evaluated by using it to implement several programming test cases.

Chapter 5 presents an overview of how behavioral predictions are produced and used, and then describes how the original compiler was modified to support behavioral prediction. Chapter 6 presents the method used to produce the behavioral prediction, called "subgraph reduction", in detail. The chapter describes several classes of reductions and how they address specific source level constructs. The chapter then discusses how the validity and accuracy of the predictions were evaluated, considers the complexity of the behavioral analysis performed, and briefly discusses how the currently implemented methods could be extended. Chapter 7 discusses in detail how the behavioral predictions produced by the subgraph reduction phase are used to construct a task group representing that behavior to the scheduler, and the evaluation of the method using a set of simple test cases.

Chapter 8 discusses how using the predictability of worst case behavior as a primary design criterion affects operating system and hardware design, concentrating on three areas of particular interest: memory management, execution time prediction, and scheduling. The main points of the chapter are illustrated with examples arising from the implementation of the Spring system, and the effectiveness of the approach demonstrated by the system's ability to support the predictable execution of real-time applications. Finally, Chapter 9 presents our conclusions, and describes the future work we are considering to extend that presented here.

CHAPTER 2

STATE OF THE ART

There are four general areas of current real-time systems research that have a direct bearing on the results discussed in this dissertation: real-time languages and the computational models they use, predicting program behavior, real-time operating systems which includes their scheduling policies, and hardware designs explicitly considering the needs of real-time systems. This chapter will discuss each of these topics. We begin by considering programming languages in which real-time computations might be implemented. The properties and assumptions of the source language determine the vocabulary available to the developer for describing computations, much of the information available to support analysis for predicting behavior, and many aspects of how the computation must be managed at run-time. We then discuss several other approaches to predicting run-time behavior differing from the methods presented in this dissertation.

The methods used by the system to manage the execution of a computation at run-time influence the type of behavioral predictions that can be made, as well as influencing their validity. Different real-time systems take widely differing approaches to system design and implementation which strongly influences, in turn, the management of computations at run-time. We discuss several current systems, both commercial and research oriented, and consider how their implementation influences their predictability. Finally, we consider how aspects of the hardware design affect

the behavior of the system and its predictability, and then summarize how the work described in this dissertation relates to other current work in real-time systems.

2.1 Languages for Real-Time

The source language level is where the developer specifies the properties of the real-time computation. This description must provide enough information for the run-time management portion of the system to produce the desired behavior using the representation of the computation described in the source. At a high level features of many real-time languages resemble one another, since they are trying to address the same or similar set of issues. However, at a slightly more detailed level differences in what information is specified, how that information is used, and the properties they assume the run-time system possesses create significant differences.

There are many real-time languages that make a claim of suitability for real-time programming using a variety of arguments. Some are based on a view of real-time systems as a simple extension of conventional (non real-time) computer systems, perhaps enhanced by methods for finding the current time and setting alarms. This view can lead to the explicit or implicit acceptance of several misconceptions about real-time systems [84]. One of the most obvious misconceptions is, "the computer is so fast everything will always be done in time". Experience has shown that creating systems which can support computations which will reliably and predictably meet their deadlines requires more than selecting a powerful CPU.

Burns and Wellings give a good description of real-time systems in general and describe their implementation using straightforward adaptations of techniques used in conventional systems to real-time [10]. Real-time languages that step beyond manipulating process priorities often add statements about the temporal constraints

of computations to the syntax of the language. However, most current real-time languages using the process model for programming also assume the conventional run-time model which manages a set of processes preempting one another according to their execution priority, competing for resources, and blocking when resources are already in use.

The most important requirement for a real-time system is that its behavior be *predictable* [84]. Predictions about the behavior of individual computations, and about the set of computations comprising the system as a whole, is the basis for most current real-time system research. The form and limits of the predictions vary, depending strongly on the design philosophy and implementation strategy used. Further, the predictions about the system behavior usually specify the *worst case* behavior, not the average case, because the run-time system using them assumes that the predictions will never be violated.

The issue of predictability arises in all phases of real-time system design, but begins at the language level since that is where the *required* behavior is specified, and from which the run-time description of the computation is derived. Using predictability and other criteria Halang and Stoyenko have, for example, considered what properties a real-time language should have and have discussed how a number of existing languages address their requirements [26]. The criteria are divided into three groups: those describing the "language elements facilitating the construction of application software", those describing "the run-time language system services needed for predictable and reliable real-time software performance", and those required for "software verification and analysis mechanisms". The results described in this dissertation, along with the work of many others as embodied in the Spring system, all but two of their criteria are either satisfied or not applicable, as discussed below.

Halang and Stoyenko note that a real-time language should contain:

- application oriented synchronization constructs,
- surveillance of the occurrence of time events within windows,
- surveillance of the sequences in which events occur,
- time-out of synchronization operations,
- time-out of resource claims, and
- availability of current task and resource states.

The language discussed in Chapter 4, Spring-C, and the system supporting it exhibit all of these properties. Spring-C contains application oriented synchronization constructs. The need for time out of synchronization operations and resource requests is eliminated by the use of a scheduler which builds explicit execution plans, and which keeps track of current task and resource states [70]. The operating system, discussed in Chapter 8, keeps track of the occurrence of time events within windows and of the sequence in which events occur.

The set of system services required to ensure a real-time system is predictable and reliable includes:

- inherent prevention of deadlocks,
- feasible scheduling algorithms,
- early detection and handling of transient overloads,
- determination of entire and residual task execution times,
- task oriented look-ahead virtual storage management,

- accurate real-time,
- exact timing of operating system services,
- dynamic reconfiguration of distributed systems when a failure occurs, and
- support of software diversity.

The scheduler building explicit execution plans inherently prevents deadlocks and provides early detection of transient overloads. It is not, however, feasible in the sense of always finding a schedule if one exists. It employs heuristic methods which can fail to find a feasible schedule, but does so rarely [70]. The Spring-C language and the compilation methods described in Chapters 5, 6, and 7, satisfy the requirement for determining task execution times, the exact timing of operating system services, and software diversity. The system provides an accurate real-time clock. The system does not currently support dynamic memory allocation or virtual memory, nor does it yet address dynamic reconfiguration when failures occur. The Spring-C language described in Chapter 4 does, however, provide some support for fault tolerance which should be useful as the system is developed further.

The software verification and analysis mechanisms Halang and Stoyenko advocate include:

- application oriented simulation of operating system overhead,
- interrupt simulation and recording,
- event recording,
- tracing,

- usage of static features if necessary, and
- schedulability analyzability.

The simulation of operating system overhead and interrupts is not applicable, since the functional decomposition of the Spring kernel design isolates application computations from interrupts, and the overhead of system services is bounded [87]. The system provides for recording events, the system debugger provides the ability to trace system execution, and the system provides for specifying only static behavior if so desired. The Spring simulation testbed [20] enables developers to extensively test the scheduling behavior of task sets with many different properties, but two factors place limits on *a priori* assertions that a feasible schedule will always be found. First, since tasks arrive dynamically it is not possible to ensure that a feasible schedule will always exist. Second, since the scheduler employs heuristic search to construct an explicit execution plan, it is not possible to guarantee that a feasible schedule will be found, even if one does exist.

Using these criteria, Halang and Stoyenko consider a number of languages which are essentially conventional with simple priority manipulation and time related features added inadequate. One such effort is the process control language PEARL, which arose from the process control community in Europe [88, 26]. It has a number of attractive features including: ways of decoupling the specifications of the computations from the architecture of the underlying hardware, ways of checking the validity of sensor data, and for other aspects of detecting and handling faults. It has a long history of successful industrial application, but its approach is essentially conventional, and does not address the problem of predictability. Halang and Stoyenko suggest a number of extensions to PEARL in the context of their views on constructing predictable real-time systems in [27].

While Ada is defined by DoD as the standard implementation language for real-time and embedded systems, there are a number of problems that make it unsuitable for real-time programming, by Halang and Stoyenko's criteria [26, 27]. The general source of their dissatisfaction is the lack of provisions in the language for ensuring predictable behavior. Baker notes the inconsistency between Ada's priority scheduling semantics and the requirement to service **rendevous** requests in FIFO order, which can lead to potentially unbounded priority inversion [2]. Sha discusses the problems encountered when applying rate monotonic theory to writing Ada programs, and makes several suggestions for how Ada might be changed [74]. The main message of these discussions seems to be that while using Ada as a real-time language may not actually be impossible, it is both ungainly and unpleasant.

Kligerman and Stoyenko produced a restricted language, Real-Time Euclid, which was designed to make *schedulability analysis* possible, under a number of assumptions about the system and process behavior, [40, 89]. Schedulability analysis addresses the question of whether or not processes in the system will satisfy their temporal constraints under worst case conditions. The analysis takes worst case execution times, blocking, and preemption behavior into account when making its assessment. The language provides only counted loops, and prohibits both recursion and dynamic memory allocation, but this work established that a system with significant levels of predictability could be built. The restrictions on loops is similar to that imposed by Spring, described in Chapter 4, but the information provided is used in different ways because of the difference between Real-Time Euclid's preemptive deadline driven process based run-time management methods and Spring's task based run-time management models. The difference between these models is discussed in Section 2.2.

A more recent language effort is Real-Time Concurrent C, which has added constructs to Concurrent C for specifying periodicity or other timing constraints, to seek guarantees that the timing constraints will be met, and to perform alternative actions when either the alternative actions or the guarantees are not available [19]. The language provides a reasonable vocabulary for specifying the real-time behavior that is desired, but since the current implementation manages multiple execution threads within a single UNIX process, there are limits to how predictable the language can be made and the strength of the guarantees it can provide without predictable system support. Another concurrent language, RTC [96], is similar in providing ways to specify temporal constraints and the desire for guarantees. In this case, however, the run-time system is supported by a real-time kernel using an earliest deadline first scheduling policy.

Research addressing the programming language for the ARTS system uses C++ its starting point [33]. They have added constructs for specifying a wide range of thread sets and relations, as well as timing constraints including start time, deadline, exception handling, and periods. They assume the use of rate monotonic scheduling in the underlying system, but little is said about how the WCET, which rate monotonic scheduling requires, can be determined in the object oriented environment.

The programming language for the Maruti system, MPL, also extends C++ and uses ideas close to those of Spring in some areas [63]. MPL provides several ways to specify temporal constraints on blocks of code within an object. Loop bounds are specified and recursion forbidden for the obvious reasons. Multiple threads within an object create somewhat more complex synchronization issues which must be considered when predicting behavior of a computation implemented by a thread visiting many objects. The implementation described produces C code running under Mach.

Their ability to predict the actual execution behavior of a computation is thus limited by the predictability of Mach's support.

The predictability of application programs is improved using an interesting application of partial evaluation to simplify the code being analyzed [62]. One drawback to the method is that it can greatly increase the size of the executable since a single routine may spawn several specialized versions. Maruti assumes an approach to real-time scheduling that is the closest to that of the Spring project of all current efforts discussed here. They consider how to extract schedulable segments from their object oriented code [31], and apply various optimizations and transformations to the code to improve the schedulability of its behavior. However, there are also important distinctions arising from differences in the programming models, how program translation and analysis are implemented, how the execution plans are constructed, and the nature of the support provided by the system.

The programming environment of the MARS project takes a slightly different approach, expressing all computations in terms of modules which receive messages as they begin execution, send messages as they complete, and execute periodically [67]. This has aspects of object orientation and data flow about it, although the language, Modula-R, is an extension of Modula-2 and is not not explicitly object oriented. Their approach has the advantage of regularity and simplicity, is well coordinated with the properties of the supporting operating system, and provides a programming environment with several interesting features.

Programmers are responsible for decomposing computations into a set of communicating modules. This is superficially similar to the Spring model, in which developers implement computations as sets of processes, but is different in two important respects. First, the programming model is restricted to processes which communicate

only by sending messages, and further restricts processes to receiving messages only at the beginning of their execution, and to sending them only at the end. This simplifies the analysis and compilation of the programs, as well as their predictable execution but is a more restricted programming model than that provided by Spring-C which permits the exchanged of messages *during* a process's execution, as well as the use of shared memory segments.

The second important difference is that the decomposition of the computation into a set of schedulable entities, in MARS' case processes, is *static.* The method described in Chapter 6 for deriving a task group representation of a process or process group and thus of a computation, while it does not currently do so, can be extended to permit a computation's properties and their interaction with those of the system and the other computations it supports to affect the number and properties of the tasks in the group generated to represent the computation. This decouples the description of a computation provided by the developer as a set of processes, from the representation used by the system to execute the computation, which is the task group. Since the methods described in this dissertation automatically produce the task group description of a computation, they also represent an opportunity to automatically adjust the task group representation to different target node configurations as well as other properties of the system.

The data flow computation model is attractive for some classes of applications, including the design of process control and other reactive systems. Faustini and Lewis describe Real-Time Lucid, which is a data flow language enhanced with time stamps on the data tokens, making time part of the enabling condition for each node in the data flow graph [16]. They recognize that using this approach at run-time is probably infeasible due to the overhead involved, and suggest compile-time analysis producing

a specification of time windows for the production and consumption of data at each node in the data flow graph. If these windows are violated, a timing exception is generated. However, the timing windows also provide information which makes it possible to consider shifting the language from dynamic to static scheduling of the data flow graph. This would require constructing a schedule executing each node only when its inputs are ready, but before its output is required.

Lee describes a system addressing the signal processing application domain which does static scheduling of data flow graphs [43]. The data flow graph for a particular signal processing application is analyzed, and a static schedule constructed which satisfies the data consumption and production constraints of the nodes in the data flow graph describing the application. This depends on knowing the WCET for the code implementing each node in the data flow graph, which is generally easy to determine for the calculations typical of the domain. Other work investigating the use of synchronous data flow languages for reactive programming [5], include work on the languages LUSTRE [28] and ESTEREL.

ESTEREL is a synchronous language developed to address reactive programming which incorporates deterministic action, the atomicity of reactions, and instantaneous broadcast in its programming model semantics [8]. ESTEREL programs are expressed as a set of modules which take a given set of signals as input, and produce output in reaction to the input signals. The language includes an operator specifying which portions of a computation may be executed in parallel. ESTEREL programs are compiled into finite state automata (FSA) using a number of target languages, including C, which embody the reactive behavior specified by the source program.

During the compilation, the parallelism is eliminated, producing a sequential program which embodies a specific execution interleaving of the parallel portions of the source program.

The implementation of ESTEREL programs as FSAs means that they can be executed with comparatively low system overhead, and that it is possible to consider verifying their correctness through the construction of proofs, as well as through simulation. The approach does, however, have some weaknesses. First, the FSA produced by some programs are very large, and require that the computation be implemented as a set of smaller FSAs, whose execution is coordinated by a higher level system executive. Another problem is that the assumption of instantaneous broadcast communication makes it difficult to see how ESTEREL programs could be effectively distributed across processor boundaries where non-zero transmission delays predominate. Finally, while ESTEREL is presented as a possible language for implementing real-time systems, the only justification for this given is the efficiency of the FSA implementation and that "the maximum transition time of an automaton is predictable". Since ESTEREL programs are compiled into other target languages, C for example, predicting their behavior is still subject to the difficulties and limitations discussed in this dissertation. It would, however, be possible to consider using Spring-C as the target language, and thus produce FSA whose execution time could be accurately bounded, and which could thus be executed predictably.

The Real-Time Mentat language combines data flow and object oriented programming to create a language with support for real-time computations with both hard and soft deadlines [24]. The language is a derivative of C++ with extensions for the support for the automatic detection of data flow relations and the generation of data flow graphs, as well as to express timing constraints and provide scheduling directives

to the run-time system. Only those portions of an application with hard deadlines are required to have bounded execution time and resource requirements that can be determined at compile time. Computations with soft deadlines can be implemented more dynamically, with data flow graphs that are determined at run-time.

Some researchers take the view that program behavior cannot be predicted adequately, and that it is better to *measure* behavior. Kenny and Lin describe the Flex language, another extension of C++, which includes a number of timing constraint expressions and exception handling clauses [36]. This research is particularly interesting because it addresses the issues of approximate processing by adopting a polymorphism analogous to operator overloading [47]. In this instance polymorphism refers to supplying several routines implementing the same function which have different properties in space and/or time. Characteristics of the system as a whole, either at compile or run-time, can then be considered when choosing which implementation to use to perform a particular instance of a computation. They propose having the developer supply a template for an execution time equation and finding coefficient values by experiment, but without discussing how to chose testing data that will produce worst case, or at least representative, behavior.

The GARTL language also investigates the use of multiple versions of a task [49]. This system is an extension of Spring's approach, and so makes the same assumptions about tasks and their characteristics. The idea of monitor variables is introduced, and the different versions of a task are identified by the tuple defining the set of monitor variable values for which it is valid. The scheduler chooses the best version at run-time based on the current values of the monitor variables. Execution times are measured by experiment, creating the same problems with knowing if the test data produced the worst case behavior or not.

Other research in object oriented languages for real-time also moves away from worst case considerations, instead predicting behavior through measurement and adapting run-time management to monitored behavior [21]. This work involves a number of interesting features. The emitted code is annotated to produce the information monitored at run-time. This makes it possible for the system to detect deviations from the expected performance or behavior of the system as early as possible. Compiler optimizations are applied which move code to improve the temporal aspects of its behavior.

Measurement cannot, however, determine the WCET unless the test data is certain to produce worst case behavior. The importance of knowing the WCET depends on how tolerant the system and application are of a computation missing its deadline, and on how the system ensures deadlines are met. GARTL uses Spring's scheduling method, and violation of the execution time estimates would be problematic. Methods for describing imprecise computations and monotonic algorithms can reduce the need for an accurate WCET estimate [47]. In this case the word "monotonic" refers to a calculation which produces an approximate result comparatively early during its execution, and then steadily improves it. Such computations are sometimes split into mandatory and optional portions. The mandatory part must complete before the first approximate answer is available. The optional part iteratively refines the first approximation. A WCET is sometimes calculated for the mandatory part, ensuring *some* answer is available by the deadline as long as the system allots at least that amount of time to the computation. However, we believe that while important, imprecise calculation is not applicable to every real-time problem, and so reliable WCET predictions will still be required.

Gopinath and Gupta address how the compiler can be used to assist with adaptive scheduling in real-time systems [22]. This work involves annotating the emitted code, and monitoring its progress during execution, but also divides the program being compiled into blocks of code with specific properties. This works assumes that code blocks can be classified as predictable and unpredictable, and as monotonic or non-monotonic. The predictability of a code block is determined by the variance of its execution time. Predictable blocks have low variance. This is still an essentially average case approach to the problem since no mention is made of the WCET, as is addressed by Spring and the work described in this dissertation. The role of the compiler is to reorder the emitted code segments so that the program as a whole becomes easier to schedule. One of the main strategies is to place the unpredictable code blocks first, so that as the execution of the program proceeds, and thus approaches its deadline, it becomes more predictable. The system, by monitoring the program's progress, can then adjust the execution time of the monotonic portions of the program as indicated by the relation between the program's progress and the time remaining before its deadline.

A prominent feature of Spring-C is the System Description Language (SDL), discussed in Section 4.2 which is used to specify, store, and distribute information about all aspect of applications that run under Spring. One part of the SDL supports the description of processes and process groups used to implement computations. The information specified includes scheduling information, precedence relations describing the structure of the group, description of shared memory segments used, and other details required by one or more portions of the system to properly compile, download, or execute the applications.

The Conic environment addresses similar issues in the distributed systems area providing, among many other features, ways to describe the configuration of processes into logical nodes, and for describing in detail the exchange of information between processes [48]. This is very similar to the SDL in some respects, since it addresses relations among processes and specifies information required to support their execution. Differences arise from the SDL's addressing the needs of distributed predictable real-time systems as represented by Spring, in contrast to Conic's addressing the needs of non-real-time distributed applications running on heterogeneous systems of conventional design.

An effort in a similar direction, but designed with real-time systems in mind, was made by Barbacci and Wing in producing the Durra [4] and Larch [3] languages. Durra provides support for describing the ports, signals, and other attributes of a process, as well as conditions under which the set of active processes should be modified. Larch was designed to support the specification of the functional and timing behavior of real-time applications on heterogeneous machines. As a result, it has a different focus than Durra, addressing the properties of the computations involved, rather than their configuration. The SDL combines the functions of both Durra and Larch in the Spring system, and so shares some of its features with both. However, the specific needs of a predictable real-time system and the Spring implementation, in contrast to the general needs of Larch and Durra's heterogeneous target machines, have created significant differences as well. The most obvious is the use of resources by Spring's scheduling method, and the SDL's corresponding description of a task's resource use. Other differences include the SDL support of shared segments, the description of the target hardware, and for the assignment of processes to specific processors within the target nodes.

2.2 Run-Time Computation Models

The differences between the requirements of real-time systems and conventional (non real-time) systems can be difficult to appreciate at first, which can lead to a number of misconceptions [84]. One of the most common is that if the computer is made fast enough, then every real-time computation will finish by its deadline because the speed of the computer will overwhelm all other considerations. This is a particularly prevalent misconception because it is *true* for simple applications with undemanding deadlines using hardware which provides abundant computing power compared to the demands of the application.

When more demanding real-time applications are considered, one of the areas most obviously requiring modification is the programming language, as discussed in the previous section, since developers require some way to specify the temporal constraints and properties of the real-time computations they are implementing. The features of real-time languages differ from those of conventional languages as thought necessary by their designers to help the system satisfy the real-time constraints of the applications using it. The language features are not, however, the only aspects of the computer system whose alteration must be considered in response to the unique demands of real-time systems.

A programming language, and the run-time management of computations described using the language, are designed within the framework of a computational *model*. The most common programming model is the *process*, a single thread of control executing in an independent address space which competes for shared resources with other processes, blocks while required resources are unavailable, and which is subject to preemption at arbitrary times. In conventional systems the process model is an excellent choice because it enables designers at the application, system, and

hardware levels to make a number of powerful simplifying assumptions. Using the process model at run-time is attractive because as long as the execution of the processes is managed according to a few relatively simple rules, then the results produced will be logically correct.

The run-time management methods in most conventional systems use a priority driven process model, always executing the process with the highest priority and dynamically modifying their priorities as a way of organizing the system's activities to ensure "fair" treatment. Process priorities are modified in response to a number of conditions, typically considering several significantly different issues in the policy setting initial process priority and determining how it changes. The issues commonly influencing process priority include how much execution time a process has consumed in the recent past, its anticipated execution time, its recent I/O behavior, as well as differences in the inherent importance of one process over another. Within the framework of the process model, designers of both hardware and software for conventional systems typically concentrate on improving the *average case* performance at all levels of the system, and consider the occasional delay or neglect of a specific computation under the designs they adopt as insignificant.

In conventional systems the techniques used to control the use of shared resources are also focussed on the average case. When a process attempts to use a shared resource that is already in use, it typically blocks. When the resource in question becomes free, then the blocked process can be made runnable again. When the aim of the system is to provide good average case performance this is a perfectly reasonable choice, because in the context of the conventional computer system every computation makes reasonable progress and is able to use the resources in a logically consistent manner.

Real-time systems, in contrast, are not generally concerned with fairness, but with ensuring that computations complete on or before their deadlines. Much of the theory and practice that has been developed concerning the scheduling, schedulability, and run-time management of real-time computations requires that their *worst* case behavior be known [46, 69, 70, 45, 89, 15]. Knowing the WCET requires that it be reliably *predicted* either by measurement or analysis which requires, in turn, that the system *as a whole* be constructed in a way which makes such predictions possible. The predictability of worst case behavior is, however, a fundamentally different performance criterion than the optimality of the average case. It is thus reasonable to assume that basic design decisions made using optimization of the average case as their performance criterion must at least be reevaluated, and often reformulated, in light of this change. Yet, while the average case orientation pervades most conventional application, system, and hardware designs, many approaches to real-time systems have simply applied conventional methods with little or no modification.

The most obvious examples of this approach are the "real-time" UNIX implementations [17] but many systems designed specifically for real-time, of which VRTX is a typical example [72], exhibit equally conventional designs. These systems generally concentrate on making the system more preemptable, decreasing interrupt service latency, and providing a specific set of real-time system calls with bounded execution times, without modifying the average case orientation of the system design or its scheduling methods. The system still executes the runnable process with the highest priority at any given time, preempts processes as priorities change, and attempts to service interrupts with the greatest possible speed. Such systems often add an application level interface for modifying process priorities, setting timers, and add a clock to the design in an effort to give the application layer influence over why and when

process priorities change, and thus over the system scheduling policy. However, processes still compete for resources at run-time, block while they are not available, and their deadlines are generally not considered by the system scheduler. These systems are thus, broadly speaking, still average case designs which have been modified to increase the likelihood that processes executing on them will meet their deadlines, but which cannot be called predictable in any meaningful sense [85].

The rate monotonic approach to scheduling in real-time systems is particularly appealing to many practitioners because it requires less radical changes to the system design than other approaches. It uses the process model and simple priority scheduling at run-time but provides a theoretical basis for predicting whether a set of periodic processes will finish by their deadlines when processes' assigned priorities are directly proportional to their execution frequency [46, 69]. One of the most important aspects of this approach is that it assumes the processes' worst case behaviors, both execution time and resource use, are known. Thus, while its run-time management scheme appears essentially conventional, it requires that the system's worst case behavior be predictable.

Another important aspect of rate monotonic scheduling is that it accounts for process execution delays from resource contention by calculating the worst case blocking time of every process, and taking this into account in the scheduling analysis. Since this analysis assumes that every possible interaction over resources is always taking place, it will often over estimate the blocking time for specific execution scenarios. A method, such as that assumed by this dissertation and employed by Spring, which explicitly constructs an execution plan only needs to consider the contention for resources which actually occurs. Finally, rate monotonic scheduling also assumes that each computation can be preempted an arbitrary number of times with no overhead.

This is clearly unrealistic, but recent work is attempting to take system overhead into account when performing schedulability analysis [35].

While its assumptions create some limitations, using the rate monotonic approach to scheduling does enable the design and implementation of predictable real-time systems for a number of application domains using the process model for programming and at run-time. One of the potentially significant limitations of the original theory is that it only considers periodic computations. Extensions for aperiodic computations essentially allocate a periodic server for aperiodic computations [80]. However, this will not necessarily guarantee the successful execution of aperiodic computations for which a feasible schedule might exist, since insufficient time may be allocated to servicing aperiodic computations under a specific execution scenario, while excess time is allocated to the periodic tasks. Another potentially significant limitation is its lack of support for synchronous communication among processes, since its schedulability analysis does not take into account the *order* in which portions of each processes execute. This clearly limits the software architectures that can be used to implement a computation under the rate monotonic approach.

The work on the specification [36] and scheduling of imprecise computations [47] addresses the question of how to make the production of a usable result by a deadline more predictable from a different perspective. It uses the monotonic computation model at programming and run-time, which enables the system to produce a usable result even when the worst case behavior of every part of the computation is not known. The system scheduler only has to ensure that the mandatory portion of the computation is completed by the deadline to guarantee a usable result. The optional portion can then be given execution time as the other demands on the system permit. The ability of the system to ensure that a computation produces a result by its

deadline is thus increased, since production of the answer is less sensitive to variation in execution time, but at the cost of increasing the variation in the quality of the answer. While this is an excellent programming model for many applications, but not all computations can be described in monotonic form.

The MARS system approaches the problem from a different direction, supporting process based programming and run-time models with synchronous communication, but limited to communication at the beginning and end of each process [67]. MARS places the responsibility for decomposing the computation into a set of communicating processes on the developer. The programming model imposes other restrictions as well, including that every process be periodic, and that information exchange between processes is limited to messages. MARS does not provide support for the use of memory segments or other resources to be shared among two or more processes. While these restrictions simplify the implementation of a predictable and fault tolerant system [15], it obviously limits the software architectures developer can use to implement applications.

The MARUTI system uses object oriented programming and run-time models with threads of control moving through many different objects as required by the computation [63]. The system assumes that the WCET and resource use for each object are known, and it builds explicit execution plans, called calendars, for each object specifying when the portion of each thread running through it will execute [45]. The run-time model thus implies that a computation is decomposed into the segments of its execution through each object. The need to consider the interactions, through shared resources, among the many possible threads running through an object can create non-trivial synchronization problems which complicated the construction of

an execution plan for the object. Further, since the deadlines apply to the computations, which are represented by threads, the system must be able to ensure that the individual execution plan for each object helps ensure that the thread will finish executing by the computation's deadline. Explicit compiler support is used to extract the schedulable segments from the object oriented source code [31].

The programming and run-time models discussed in this dissertation, and exemplified by the Spring system, takes an approach having aspects similar to both MARS and MARUTI, but with important differences from both. The programming model discussed here is similar to MARS, but supports significantly more flexible software architectures since it permits computations to be described by groups of processes which interact through shared memory segments and other resources, can exchange synchronous messages at any time during their execution, and can be periodic or aperiodic. The System Description Language and Spring-C programming language implement the programming model and are described in Chapter 4.

The run-time model described by this dissertation is similar to that of MARUTI in assuming that computations are represented by a set of segments which have known WCET and resource use, and for which the system builds explicit execution plans. The method for decomposing a computation into such segments described in this dissertation is similar to that used by MARUTI in some ways. Chapters 5, 6, and 7, describe the methods for deriving the WCET and worst case resource use of a set of processes implementing a computation, and constructing a representation for the computation's behavior as a set of tasks. This method uses each process's suspension points to define the task boundaries. The MARUTI system uses the points where execution threads cross object boundaries to defined the segments of a computation. However, if the scheduler considers each object as a separate resource, as

the MARUTI scheduler essentially does, then those points where the thread crosses object boundaries *are* suspension points. However, MARUTI only considers the behavioral segments defined by the object boundaries, whereas the method described in this dissertation considers *all* points at which execution may be suspended.

2.3 Behavioral Prediction

Predictions about program behavior are basic to nearly all approaches to the design and construction of real-time systems. These predictions can be loosely divided into two classes which we call *deterministic* and *stochastic.* Deterministic predictions are those which are *always* correct. This class obviously includes WCET predictions consisting of a single number, but could also refer to predictions which give the WCET as functions of key input parameters, or as a function of the number of items in a set to be processed. The important point for system design is that deterministic predictions are *never* violated, subject to the assumptions under which they were made remaining valid. Stochastic predictions, in contrast, describe the expected behavior. This can be expressed as a single average case execution time, as a distribution of times, or even in combination with a worst case time.

It is also important to remember that resource use, where resources are protected by critical sections in the source code, is a vital part of the behavioral predictions. How the description of resource use is used to produce a prediction depends on the system's run-time model and its approach to scheduling. Under the rate monotonic and other process based approaches, contention over resource use is accounted for by calculating worst case blocking times. Under Spring, the portion of the process's execution that uses the resource is represented as a separate task, and conflict among

tasks using the same resource in exclusive mode is avoided when the schedule is constructed.

Systems designed using the deterministic view are simpler, in one sense, because exception processing does not have as prominent a role in the system design. Exceptions still occur when the predicted behaviors are violated, but in a deterministic system this is an error condition. As such, it can be addressed simply. A system designed from the stochastic viewpoint expects the behavioral predictions to be violated, and handles at least some of these violations as a normal part of the system's duties. As a result, a system using the stochastic viewpoint will generally require, or at least desire, far more elaborate methods for specifying both the exceptional conditions, and the computations that must be executed when the exceptions occur.

Spring is currently implemented as a hard real-time system and requires deterministic predictions about each task's WCET, resource use, and precedence relations. Our discussion of the related work in behavior prediction thus concentrates on methods of producing deterministic predictions. At the most general level, any method for calculating WCET is engaged in a problem of simple addition. The difficulty comes in deciding how to control the summation in a way that increases the ease of the calculation, its accuracy, its generality, or its ability to be extended. An estimate of the WCET is *valid* when it is greater than or equal to the true WCET. A WCET estimate that is less than the true WCET is *invalid* since it will lead to an exception when the execution path requiring the true WCET is taken. The goal of behavior prediction is to produce estimates of the WCET that approach as closely as possible to the true value while remaining valid.

At a fundamental level, all prediction methods depend on a model of the target processor, since this determines how long specific processor instructions will take.

An ideal model would account not only for the basic instruction times, but also the effects of pipelining in the processor, non-uniform access times in the memory hierarchy, the presence of instruction or data caches, and run-time management methods among others. This is one reason we believe that an integrated approach to real-time system design is required. In the context of the Spring system, decisions ranging from limitations on programming language through program translation, run-time process management, all the way to specific properties of the target hardware have been carefully coordinated to ensure that a reasonably simple method of predicting program behavior will be valid.

Amerasinghe developed a tool which takes the assembly language emitted for a program as its input, and analyzes it with respect to a model of the target processor [1]. However, this tool has several important limitations. It does not take the effects of pipelining and instruction caching into account, and since it works with the assembler output of the compiler, no transformations of the program are possible in response to the results of the timing analysis. The tool first derives the original block and looping structure of the program, then interactively asks the user to specify bounds. Since user interaction is not always convenient, a timing analysis language (TAL) was developed which made it easier to specify simple properties of the program, and made it possible to calculate execution times for more complex structures [13].

While more convenient than the interactive tool, the developer is still responsible for specifying many properties of the compiler's translation of the source code. This obviously creates the possibility for inconsistency between the actual properties of the assembly code produced by the compiler, and the temporal description of it produced by the developer. Some effort was made to reduce this source of error by modifying the tools to take C programs annotated with TAL statements as input, feeding them to

the compiler and the timing tool in parallel [51]. This automated some aspects of the existing tools, but did nothing to ensure the consistency of the assembler translation and its temporal description, other than placing the two descriptions in the same source file.

Harmon recently produced a similar tool performing what he called *micro-analysis* [29]. The tool worked directly with the executable code, first disassembling it, and then deducing the looping structure. The innovative aspect of the work is that the analysis breaks each machine instruction into several component steps, and considers the effect of each on the final WCET estimate. Harmon claims more accurate WCET estimates than other methods as a result, and his approach clearly comes closer to considering the effects of pipelines. However, this tool suffers the same limitation on the ability to transform the program in response to the results of the temporal analysis as Amerasinghe's tool does.

Shaw has concentrated his predictive efforts at the other end of the translation process, the source code. His first approach to the problem was from a verification and logic perspective [75]. Later efforts moved away from the logical assertions and attempts at deriving temporal properties as an extension of proof techniques, and settled on the use of source level *timing schema* for the basic elements in a restricted subset of C [65]. Each timing schema describes the execution time for a C language statement or construct. They did not construct schemata for every aspect of the full C language, so their method is limited to considering programs written in a subset of the full language. This approach avoids many of the problems with that using the compiler's assembly code output, but suffers other limitations. One problem is that since the schema are for source level constructs, the method has difficulty taking into account any optimizations performed by the compiler which cross schema

boundaries. To the extent the source level schema fail to predict the assembler code actually produced by the compiler, they cannot take the properties of pipelines and caches in the target hardware into account effectively. The subset of C considered is also fairly restricted, and would thus be less useful for program implementation.

Shaw also points out a number of interesting issues when he considered the design of a real-time system in general, beyond the basic timing schema approach [12]. His experience points out the nondeterministic nature of many current hardware designs, and the subtle difficulty of determining some elements of execution time. He attributed his problems to an underestimation of interrupt latency, cycle loss due to a lack of synchronization between the CPU and memory refresh clocks, low level PROM activity, and nondeterministic features of the hardware. It is instructive that he includes the instruction loop buffer, an instruction cache, in the list of nondeterministic features, since it is nondeterministic only from the perspective of his schema based approach, not from a perspective that considers instruction sequences and controls context switches, as the Spring system does. However, perhaps the most important observation for the research discussed in this dissertation is that construction of real-time systems with predictable performance requires attention to the design of all levels of the system, and that current hardware designs have emphasized average case performance rather than worst case. This supports our view that a highly integrated design is required.

An earlier effort discussed in Section 2.1 was made by Kligerman and Stoyenko in designing Real-Time Euclid, and the methods for analyzing the schedulability of programs written in it [40, 89]. Their work assumes the process as the run-time representation, and considers the blocking time arising from several sources including: IPC, preemption, and resource contention. The language also assumes priority

based scheduling. They also used a limited language, using only counted loops and precluding recursion and dynamic memory allocation.

Leinbaugh also did some early work on predictable real-time systems [44]. He made a number of interesting assumptions, and introduced several good ideas. One point that is particularly relevant to the research discussed in this dissertation is the use of segmented computations, though he never specifies how the segments are identified. A calculation is specified as a group of these segments, with possible internal parallelism using fork-join semantics. However, his view of the tasks and segments seems to be physical since he precludes looping back to a previous task. This is interesting in contrast to our representation of an execution episode of a process as a task, as described in Chapter 7. His treatment of critical sections is interesting, particularly in the representation which permits identifying impossible interactions, termed *incompatible blockages.* Transformations on this representation implement processing which iteratively lowers the worst case blocking time estimate. The tactic is not to try to find all possible blockages, but to begin by assuming all blockages can happen, and then identifying some of those that are provably impossible. However, the work does not address how any of the execution times are derived, nor how a system could reliably produce the behavior he discusses.

Puschner and Koza take an approach to determining execution times in the context of the MARS system [68], that is much closer to that described in this dissertation than any discussed so far. They bound the execution of loops in two ways: either by iteration count, or by cumulative execution time. They also introduce the idea of scope and markers into their analysis, which enables them to produce tighter bounds than they would otherwise. The idea is to use semantic information to produce constraints that the timing analysis can use. Their example is a marker which places

an upper limit on the total number of times the expensive branch of a conditional can be taken out of the total number of iterations of the loop that contains it. Clearly this is a good idea, where applicable. They also introduce the idea of describing a set of loops where the total iterations of all loops within the set can be bounded, but less helpful constraints are available for the loops individually. This work is interesting, and has features not currently included in the research described in this dissertation, particularly the use of markers and aggregate bounds on sets of loops.

They take a two phase approach, the first phase combines information about program structure and timing, and the second analyzes this representation to determine the worst case execution time. This permits them to consider optimizations performed by the compiler and hardware features, but does not permit them to perform transformations on the code during the same phase as their temporal analysis. They also assume a synchronous model for computations as sets of periodic processes that accept messages when they begin execution, and produce messages as they complete. This closely resembles some data flow approaches [43], and places the responsibility for decomposing the implementation of a computation into a set of communicating processes with the properties required on the programmer. Since all process interactions take place through messages, resource contention and analysis of resource use is irrelevant.

It is worth mentioning again the portion of the Maruti system associated with execution time prediction. The partial evaluation [62] and use of code motion during compilation [31] both assume one or more of the methods of determining WCET for specific code segments already discussed [1, 65]. However, the use of partial evaluation produces a specialized version of a procedure whose WCET will often be lower than that of the more general version. This gives the compiler an important way to affect

the WCET of the code emitted for a computation. Similarly, code motion provides the compiler with a second way to affect the WCET of the emitted code. The work discussed in this dissertation does not currently consider either partial evaluation, or code motion, although either or both could be incorporated in the work described here. Partial evaluation, code motion, and other compiler optimization techniques are not considered because they do not substantially change the fundamental problem of producing a valid worst case behavioral prediction. When partial evaluation is used to create specialized versions of a routine, worst case behavioral predictions must still be made for each version. Similarly, while code motion and other compiler optimization techniques may lower the WCET and make the prediction problem either simpler or harder, *predictions must still be made.* We decided that it would be more prudent to establish the validity of our technique first, and then extend it to consider code transformed by optimization or specialization.

2.4 Operating Systems

The previous sections discussed how adopting predictability as a primary system design criterion can affect the design of a real-time system's programming language, how it can affect the choice of programming and run-time computation models used, and discussed methods used by various researchers to predict worst case behavior of a computation. These discussions showed that assumptions and design decisions made at every level of the system often interact with one another, and can have a significant affect on the predictability of the system as a whole. This section discusses how, or if, a number of systems developed specifically for real-time address these issues.

The most obvious approach is to apply a conventional system to real-time problems and depend on processing power to overwhelm all difficulties. This approach clearly

ignores all language issues, adopts conventional programming and run-time models, and makes no effort to predict or consider worst case computation behavior. While it is taken by some, it is not viable for more demanding applications [84].

A slightly more sophisticated approach applying UNIX to real-time systems has substantially rewritten the operating system to make processes executing system calls preemptable [17]. Other modifications supported preallocation of disk space, fast initial responses to interrupts, and lighter weight processes. This version of UNIX also modified the scheduling algorithm to use fixed priorities, discarding the aging of process priorities performed by the conventional scheduler. These modifications were nontrivial, but did not substantially address the problem of making the system predictable. The programming languages were not modified, giving programmers no way to specify temporal constraints on program execution. The programming and run-time models were unchanged, and no provision was made for predicting program behavior.

Operating systems for real-time embedded systems are often far simpler, and faster, than a conventional system modified for real-time use. A representative example is the VRTX operating system, which is carefully written in assembly language to make it as fast as possible [72]. VRTX is essentially a conventional operating system using a fixed priority scheduling policy. The system uses conventional programming languages which do not support the specification of temporal constraints on program execution. The system uses the process model at both programming and run-time. Significant effort was made to bound the execution time of specific system services, but the system documentation does not explicitly discuss any behavior prediction method. The system uses a single physical address space for the system and all application code, and is limited to a single statically loaded executable image for each

processor containing all the system and application code. This clearly raises pragmatic development issues with respect to protection and debugging. The use of the fixed priority scheduler, the bounded execution time of the system services, and the ability of the application code to change process priority does make it possible to implement other scheduling policies at the application level. A system call supporting priority manipulation was used by the first implementation of the Spring scheduling algorithm to choose the next task for execution [52], and would also support a similar implementation of the rate monotonic approach to real-time scheduling.

The Hawk operating system is slightly more interesting, being designed for a multiprocessor target node, but is of essentially similar design [30]. Hawk is limited to a single statically loaded task set, but this is consistent with its application domain. The programming language is not modified to include any real-time features, and the system uses the process model at both programming and run-time. The system employs priority scheduling, and process priorities can be changed under application control. However, within an essentially conventional approach, some care has been taken to consider predictability. Critical sections are carefully managed by imposing a strict ordering of request and release on the resources, thus preventing deadlock. The target hardware also has some minor features which help address some of the multiprocessor issues, particularly synchronization.

One point of particular interest is that Hawk uses both local and global memory in its design. The application code running on a processor uses the memory local to it by preference, but can access all memory in the system. In particular, a separate bank of memory is purely global, and used to hold items of interest to all processors. This is an important feature, since local memory accesses do not interact with one another, while global accesses do. Both the system and application software are carefully designed to

minimize interactions, and thus increase predictability. No explicit mention is made, however, of any specific behavioral prediction method. The system has a number of features of practical interest which are oriented toward effective software development and debugging. These include the ability to interact with each of the processor boards individually, and to collect streams of debugging output from each processor while minimizing the effect on system behavior.

The HARTS project features sophisticated hardware, but an essentially conventional approach to the operating system [34, 78]. This project primarily focuses on the communication aspects of distributed real-time systems, and by extension on fault tolerance. The hardware design uses a hexagonal mesh layout for the processor interconnections, and a special network processor takes care of much of the networking overhead. This scheme provides good redundancy, and some opportunity for load balancing. However, the operating system is currently based strongly on a conventional commercial product, pSOS, which is similar in its features and limitations to VRTX. The system uses a single physical memory space shared by all the application processes, and conventional fixed priority based process scheduling. The programming and run-time models are process based, and few explicit measures have been taken to ensure predictable execution.

The real-time extension of the Mach operating system is part of the ART (Advanced Real-Time) project [92]. It uses the rate monotonic approach to scheduling, and thus assumes that the worst case execution time for each task is known. Mach's conventional priority driven scheduler is replaced with the Integrated Time-Driven Scheduler (ITDS), which attempts to handle both hard and soft real-time tasks. The programming and run-time models are unchanged, but the system attempts to avoid

unpredictable execution behavior arising from virtual memory management by providing the ability to lock pages in physical memory. The programming language is based on C++, with added constructs for specifying a wide range of thread sets and relations, as well as timing constraints including start time, deadline, exception handling, and periods [33]. However, little discussion is devoted to *how* the WCET of each task can be predicted, as required by the rate monotonic analysis. This is particularly significant since the system supports active objects and assumes a shared memory architecture [50]. This raises issues for accurate behavior prediction since the influence of the active objects and contention of the processor for access to the shared memory must be taken into account when worst case behavioral predictions are made.

CHAOS is another object oriented system that "offers kernel-level primitives that allow high-performance, large-scale, real-time software to be programmed and represented as a system of interacting objects" [23, 73]. The discussion is at a very high level, and is not sufficiently specific for a reader to tell whether or not the system can achieve any degree of predictability. There is no focussed discussion of how to determine the execution time of a computation, as opposed to the time required for a thread of execution to pass through an object. The general approach is clearly derived from Mach, and has much in common with Mach's threads interface. A degree of control is given to the programmer by the ability to place code in local or global memory having different access times. The use of active objects where each object can have an individual scheduling policy raises questions about execution time calculation and rate monotonic analysis that may be solvable, but are not discussed in any detail. On balance, it is difficult to tell how suitable the system is for real-time, but that in itself is probably a sign of insufficient predictability.

MARS is a system which *does* address predictability, but restricts itself to using a simple cyclic scheduler for a set of tasks with no resource conflicts [15]. Tasks are strictly periodic, and communicate only at the beginning and end of each process [67]. The system is thus *time triggered* since all events in the system, process execution and message transmission, are explicitly triggered by the system clock. MARS makes the developer responsible for decomposing the computation into a set of communicating tasks. The programming model imposes other restrictions as well, including that every task be periodic, and that information can only be exchanged between tasks using messages. MARS does not provide support for the use of memory segments or other resources shared among two or more tasks. The programming language is significantly modified to permit the specification of temporal constraints, as well as other information required to support behavioral prediction [68].

The Maruti system uses an object oriented model at programming and run-time, and has added statements to the programming language for specifying temporal constraints and other properties that are important when making behavioral predictions [63]. Partial evaluation is used to create specialized versions of specific routines [62], which is supposed to simplify the behavioral prediction problem. The system also employs code motion and other compiler optimization techniques to improve the schedulability of the emitted code [31]. They do not, however, address the problem of producing behavioral predictions, instead citing the methods of Park and Shaw [65]. The system scheduler builds explicit execution plans for the set of threads running through each object [45]. The system is, however, being implemented on top of the Mach platform and its predictability will thus be limited to that provided by the underlying operating system support [45]. Since Mach was originally developed

to support conventional applications whose average case performance was the relevant system evaluation criterion, it is reasonable to question how well it will support *predictable* execution of real-time computations.

The Spring system is designed with predictability as the primary design criterion [87]. The system has made significant changes to its programming language to permit the specification of temporal constraints and other information required to produce behavioral predictions, as discussed in Chapter 4 of this dissertation. The system supports the process model for programming, but uses a task based model at run-time for scheduling, as required by the system scheduler [70]. The scheduler builds an explicit execution plan using the task based representation of computations, which is constructed by the methods described in Chapters 5, 6, 7.

2.5 Hardware

The discussion in previous sections considered how the current state of the art addresses the needs of real-time systems at the language level, presented several methods of producing the behavioral predictions required by many real-time systems, and considered how the design of the operating system affected the ability of a real-time system to ensure that computations are executed according to their temporal constraints. Virtually all of the systems and approaches considered, however, used conventional hardware. The economic motivation for this is clear, but the average case orientation of most hardware designs imposes limitations on the degree and type of predictability the design and implementation methods described can achieve.

Optimization of the average case pervades most hardware designs. For example, the design of a processor's pipeline is generally optimized for the most common case where instruction and data cache hits occur, and often is *less* efficient for the cases

where they do not than other less optimized designs. This is the correct approach when designing a system for which the average case performance is the primary design criterion. It is not necessarily the best approach when predictability of a computation's worst case behavior is adopted as a primary design criterion.

Comparatively little work has been done on the design and implementation of hardware specifically targeting real-time systems for several reasons. First, the constraints on predictability arising from the hardware's properties are often more subtle than those arising from other levels of the system, and can thus remain masked until the methods and designs addressing predictability at those other levels become accurate enough to reveal the limitations of the hardware. Other reasons arise from the significantly greater effort and expense of designing and implementing hardware compared to that required for software. The design of other levels of the system clearly influence the set of properties the hardware must exhibit to provide the best support. The design of the other system levels must thus be developed fully enough to stabilize their requirements before designing specialized hardware can be considered. A closely related issue is that developing specialized hardware must, clearly, be justified by a substantial performance advantage. Predicting such an advantage with any assurance also clearly requires stable designs at the system levels which interact with the hardware.

One of the more interesting real-time hardware design efforts was called SMART, Strategic Memory Allocation for Real-Time, caching and addressed the question of how to use caches to decrease the WCET of tasks running on a system using rate monotonic scheduling [37, 39, 38]. The approach segmented the cache, dedicating a portion of it to each active task, and thus isolated the segment of the cache devoted to each task from the effects of context switching on a single cache. The contents of

each cache segment were thus the same as they would be for a conventional cache of the same size when a task ran without being preempted. This research showed that it was at least possible to consider using caches in real-time systems, but it did have a significant limitation. The problem is that while segmenting the cache certainly isolates each segment from context switching effects, determining the WCET of the task requires the behavioral prediction method to identify the execution path through the code with the WCET, which we can call the worst case path. The worst case path, however, *depends on the cache partition size.*

The SMART approach assigns cache segment sizes based on an assessment of the average hit rate exhibited by the worst case path through the code as determined by a predictive method which does not take caches into account [1]. The segment size, the average hit rate, and the WCET derived from this information are thus derived using an execution path, *which is not necessarily the worst case path for the cache segment size used.* This is easy to see for a conditional statement within a loop. When no cache is present the worst case path takes the branch of the conditional with the greatest uncached execution time on every iteration. When a cache is present, however, the worst case path will include the second branch of the conditional if its uncached time is greater than the cached time of the first. It is also possible, if the cache footprints of the two branches of the conditional conflict, that the worst case path through the loop will alternate between the two branches on each iteration of the loop. In either case, the worst case path assumed by the SMART approach is *not* the correct one. This clearly limits its ability to accurately predict the WCET of tasks.

The MACS architecture is a processor design which interleaves the execution of several threads of execution through a shared pipeline, which eliminates the problem

of memory access latency without having to use caches [14]. This approach improves the predictability of the WCET for a task, but is subject to some limitations. First, it works best when enough tasks are ready to execute simultaneously for it to remain busy by interleaving their execution. If this is *not* true, then the throughput of MACS is reduced to that of an uncached processor. Further, while the *throughput* of MACS with enough runnable threads available is good, the WCET of each individual thread is greater than it would be for a processor with a cache. This is not necessarily bad, but is clearly a significant factor in determining the suitability of MACS for particular applications.

The design of the Spring system makes certain assumptions, and ensures some properties that are relevant to the design of specialized supporting hardware. One of the assumptions made in the design and implementation of the operating system is that bounded synchronization primitives can be implemented which are used to help coordinate the actions of the system code on each of the processors in a multiprocessor node. However, when the implementation of these primitives was addressed, several problems with the existing hardware were discovered. Creating a solution lead to the formulation of more general results addressing synchronization in multiprocessor real-time systems [53]. The design of the Spring system makes it possible to consider predicting the effects of caching on the WCET of a task [59]. The viability of the approach depends on the fact that under Spring application tasks are shielded from external interrupts, and are not subject to preemption at arbitrary times [87]. The explicit execution plan construction ensures that a task will complete without interruption, and analysis of the cache's effect on its worst case behavior can proceed. We have also considered how a set of predictable Spring nodes can be connected to form a network capable of supporting predictable computation execution [86].

Finally, since the performance of the Spring scheduling method has such a significant effect on the performance of the system as a whole, the Spring Scheduling Co-Processor (SSCoP) was designed, implemented, and is being tested [9, 60]. This is an application specific integrated circuit which implements a substantial portion of the Spring scheduling method in parallel hardware. It thus has the potential to substantially accelerate the scheduling operation. Its performance is, however, currently limited by the time required to preprocess the information describing the set of tasks for which an execution plan must be constructed, and the time to post process the output of the chip into a usable schedule. Work is proceeding on the SSCoP, and experience with its use may indicate that additions to the current are required.

2.6 Summary

This chapter has discussed many related research efforts in several areas of real-time system design including programming languages, behavioral prediction, operating system design and implementation, and hardware specifically designed for real-time systems. Some of our concerns are shared by other researchers, giving rise to similarities, for example, between our approach to programming language design and other existing real-time languages, or between our approach to operating system design and the designs of other real-time systems. None of the existing efforts have, however, addressed *all* of the interrelated issues in the integrated way described in this dissertation. This is extremely important since, as our discussion shows, overall system predictability is affected by the predictability at all levels of the system: hardware, operating system, program translation, analysis, and programming language support.

An integrated effort is essential to ensure the only kind of predictability that really matters, that of the real-time computations executing in the context of the system *as a whole.*

CHAPTER 3

SPRING SYSTEM OVERVIEW

This chapter presents an overview of the Spring system, establishes a context for the research presented in this dissertation within the system as a whole, and provides the background required to understand the issues discussed in subsequent chapters. Figure 3.1 illustrates the view of real-time system design that has guided our work. The three largest rectangles distinguish between the application, the virtual real-time machine, and the physical machine from which the system must elicit predictable real-time behavior. The arrows pointing down stress how assertions at higher levels create requirements for the behavior of the lower levels. Examples of this are behavioral assertions, including time constraints, made in the source language which are then used by the system in the course of supporting the specified behavior. A similar relationship also exists between the virtual real-time machine and the physical hardware, since the virtual machine issues hardware commands and depends on predictable responses.

The arrows pointing up stress that lower system levels often have properties which constrain the requirements that can be satisfied. For example, imagine that the target hardware design limits the predictability of the execution time for a given instruction. Since the instruction's execution time is inherently variable, the virtual machine operations using that instruction are less predictable than they might have otherwise have been. With this in mind, it should be apparent that the designs

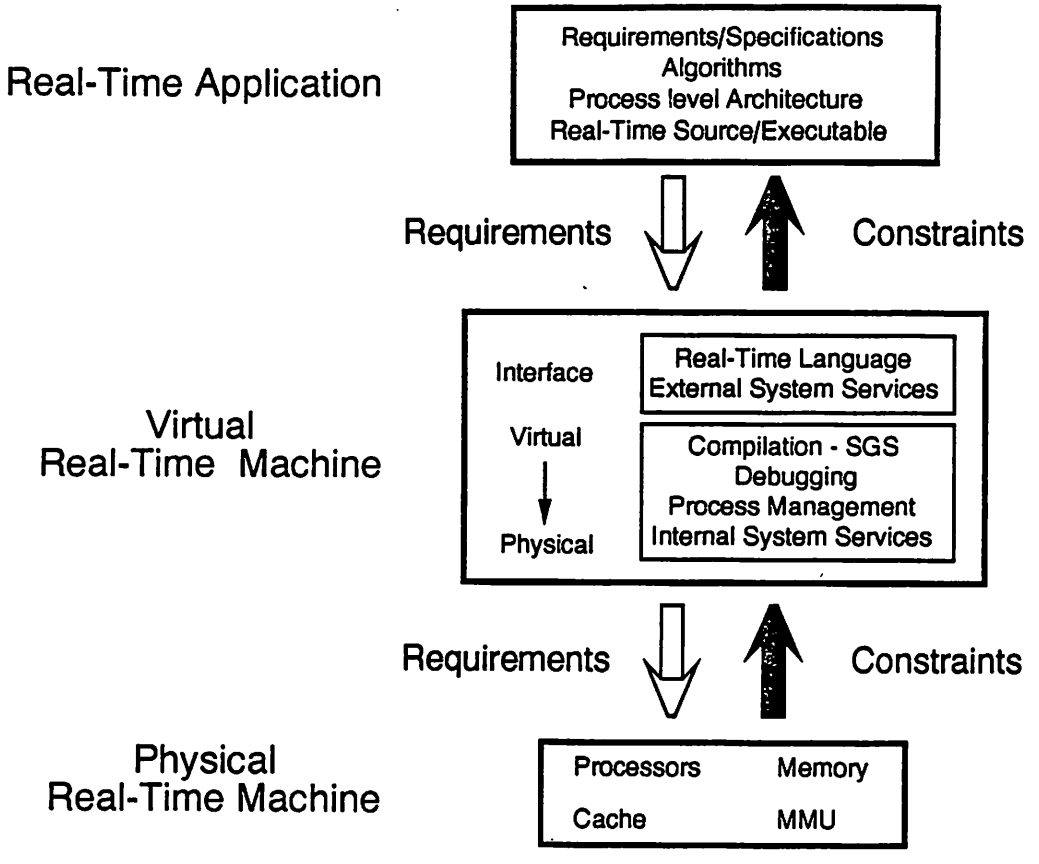


Figure 3.1. System Layer Relationships

of different layers of a real-time system must be carefully coordinated to create a system with truly predictable behavior. We use the term *vertical slice* to describe this approach to system design, since it slices across the boundaries between system layers whose design and implementation are often addressed almost independently in conventional systems [27].

Figure 3.2 illustrates the environment for real-time program development and execution provided by the Spring system. The portion below the heavy black line shows the components of the current system, while the portion above it describes the current system's potential to act as a target for higher level real-time languages. There is a significant body of research in languages aimed at describing real-time

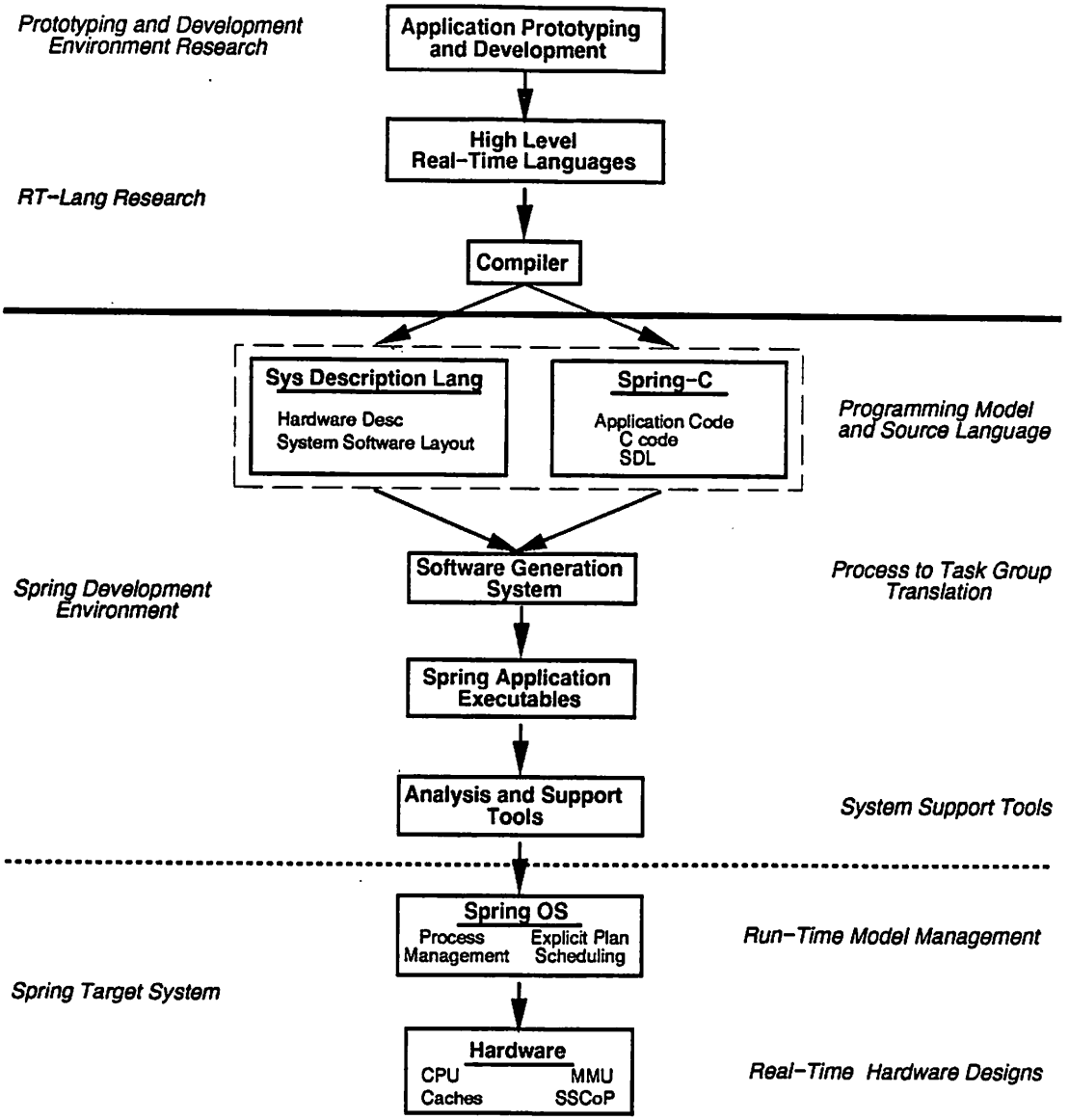


Figure 3.2. Spring System Overview

computations, which does not always consider in detail whether the platform upon which those computations execute can reliably produce the behavior specified by the language. Spring-C has the potential to act as a bridge between such higher level languages and the predictable execution platform provided by the Spring operating system, by providing a target language into which some of these other real-time languages might be translated. This idea is consistent with several existing research

efforts [63, 36, 49, 8, 88, 25], but for the moment all programs executing under Spring are written directly in Spring-C.

The dotted line distinguishes between the host and target portions of the system. The programming model, source language, and software generation system (SGS) are the most prominent sections of the host's development environment with respect to the research presented here. However, the analysis and support tools, which include the system debugger, are of significant practical importance. Spring's development environment is currently implemented on UNIX workstations, and the SGS provides support for program compilation, linking, analysis, and debugging. The executable files prepared by the SGS are downloaded onto the Spring target node using the debugger, and then executed under the supervision of the operating system.

The SGS's support for cross compilation decouples the selection of host and target architectures. The SGS currently supports two host architectures, DEC Vaxstations and DECstations, and one target architecture, the Motorola 68020. However, many parts of the SGS are extensions of the Free Software Foundation's tools[81], which support a wide range of hosts, and share in their portability. The other parts of the SGS have been written with portability in mind, making support for other host architectures easier.

The target system implements the operating system half of the virtual real-time machine which manages computations' execution according to their requirements and constraints. This obviously includes Spring's scheduling algorithm extended for computations represented by precedence related groups of tasks, but also required careful design and implementation of all other parts of the system to support predictable execution. The design and implementation of the operating system also required us to consider how well the target hardware supports predictable execution, and how

designs focussed on predictable worst case performance could improve on current designs.

The rest of this chapter discusses three basic aspects of the Spring system. Section 3.1 discusses how information flows through various elements of the programming environment as application programs are compiled and downloaded. Section 3.2 considers the Spring node architecture, whose properties obviously exerted a significant influence on system design decisions and performance. Section 3.3 finishes the chapter by discussing support for communication at the system level.

3.1 Software Level

This section discusses how information about application programs flows among the various elements of the programming environment in the course of creating and downloading the application's executable images, as illustrated in Figure 3.3. A novel feature, and important contribution, of Spring's programming environment is the richness and extensive coordinating role played by the system description language (SDL). The figure is divided into three horizontal sections, illustrating the processing of Spring-C source files, SDL source files, and how the information specified or derived from the SDL and Spring-C files is used to control scheduling and Spring scheduling co-processor (SSCoP) simulations as well as to control downloading of the target system for execution.

The middle section illustrates the processing of the source files which contain only SDL statements. This is emphasized by showing that the compiler used is **sdl_cc**, although the Spring-C compiler **spr_cc** can also be used. The SDL source is compiled, producing machine readable files containing descriptive information. The **sdl_merge** tool is then used to accumulate the descriptive information from these and other

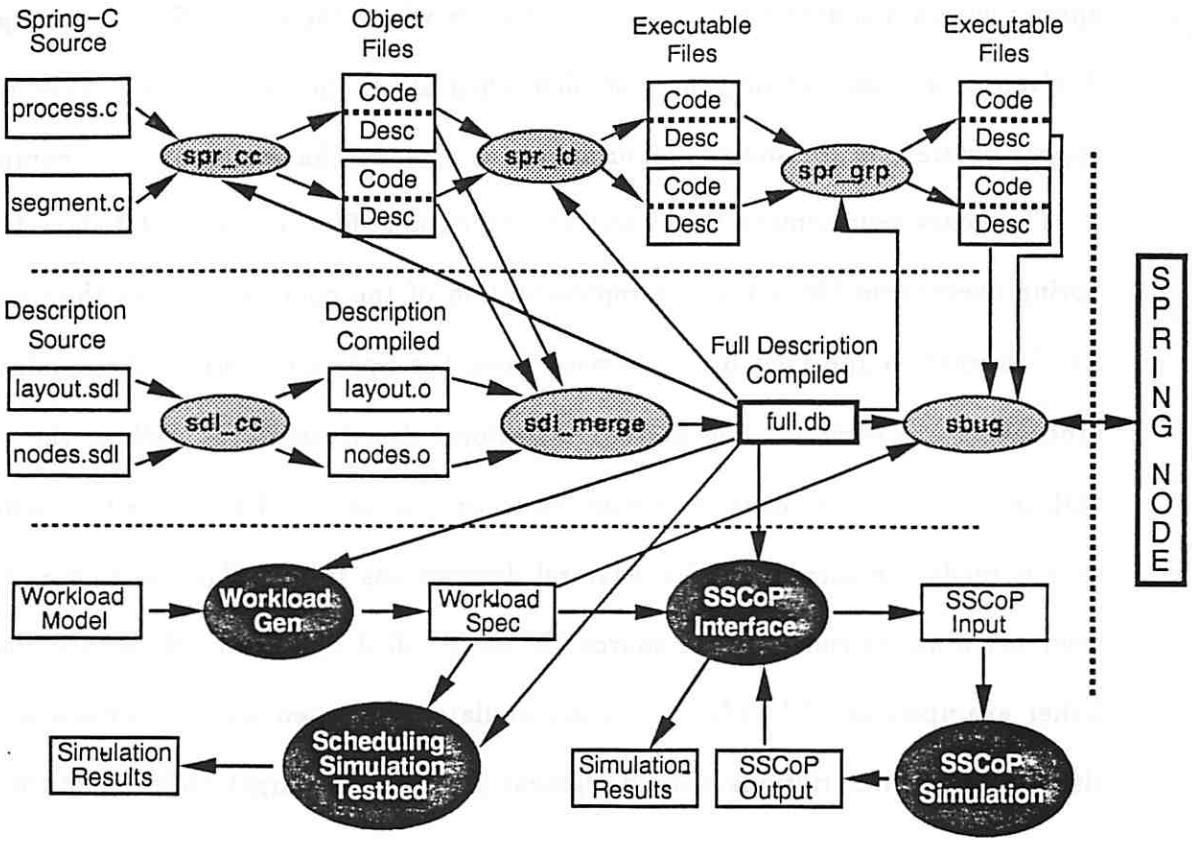


Figure 3.3. Programming Environment Information Flow

sources as it becomes available, producing a cumulative description of the application in the file *full.db*. This information is then available for use by all elements of the programming environment.

Some of the information in these descriptive files is required to compile and link Spring-C programs, as illustrated by the arrows from *full.db* to **spr_cc** and **spr_ld** steps in the top section of Figure 3.3. For example, a procedure in one file often calls procedures described in other source files. In conventional systems each source file can be compiled independently, because all unresolved references are handled at link time. However, when behavioral predictions are derived during compilation, the compiler must have access to a behavioral description for all routines called by the procedure being compiled. This implies constraints on the order in which procedures

appear within a source file, and on the order in which the source files are compiled. A given procedure can only be compiled when all of the procedures it calls either appear before it in the source file, or appear in files that have already been compiled.

The boxes representing the object and executable files in Figure 3.3 show that a Spring executable file contains a representation of the code, as well as the descriptive information specified by, *or derived from*, the Spring-C source. As application procedures are compiled the derived behavioral descriptions, as well as the other SDL information they contain, are added to the *full.db* file. Each time the compiler runs it reads the accumulated behavioral descriptions from *full.db*, as well as those from any libraries called by the source file, as specified by command line arguments. Other examples of SDL information accumulated and used during compilation include: resource descriptions, shared segment descriptions, target node structure, and the assignment of processes and shared segments to specific locations in the target node memory. A resource is defined by SDL statements described in Section 4.2.2, which specify a list of the shared data structures comprising the resource, its name, modes of access, and its support by a shared segment. A shared segment is defined by SDL statements described in Section 4.2.3, which specify the resources it supports, a virtual and/or physical base address, and a name. The SDL statements described in Section 4.2.4 specify the structure of the target node, including: a list of each processor and memory board, while the statements described in Section 4.2.6 specify the board assignment of each process and shared segment within a node.

When all the application code is compiled, executable files can be produced for each process. SDL information about shared segments is used by the loader **spr_ld** to resolve a process's references to shared data structures. Executables for independent processes are complete when linked, and contain all of the SDL information required

to describe them to the Spring scheduler. However, for processes that are part of a group engaging in synchronous communication, some post-processing is required to complete the SDL description. The **spr_grp** command reads the SDL information contained in the executable of each process in the group, and performs the analysis require to complete the behavioral description in the executable files. This analysis is described in Section 7.2.

The *full.db* and process executable files provide all the information required by the debugger, **sbug**, to download the system and application code onto the Spring node(s), as described by the layout section of the SDL, and begin execution. The vertical dashed line symbolizes the fact that the download operation crosses the boundary separating the development machine from the Spring target node. However, the SDL can also be used to provide input information to simulations associated with the Spring system. This information can describe actual application software, or describe an imaginary workload.

The bottom section of Figure 3.3 illustrates the workload generator, scheduling simulation, and SSCoP simulation testbeds. The workload generator uses the information in *full.db*, as well as a model of the workload, to generate a detailed system workload for specific experiments. If the workload is intended to drive the actual system, then **sbug** takes care of providing the workload to the running system as required. The system description and workload can also be used as input to the scheduling simulator to conduct a scheduling experiment. The scheduling simulation testbed enables researchers to experiment with different scheduling algorithms within a context that accurately reproduces many aspects of the actual Spring system [20].

The workload and the information in *full.db* can also be used as input to the SSCoP simulation. The SSCoP interface simulation embodies algorithms and code

drawn from the Spring system subsections addressing both process activation and the scheduler. It uses the information in *full.db* to construct the run-time data structures used by the scheduler, while the workload specifies the order and timing of the tasks to be scheduled. The SSCoP interface considers the set of computations requiring scheduling, and performs a number of preprocessing steps required to prepare the information in the form required by the SSCoP. This is then passed to the SSCoP simulation, whose output is used to update the system schedule, and also records the results of each part of the simulation.

The two testbeds can be used on the same input information to compare the results produced by a simulation of the scheduling algorithm, and by the SSCoP. This is an important part of the testing process for the coprocessor. Further, results of the simulations will also be compared to the behavior of the system actually using the SSCoP, when a working SSCoP is available.

This completes the discussion of how the SDL's descriptive information is accumulated and then used by various portions of the Spring programming environment. While a bit more complex than those for conventional systems, the compilation ordering constraints are no different in principle from those requiring that all the object files required to build an executable be produced before linking occurs. Further, while they illustrate the central role played by the SDL, the details of exactly what information is used by which tools is less important than the idea that the descriptive information is accumulated and stored in a common form, thus making it available to all present and future tools.

The uses of the SDL information are by no means restricted to those discussed. For example, the download function of **sbug** was mentioned, but the descriptive information is also used during debugging. The SDL was explicitly designed to support

the long term development of the system in several ways. Other tools will be developed over time which use the current descriptive information, and which may derive portions of what is now explicitly specified by the developer. For example, one could imagine a tool which takes process requirements and properties into account, and derives a system layout. Some of the requirements information might be given by or derived from a higher level specification written in a separate requirements language or an extension to the SDL. Some existing approaches address precisely this problem, and would be good candidates for creating such tools[91]. Also related are languages and tools addressing the problem of configuring distributed application software [48, 4].

Other tools might conduct analyses related to specified fault tolerance requirements, and produce an appropriate process group specification. The SDL is thus intended to support the orderly evolution of the system by serving as a target language within which to express the results of higher level languages, analyses, and requirements specifications; and by supporting a common format for the use and exchange of all relevant information among tools in the programming environment. So, for example, it is possible to imagine, as illustrated in Figure 3.2, that a higher level real-time language's compiler might generate Spring-C, using the SDL to describe the properties of the processes it has decided are required to implement the computations described by its input. Finally, the definition and implementation of the SDL makes it easy to add to or modify descriptive information, ensuring the system's flexibility and extensibility.

The SDL thus represents an important contribution for three reasons. First, it considers information of a wider range and at a greater level of detail than most other systems. Second, the generation and use of this information has been fully integrated

into all aspects of the SGS and run-time system. The SDL thus facilitates the vertical slice approach to integrated design illustrated in Figure 3.1 by simplifying extensive information exchange among normally separate system layers. Third, its extensibility greatly simplifies creating and integrating new features and tools into the system, thus making it easier to maintain an integrated design as the system evolves.

3.2 Spring Node Architecture

Spring is a physically distributed network of multiprocessor nodes each running the Spring operating system[87]. Each multiprocessor (see Figure 3.4) contains one (or more) application processors(AP), one (or more) system processors(SP), and an I/O subsystem. Ultimately, SPs could be specifically designed to offer hardware support to system activities such as guaranteeing computations. The SSCoP is our initial effort in this direction[9], and is described in Section 8.3.2. The I/O subsystem is partitioned from the Spring Kernel, handling non-critical I/O, slow I/O devices, and fast sensors.

One of the most important features of a Spring node is its *functional partitioning*, which enhances predictability by *shielding applications running on the APs from external interrupts.* Environmental interrupts directly affect only the SP and I/O processors, indirectly affecting the APs and the applications executing on them by generating work whose execution must be added to the execution plan maintained by the scheduler. Functional partitioning provides other benefits, including the ability to manage different classes of processes separately.

Each Spring node, as illustrated in Figure 3.4, currently contains 5 processors, one SP, three APs, and an I/O board; as well as a "global" memory (GM) board not associated with any processor. Each SP and AP has a Motorola 68020 CPU, 68851

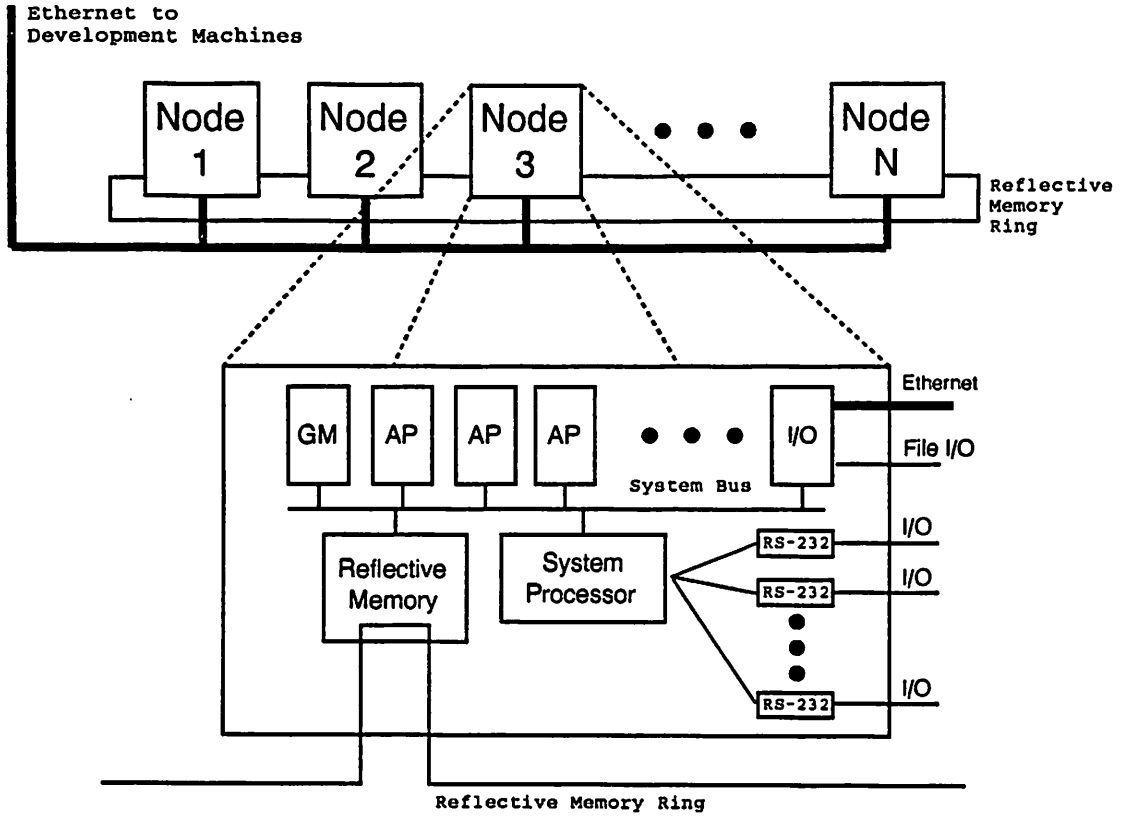


Figure 3.4. Spring Node Architecture

MMU, 68881 FPU, and 4Mb of local memory which is also visible on the system bus. We currently have 3 multiprocessor nodes connected via two networks; an Ethernet to support non real-time traffic and for downloading the system from the development platform, and a fiber optic register insertion ring connecting 2 Mb "reflective" memory boards in each node. The I/O board is currently a stand-alone UNIX system supporting both the Spring file system and the node's Ethernet connection. The SP can interact with other external devices and sensors through its serial (RS-232) ports. Other I/O boards, not running UNIX, could be added to support time critical devices.

The reflective memory provides a shared memory model for its 2 Mb (of physically distributed but logically centralized memory), and is implemented via the off-the-shelf product called Scramnet[90]. The memory is "reflective" in that the reflective memory

boards in each of the Spring nodes always contain the same information, subject to transmission delays on the fiber optic ring connecting them. The reflective memory boards thus effectively represent a single memory board shared among all the nodes on the ring.

Its small scale helps achieve predictability within a single Spring node, while predictability across physically distributed nodes is supported by the replicated memory, real-time virtual circuits implemented using it, and higher level software for end-to-end scheduling. The replicated memory can also be used to support shared memory across node boundaries and for fault tolerance. Important data structures written to the replicated memory are automatically reflected in the other boards on the ring. The duplicate information is useful in recovering from several classes of node failure faults, and it is possible to enhance fault tolerance by adding *parallel* replicated memory rings[86]. That and other fault tolerance issues are, however, not addressed in this dissertation[95].

The protection of application processes from external interrupts, along with the scheduler providing execution guarantees, allows us to construct a more *macroscopic* view of a predictable system. Context switches are reduced, predicting computation behavior is made simpler, which simplifies guaranteeing completion by its deadline. Our strategy also partitions the real-time processes into those that require static resource allocation, those requiring a dynamic scheduling algorithm in the front-end, and those, which typically have higher levels of functionality and greater latency, handled by the dynamic on-line guarantee routine. Those requiring static allocation on dedicated I/O boards are typically fast I/O device drivers and critical processes. Slow I/O devices can be multiplexed through a front-end processor, which might use

a cyclic or rate monotonic scheduler. The APs support the higher level application processes given dynamic on-line guarantees.

There are two levels in the non-uniform memory access (NUMA) hierarchy of the current Spring node architecture. Access by a processor to the memory on its board, a *local* access, is fastest. Access by a processor to the memory of other processors or to the GM board, a *global* access, is substantially slower for obvious reasons. A global access suffers delay contending for the system bus, and must also contend with the local processor if the access is to memory on another processor board. The system ensures a predictable worst case global access time by using the VME backplane in *round robin* mode, which enforces fair contention[54].

The *reflective memory* represents a potential third level in the NUMA hierarchy, especially if its access time included the time for updating all other boards on the ring, but in the current configuration its access time is the same as to any other memory on the system bus. An additional level in the NUMA hierarchy would also be created by instruction and data caches, which the current Spring target architecture does not possess. Calculating at least a portion of an instruction cache's effects on the *worst case* behavior of a procedure is a non-trivial problem, but can be done under some circumstances[59]. This issue is relevant, since the next target architecture for Spring will almost certainly have instruction and data caches, which must be used predictably, if they are to be used at all.

The NUMA hierarchy is of considerable interest during compilation, since the time for a memory access is determined by the location within it of the data structure being referenced relative to that of the process generating the reference. The portion of the SDL described in Section 4.2.4 enables a developer to specify the structure of the node within which the application and system code run, and thus of the NUMA hierarchy.

This information, along with the assignment of processes and shared segments to processors given by the portion of the SDL described in Section 4.2.6, is required to distinguish local and global memory accesses.

3.3 System Level

This section describes support for two types of communication in Spring, both of which are used by *system* software, and have no impact on Spring's support for *application* level IPC. The first set of routines is socket-based, and supports communication between SGS applications running on the host system and processes running on the Spring node's I/O board[41]. The second set uses shared memory and a simple handshaking protocol to control communication between software on different boards in the Spring node at boot-time and during debugging. Run-time communication between Spring processors in the current implementation is largely asynchronous. The minimal synchronization required to manage event and dispatch queues is supported by semaphore control of critical sections implemented using the *test-and-set* instruction [53].

Figure 3.5 illustrates the communication path between applications on the host system and software within the Spring operating system; specifically, that of the Spring system debugger *sbug* and the Spring file system. When it begins execution, *sbug* reads the SDL layout and other information in "full.db", as discussed in Section 3.1, to determine how many APs will be used, what system code to load onto the SP and APs, as well as finding any non-default values for various boot parameters. In the current implementation, a separate *sbug* process runs on the host for each of the SP and AP boards, so the original *sbug* process forks off the number of siblings required.

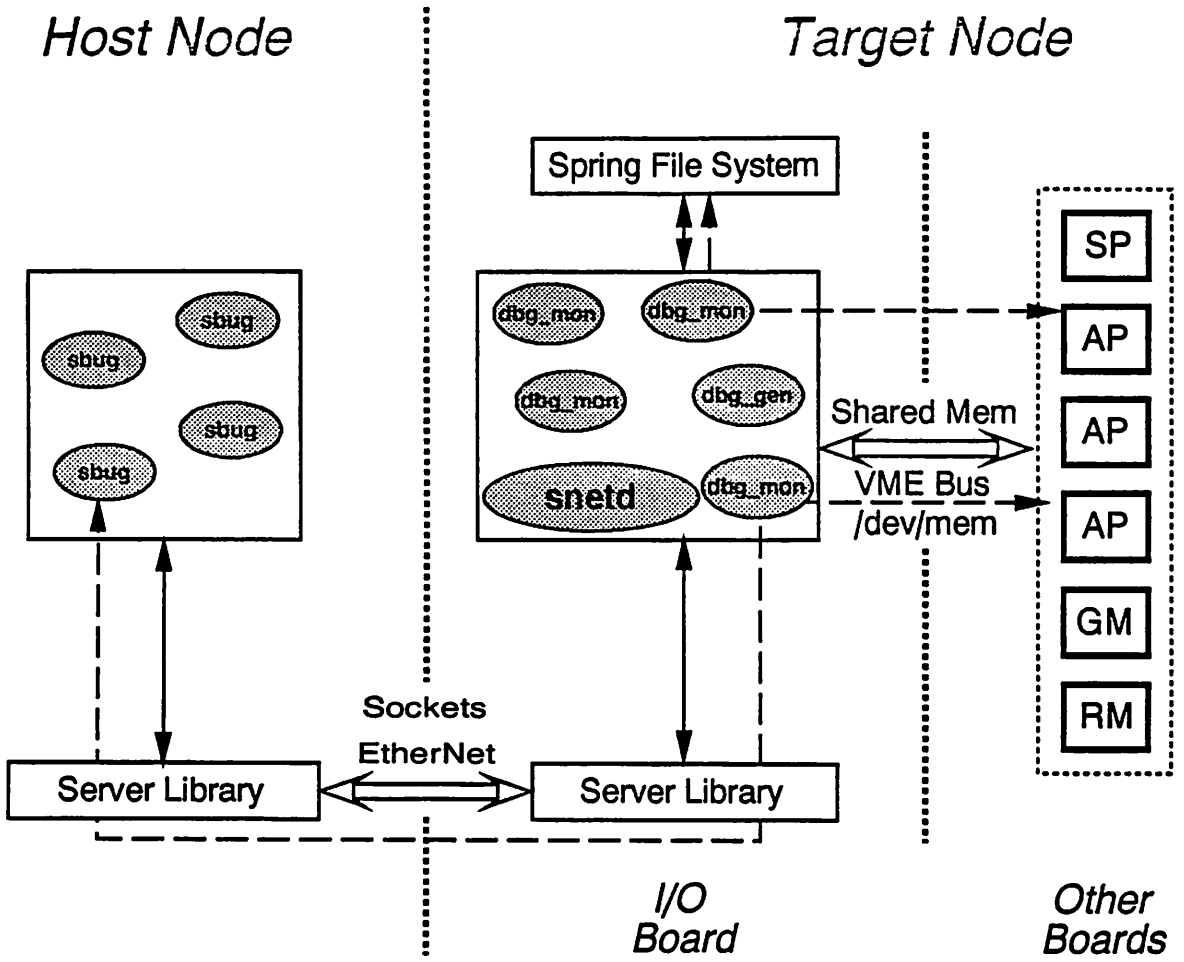


Figure 3.5. Host–Target Communications

Each *sbug* contacts the *snetd* demon running on the Spring I/O board, and establishes connection to the general debug server *dbg_gen*, using the socket based communication routines. Each *sbug* uses the general server to initialize its board, and then requests the creation of a debug monitor *dbg_mon* from *snetd*. As each *sbug* establishes contact with its *dbg_mon*, it begins execution of the system code on its processor board. From then on, each *sbug* process speaks to the debug monitor on its SP or AP through the *dbg_mon* process. This is illustrated by the lower dashed arrow in Figure 3.5 running from the *sbug* process on the host through the *dbg_mon*

process on the I/O board to the AP. The socket based library supports communication between *sbug* and *dbg_mon*, which communicates with the AP's debug monitor using a simple handshaking protocol.

Spring's handshaking protocol supports communication between the SP and AP processors at boot time, as well as between the debug monitor on each SP and AP and the corresponding *dbg_mon* process during debugging. The protocol manages communication between clients and servers, one of which is always in *sending* and one in *receiving* state. The client and server each have a message buffer in their *local* memory dedicated to the exchange.

The debug monitor on the AP is, for example, the client of *dbg_mon* which forwards messages to and from the *sbug* process on the host. When a breakpoint occurs, the debug monitor announces the fact with a message to *dbg_mon* which is forwarded to *sbug*. The debug monitor on the AP waits for a reply, which is either a request for a debugger operation, or a command to continue execution. At each point, one of the two communicating parties is waiting for a message from the other.

The protocol is also used to support the Spring file system. As illustrated by the upper dashed arrow in Figure 3.5, a file request originating on an AP (or SP) is submitted to the *dbg_mon* server, which performs the required file system operation and writes the appropriate reply. The file system is currently only used at boot time to support process activation, and is thus not subject to real-time constraints. Further work is required to design and implement a file system capable of satisfying real-time constraints.

The other application of the handshaking protocol is between the SP and the APs at boot time. Each process is activated in response to a request from *sbug* sent to the boot code on the SP, which initializes a number of run-time data structures and then

forwards the request to the AP in whose local memory the process should reside. The AP then opens the file and creates the executable image by making a series of file system service requests to *dbg_mon*. During the process activation phase of system initialization, the SP is the client, and each AP is a server.

CHAPTER 4

SPRING-C PROGRAMMING LANGUAGE

The Spring-C language must fulfill two significant goals: it must provide developers with a vocabulary for describing a computation which is as rich and familiar as possible, while ensuring that it is possible to produce executable files containing all the information required to load and predictably execute the computation. The Spring scheduler assumes that every computation is represented by a finite group of tasks with known worst case execution time (WCET), resource requirements, and precedence relations constraining execution order [70, 98]. This representation is in stark contrast to the familiar process based programming model within which most programmers are accustomed to describe computations for execution on conventional systems.

Within Spring's programming model, a computation is described as a single process or group of cooperating processes. Each process executes in an independent address space to which memory segments shared with other processes may be attached. The programming language includes statements by which processes: delay their execution for a specified minimum period, specify blocks of code within which they use "resources", and specify points at which they exchange messages synchronously with another process. Resources are names used to represent any set of elements in the system which can be used by two or more processes executing concurrently, and to which at least one process requires exclusive access. Resources are most often used

to represent sets of shared data structures, but can represent other system components, such as memory mapped control registers for external devices, communications channels, or processors.

The delay, resource use, and synchronous communication statements in Spring-C are points within the process code where its execution can or must be suspended. These *suspension points* are of interest when constructing the task group representation through compile-time prediction of each process's worst case behavior, as described in Chapters 5, 6, and 7. However, the analysis involved requires the developer to specify much more detailed information about the requirements, constraints, and properties of each process than conventional source languages generally either require or support.

The Spring-C language ensures that the required task group representation can be derived for every computation by extending the original C language in two significant ways. First, we *avoid* the halting problem by defining several syntactic changes which ensure that every Spring-C process is *finite*. The obvious way to do this is to place explicit limits on the sources of unbounded execution: loop iteration and recursive subroutine call depth. The obvious limitation of this approach is that while enforcing the loop iteration and recursion depth bounds ensures a finite process, for which it is at least not obviously impossible to derive a worst case behavioral description, determining the proper bounds can be difficult. For the purposes of this dissertation, we assume that valid bounds are specified by the developer. Methods for deriving such bounds are a separate problem, to which there are a number of possible approaches [62, 64]. Other syntactic changes include statements supporting all the suspension points, thus making their location and type available during compilation and analysis.

The System Description Language (SDL) is the second significant extension to the original C language. It supports the specification and representation of a wide range of information required to describe, compile, perform behavioral predictions, load, and predictably execute Spring applications. This includes descriptions of processes, process groups, tasks, task groups, resources, shared segments, the target hardware architecture, and the assignment of system and application software to that target architecture. The SDL is fully integrated into the Spring-C language and is supported by the Spring-C compiler, thus permitting descriptive statements to be inserted into application source files wherever appropriate.

The design of Spring-C is an example of how highly integrated the design of a predictable real-time system must be. Each feature of the source language and the programming model it presents have been carefully considered with respect to the properties and duties of specific elements at *every* other level of the system, including: the behavioral predictions and task group construction performed during compilation, the scheduler, the dispatchers on each application processor, and the properties of the target hardware. Changes in any of these areas can have significant implications for the features Spring-C can and should support, just as changes to Spring-C can imply significant changes to the design or implementation of one or all of these areas.

The rest of this chapter discusses the Spring-C language in greater detail. Section 4.1 presents the syntactic differences from basic C, and why all Spring-C programs are finite as a result. The pivotal role of the SDL in making Spring a highly integrated system was introduced in Section 3.1. Section 4.2 presents each part of the SDL, considers in some detail the information it represents, and illustrates the role for that information in the system as a whole. Section 4.3 finishes the chapter by

discussing several ways in which the effectiveness of Spring-C for describing real-time applications has been and is being investigated.

4.1 New Language Features

The changes made to the original C language, creating Spring-C, were motivated by the need to make worst case behavioral predictions during compilation. These predictions, described in Chapters 5, 6, and 7, require information constraining the sources of unbounded execution time in a program, and information about where process execution may suspend. Features of Spring-C syntax ensure that this information is available to the analysis producing the behavioral predictions.

It is important to note that Spring-C is a block structured language, and that the behavioral analysis is block oriented. Spring-C is specifically designed to ensure that the behavioral analysis can be performed for all Spring-C programs, and to simplify the analysis when possible. Unbounded execution arises from two sources: loops and recursion. Syntactic changes to the original language enforce the requirement that the developer provide iteration bounds for every loop, and recursion depth bounds for every call to a recursive procedure.

Loops in a C program can arise in two ways: use of a *goto* statement; and use of a *do*, *while*, or *for* statement. Consider the assumption that the developer supplies iteration bounds for every loop. In the case of the *do*, *while*, and *for* loops, it is easy to add upper and lower iteration bounds to their syntax, as illustrated in Figure 4.1. In contrast, even when a *goto* is used to create a well structured loop, it provides no syntactic opportunity to specify iteration bounds, nor any ready way to identify the boundaries of the loop. Instead, the loop structure must be derived from analysis of the emitted code. Separately specified iteration bounds might then be applied, and

```
while (exec-condition)(min_expr, max_expr) {
        loop-body;
}

for (init; exec-condition; incr)(min_expr, max_expr) {
        loop-body;
}

do {
        loop-body;
} while (exec-condition)(min_expr, max_expr);
```

Figure 4.1. Spring-C Bounded Loop Constructs

the behavioral predictions made. This is the approach taken by one popular WCET prediction method [1, 13]. Further, the *goto* can obviously be used in *unstructured* ways which present far more complex analysis problems.

While the *goto* would add significantly complicate the analysis performed during compilation, it would confer little benefit. Many software developers never use it at all, without distress. The cost/benefit ratio of the *goto* for Spring-C is thus very low, and it is omitted from the language. Spring-C does, however, support what might be called C's *structured* jumps: *break*, *continue*, and *return*. They are also capable of transferring control between widely separated sections of a program, but their semantics are well defined for the block structures within which they operate. How the behavioral analysis takes them into account is discussed in Chapter 6.

The syntactic modification of the C loop constructs illustrated in Figure 4.1 makes the iteration bounds available to compile-time behavioral analysis. Note that the bounds expressions are evaluated at *compile* time, and thus may not depend on variables whose values are known only at run-time. The bounds are enforced at run-time by code emitted as part of loop infrastructure. If either the lower or upper bound is

```
recursive return_type
subr_name(param_defs) {
        body;
}

subr_name(param_list)(min_expr, max_expr);
```

Figure 4.2. Spring-C Recursive Routine Declaration and Call Syntax

violated, the *bounds_violation* system call is executed giving the system an opportunity to abort or continue executing the program, as it sees fit. This and other aspects of how the emitted code enforces loop and recursion bounds are discussed in Section 5.1.

The other source of unbounded execution, recursion, is addressed by two simple syntactic changes illustrated in Figure 4.2. The keyword *recursive* tells the compiler to emit the additional code required to enforce the recursion bounds as it compiles the procedure. The emitted code of a recursive procedure uses two additional parameters to enforce the recursion depth bounds. As with the loops, if the stated bounds are violated, the *bounds_violation* system call is executed giving the system an opportunity to abort or continue executing the program, as it sees fit.

The originating call to a recursive procedure supplies the recursion bounds using an additional pair of compile-time evaluable expressions, just as loops specify their iteration bounds. These values are supplied to the first call frame of the procedure as the initial values of the additional bounds parameters created when the routine was compiled. It is important to note that only simple recursion is currently considered, meaning the routine always calls itself directly until the base case is reached, and then returns through the stack of call frames to the original call. It would be far more

difficult to bound the execution time of more complex forms of recursive behavior, though it may be possible to extend the current methods to include some of them.

The syntactic changes to loops, recursive procedure calls, and recursive procedure declarations eliminate sources of unbounded execution. This was one of the two major requirements of the compile-time behavioral analysis. The other is to have all the places in the process's code where process execution might suspend clearly marked. This is addressed by a new set of statements added to the Spring-C language. The behavioral analysis can safely assume that these statements define the *only* places where a Spring process' execution will suspend, because of design decisions made at several other levels of the system; yet another example of Spring's highly integrated design and implementation.

The system is designed to eliminate the traditional sources of unpredictable interruption common on conventional systems [87]. Spring's functional partitioning shields application processes, running on the APs, from external interrupts. Timer interrupts are handled by the SP, which schedules new work, sends signals to specific APs, and updates the software clock on each AP as appropriate. Spring's scheduling method integrates CPU and resource scheduling, eliminating the need for a process to block during scheduled execution to wait for a resource to become available. The current version of the scheduler is non-preemptive, so it generates no interrupts to an executing task. Thus, a process's execution will suspend only when it terminates, or when it executes one of the new statements which suspends its execution: the explicit delay, synchronous send or receive, and the resource use statement.

An explicit *delay* statement is placed in Spring application code where the semantics of the computation require that execution of the process should suspend for a specified period. The statement takes a *compile* time evaluable argument specifying

the *minimum* time that should elapse between when a thread of control enters the *delay* statement, and when it leaves. The argument is evaluated at compile time because it must be taken into account by the behavioral analysis which constructs a task group representation of a process's worst case run-time behavior.

The *delay* statement will, as described in Chapter 7, eventually be represented by a precedence relation with the specified delay between tasks in the group representing the process, rather than by a *delay* system call having the delay value as an argument. The task boundary will, of course, be implemented by a system call that suspends the process, but Spring contains no *delay* system call. The distinction between the Spring-C *delay* statement and the system call actually used to suspend the process is important, and illustrative of a recurring theme in Spring's program translation method.

The system explicitly represents all the blocking and resource use aspects of a process's behavior as attributes of the task group, rather than leaving it to emerge at run-time. The obvious reason for this is that the scheduler requires this information *before* the process executes to construct an execution plan guaranteeing the process will complete by its deadline. One requirement for building a task group representation is knowing where the process can suspend, thus motivating the addition of the new set of statements to the language. These and related issues are discussed at greater length in Section 5.1.

Figure 4.3 gives the syntax of the *sync_send* and *sync_recv* statements. Synchronous communication statements are part of the Spring-C language for the same reason as *delay* is: to mark a process suspension point for the behavioral analysis. The *port_name* parameter is required so that the algorithm constructing the task group, described in Chapter 7, can find the corresponding send and receive statements for

each virtual circuit in each communicating process. Simply using a variable containing a port ID, as conventional systems do, is not sufficient since the variable is set at run-time and the synchronous communication patterns must be determined at compile time.

```
sync_send(''port_name'', param_list);

sync_receive(''port_name'', param_list);
```

Figure 4.3. Spring-C Synchronous Communication Statements

The synchronous communication statements are, in contrast to the *delay* statement, represented in the emitted code by the corresponding conventional system calls which take the *param_list* parameters as arguments, including the port ID variable. Note that *asynchronous* communication is a normal system call; it plays no special role in the behavioral analysis, since it creates no scheduling constraints.

The last statement added to Spring-C is the *with*, which is used to define the boundaries of each code block in the program source which uses a resource. Figure 4.4 illustrates the syntax of the statement, which takes a list of strings as arguments. Each string specifies the name of the resource, and the mode (*shared* or *exclusive*) in which the resource is used. Each resource use block of a process is represented by a task in its run-time representation using that resource, thus constraining how the task can be scheduled. When used in *shared* mode, other tasks using the resource may execute concurrently. When used in *exclusive* mode, other tasks using the resource may not execute concurrently. Obviously, if a part of the system is only used by a single process, or is always used in *shared* mode, then it need not be represented by a resource, since it represents no constraint on how tasks are scheduled. The Spring System Description Language (SDL) provides support for describing resources to the

```
with (''resource_name mode'') {
        body;
}

with (''resource1 shared'', ''resource2 exclusive'') {
        body;
}
```

Figure 4.4. Spring-C Resource Use Statement

system, as discussed in Section 4.2.2. The correspondence between resource use blocks and tasks is discussed in Chapter 7.

That completes the discussion of the syntax changes made to the original C in creating Spring-C. There is, however, a related issue: how a real-time computation is described at the highest level. In conventional systems, a computation runs once, and is described by a process with a single entry point. The convention in C is to name the entry point procedure *main*. The reason for this is pragmatic; the convention provides a simple way to perform appropriate system level initialization to prepare the process for execution, and then transfer control to the process's code. The compiler links every program with system specific initialization code, which contains a subroutine call to *main*, thus transferring control to the program.

A similar mechanism is used in Spring, modified in a way which reflects the difference between conventional and real-time computations. Most real-time computations can be divided into two sections; one performing initialization, preparing the address space to support real-time constraints, and the real-time portion. When the computation is periodic, an instance of the real-time portion executes during each period.

Figure 4.5 illustrates a technique commonly used to implement such a computation using conventional C. The *main* procedure begins with initialization code preparing

```
int
main () {
    initialization_code;
    while (another_iteration_desired) {
        real_time_code;
        period_control;
    }
}
```

Figure 4.5. Conventional Approach to a Real-Time Computation

```
int                          void
proc_init () {               proc_exec () {
    initialization_code;         real_time_code;
    return(1);               }
}
```

Figure 4.6. Spring-C Entry Points

the process to execute under real-time constraints, and then falls into a loop containing the real-time portion, and some code controlling the period of successive loop iterations. This approach can be effective, although obviously subject to the limitations of the period control implementation. However, it combines the computation with the control of its periodic execution, which is neither necessary nor appropriate under Spring.

Instead, Spring's programming interface separates the definition of the initialization section, real-time section, and execution constraints of a computation. As illustrated in Figure 4.6, the initialization section of a computation is specified using the *proc_init* entry point. This procedure is executed by the system *once*, when the process is first activated but before it begins executing under real-time constraints. It returns an integer value: non-zero for a successful initialization, and zero if initialization fails.

All operations required to prepare the address space of the program for execution under real-time constraints comprise the body of *proc_init*. It is important to note that the activation of the process and initialization of its address space are not subject to real-time constraints in the current implementation. This is convenient for two reasons. First, while the initialization must be completed for the program to function properly, they must only be done once and are not part of the computation executed under real-time constraints. Second, process activation occurs at boot time before real-time constraints apply. Processes can initialize their data spaces at the same time, and so bounding the behavior of the initialization code is not necessary. If the system develops to the point where processes are *activated* under real-time constraints, then deriving the worst case behavior for the *proc_init* routine will be worth the effort.

The real-time section of a computation forms the body of the *proc_exec* procedure. It is thus the *proc_exec* procedure which is subjected to the compiler's behavioral analysis, for which a task group representation is constructed, and for whose execution the scheduler constructs a plan. The system transfers control to *proc_exec* each time a new instance of the real-time computation is started. So, for example, the system transfers control to the *proc_exec* entry point of a periodic process once during each period. *Proc_exec* executes to completion, just as any other procedure does, and returns to its calling context when the computation is complete.

A computation's period, as well as other conditions and constraints used by the scheduler when building execution plans, is specified using the SDL, as described in Sections 4.2.1.1 and 4.2.1.2. Spring-C thus provides clearly defined, and separated, support for the initialization, real-time, and execution control aspects of a computation.

This concludes the discussion of the syntactic changes required to create Spring-C. The motivations for each of them should be clear; the discussion showing that they are sufficient to support Spring's behavioral analysis, and to ensure that a task group representation can be constructed for every Spring-C program, is presented in Chapters 5, 6, and 7. This section concludes with a short discussion of why these changes ensure that all Spring-C programs are finite, and why this is necessary to make Spring's behavioral analysis possible.

4.1.1 Spring-C Programs are Finite

The syntactic additions to Spring-C were motivated by the need to produce the behavioral predictions used to construct the task group representation of each process's behavior. This representation is required by the scheduler to construct an execution plan guaranteeing a computation will complete by its deadline. We will eventually demonstrate, in one of the main results presented by this dissertation, that a valid task group representation can be constructed for every Spring-C program. While that must await the information presented in Chapters 5, 6, and 7; it should be clear that this result depends on the ability to produce valid behavioral predictions for every Spring-C program. This depends, in turn, on every Spring-C program being *finite*, since it is impossible to produce a worst case behavioral prediction for a program which may not halt.

The halting problem presents obvious objections to such a claim, however, since it implies that all Spring-C programs *halt.* It is important to note, however, that we are claiming to have *avoided*, not *solved*, the halting problem. The gist of the argument is simple: all statements in Spring-C describe finite operations, any program of finite length written in Spring-C must thus represent a computation that will execute a finite

number of steps, and thus *halt.* The argument is based on a Assumption 1, which is supported by the discussion of the enhanced RTL and properties of the emitted code presented in Chapter 5, that all loop constructs enforce the loop bounds which were evaluated at compile time. Note that since the *goto* does not exist in Spring-C, its explicit loop constructs are the only way to create a loop.

Assumption 1: The code emitted for Spring-C loops and recursive procedures *enforces* the iteration and recursion depth bounds required by the Spring-C syntax and evaluated at compiled time, terminating the program when a bounds violation is detected.

Given this assumption, a simple proof by contradiction suffices to show that all Spring-C programs are guaranteed to described computations with a *finite* number of steps, and are thus guaranteed to *halt.*

Theorem 1: All Spring-C programs specify computations comprising a finite number of steps and, as *finite* computations, are guaranteed to *halt.*

Proof:

Assume that some Spring-C program P is *not* finite, and thus fails to halt. Since the source files of P must be of finite length, P can fail to halt if and only if it contains a statement describing an operation with an infinite number of steps: an *unbounded* construct. The only potentially unbounded constructs in Spring-C are loops and recursive procedure calls. However, this contradicts Assumption 1, that the bounds on loops and recursion required by the Spring-C syntax are enforced. The assumption that a Spring-C program P fails to halt produces a contradiction. All Spring-C programs will thus *halt*, and are *finite.*

The obvious limitation of this result is that while Spring-C programs will always halt, they may do so only because they are aborted after violating one of the iteration or recursion bounds.

4.2 System Description Language

The syntax changes made to the original C, creating Spring-C, ensure that all Spring-C programs are finite and that every place in the code where program execution can suspend is marked. However, performing the behavioral predictions, constructing the task group representation, loading the system, and then executing the computations predictably requires the ability to represent, use, and modify a great deal of information in addition to that provided by the syntactic changes described in the previous section. The System Description Language (SDL) was developed to address this need.

```
SDL {
    sdl-statements;
};
```

Figure 4.7. Spring-C SDL Block Statement

The SDL supports every piece of information required by the system at all stages of program compilation, downloading, execution, and debugging. In compiled form it makes existing information available to all tools for use or modification, as well as providing a way to preserve any new information they produce. While originally developed as a separate language, it has been fully integrated into Spring-C. The easiest way to accomplish this was to create the *SDL* statement, illustrated in Figure 4.7.

```
config_info  ::=   [conf_item]*
conf_item    ::=   layout
                 | network
                 | node_desc
                 | process_desc
                 | process_group_desc
                 | resource_desc
                 | shared_seg_desc
                 | task_group_desc
```

Figure 4.8. System Description Language Top Level

The Spring-C compiler accumulates the information from each SDL statement, making it available for use during compilation and analysis, and storing the accumulated information in the executable file produced. The information from each executable, as well as that from system level description files, is used by the system during booting and execution. Spring is a *reflective* system, since it must take the effects of its scheduling and process management activities on the real-time computations being supported into account [82, 83]. The SDL represents this reflective aspect of the system by representing the wealth of detailed information required by the kernel to properly schedule and support real-time computations [87]. These and other aspects of the SDL's role in supporting information exchange among SGS tools, and thus both mediating and promoting a more highly integrated system design, was discussed in Section 3.1.

Figure 4.8 shows the highest level of the SDL grammar, which permits an arbitrary stream of language elements. This is done to preserve flexibility in processing the application source files, since portions of the system description may appear at many locations within a source file, and can thus be processed in almost any order. However, it is reasonable to require that a single system layout specification and a single network

topology specification appear within the stream of definitions, which is a restriction enforced by the compiler.

Since it plays many roles, the SDL has several fairly distinct sections. Those already alluded to in previous discussions include the SDL's support for describing computations in terms of processes, process groups, and their run-time representations as groups of precedence related tasks. The SDL also supports the description of the resources and shared memory segments in the system, which are used by processes and considered by the system scheduler. A section of the SDL provides for describing the target nodes onto which the software must be loaded. This includes a description of each processor and memory board within each node, and the topology of the network connecting the nodes. The last part of the SDL presented specifies the system layout which lists the processor or memory board within each node to which every process, and shared memory segment is assigned.

The rest of this section will present each part of the SDL in detail, illustrate why the information it represents is required, how the information is used, presents an an example SDL description of a robot control application, and briefly describes the SDL interface library supporting all tools wishing to use, modify, or create SDL information.

4.2.1 Computation Descriptions

The specification of computations for a real-time system, or any system, can be done in a myriad of ways. In the Spring system we have adopted the *process*, a single thread of control in an independent logical address space, as the basic unit of computation. A *computation* can be implemented as a single process, or as a group of processes, where the execution order of processes in a group is constrained using precedence

relations. Process groups are useful for representing coarse grain concurrency within a computation, as well as describing structures used to support fault tolerance.

Each process is represented to the scheduler as a group of tasks, whose execution order is constrained by precedence relations. If a computation is represented by a process group, the precedence relations defining the structure of the task group representing the computation are derived from those of the task groups representing each process, and from the precedence relations defining the structure of the process group. How the structure of a task group representing a computation is determined is discussed in Chapter 7. There are, however, other sources of scheduling constraints beyond process and task group structure.

Processes communicate with one another either through messages or through shared data structures. Messages are sent and received across virtual circuits through *ports*, using the interprocess communication (IPC) facilities. Synchronous communication can be used between processes in a group implementing a computation. It creates scheduling constraints between tasks in the groups representing the communicating processes, as discussed in Section 7.2. The SDL and Spring-C include statements describing synchronous communication, but asynchronous communication is *not* explicitly represented since it has no impact on the task group representation of a computation, and creates no scheduling constraints. The Spring-C statements marking places in a process's code where it engages in synchronous communication were discussed in Section 4.1.

A resource is an abstraction most often, but not always, representing a set of items in the system that are shared among two or more processes, of which at least one process uses the resource in *exclusive* mode. The portion of a computation using a resource in *exclusive* mode cannot be scheduled concurrently with any portion of

another computation that uses the same resource. Resource use by a process, and therefore by the tasks in the group that represent the process to the scheduler, thus create constraints on how a computation can be scheduled. Resources generally, but not always, represent sets of shared data structures. The definition of resources is described in Section 4.2.2, the Spring-C *with* statement describing their use by processes was discussed in Section 4.1, and their use by tasks is described in Section 4.2.1.3.

A shared segment describes an area of memory that can be attached to the address spaces of one or more processes often, but not always, representing areas of address space *overlap* among the processes. The shared segment is visible in the address space of each process which *attaches* the segment to its address space during initialization. This is currently supported by system calls made from the process's *proc_init* initialization entry point, but will eventually be handled automatically. The SDL process description includes a list of the shared segments it uses, enabling the system to attach the required shared segments prior to transferring control to the *proc_init* entry point. Shared segments are generally used to support sets of shared data structures, but can also be used to represent memory mapped control and status registers of external devices. Shared segments are discussed in Section 4.2.3.

It is important to note the distinction between a computation's use of a resource and shared segment in Spring. A resource is an abstraction often, but not always, representing shared data structures that reside within a shared segment, and also representing constraints on how portions of a computation (tasks) are scheduled. A shared segment describes an area of memory often, but not always, containing one or more sets of data structures shared among two or more processes, where each set of shared data structures is represented by a resource. The shared segment and the

resource are thus closely related ideas, but ones which have distinctly different roles in the system.

Definition 1: A *group*, $G = (I, PR)$, is described by a pair of sets I and PR. I is the set of items in the group, while PR is the set of precedence relations $pr \in PR$, $pr = (i_1, i_2, d)$, where:

- $i_1, i_2 \in I$, and
- d is an integer specifying the minimum non-negative delay that should elapse between the completion of i_1 and the beginning of i_2's execution. Note that the "delay" specified can be zero.

Given $(i_1, i_2, n) \in PR$:

- i_1 is the *predecessor* of i_2, and
- i_1 is the *successor* of i_2,

Definition 1 presents the abstract notion of a *group*, which describes a set of items whose execution order is constrained by precedence relations. This representation is used to describe both process and task groups in the SDL, as well as to efficiently schedule a set of precedence related tasks, as discussed in Section 8.3.

Definition 2:

Given a group $G = (I, PR)$, an item $i \in I$ is *eligible* when all of its predecessors have completed their execution. An item with no predecessors begins in the eligible state.

When specifying the structure of a group in the SDL we use the simple, but effective, successor list representation. As a matter of convenience and efficiency when processing the compiled SDL form, the list has an additional item, always called *Begin*,

```
succ_list     ::=    succ_begin [succ_item]*
succ_begin    ::=    Begin: prec_list;
succ_item     ::=    name: prec_list;
prec_list     ::=    prec_item
                   | prec_list, prec_item
prec_item     ::=    list_item
                   | (list_item delay_val)
list_item     ::=    proc_group_name
                   | proc_name
                   | (proc_name task_name)
                   | task_name
delay_val     ::=    INTNUM
                   | (Comm_delay port_name_list)
                   | (Comm_delay INTNUM port_name_list)
```

Figure 4.9. Successor List

which is not part of the group being described. Its successor list, however, provides a convenient way to list the initially eligible items in the group. Figure 4.9 specifies the SDL grammar for a successor list.

The basic structure of the list is obvious, although the grammar is slightly complicated by the fact that successor lists are used for groups of tasks, processes, or groups containing process groups. The simplest form of a *list_item* identifies another element of the same group by name. However, it is also necessary to specify precedence constraints that cross task group boundaries, since they are required to represent synchronous communication. A member of a group is uniquely identified by the pair giving the name of the group and the name of the group member. The simple form for the delay value associated with a precedence constraint is an integer number of the system's basic time units. This is useful for describing delays resulting from the use of the *delay* statement in Spring-C.

However, delays can also be created by synchronous communication, which is more complicated, since delay associated with a communication channel is only known at boot time. This is why the grammar permits expressing a delay value in the communication delay form to specify a list of port names. More than one port name is permitted since the delay between tasks in a group may represent more than one communication act across more than one virtual circuit, as discussed in Chapter 7. Further, a given precedence constraint may represent the delay associated with both communication and *delay* statements, and so permits an integer argument. When communication is involved, the actual delay value used is determined at process activation or scheduling-time by taking the maximum of the delays associated with all of the virtual circuits involved, specified by the ports named, and that arising from *delay* statements, if any.

For example, consider a computation implemented by a group of two processes synchronously communicating through port *A*. The derivation of precedence constraints associated with synchronous communication is addressed in detail in Section 7.2. For the moment take it as given that the *sync_send* call will produce a task boundary where the predecessor task ends after the first part of the *sync_send* has been executed, and the successor task begins with the return from the *sync_send.* Since the sending process should not return from the *sync_send* call until enough time has passed to ensure the message's arrival on the receiving side, the delay associated with the precedence constraint is *(Comm_delay A)*. However, the communication delay can only be evaluated when the locations within the network of the sending and receiving processes using virtual circuit *A* are known. The SDL description of the task group in the process's executable thus contains the *(Comm_delay A)* form, and is evaluated during process activation.

The SDL must support the specification of *both* actual application code *and* of a simulated workload based on that application, as discussed in Section 3.1. As part of the support for simulations the SDL includes the *distribution* type used to specify numeric values; particularly execution times, deadlines, and laxities. The distributions are appropriate when specifying a workload for simulation, but have in several cases also been defined as specifying worst case values. The reason for this is that some distribution types, technically speaking, place no limit on the maximum value of the random variable. In such cases, the values in question play a slightly different role in the simulation where a distribution is required, and in the actual system, where a maximum value is needed. The distributions supported include the constant, uniform, exponential, and normal distributions. The grammars for the distribution types and those of other simple language elements such as names, numbers, and lists are specified in the complete SDL grammar in Appendix A.

4.2.1.1 Process Descriptions

The parameters describing a process, as presented in Definition 3 and the corresponding SDL grammar illustrated in Figures 4.10 and 4.11, are divided into three sets: those required if the process comprises the whole representation of a *computation*, those related to the process's *execution*, and those related to *scheduling* an instance of it.

The computation parameters describe the computation as periodic or non-periodic, give the period when relevant, specify a minimum separation that should be maintained between consecutive instances of the computation, and give the computation type. A computation can be *critical*, *essential*, or *non-real-time*. It is important to note, however, that the computation set is relevant *only* when the computation is

implemented using a single process. If the computation is implemented by a process group, of which this process is a member, then the computation parameters of the group are in control, rendering the computation parameters of each process in the group irrelevant.

Definition 3: A *process Proc* = (*Comp*, *Exec*, *Sched*) is defined by three sets of parameters, where:

- *Comp* is the computation parameter set, describing *Proc* as *periodic* or *non-periodic*, giving its *period* when relevant, and specifying a minimum *separation* between consecutive instances of the computation, periodic or not. It also specifies the computation's type, which can be *critical*, *essential*, or *non-real-time*.
- *Exec* is the executable parameter set which names the file containing the process's executable code, lists the shared segments *Proc* attaches to its address space, lists the data structures imported from those shared segments, and lists the synchronous communication virtual circuit ports used by *Proc*.
- *Sched* is the scheduling parameter set, which gives the deadline, deadline type, and laxity of *Proc*. A deadline can be *hard*, *soft*, or *non-real-time*.

The execution parameter set specifies what file contains the process's executable image, what data structures it imports, what shared memory segments it uses, and the virtual circuit ports through which it engages in synchronous communication. The list of imported data structures specifies the shared data structures used by the process, which reside within one or more shared memory segments, but may or may not be represented by resources. As discussed in Section 3.2, the classification of a data structure as part of a shared segment or private to the process is an important

```
process_desc  ::=   Process(name) { [proc_attr]* } ;
proc_attr     ::=   comp_spec;
                  | exec_spec;
                  | sched_spec;

comp_spec     ::=   Comp_type comp_type;
                  | Non_Periodic;
                  | Period INTNUM;
                  | Periodic;
                  | Separation INTNUM;

exec_spec     ::=   Code name
                  | Import name_list;
                  | Sharing name_list;
                  | Sync_ports port_list;

sched_spec    ::=   Deadline dist_spec;
                  | Deadline_type dln_type;
                  | Importance INTNUM;
                  | Laxity dist_spec;
                  | Value INTNUM;
```

Figure 4.10. Process Specification

```
comp_type  ::=   Critical
               | Essential
               | Non_essential
dln_type   ::=   Hard
               | Soft
               | Non_real_time
port_list  ::=   port_item
               | port_list, port_item
port_item  ::=   name
               | (name use_type)
use_type   ::=   Receive
               | Send
```

Figure 4.11. Process Related SDL Elements

part of determining the time required to access it in an architecture with non-uniform memory access (NUMA).

The specification of communication ports, *Sync_ports*, only lists those used for synchronous communication because synchronous communication has an impact on the computation's representation and scheduling, while asynchronous communication does not. The communication ports are listed at this level in the SDL because each process attaches to a port and then sends to or receives from it. The synchronous port list for each process is used by the algorithm constructing the task group, as discussed in Section 7.2, and enables it to detect some classes of synchronous communication errors at compile time, including contradictory communication patterns.

The scheduling parameters specify the process's deadline, type of deadline, value, and importance. Deadlines may be hard, soft, or non-real-time, while processes are critical, essential, and non-essential. The laxity statement is there to support scheduling simulations. In the real system the initial laxity of a process is determined by its deadline, and the time when the system first becomes aware of the need to schedule it. The value of the process specifies how much the system earns by completing its execution. The importance rates the significance of the process to the system, and is used by some scheduling algorithms for load shedding. The distinction between value and importance is subtle, but the SDL supports both, since some scheduling algorithms use one, and some use the other [7, 98].

Note that not all combinations of computation and deadline type are allowed; the combination of a critical computation with a non-real-time deadline is, for example, nonsense. Note also that when *Proc* is part of a group, the values of its scheduling parameters may be derived in whole or in part from the scheduling parameters of the group.

4.2.1.2 Process Groups

The parameters describing a process group are divided into several sets, as described by Definition 4, and Figures 4.12 and 4.13. The reason for supporting the definition of process groups is simple: some computations are more conveniently described as a set of cooperating processes than as a single process. It is important to remember, however, that the system's primary goal is to ensure that *computations* execute according to their real-time constraints. Describing, manipulating, and managing processes, singly or in groups, is simply its current means of achieving that primary goal.

Definition 4: A process group $PG = (Procs, PR, Comp, Fault, Sched)$, represents a set of two or more processes implementing a computation, where:

- *Procs* specifies the set of processes which are members of the group, also denoted $Procs(PG)$,
- *PR* is the set of precedence relations among the processes in the group, and is also denoted $PR(PG)$.
- *Comp* is the computation parameter set, describing *PG* as *periodic* or *non-periodic*, giving its *period* when relevant, and specifying a minimum *separation* between consecutive instances of the computation, periodic or not. It also specifies the computation's type. A computation can be *critical*, *essential*, or *non-real-time*.
- *Fault* specifies the fault tolerance structures of the process group.
- *Sched* is the scheduling parameter set, which gives the deadline, deadline type, and laxity of *Proc*. A deadline can be *hard*, *soft*, or *non-real-time*.

```
proc_grp_desc  ::=   Process_group(name) { [pg_attr]* } ;
pg_attr        ::=   comp_spec;
                   | Fault_tolerance { ft_spec };
                   | Process_graph { succ_list } ;
                   | sched_spec;
```

Figure 4.12. Process Group Specification

The most obvious aspect of a process group is the specification of the group structure using the successor list described in Figure 4.9. The precedence constraints between processes are translated into precedence constraints that hold between tasks in the groups describing the processes involved. A larger group representing the computation as a whole is thus created from the task groups representing the component processes. Almost arbitrarily complex structures are possible, since a given process group can be an element of another process group at a higher level.

The computation parameters, as explained in Section 4.2.1.1, describe the computation as periodic or non-periodic, give the period when relevant, and so on. It is important to note that, as with the process, the computation set is relevant *only* when the computation as a *whole* is implemented by the process group described. If the computation is implemented by a higher level process group, of which the group described is a member, then the computation parameters of the higher level group are in control, rendering the computation parameters of each process and lower level group irrelevant.

The scheduling parameters are also the same as those for the process. Their values may be used to help calculate the values of the scheduling parameters for the processes comprising the group, and may be calculated from those of a higher level group containing this one, if it exists. The methods for deriving the scheduling

```
ft_spec            ::=   alternative_spec
                       | copies_spec;
                       | pb_spec;
                       | voting_spec;

alternative_spec   ::=   Alternative proc_name_list;

copies_spec        ::=   Copies proc_name (min_cp, max_cp);
                       | Copies proc_name INTNUM;
min_cp             ::=   INTNUM
max_cp             ::=   INTNUM

pb_spec            ::=   primary_proc backup_proc
                       | backup_proc primary_proc
primary_proc       ::=   Primary proc_name;
backup_proc        ::=   Backup proc_name;

voting_spec        ::=   Voting { voters arbiters };
voters             ::=   Voters proc_name_list;
arbiters           ::=   Arbiters proc_name_list;
```

Figure 4.13. Fault Tolerance Specification

parameter values of subordinate components from those of the component containing them are specific to each scheduling algorithm.

Figure 4.13 presents four varieties of fault tolerant structures: alternative, copies, primary-backup, and voting. The alternative type is useful for handling *scheduling* faults. The alternative list specifies that members of the group represent the computation in different ways, and the order in which they should be considered. If the scheduler fails to find a feasible schedule using one process, then the next process on the alternative list can be tried.

The other three types are meant to address *execution* faults in different ways. The *primary-backup* type specifies a pair of processes which are *both* scheduled, but where the primary should be scheduled in such a way that if it completes successfully, the

backup can be canceled. Note that the backup can be canceled whether or not it has already started execution, although doing so *before* it begins execution is clearly simpler. If the primary fails, then the schedule constructed will run the backup process by its deadline. This is useful for specifying a preferred version of a computation, the primary, and a minimal version, the backup.

The *copies* type of fault tolerance is appropriate to situations where only one version of a computation is available, but the developer wishes to guard against hardware faults. In that case, the number of copies of the process may be specified. Either an absolute number of copies, or a minimum and maximum can be given. In the latter case, the scheduler has some freedom to choose the number in the context of the current system load. Finally, the *voting* type of fault tolerance specifies the set of voting processes, and the arbiter process that collects the results. Note that the arbiter process could itself be a group.

These constructs are a first approximation of those that will eventually be required as fault tolerance issues are addressed within Spring as a whole, and were intended to help support research focusing on fault tolerance issues [95]. The role of the SDL is limited, but vital, since it provides the ability to specify the level and type of fault tolerance required. The statements currently supported may be used, modified, or replaced as research addressing fault tolerance in Spring continues.

4.2.1.3 Task Groups

A task group represents the worst case run-time behavior of a computation, and is used by the scheduler when constructing an execution plan. Definitions 5, 6, and 7 present the task group, task, and resource use, respectively. The SDL grammar supporting these ideas is given in Figure 4.14. The task group definition is separated

from that of the process in the grammar because the description of the process appears in its source code, while the task group representation is generated in compiled form during process compilation and linking. However, the system must *also* provide the ability to specify a task group at the SDL source level to support scheduling simulations. While we are interested in simulating application code, we are also interested in constructing simulations for workloads that may *not* represent actual application code.

Definition 5: A task group, $TG = (name, Tasks, PR)$, represents a process's worst case run-time behavior to the scheduler, where:

- *name* gives the name of the process whose worst case run-time behavior is represented,
- *Tasks* is a finite set of tasks, and
- PR is a set of precedence relations among the tasks in the group.

The task groups representing the behaviors of two processes or process groups A and B are denoted $TG(A)$ and $TG(B)$ respectively. The task set and precedence relation set of $TG(A)$ are denoted $Tasks(TG(A))$ and $PR(TG(A))$, respectively. When not ambiguous, the more compact notation $Tasks(A)$ and $PR(A)$ can be used. Ambiguity, for the PR set at least, arises when A is a process group, and $PR(A)$ already refers to the *process* level precedence relations rather than those at the task level.

Definition 6: Each task $task = (WCET, RU, Sched)$, is described by three sets of parameters, where:

- $WCET$ is the worst case execution time of the task, also denoted $WCET(task)$,

- *RU* describes the resource use of the task, also denoted *RU*(*task*), and
- *Sched* is the scheduling specification of the task.

Definition 7: Each resource use description $r = (res, mode, eu, lu)$, is described by four items, where:

- *res* is the name of the resource;
- *mode* is the mode, either *exclusive* or *shared*, in which the resource is used by the task;
- *eu* is the earliest time *relative* to the beginning of *task* that *res* is used;
- and *lu* is the latest time relative to the beginning of the task during which it is used.

The name cited in the SDL task group specification, illustrated in Figure 4.14, must be the name of the process it represents. The definition of a task group has two parts. The *Group_list* specifies the structure of the group using the successor list method. Following that is a description for each of the tasks in the group. Each task has a name, which need only be unique within the group. The scheduler, when working with all the tasks on the system can uniquely identify a task by the process name and task name pair. Note that for processes engaging in synchronous communication, the successor list of a given task group contains references to tasks in the group representing the other communicating process. Such references are of the form ((proc_name task_name) (Comm_delay port_name)) presented in Figure 4.9. This constraint says that a successor of the task in whose list it appears is the task named *task_name* within the group representing the process *proc_name*, and having a delay associated with communication through the virtual circuit port *port_name*.

```
task_grp_desc   ::=   Task_group(proc_name) { tg_def } ;
tg_def          ::=   group_def [task]+
group_def       ::=   Group_list { succ_list } ;

task            ::=   Task(name) { [task_attr]* } ;
task_attr       ::=   M_time dist_spec;
                    | Non_Preemptive;
                    | Preemptive;
                    | Resources [ru_spec]+;
                    | sched_spec;
                    | W_time dist_spec;

ru_spec         ::=   (name [use_attr]+)
use_attr        ::=   Start time_val;
                    | End time_val;
                    | Exclusive;
                    | Sim_prob use_prob excl_prob;
                    | Shared;
use_prob        ::=   FLOATNUM
excl_prob       ::=   FLOATNUM
```

Figure 4.14. Task Group Specification

The task uses the same scheduling parameter specification, *sched_spec*, as a process or process group. As discussed in that context, the deadline and other scheduling parameters of the task are derived from those of the higher level item, in this case a process, that it helps represent. The method used to derive the task's scheduling parameters is specific to the scheduling algorithm being used. For example, some scheduling algorithms require intermediate deadlines to be derived for each task in the group from the deadline of the process, during a preprocessing phase executed after creation of the process's executable but before scheduling the task group at run-time. Other scheduling algorithms may assume that the deadline of each task is that of the process or process group it is helping represent.

The use of the worst case execution time is obvious. A mean time is also supported to aid in simulation studies. Both mean and worst case execution times are represented by distributions to aid simulation, although only a constant value for the worst case time is meaningful for software executing on the real system. As development of the system proceeds, we also envision expanding our representation of the WCET. One such extension would represent WCET as a function of one or more inputs to a computation known at scheduling time.

The specification of the task's resource use can take two forms. The form used to describe resource use of tasks representing real computations, and a form used only for simulations. The form representing real computations gives the resource name and its mode of use, which is either *shared* or *exclusive*. This form can be used for simulations as well, of course, but the other form can *only* be used for simulations. It gives the name of the resource, the probability that the resource will be used and, if used, the probability that it will be used in exclusive mode. The other two parameters make it possible to specify, relative to the beginning of the task, the earliest time at which the use of the resource can *start*, and the latest time at which it can *end*. This information has the potential to increase the schedulability of a task set, since it results in lower demands on resources, and thus loosens constraints on concurrent task execution, as discussed in Chapter 6.

4.2.2 Resources

From the Spring scheduler's point of view, a *resource* is simply an abstract object used by tasks in either shared or exclusive mode. Their use of resources implies constraints on what tasks can be scheduled to execute concurrently, and thus constrains the execution plans that can be constructed by the scheduler. There is currently no

formal definition of a resource, because it need consist of nothing more than a name. A resource usually, but not always, represents a set of data structures shared by two or more processes. An obvious exception to this is that the processors on which the processes run are resources used by the tasks representing the process for scheduling purposes.

Within the programming model, the resources represent sets of objects to which exclusive access is at least *sometimes* required. Spring-C supports the *with* statement describing the use of a resource in either exclusive or shared mode, as described in Section 4.1. Critical sections in a program can thus be represented as exclusive use of a resource representing the shared data, and the scheduler will enforce the exclusive use constraint. While a resource is used to *represent* a set of shared data structures to which access must be controlled, a shared memory segment is required to *contain* them. Shared segments are described in Section 4.2.3.

Note that this approach works well even when using a resource to represent a set of hardware control and status registers, since they are usually memory mapped. The basic problem, then, is to provide a way to describe the abstract idea of a resource in a way that is appropriate to the requirements of the scheduler, and to describe shared memory support for the data structures comprising resources in a way that is appropriate to the requirements of the SGS and operating system.

Figure 4.15 presents the SDL grammar for describing a resource. Each resource has a name, and specifies the type of access permitted: shared, exclusive, or both. Note that it is superfluous, from a scheduling point of view, to define a resource unless at least one process uses it in exclusive mode, since otherwise it produces no constraint on scheduling and could be ignored. However, the resource definition is

```
resource_desc  ::=   Resource(name) { [res_attr]* };
res_attr       ::=   Access access_type;
                   | Segment name;
                   | export_sym
                   | Instances INTNUM;
                   | Mode mode_type;
                   | Type res_type;

res_type       ::=   Read
                   | Write
                   | RW
access_type    ::=   Both
                   | Exclusive
                   | Shared
export_sym     ::=   Export name_list;
                   | Export export_type name_list;
export_type    ::=   Direct
                   | Indirect
mode_type      ::=   Appl
                   | Both
                   | Sys
```

Figure 4.15. Resource Description

still useful for grouping shared data structures, and exporting them for use by the processes requiring them, even when no process uses them in exclusive mode.

Describing such groups of shared data structures as a resource is also prudent considering the extent to which software commonly evolves. A program modification requiring exclusive use of a data structure which was not already represented by a resource would require creating a new a resource to represent it, finding all uses of the shared data structure in the application code, and placing each such use in the body of a *with* statement. It is much simpler to represent sets of shared data structures as resources from the beginning. Resources used only in shared mode can be identified during compilation, and the software optimized by eliminating the

resource in question from the view of the scheduler. The advantage of this is that when an exclusive mode use of the resource is added to the application, the code only needs to be recompiled, not rewritten.

Each resource also has a type, specifying whether access to it is limited to reading, writing, or if both are permitted. This information can eventually be used by the SGS and operating system to enforce such restrictions, increasing the security and robustness of the system. The description specifies a mode for the resource as an aid in checking the final system layout, and as a way of supporting future development. The mode declares whether the resource is used by system or application processes, and provides for the possibility that a resource could be used by both.

For those resources representing sets of shared data structures, the grammar includes a list giving the names of the data structures comprising the resource. These structures are *exported* from the resource by name, either *directly* or *indirectly.* The distinction has important implications for the ability of the SGS and system to ensure proper execution of the process. A data structure can be exported *directly* if all processes using the data structure reference it directly by name, or by address through a pointer that is not shared. A data structure *must* be exported *indirectly* if a process can access it using a pointer, *which is itself shared.* Direct exportation of shared data structures places a less stringent constraint on the placement of the shared data structures within the address spaces of the processes using them than does indirect exportation, so the distinction can have a significant impact on the address space structure of the processes using the resource.

The reason for this arises from the fact that the data structures represented by the resource are contained within a shared segment attached to the address spaces of two or more processes. If the data structures within a resource are exported directly,

then each process can map the segment to a different part of its address space, and still be able to access the data without error. If the data structures are exported indirectly, however, that means there is a shared pointer giving access to other shared data structures, and the value of that pointer must be valid in the address spaces of *every* process using it. This implies that the segment containing the structure being accessed through the shared pointer must be assigned to the *same* base address in all spaces to which it is attached. If the segment containing the indirectly exported data structure was *not* assigned to the same base address in all spaces, then the shared pointer's value would be invalid for at least one process using it.

4.2.3 Shared Segments

Shared segments in the Spring system have a number of interesting properties. A set of system calls enables processes to create and attach shared segments to their address space at run-time. This interface is currently intended for use only during process initialization, but could in principle be used when executing under real-time constraints. Such shared segments have no internal structure from the system's point of view, and must be accessed using conventional methods of pointer assignment and manipulation. However, this makes the problem of determining the memory access time for the structures more difficult, since the analysis must decide the relative location within the NUMA hierarchy of the code executing the pointer reference, and the memory referenced. While the system will eventually be able to handle this conventional method of access to dynamically created shared segments through pointers, it is also convenient to permit the definition of shared segments at compile time within which data structures are accessed by name.

In the Spring system, memory segments can exist as independent objects, defined at compile time using the SDL, and created during system boot, prior to the activation of any processes. Figure 4.16 shows the SDL grammar for a shared segment description. A formal definition is not presented, since a shared segment consists of nothing but a name, and some memory with which specific data structures may or may not be associated. The issues of interest arise from how these memory segments are created and used during compilation and process execution. System level implementation issues related to supporting and using shared segments are discussed elsewhere [61].

Segments defined at compile time are said to be *predefined*, since they already exist when processes wishing to attach such segments to their address space are activated. The SDL *layout* specification is used to specify where predefined segments should reside within the various areas of memory available in the target node. On the other hand, shared segments can also be created during process initialization using the *shm_attach* system call. In that case, the order in which the processes using the segment are activated can influence the location of the segment within the node's memory. The reason for this is that the physical memory supporting the segment is allocated on the processor which runs the *first* process to attach the segment.

Predefined shared segments are useful in a number of ways including: as a way to provide access to memory mapped control and status areas for external devices of all kinds, to support system status information made available to application programs, and to ensure unique logical addresses for segments supporting shared data structures that are exported indirectly. Every predefined shared segment has a name which is used at compile time by the SGS, at boot time by the system when activating a

```
shd_seg_desc  ::=   Shared_seg(name) { [seg_attr]* };

seg_attr      ::=   Code name;
                  | Logical_base HEXNUM;
                  | Matching;
                  | Memory_mapped;
                  | Mode mode_type;
                  | Physical_base HEXNUM;
                  | Predefined;
                  | Resources name_list;
                  | Size HEXNUM;
```

Figure 4.16. Shared Segment Description

process, and by each process attaching the shared segment to its address space using the shared memory system calls during its initialization.

Segments which are *Memory_mapped* must also specify the *Physical_base* address of the existing area where, for example, a device's control and status registers are mapped. No physical memory is allocated for these segments from the page pool maintained for each processor within a Spring node. A mode is specified for each segment for the same reasons of consistency and error checking as applied to the resource descriptions given in Figure 4.2.2. The *Matching* attribute specifies that the segment must appear at the same logical address in all spaces to which it is attached. This is required to properly support indirectly exported data structures, as discussed in Section 4.2.2. Since a *Matching* segment is always predefined, *Matching* implies *Predefined.*

The logical base address of a segment can either be specified using the SDL or assigned during system boot, but the choice is constrained by how processes sharing the segment expect to access the data structures. If the application wishes to access the data structures in the segment by name, then a logical address for the segment

must be specified in the SDL description. If the segment has the *Matching* attribute because it contains indirectly exported data structures, then the *Logical_base* is the address at which the segment will appear in each space to which it is attached. The *Logical_base* must be specified, and an executable for the segment produced, because an executable file for the process accessing the shared data structures by name cannot be successfully linked unless the logical addresses of those shared variables are known.

This information is, however, available from the shared segment's executable files. Each shared segment containing symbols referenced by name is described by an executable file whose symbol table gives the logical address of each symbol, as determined by the base address specified when the segment's executable is produced. The *Code* statement specifies the name of the executable file, enabling the linker to find the executable files for the shared segments a process uses, and thus the addresses of the shared symbols referenced by name.

A segment containing only directly exported data structures might specify a *Logical_base* of zero. Then, when the process using it is linked, a base address for the segment would be chosen, and the addresses of the shared structures relocated relative to the assigned base. Thus, each process using this shared segment could choose to attach it to their address space at a different place. However, the modifications to the linker, the SDL information included in the executable file, and changes to the system required to support this have not yet been implemented. If the logical base address is *not* specified, then it will be assigned during process initialization, either as the system activates the process or when a shared memory system call is made.

A segment's size is always specified, and the set of resources supported by the segment is also listed. Under the current system implementation, each segment must use at least one page of memory. Since this will often be much larger than required for

```
node_desc          ::=   Node(name) { [node_attr]* };
node_attr          ::=   processor_desc
                       | mem_board_desc

processor_desc     ::=   Processor(name) { [processor_attr]* };
processor_attr     ::=   memory_area_spec
                       | Use processor_use;
mem_base           ::=   HEXNUM
mem_size           ::=   HEXNUM
processor_use      ::=   Appl;
                       | IO;
                       | Sys;
mem_board_desc     ::=   Mem_board(name) { [mem_board_attr]* };
mem_board_attr     ::=   memory_area_spec
memory_area_spec   ::=   Memory_area(mem_board_use, mem_base, mem_size);
mem_board_use      ::=   Data;
                       | Memory_mapped;
```

Figure 4.17. Node Structure Grammar

the data structures of a given resource, it is prudent to place more than one resource in a segment. While the method of supporting logical memory may change, obviating the need to group large numbers of shared structures into a single segment, some need to group resources onto segments will probably always exist.

4.2.4 Target Node Description

The portions of the SDL which describe application software have been described in previous sections. This section discusses the portion of the SDL used to described the hardware onto which the software is loaded. The grammar for this section of the SDL is specified in Figure 4.17. Each node is given a name, and contains boards which are either processor or memory boards. Each processor board is given a name, and the global address range of its on-board memory, if any, is defined. Every memory

board is also named, and its address range specified. The names given to boards only need to be unique within the node.

Note that the memory board notation is used to describe all memory mapped devices, as well as normal memory. For example, the memory mapped control and sensor registers for a robot would be described as a memory board. For such devices, it is reasonable to permit defining more than one memory area under a single name. Note that in the case of the memory board, the memory area has a use attribute. This distinguishes between memory used for data and that mapping device registers, which gives the SGS information helping it to detect and avoid the error of misusing a memory mapped board by assigning data structures to it.

Finally, note that the node description does not contain a section specifying how the processors within a node are connected. Several descriptions are possible, including one analogous to that given for the network topology in Section 4.2.5. We currently assume a simple bus connection among all boards in the node, which is why a single base address is sufficient to identify the location of the memory associated with the board. Arbitrarily complex target hardware could be supported by fairly straightforward additions to the grammar to represent interconnections, and their access time characteristics.

4.2.5 Network Topology

The compilation and downloading of the system and application software only requires a view of the structure of nodes, but if the system is meant to run on a network of nodes it is prudent to permit a description of the connections among them. This information can be of use to the system as it boots each node, since it must establish connections to its companions on the network. Figure 4.18 gives the grammar for

```
network_topology   ::=   Network { [node_connections]* };
node_connections   ::=   Node (name) { connection_list };

connection_list    ::=   connection_spec
                     |   connection_list, connection_spec
connection_spec    ::=   Connection (node_name) { [connection_attr]* } ;
connection_attr    ::=   Latency INTNUM;
                     |   Speed INTNUM;
```

Figure 4.18. Network Topology Grammar

describing the network topology. Each node in the network has a section specifying the names of the nodes to which it is connected, and the properties of the connection.

The current version of the network specification is quite simple, because the current needs of the Spring system are modest. Each connection has a basic *Latency* value associated with it, which gives the time required to begin transmission of a message across the network, and the *Speed* attribute, which gives the rate at which information is transmitted, once transmission has begun. These are intended as generically useful attributes, but others will be required to properly describe networks supported by different types of hardware or communication protocols.

4.2.6 Layout

The sections of the SDL describing all the objects that can be loaded onto a Spring node have been discussed in previous sections, as have the portions of the SDL describing the target node and interconnection architecture. The only part of the SDL left is that used to describe the assignment of the software to specific places within the target hardware, which requires specifying the set of items assigned to each processor

```
system_layout       ::=   Layout { [node_layout]* };

node_layout         ::=   Node_layout(name) { [node_layout_attr]* };
node_layout_attr    ::=   processor_layout
                        | mem_board_layout

processor_layout    ::=   Processor_layout(name) { [proc_layout_item]* };
proc_layout_item    ::=   Appl_set { [proc_load_item]* };
                        | System_set { [proc_load_item]* };

mem_board_layout    ::=   Mem_board_layout(name) { [mb_layout_item]* };
mb_layout_item      ::=   Appl_set { [mb_load_item]* };
                        | System_set { [mb_load_item]* };

proc_load_item      ::=   Boot_proc name;
                        | Process_set name_list;
                        | Resource_set name_list;
                        | Shared_seg_set name_list;

mb_load_item        ::=   Resource_set name_list;
                        | Shared_seg_set name_list;
```

Figure 4.19. System Layout Grammar

and memory board within each node. Figure 4.19 shows the grammar for describing the system layout.

The layout specification reflects two basic ideas: that the application software and system software are described separately, and that each set of software can be described in terms of processes, resources, and shared segments. Memory boards, obviously, can only support resources and shared segments. We list the system and application items for each board separately because while research in real-time applications may use a standard set of system code, many different versions of the system code may be used for research at the operating system level. Specifying the

sets separately makes it easier to understand the software structure of a particular experiment.

While the application and system code are both described as sets of processes, resources, and shared segments, there are some subtle differences. For example, the system set for each processor must identify the booting process (*Boot_proc*), whose executable is loaded first and its execution started. The boot process takes care of properly initializing the board, and establishes a communication channel with the boot processes on the other boards within the node and with the debugger *sbug*. The boot processes, in cooperation with *sbug*, are then ready to download the rest of the system and application objects, activating them on the boards of the target hardware as required by the layout specification.

Using separate resource and shared segment lists creates a certain amount of redundancy, but we accept it for two reasons. First, the two lists address information used by different parts of the system. The resource list applies to scheduling, since resources represent scheduling constraints, while the segment list specifies some of the executable files loaded on each board. Second, while most resources will be supported by a shared segment, not all will require it, and not all shared segments will support resources. Separate lists let the system retain enough flexibility to handle every situation.

4.2.7 SDL Example

The SDL is large, and while the purpose of any given section is often easy to understand, an illustrative example may make it easier to see how it is applied to the description, compilation, and execution of an application as a whole. Its support for the SGS information flow described in Section 3.1 is of particular interest. Consider,

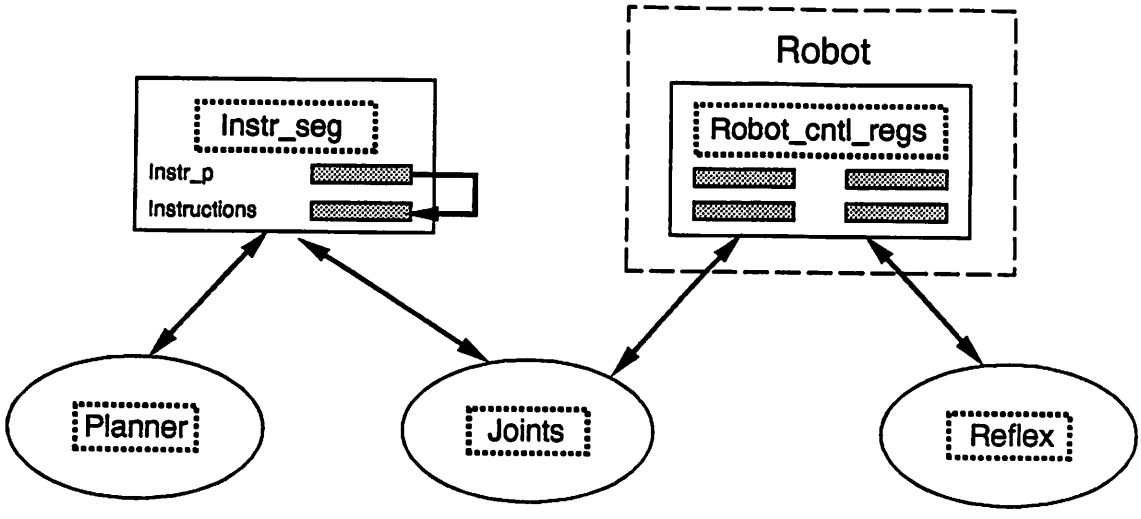


Figure 4.20. Robotics Example Process Architecture

for example, the software controlling a robot. A simple software architecture for this might consist of three independent computations, each implemented as a process, which communicate with the robot and one another through the *Instr_seg* and *Robot_cntl_seg* shared segments, as illustrated in Figure 4.20.

The *Reflex* process is responsible for periodically examining the control and status registers of the robot, made visible by attaching the *Robot_cntl_regs* segment to its address space, to determine that the robot is operating within acceptable limits, and for taking corrective action if it detects anything untoward. Recall that the *proc_init* entry point of *Reflex* is responsible for attaching *Robot_cntl_regs* to *Reflex*'s address space when the process is activated. Figure 4.21 shows the SDL process description for *Reflex*. The computation is *critical*, periodic, and has a period of 25 time units. The executable file containing the process code is named "reactive", and the *Robot_cntl_regs* shared segment will be attached to the process's address space. Its deadline is 25, relative to the start of the period, implying that the scheduler is free

to plan its execution anywhere within each period. No minimum separation is specified, so two consecutive instances of the computation might be scheduled very close together; one at the end of its period, and the next at the beginning of the following period. The deadline is *hard* which, in combination with the critical computation type, means the system is not permitted to fail to scheduling the computation during each period.

```
Process(Reflex) {
 *** Computation Spec ***
   Comp_type      Critical;
   Periodic;
   Period         25;
 ***      Exec Spec     ***
   Code           reactive;
   Sharing        Robot_cntl_regs;
 ***  Scheduling Spec ***
   Deadline       25;
   Deadline_type Hard;
};
```

Figure 4.21. Single Reactive Process Example

The other two application computations are implemented by the processes named *Planner* and *Joints*. The *Planner* process, described by the SDL in Figure 4.22a, is responsible for formulating a movement plan for the robot whenever a new command is received from the external environment, and is thus non-periodic. It is classified as essential, and its executable code is contained in the file "planner". It imports the pointer variable *Instr_p*, and uses the shared segment *Instr_seg* containing the imported variable. It has a soft deadline of 300, which is relative to the time of the environmental event invoking the computation since it is non-periodic.

Figure 4.22b describes the joint controller process *Joints*. The file "joint_control" holds the executable code, and the process requires access to *Instr_seg*, since it also

```
Process(Planner) {
 *** Computation Spec ***
   Comp_type  Essential;
   Non_Periodic;

 ***      Exec Spec     ***
   Code         planner;
   Import       Instr_p;
   Sharing      Instr_seg;

 ***  Scheduling Spec ***
   Deadline       300;
   Deadline_type Soft;
   Value          30;
};
```

(a) Planner Process

```
Process(Joints) {
 *** Computation Spec ***
   Comp_type  Essential;
   Periodic;
   Period       50;
 ***      Exec Spec     ***
   Code         joint_control;
   Import       Instr_p;
   Sharing      Instr_seg,
                Robot_cntl_regs;
 ***  Scheduling Spec ***
    Deadline       40;
    Deadline_type Hard;
    Value          60;
};
```

(b) Joints Process

Figure 4.22. Process Description Example

imports *Instr_p*. This process is periodic, and has a hard deadline of 40 time units from the beginning of its period, 10 units before its end. Note that specifying the deadline in this way is similar but not exactly the same as specifying a separation of 10. The process also requires access to the control registers of the robot, and so uses the *Robot_cntl_regs* segment. The semantics of the computations are simple; *Planner* formulates a plan as a series of joint motions, while *Joints* is responsible for executing the plan.

Figure 4.23 shows the specifications of the two shared segments and the resource. The *Instr_res* resource definition specifies that it is exporting the data structure *Instructions* indirectly, and *Instr_p* directly. Access to the elements of the resource is always exclusive, and the resource is supported by the segment *Instr_seg*. Only application processes use the resource, and it can be both read and written.

The shared segment named *Instr_seg* is used only by application processes. The logical addresses of the data structures it holds are given by the symbol table of

```
Shared_seg(Instr_seg) {
  Predefined;
  Mode          Appl;
  Matching;
  Logical_base 0x50000;
  Size          0x2000;
  Code          Instr_seg;
  Resources     Instr_res;
};

Shared_seg(Robot_cntl_regs) {
  Predefined;
  Mode          Appl;
  Memory_mapped;
  Physical_base 0xc00000;
  Size          0x200;
};

Resource(Instr_res) {
  Access           Exclusive;
  Segment          Instr_seg;
  Export  Indirect Instructions;
  Export  Direct   Instr_p;
  Mode             Appl;
  Type             RW;
};
```

Figure 4.23. Resource and Shared Segment Descriptions

the executable file *instr_seg*. The segment has the *Matching* attribute, specifies its *Logical_base*, supports the *Instr_res* resource, and is $8K$ in size because of the page size in the current Spring system implementation. Clearly we would wish to group more resources on the page, if possible, to avoid wasting space.

In contrast to *Instr_seg*, the shared segment *Robot_cntl_regs* is not associated with any resource. It is *Memory_mapped* and assigned a physical base address since it represents the robot's control and status registers which are mapped to the specified physical memory. The segment is predefined, and is thus available to any process wishing to attach it to their address space. When it is attached to each process's address space, part of the operation is assigning it a logical address and adjusting the process's memory map accordingly. No resource is associated with it, and no executable file is used to describe its internal structure. The processes using it must thus gain access to its control registers through pointer assignment and manipulation.

Now consider how the source files describing the processes, resource, and shared segments are compiled, and the novel constraints on the compilation order which arise from the need to produce worst case behavioral predictions and to access shared data structures by name. Section 3.1 discussed the need to impose a partial ordering on procedure compilation so that the behavioral predictions for a procedure A are available when all procedures calling A are compiled. This constrains the structure of source files, and the order in which they are compiled, in a way that is not common in conventional systems, although even some conventional systems might benefit from the use of such information. A similar constraint arises from the need to resolve references, by name, to shared data structures.

The *Instr_seg* shared segment supports the *Instr_res* resource, which represents the two exported variables *Instr_p* and *Instructions*. Since both the *Planner* and *Joints* processes access the *Instr_p* variable by name, its logical address must be known by the time the process executables are linked. We ensure this by compiling and linking the source file for shared segments containing the declarations of variables referenced by name, before linking the process executable files. The logical base address assigned to each shared segment is used by the linker to determine the logical addresses of the data structure contained within the segment, which are recorded in the symbol table of the shared segment's executable file. When the linker is called to produce a process's executable file, the names of the executable files for each shared segment the process uses are specified, thus giving the linker access to the symbol tables required to resolve references to shared data structures.

The source file for the *Instr_seg* is thus compiled first, and an executable produced which gives the logical address of *Instr_p* in the address space of every process attaching *Instr_seg*. The segment's *Matching* attribute ensures this, by telling the system

that the segment *must* be attached to each process address space at the logical address specified in the segment description. When the *Instr_seg* executable is available, the executables for the *Planner* and *Joints* processes can be linked. The *Reflex* process gains access to the robot control registers supported by the *Robot_cntl_regs* segment using conventional techniques. No executable is produced for the *Robot_cntl_regs* segment, and the compilation and linking of *Reflex* is thus constrained only by the need to ensure that behavioral predictions of procedures are available when procedures calling them are compiled.

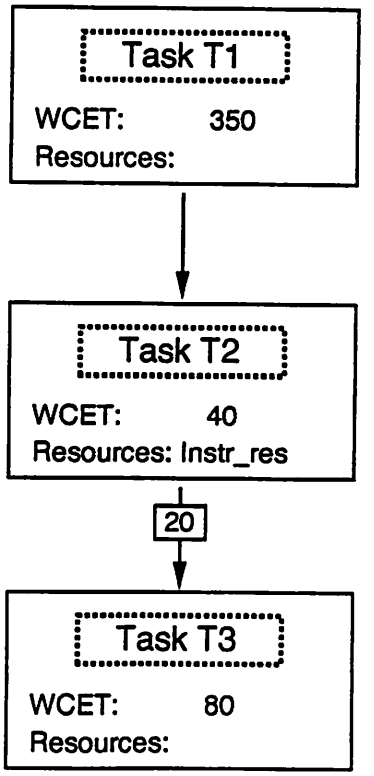
In the course of compilation, as detailed in Chapters 5, 6, and 7, a task group is constructed which represents the worst case behavior of each process to the scheduler. The SDL description of each task group is included in the executable file produced, which is where the system finds the representation during process activation. Let us assume that the *Planner* process source contains a substantial preamble within which the plan is formulated, a resource use block where the plan is posted by copying it into the *Instructions* shared variable, and a small amount of postprocessing. Let us also assume that the semantics of the planner require a delay between posting the plan into *Instructions* and the postprocessing. The process thus has three execution episodes, preamble, plan posting, and postprocessing.

Figure 4.24(a) gives the SDL task group description produced for the *Planner* process, while Figure 4.24(b) illustrates the group structure. It is a simple linear group, with each task giving the execution times of the corresponding execution episodes of the process. The first task represents the preamble during which the plan is formulated. The second task represents the resource use block which posts the plan into *Instr_seg*, and the task thus uses the resource containing *Instr_seg* in exclusive mode. The third task represents the postprocessing, which requires no

resources. The tasks are nonpreemptive, and any deadlines within the group are left for specification by scheduler specific processing when the process is activated. Note the delay associated with the precedence constraint between *T2* and *T3*, which implements the delay required between posting the plan and postprocessing.

```
Task_group(Planner) {
  Group_list {
    Begin: T1;
    T1:    T2;
    T2:    (T3 20);
  };
  Task(T1) {
    Non_preemptive;
    W_time   350;
  };
  Task(T2) {
    Non_preemptive;
    Resources (Instr_res Exclusive);
    W_time   40;
  };
  Task(T3) {
    Non_preemptive;
    W_time   80;
  };
};
```

(a) Task Group Description



(b) Task Group Structure

Figure 4.24. Task Group Example

The software architecture described must, eventually, be loaded onto a Spring node and run. For this example we assumed the target node structure described in Figure 4.25. The node is given the name *Robot_controller*, and contains five boards; three processor and two memory boards. The *SP* processor board, used to support system functions, has $4Mb$ of memory accessible across the system bus at the physical address $0x1000000$. The two application processor boards each have $4Mb$ of memory

```
Node(Robot_controller) {
  Processor(SP) {
    Memory_area(Data, 0x1000000, 0x400000);
    Use   Sys;
  };
  Processor(AP_1) {
    Memory_area(Data, 0x1400000, 0x400000);
    Use   Appl;
  };
  Processor(AP_2) {
    Memory_area(Data, 0x1800000, 0x400000);
    Use   Appl;
  };
  Mem_board(Scramnet) {
    Memory_area(Data, 0x14000000, 0x200000);
  };
  Mem_board(Control_board) {
    Memory_area(Memory_mapped, 0xc00000, 0x200);
  };
};
```

Figure 4.25. Node Structure SDL Example

also accessible from the system bus at the addresses specified. The *Scramnet* board is described as a memory board $2Mb$ in size capable of holding data, and visible at the specified bus address. The robot control registers are defined by the *Control_board* memory board description, and occupy the $0x200$ bytes of memory at the specified address.

The descriptions of all the software, processes, resources, and segments, have been specified. The structure of the target node has been described. The only step left is to describe how the software elements are loaded into the target hardware, as illustrated in Figure 4.26, and described by the SDL in Figure 4.27. The layout specifies that the SP processor will be loaded with the executable for the process required to boot it, as well as the process activation process (Pap) and the scheduler. The *AP_1* application processor supports the *Planner*, while *AP_2* supports the *Joints* and *Reflex* processes

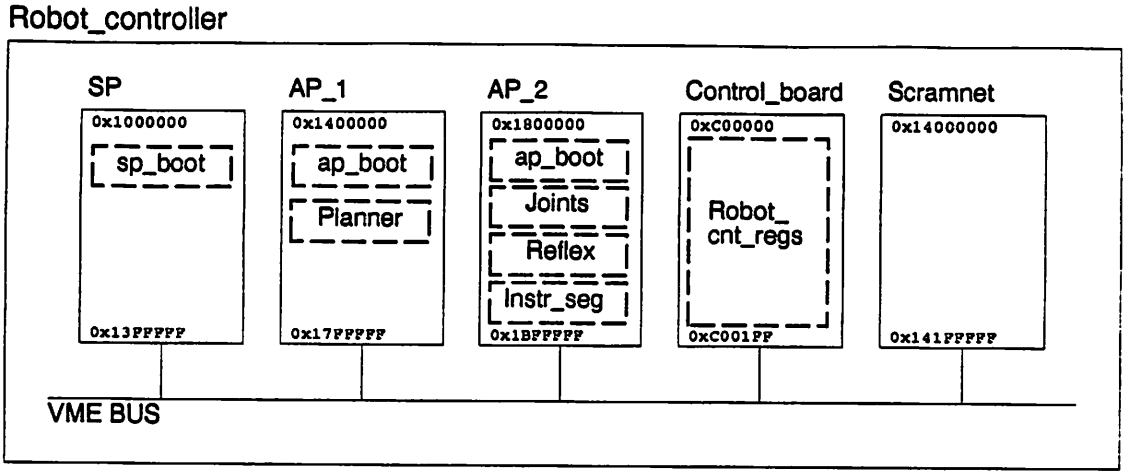


Figure 4.26. Target Hardware Loaded

as well as the shared segment *Instr_seg* and the resource *Instr_res* it contains. Each of the application processes boots using the same executable image. The memory board *Control_board* obviously supports the shared segment describing the robot control and status registers.

This example has shown how processes, shared segments, and resources are described, and their arrangement on the target hardware specified. The important point should be clear; that the SDL provides a way to precisely describe the information required to load and run Spring applications. The fact that specifying this information requires a great deal of space is inconvenient when giving examples, but is a direct result of the level of detail being represented. This level of detail is not frivolous, but is that required to support the specification, loading, and running of real-time programs in a predictable manner. It is also important to emphasize that the SDL is designed to support the derivation of much of the descriptive information by software, rather than forcing the programmer to specify all of it. The compiled form of the SDL can be used as input by tools which derive some of the information required by

```
Layout {
  Node_layout(Robot_controller) {
    Processor_layout(SP) {
      System_set {
        Boot_proc      sp_boot;
        Process_set    Pap, Scheduler;
      };
    };
    Processor_layout(AP_1) {
      Appl_set { Process_set Planner; };
      System_set { Boot_proc  ap_boot; };
    };
    Processor_layout(AP_2) {
      Appl_set {
        Process_set    Joints, Reflex;
        Resource_set   Instr_res;
        Shared_seg_set Instr_seg;
      };
      System_set { Boot_proc  ap_boot; };
    };
    Mem_board_layout(Control_board) {
      Appl_set { Shared_seg_set Robot_cntl_regs; };
    };
  };
};
```

Figure 4.27. Layout Example

the system from the set of descriptive statements specified in the source code by the programmer, while the derived information can be output in compiled SDL form for use by tools performing further analysis or for use by the system at run time. The design of the SDL thus supports the gradual refinement and automation of the system since tools or methods for deriving specific pieces of the information represented by the SDL can be developed independently.

4.2.8 The SDL Interface Library

The SDL supports information exchange among various parts of the Spring development and execution environments, as discussed in Section 3.1, by providing a standard interface with which all interested tools can access, modify, and add to the compiled SDL information describing a Spring application. The SDL interface library routines provide the standard interface used by such tools.

The most important interface routines support reading and writing files containing SDL information in compiled form. The *read_db* routine copies the compiled SDL in a file into data structures in the working memory of the program, thus making it possible for the program to use, modify, or add to it. The *write_db* routine copies the memory based data structures into the specified file, thus preserving any alterations made. Other routines help programs using the interface to manipulate the SDL sections of Spring executable files, merge sets of SDL descriptive information read from separate files, search for an item of a particular name and type, add and delete items, and free the memory allocated when a file containing SDL information is read. The complete set of interface routines is discussed in detail elsewhere [58].

4.3 Evaluation

The purpose of Spring-C is to provide a way for a developer to describe real-time computations, and to ensure that all the information required by the system to compile, load, and execute application and system programs under Spring is available to every part of the system when needed. Evaluation of Spring-C must address how well it performs this role, which means evaluating how well it performs in the context of current system development, as well as using it to implement and execute application programs. Our assessment has an objective and subjective component. We

can objectively say whether Spring-C, particularly its SDL component, represents the information required for the system to function properly by *using* it and noting problems or shortcomings that use reveals. We subjectively evaluate how well it supports our activities by the ease with which users can express the descriptive information required, and how effectively it presents the descriptive information for inspection when a developer is debugging an application.

The information flow discussed in Section 3.1, and illustrated in Figure 3.3, has been fully implemented. The appropriate portions of the SDL information describing application and system software are used by the compiler, linker, associated inspection and analysis tools, the debugger, and the Spring operating system in the normal course of their execution. Ongoing development will continue to increase the degree to which the SDL integrates and coordinates operations at all levels of the system.

There are several current application and system level development efforts. Some are more demanding than others, but all are reasonable uses of the programming environment provided by Spring-C, and thus present opportunities to evaluate it. We will discuss how four sets of software use Spring-C, and what the experience of the developers involved has revealed of its properties. The first is its role in supporting test cases created as part of system development, and validates the basic information flow in the SGS, as well as *sbug's* use of the information for downloading and debugging. The second application is an implementation of the example given in Section 4.2.7, which demonstrates Spring-C's ability to describe processes, shared segments, and resources, and to successfully run the computations described.

The third is a robotics application adapted from an existing automated robotic manufacturing cell, and provides a much more demanding use of the SDL's ability

to describe computations, shared segments, resources, and synchronous communication. It demonstrates that the SDL is capable of supporting realistic applications. The fourth example is another automated manufacturing application, and is part of current research. The application is similar to the previous one in several ways, but exhibits more sophisticated demands, and has already suggested several ways in which to expand and elaborate Spring-C's current features. The rest of this section discusses each of these evaluations in turn.

4.3.1 Development Test Cases

The simplest evaluations of Spring-C are performed by the test cases associated with development of the compiler, debugger, scheduler, and kernel. Spring-C has been easy to use for these problems, and its application straightforward. The test programs used to evaluate the compiler's behavioral prediction accuracy are a good example of the kind of demands made on Spring-C by these tests. Each performance test is a single process, which attaches a shared segment to make the system clock's control and value registers available in the user address space. Library routines support the collection, accumulation, and display of execution time measurements. Since each test only uses a single processor board, multiple tests are often run on the node's AP processors in parallel. The SDL layout specification is used to specify the set of test programs using each AP board within the target node.

The debugger *sbug* reads this and other SDL information from the *full.db* file accumulated during test case compilation, uses it to control loading the system and application executables into the target node, and then begins execution. One indication that the SDL represents all the required information is that it was possible to fully automate running various sets of test cases with a few simple C-shell scripts,

within which the *sbug* command lines need only specify the target node name, the appropriate *full.db* file, and the name of the file within which to accumulate the test results. The SDL information available to the debugger is sufficient for it to support debugging any or all of the application and system code. Other test cases are implemented using more than one process sharing resources and engaging in synchronous communication. These test the SDL's ability to describe more complex structures, as well as the system's ability to consider the structures represented efficiently at run-time.

4.3.2 Example Implementation

The second evaluation involved implementation of the example described in Section 4.2.7. An interesting aspect of this test of Spring-C was that it was done by a new student as a way of introducing him to the system, as well as seeing how a novice reacted to the unique aspects of the Spring environment. He was able to use the SDL already given, adding Spring-C code to perform or simulate various portions of each computation. Using the SDL description already given was straightforward, but his effort to understand how the Spring environment, and thus programming practices, differed from that of conventional systems was instructive.

His main sources of confusion seemed to be the existence of shared segments as independent objects, the difference between a shared segment and a resource, and difficulty in deciding the best way to think about the implementation of the desired computation in terms of processes that will be translated into a set of tasks. Shared segments can exist as independent objects for several reasons, most notably the need to access the data structures they support by name, but also as an aid in designing software architectures for embedded systems. A good example of the latter is a shared

segment giving access to a device's control and status registers. It exists within the system independently, regardless of whether it is attached to the address space of a process or not. Supporting such independent shared segments enables the system to decouple use of such memory mapped control registers from their location in the physical address space, which increases software portability and lowers maintenance costs resulting from hardware configuration changes.

A shared segment supporting data structures that are accessed by name has a source file containing the data structure definitions, as well as the SDL description of the segment. Recall, as discussed in Section 4.2.3, that such a segment will also have the *Matching* attribute to ensure that it appears at the same place within every address space to which it is attached. The source file is compiled and linked, producing an executable whose symbol table gives the logical address at which each data structure will appear in the address space of a process attaching the segment. The symbol tables of these executables are used by the loader to resolve the references to the shared data structures when linking processes accessing them by name. This is reflected in the SGS information flow as presented in Section 3.1 and is an unusual feature of Spring, although similar techniques are being considered for other reasons in systems specifically designed for 64-bit architectures [18].

The difference between a shared segment and a resource was discussed in Sections 4.2.3 and 4.2.2. Shared segments represent objects within the memory space that are accessed by processes. Resources are abstractions representing constraints on concurrent execution of processes. They generally represent sets of shared data structures, but can also represent constraints that arise from application semantics, without involving the use of a physical part of the system. The *with* statement in Spring-C, indicating resource use by a code block, is generally represented by a task

in the group representing a process, thus enabling the scheduler to take the constraint on concurrency represented by the resource use into account. Once these ideas became more familiar, it was easier for the student to see why the shared segments and resource were defined in a particular way.

The need for a period of adjustment to new programming practices touches on the last major adjustment required by the student to work with Spring-C. He found that he had to be careful to *begin* thinking about the software architecture in terms of the *computations* he wanted to accomplish, and their implementation by one or more processes, deferring any thoughts about the scheduling representation until later. When he had formed a clear idea of the process structure of the computation, including their resource use, explicit delays, and synchronous communication, then it was fairly easy to see how the processes would be represented by a group of tasks for scheduling purposes. When he tried to begin by thinking in terms of how and when specific actions would be scheduled, it was much more difficult to see how the Spring-C source should be structured.

In general this novice's experience with his first Spring-C application was that the language was reasonably well suited to describing real-time applications for execution under the Spring system, but that understanding the subtle differences between designing Spring applications and those for conventional systems required some effort. These differences in programming practice arise primarily from the fact that Spring-C's syntax forces the developer to write programs whose worst case behavior can be predicted, but were also related to Spring's support for shared segments as independent objects. Overall, however, he found that he was able to start thinking in Spring-C terms fairly quickly, and was able to move on to the more ambitious project which provides the, currently in progress, fourth evaluation of Spring-C.

4.3.3 Robotic Assembly Station Application

The third evaluation of Spring-C was also undertaken by a novice, and was modeled after an existing application with which the student had previous experience: an automated manufacturing application, developed at DEC, which controlled a robotic assembly station soldering chips onto circuit boards. The application had been designed to place custom VLSI chips that commercial products were unable to handle, and involved several interesting subsystems, including robot joint and tool control, vision, and higher level control and decision making required to customize the assembly operations for several different types of boards.

The student's Masters project was to port this application to Spring, which raised significant issues at several levels, and illustrated the significant differences between conventional and predictable real-time programming [6]. Since we did not have the robotic equipment involved, the Spring version necessarily simulated portions of the application, most notably the vision and servo control subsystems. However, we did not consider this a flaw, since the goal was to create a realistic test case for Spring, not to actually control a robot. Simulating portions of the application, under real-time constraints, turned out to require a software architecture just as demanding as controlling the actual hardware would have entailed.

Figure 4.28, taken from the student's thesis [6], illustrates the software architecture of the application as ported to Spring. The *assembly manager* process runs on the UNIX development machine, and generates a stream of chip placement operations handled by the assembly computation running under Spring. The assembly computation is implemented as a process group containing the *pick* process which finds the part required by using the vision subsystem to locate it in the appropriate parts bin, and its successor *place* which also uses the vision subsystem to locate the proper board

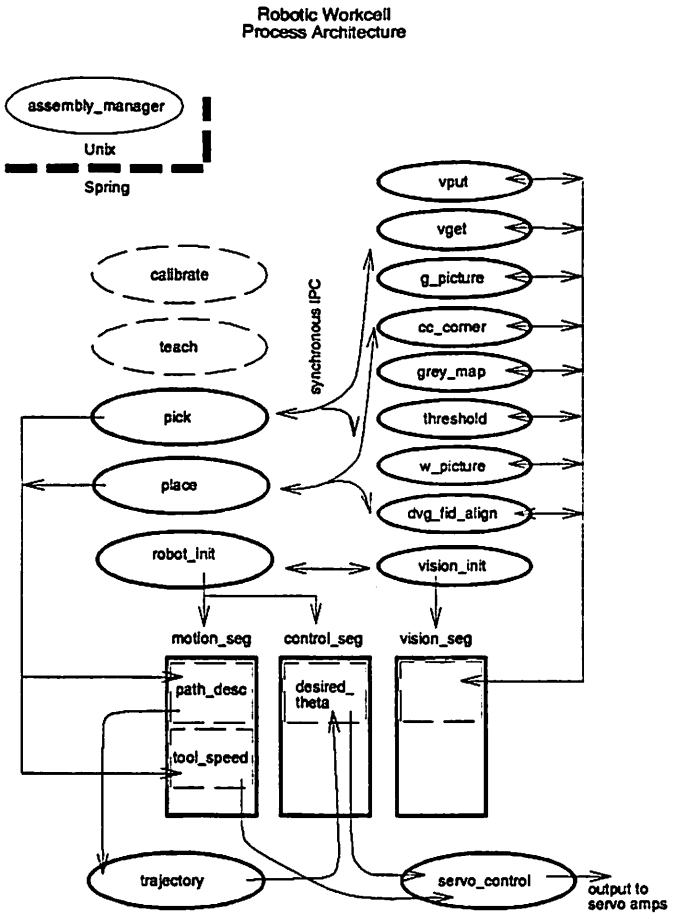


Figure 4.28. Automated Manufacturing Application

position at which to solder the part. The vision subsystem is implemented as a set of server processes which are called in an RPC style, using Spring-C *synchronous* IPC calls to make the request and receive the reply.

The assembly computation is aperiodic, as it is scheduled in response to a chip placement request from the *assembly manager*. The *pick* and *place* processes generate path and tool speed commands, which they post to the appropriate data structures in shared segment *motion_seg*. The *trajectory* process is periodic, regularly checking the path descriptions to see what the next move of the robot should be. It performs inverse kinematic calculations to determine desired joint angles, and posts these to

data structures in the shared segment *control_seg*. The *servo control* process is periodic, regularly checking the joint angle and tool speed specifications, calculating the commands needed to make the robot comply, and issuing them.

The experiences of this student in porting the application, originally written in an interpreted language called V+ and with no notion of real-time or predictability as it is understood in the context of Spring, were varied and interesting. They are discussed at length elsewhere [6], and we present only a synopsis of the most salient points. His experience was consistent with that of the student who implemented the example of Section 4.2.7. The hardest part was learning to think in the right terms, adopting predictability as a design criterion, and applying it faithfully.

Determining specific loop bounds was among the more difficult problems since the original software could afford to be "sloppy", from a Spring point of view, by writing loops with conditional statements that could detect when to terminate, but which made no assertion about how many iterations might be required. This is clearly the most limiting aspect of the syntactic changes required to ensure that Spring programs are finite, as presented in Section 4.1, although the need to have such numeric limits available at compile time should be clear from the discussion of Section 4.1.1 about why Spring-C programs are finite.

These simple forms for expressing the loop bounds can significantly overestimate the actual execution time in some situations. Some of this is an inevitable consequence of having to consider the computation's worst case behavior, but some of the excess is a result of not taking constraints expressed at the level of application semantics into account. The experience of porting this application has, however, pointed out several ways of refining the predictions by using knowledge of application semantics to apply

tighter constraints. At least one of these has been suggested by other investigators [68], while others are extensions of the reduction methods described in Chapter 6.

In other respects, porting the application was reasonably straightforward once the student adopted the proper design perspective. Deciding which data structures needed to be shared, how they should be grouped onto shared segments, and which sets should also be represented by resources since concurrent access to them was constrained were among the first design activities. Decomposing the previously monolithic code for the vision subsystem into a set of server processes came next. The original version was implemented as a single process which provided a number of different services. Service requests specified a service number, which was used to choose and entry from a table of function pointers. This design was obviously inappropriate for Spring, since the worst case behavior of the server process was that of the most costly request. Instead, the various vision services were decomposed into a set of server processes with which the clients communicated using synchronous IPC. This effectively converted each service request into a remote procedure call, and enabled the compiler to consider the worst case behavior of each service separately. When the processes implementing the vision services were established, it was then possible to address the code for the *pick* and *place* processes using those services.

One of the most instructive features of the ported code was the place where it used a loop to implement a series of attempts to position the tool holding the chip over a specific spot on the circuit board. Each attempt makes a RPC to one of the vision server processes to determine if the position of the tool is within specified error limits. This places the suspension points associated with the RPC within the loop, and thus requires a set of tasks to represent each iteration of the loop. This type of situation is handled by the prediction methods described in Chapter 6, but

the example stresses that a large number of tasks can be required to represent the behavior of some processes. However, the number of tasks required can be reduced in a number of ways, which will be addressed as Spring-C is developed further. Other aspects of the project suggested some minor refinements in Spring-C, and revealed a few rough spots in the SGS information flow, but no significant problems were encountered.

In summary, the third evaluation of Spring-C was consistent with those of the other tests. Spring-C is reasonably well suited to describing application programs that will run on the Spring system, but the programming methods and design perspective of the developer must be modified to fit the unique aspects of real-time systems. The experience of two novices has shown that it is possible to do so, but that it requires a non-trivial period of effort and adjustment.

4.3.4 Automated Manufacturing Application

The fourth evaluation of Spring-C addresses a more sophisticated automated manufacturing application [94]. It involves a series of scenarios of increasing complexity, requiring a corresponding increase in the sophistication of the software controlling the manufacturing cell. The problems involve variations of the following basic situation. A conveyor carries a varied series of parts past a gate that can be opened to transfer a particular part from the conveyor to a table top. The opening and closing operations must be scheduled to ensure the transfer of the part desired. A robot is responsible for transferring parts from the table top to one of several destination bins. A detector views the conveyor and identifies each part some fixed time prior to when it passes the gate, and is responsible for scheduling the computation which opens and closes the gate, and then uses the robot to transfer the part to the proper bin.

Requests for delivery of various types of parts to specific destination bins arrive at the system at random times. The system checks each part seen by the detector against outstanding requests. When a part that satisfies one of the outstanding requests is seen, the system tries to schedule the process(es) controlling the gate to transfer the part to the table top, and to schedule the process(es) controlling the robot that transfers the parts from the table top to the destination bin. Timing constraints arise from several sources. The requests for part delivery may or may not have deadlines, and those deadlines may be hard or soft. The speed of the conveyor belt determines the amount of time available between identifying a requested part and execution of the process opening the gate. Other constraints arise from the order in which various part types appear on the conveyor, and from the properties of the robot which must find a part of a specific type on the table and transfer it to the proper bin.

While the project is still in its early stages, it appears that the general approach of Spring-C is adequate, but that some aspects of the application would require extensions to some features of the language. One such extension is a more flexible way to acquire and release resources, whose use will span task boundaries. Such a change in resource use semantics will also require changes to the behavior prediction and scheduling methods, but none of the modifications are difficult. Another extension that seems desirable is a *delay_until* statement that would ensure that a portion of a computation would not begin before a specific time, relative to the arrival time of the computation. This extension would require changes to Spring-C and SDL syntax, modifications in the behavior prediction phase to track the new information, and changes to the scheduler to work with the new type of constraint.

4.3.5 Summary

While these examples clearly show that Spring-C does not yet have all the features required for every application, they also demonstrate what may be the most important aspect of Spring-C's current design and implementation, its flexibility and support for further development. Spring-C, and particularly the SDL, is designed to be easily extensible so it can and *should* evolve as system implementation continues and new types of application programs are written which present new types of demands on the system.

However, the work already finished has demonstrated Spring-C's ability to fulfill its assigned role. It accumulates descriptive information contained in application source files, and that derived during compilation. It also makes that information available, through a standard interface library, to any program wishing to use, modify, or add to it. Its support for universal access to the descriptive information it represents plays a vital role in achieving the level of integrated design and implementation that the Spring system requires [82, 83].

CHAPTER 5

PROGRAM TRANSLATION

The goal of the compiler and related tools when translating from Spring-C source to executable form is to produce an executable file for each process, containing an SDL task group description of the process's behavior that correctly represents every path which the process can take through its executable code. Another way of saying this is that when the tasks in a group representing a process's behavior are scheduled according to their WCET, resource use, and execution precedence constraints, then every possible execution path through the process's code will be properly supported by the system.

The essence of our translation method is to predict the worst case episodic execution behavior of a process, and then represent each episode as a task for the purposes of scheduling its execution. A schedule that satisfies every task's resource, execution time, and execution precedence constraints will thus satisfy the needs of each execution episode, and thus the needs of the process. The translation method described in this chapter, and in Chapters 6 and 7, addresses the problem of how to construct such a task group representation in the course of program compilation.

It is important to note that the task group representation only accounts for *specific categories* of program behavior. Specifically, it represents how many times the program can suspend its execution, the WCET of each interval of execution between suspension points, what resources each execution episode may require, the minimum

delay required between execution episodes, and when synchronous communication through particular virtual circuit ports, may occur. We call these the *target behaviors*, to distinguish them from other behaviors that are not considered in the compiler's analysis. We do *not*, for example, consider behaviors resulting from code that is logically incorrect, although in the course of the analysis we *can* detect certain classes of logical errors.

The syntactic features added to conventional C to produce Spring-C, as discussed in Chapter 4, were designed to explicitly represent aspects of the source code affecting the target behaviors. The information they provide makes it possible to analyze a process's target behaviors during compilation, producing the behavioral predictions required to build the task group representation of the process's target behaviors which is used by the scheduler.

Other approaches to representing process behavior are possible, with various advantages and disadvantages in expressiveness, ease of implementation, and aesthetic appeal. We could, for example, assume that every process uses every resource it requires for its entire execution. This would enable us to always represent computations as single tasks, which simplifies scheduling. However, it also holds resources for longer than necessary, and thus tends to reduce schedulability by increasing demand for resources.

Another possible approach is to represent computations using more than one task, but require the developer to do the decomposition. Source code for each task would be written separately, and the information exchange between each task would be explicitly programmed either by sending explicit messages, or through shared memory. This is essentially the approach taken by the MARS system[15]. This tends to increase schedulability by decreasing the granularity with which the system allocates

resources, but places a greater burden on the programmer. Further, the developer must either decompose, by hand, the application *at every possible suspension point*, or the blocking time for suspension points within tasks will also have to be considered. This is, however, not always a problem. When, for example, the blocking time is smaller than the system and scheduling overhead of context switching between tasks, then simply adding the blocking time into the task execution time may be preferable to explicitly suspending the process. The main disadvantage of this method is that it requires the programmer to do a lot of the work. However, many applications could clearly be written using this method, without undo difficulty.

Our choice is to represent computations as sets of communicating processes, which has the advantage of being familiar. The compiler, by considering every place in the process code where it may suspend, can construct a representation of the process's behavior using a task for every episode. This is similar to the method depending on decomposition by the developer, since a computation is still decomposed into a set of processes, but is less labor intensive, since decomposition need not be done at every suspension point. Also, automatic code *transformation* ensures that every execution path is consistent with the task group representation, and makes various optimizations possible. For example, we might choose to optimize the total execution time of a computation (ignoring delays), the total elapsed time of a computation (execution plus delay), the utilization of a particular resource or set of resources, or a metric combining two or more of these factors. As a practical matter, it also made implementation more convenient, since the gross structure of application programs remains the same, and thus the Software Generation System (SGS) tools with which we began development required less ambitious modification than might otherwise have been the case.

Regardless of the details of the decomposition, however, combining behavioral prediction with compilation has the advantage of letting us analyze the actual assembler emitted, thus reflecting any optimizations performed by the compiler, while preserving the ability to transform the program as required to improve its behaviors. The translation method described here compromises between the limitations of working only with the program source[65], and those of working only with the compiler's assembler output[1, 29], by using the compiler's intermediate representation. We use the GNU C compiler as the base for our work, which uses register transfer language (RTL) as its intermediate representation of the program[81]. Implementing the translation method inside the compiler and working with the RTL representation enables the analysis to work with the assembler code *actually emitted* by the compiler, while preserving the ability to transform the program. This is similar, in its broad outlines, to the approach taken by the MARS project to its compiler[68].

This chapter begins the explanation of how Spring's SGS integrates conventional aspects of program compilation and the behavioral prediction required to build a task group representation of a process's worst case behavior. The source language Spring-C, including the syntactic features supplying information required for the behavior prediction and guaranteeing a finite program whose worst case behavior can be predicted, was described in Chapter 4.

Our translation method uses an enhanced version of the RTL, described in Section 5.1, which represents the block structure of the source program, records the loop iteration and recursion depth bounds, notes resource use, and marks the suspension points resulting from explicit *delay* statements or synchronous communication. The enhancements to the RTL thus record the information made available by Spring-C's new syntactic features. The translation method starts with the enhanced RTL of each

procedure just prior to assembler code emission, and has has three major phases. We briefly present an overview of each, but they will be discussed in detail later in this and in subsequent chapters. The three translation phases are:

- Phase 1: Time Graph Construction
- Phase 2: Subgraph Reduction
- Phase 3: Task Group Construction

Phase 1, discussed in Section 5.2, constructs a new graph that is isomorphic to the RTL representation's basic block graph, called a *time graph* (TG). Nodes in the RTL that represent the procedure's block structure, resource use, or suspension points appear unchanged in the TG. A set of RTL nodes that represents a sequence of machine instructions is represented in the TG by a node giving the WCET of the corresponding instructions, and is thus called a *time node.*

Phase 2 reduces the size of the TG by performing *subgraph reductions*, as discussed in Chapter 6. The reductions replace sections of the original TG with simpler graphs which preserve the information required to predict the target behaviors. This involves retaining the WCET and resource use of each section of the procedure lying between suspension points, and accumulating the properties of the suspension points which are combined as the TG is reduced. The reductions proceed until the TG is reduced to its simplest form. This will be a linear TG, consisting of alternating time nodes and suspension points, which represents the procedure's target behaviors. The minimum size linear TG is called the irreducible time graph (ITG). Some subgraph reductions require transformations of the procedure, which involves modifications of the original RTL.

Phase 3, discussed in Chapter 7, analyzes the ITG and constructs a task group representation of each procedure's target behaviors. Phase 2 accumulated information about the WCET of each execution episode and the type of suspension points creating boundaries between execution episodes. Some analysis is required to construct a task group which properly represents the target behaviors specified by the ITG, but the ITG is generally simple enough that the analysis required is quite simple.

The rest of this chapter describes our enhancements to the compiler's intermediate representation (RTL) in Section 5.1, while Section 5.2 describes the construction of the *time graph*. Chapter 6 describes subgraph reduction while Chapter 7 discusses the construction of the task group representation of a computation's worst case behavior.

5.1 Enhanced RTL

The enhancements to the compiler's intermediate representation are required to provide the time graph (TG) construction, subgraph reduction, and task group construction phases of the compiler with the information they require to perform properly. The information represented by the new RTL features falls into two major categories: structural information and process suspension points. Structural nodes delineate the sequences of RTL nodes which correspond to each section of the original Spring-C source code's block structure. For example, structural nodes delineate the RTL sequences corresponding to the *if* statement's conditional expression, true, and false clauses. Suspension point nodes in the RTL correspond to those places in the source where the execution of the program either must or may be suspended.

Several types of structural nodes exist, corresponding to the various components of Spring-C source program blocks. It is important to note, however, that the structural and suspension point nodes also record important information about the target

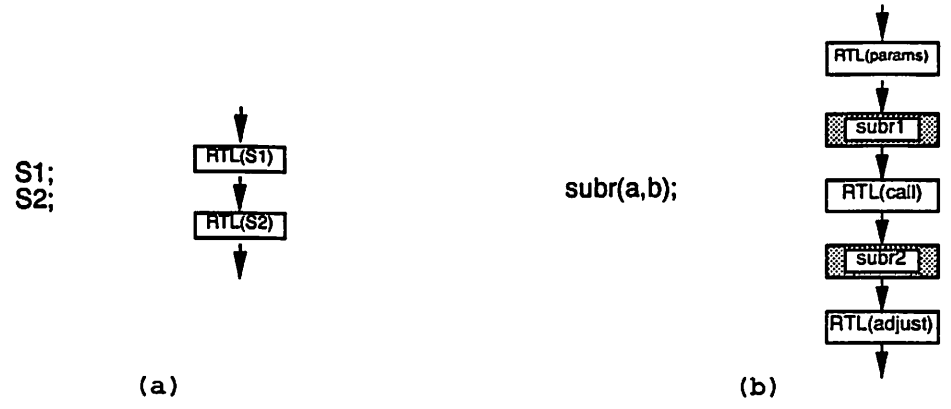


Figure 5.1. Enhanced RTL for Sequential Code and Subroutine Calls

behaviors being predicted. For example, one of the structural nodes used to record the structure of Spring-C loops is also used to record the iteration bounds. Similarly, the suspension point nodes record information appropriate to their type, delay nodes record the minimum delay, nodes associated with resource use record the resource name and its mode of use, and synchronous communication nodes note the virtual circuit port used and whether the message is being sent or received. The rest of this section will introduce the new node types as the enhanced RTL representations of each of the Spring-C statements are discussed in turn.

5.1.1 Sequential Code and Suspension Points

Figure 5.1a illustrates that the RTL emitted for sequential Spring-C assignment statements is the same as that emitted for the conventional C language. Each source statement *S1* and *S2* is translated into a linear sequence of RTL nodes. Figure 5.1b illustrates the structural nodes associated with calling a procedure. When compiling a conventional program, the compiler first produces the RTL to push any parameters on the stack, then the RTL for the procedure call itself, and finally the RTL for any instructions required to adjust the stack in compensation for any parameters that were pushed on it. In Spring-C, we have to be careful to note where the subroutine

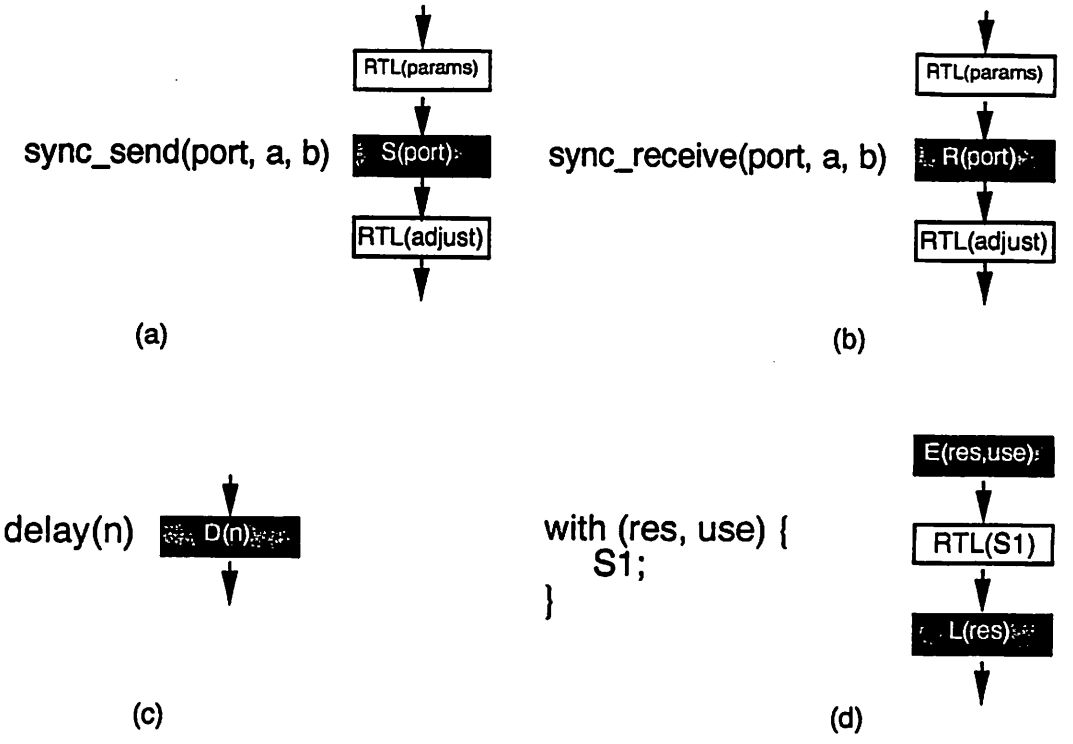


Figure 5.2. Enhanced RTL for Scheduling Points

call happens, and use the structural nodes *subr1* and *subr2* to bracket the call. The reason for this is that during the subgraph reduction phase, the ITG of the called procedure will be substituted for the nodes representing the call, thus taking the behavior of the called procedure into account. This is discussed in greater detail in Chapter 6.

Figure 5.2 illustrates the various suspension points which note where the process can either explicitly or implicitly suspend its execution. Figures 5.2a and 5.2b show the RTL generated by the synchronous communication statements in Spring-C. Each black node in the figure actually represents a pair of nodes surrounding the instruction calling the synchronous communication system call, as was the case for normal procedure calls. The RTL handling the preparation of the parameters and pushing them on the stack precedes the suspension node, and the code doing any necessary stack adjustment follows it. The suspension points note whether the program is sending

(S) or receiving (R), and on what *virtual circuit port.* A program can also explicitly suspend its execution by using the Spring-C *delay* statement. Figure 5.2c illustrates the RTL generated for the *delay* statement. The suspension point actually represents a pair of nodes surrounding a call to the *suspend* system call. Note that the delay suspension point (D) records the value n of the delay statement's argument.

The resource use statement *with* represents an implicit, or at least potential, suspension of the process. The use of a resource by a process is often represented by a separate task, but *may* be represented as a portion of a task's execution. In either case, any attempt to analyze the program must know where a block of code using the resource is entered (E), where it is left (L), and in what mode the resource is used. This is precisely the information represented by the *with* statement in Spring-C, which is translated into the RTL illustrated in Figure 5.2d. Each suspension point actually represents a pair of nodes surrounding a call to the *suspend* system call, as with *delay.*

5.1.2 Conditionals and Switch Statements

Figure 5.3a illustrates the RTL template used for a simple *if-then* statement, while Figure 5.3b illustrates the template for the *if-then-else.* The structural nodes delimit the bounds of the conditional statement, as well as marking the end of the conditional expression where the choice between the true and false clauses is made. The dotted arrow linking the structural nodes marking the beginning and end of the conditional statement's RTL graph emphasizes that the structural nodes contain references to one another which are useful during TG construction and subsequent reduction. Figure 5.4 illustrates that the enhanced RTL is well suited to representing the nested block structure of the Spring-C source program.

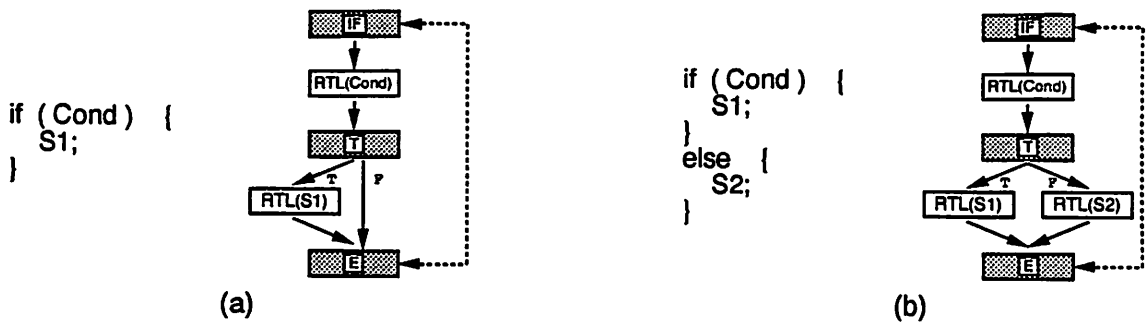


Figure 5.3. Enhanced RTL for Conditional Statements

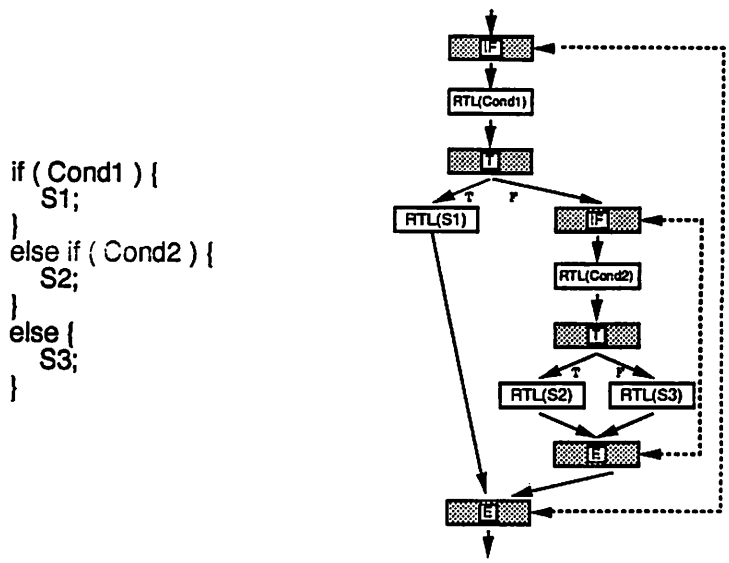


Figure 5.4. Enhanced RTL for Nested Conditional Statements

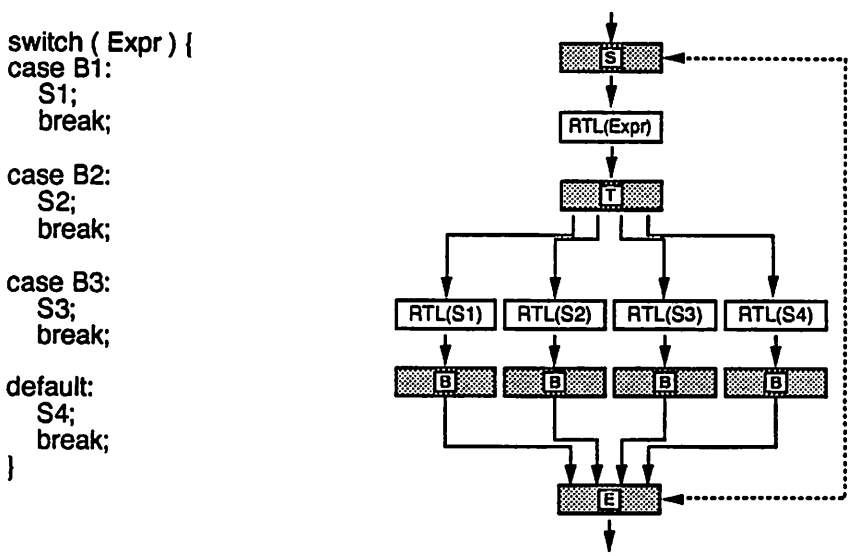


Figure 5.5. Enhanced RTL for Switch Statement

```
switch ( Expr ) {
case B1:
   S1;
   break;

case B2:
   S2;
/* fall through */

case B3:
   S3;
   break;

default:
   S4;
   break;
}
```

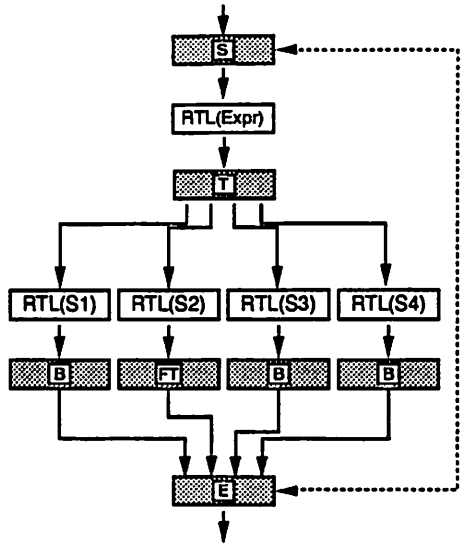


Figure 5.6. Enhanced RTL for Switch with Fall Through

Figure 5.5 illustrates the RTL structure used to represent the *switch* statement. The expression *Expr*, which calculates the value determining the portion of the *switch* to which control is transferred, is delimited by the S and T nodes, while the end of each switch case is usually delimited by the B node representing the *break* statement. Occasionally, one of the statement's cases will not be terminated with a *break* statement, but instead falls through into the subsequent case. This is easily represented, as illustrated in Figure 5.6. The *FT* structural node represents the fact that the case executing *S2* falls into that executing *S3*. Note that within the RTL data structures the *FT* node is actually linked to the end node of the *switch* block. This is done to make the structure of the RTL graphs as regular as possible, and so that the execution time for each case can be considered separately during subgraph reduction.

5.1.3 Loops

The basic issue for Spring-C loops is knowing and enforcing the iteration bounds. As discussed in Chapter 4, the syntax for Spring-C loops requires expressions giving

upper and lower iteration bounds which can be *evaluated* at compile time. However, some means of enforcing these bounds at run-time, and notifying the system if the program tries to violate them, is required. The validity of the task group representation of a process's behavior depends on the validity of the WCET predictions, which depend, in turn, upon the validity of the iteration bounds. Any violation of those bounds thus, at least potentially, invalidates its task group representation, and requires that the system be given a chance to consider what it should do in response.

```
while (condition)(lb, ub) {

    loop-body;
}
```

(a)

```
{
    int lc = 0;
    while(condition) {
        if(++lc>ub) {
            break;
        }
        loop-body;
    }
    if ( (lc < lb) || (lc > ub) ) {
        bound_violation(lc);
    }
}
```

(b)

Figure 5.7. Iteration Bound Enforcement

Figure 5.7a illustrates a Spring-C *while* statement, while Figure 5.7b gives its conventional C equivalent. The variable *lc* is local to the loop, and is used to count loop iterations as they occur. The conditional statement incrementing *lc* and comparing it to the upper bound ensures that the body of the loop cannot be executed more times than the upper bound dictates. When the loop is exited, the value of *lc* is tested against the iteration bounds, and the *bound_violation* system call is made if either the upper or lower bound was violated.

The *bound_violation* routine records the value of *lc* and the program counter value, so the offending loop can be identified. The system records this information and aborts the process. Figure 5.7b illustrates the logic of what is done, but the Spring-C compiler implementation does not actually change the Spring-C source, and then compile it. Instead, the Spring-C compiler's interpretation of the *while* statement generates an RTL graph equivalent to Figure 5.7b.

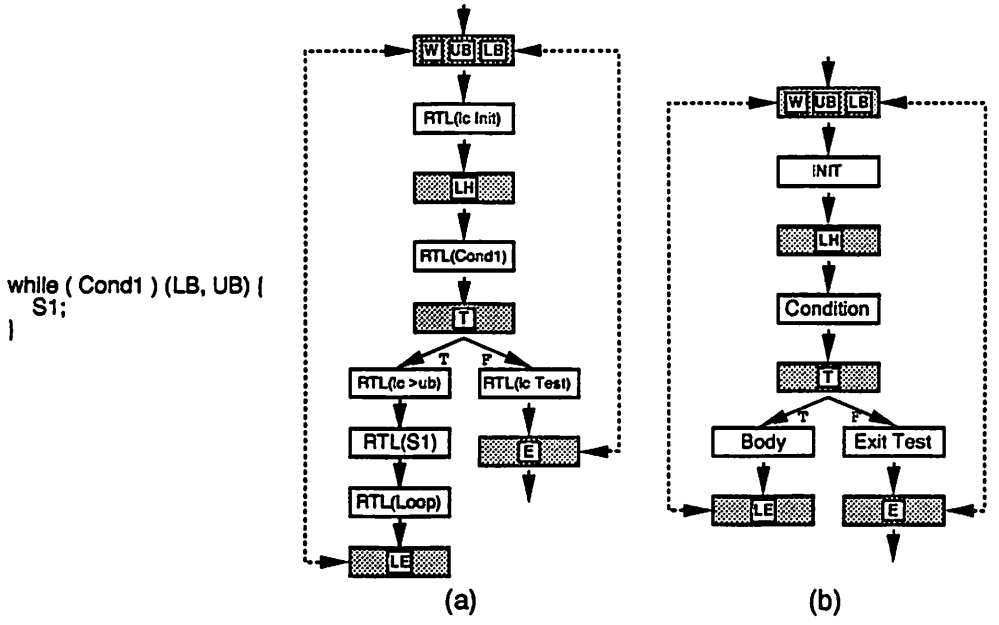


Figure 5.8. Enhanced RTL for While Loop

Figure 5.8a illustrates the RTL structures used to represent the *while* loop, which corresponds to the logical structure of Figure 5.7b. The *W* structural node marks the beginning of the RTL subgraph representing the *while* statement, which holds the lower *LB* and upper *UB* bounds on iteration. The RTL statements required to create and initialize the loop counting variable *lc* lie between the *W* and loop header (*LH*) nodes, which marks the beginning of the code executed during each iteration.

The RTL required to described the loop's conditional expression, lies between the *LH* and *T* nodes. Either the *true* path into the body of the loop, or the *false* path

exiting the loop will be taken. The loop body lies between the T and loop end (LE) nodes, consisting of the increment and test of *lc* against *UB*, the RTL for the body, and then the branch back to *LH*. The RTL for the loop exit code, lying between the T and E nodes, checks *lc* against the upper and lower bounds, calling *bound_violation* to inform the system of any violation. Figure 5.8b gives the more compact illustration of the *while* loop that is used in subsequent discussions.

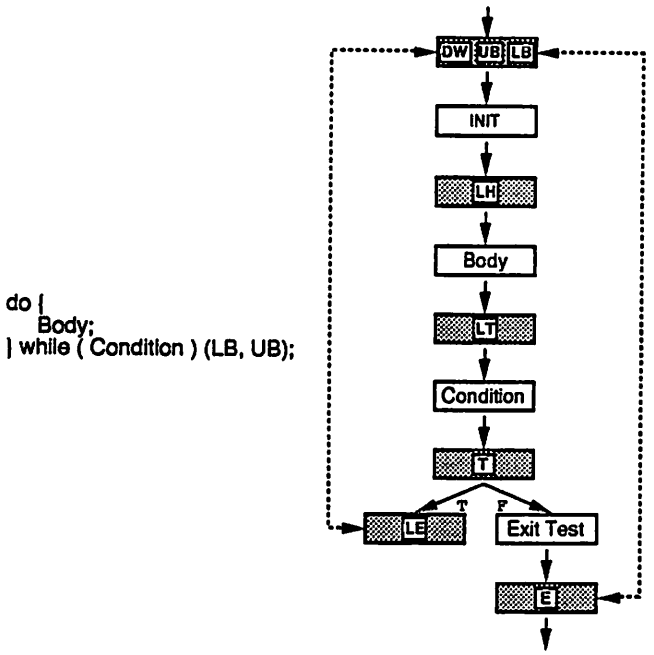


Figure 5.9. Enhanced RTL for Do-While Loop

Figure 5.9 illustrates the RTL structure representing the *do-while* loop. The initialization of the loop counter is done in the *INIT* block, but this time *lc* is initialized to one, to reflect the loop's semantics of executing the body at least once. The *LH* structural node marks the beginning of the body of the loop, which includes the same test of *lc* against the upper iteration bound as the *while* loop had, while the *LT* node marks the beginning of the conditional expression. This is important when considering the effect of a *continue* statement on the execution time of the loop, since it transfers control to the beginning of the condition test from anywhere in the loop

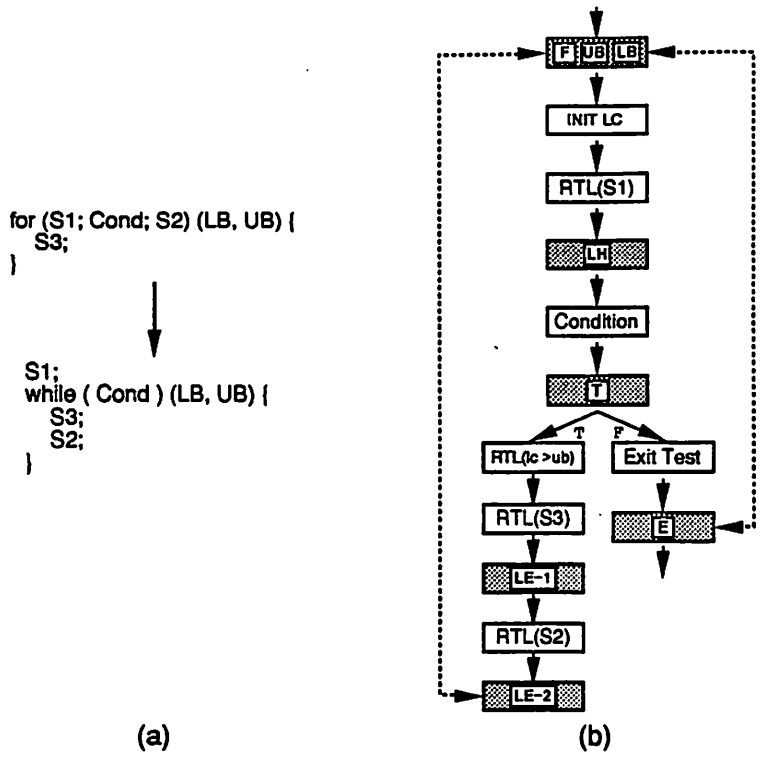


Figure 5.10. Enhanced RTL for For Loop

body, creating an alternative path through the loop body. The *Condition* node in the diagram represents the RTL of the loop's conditional expression, as was true of the *while* loop. As with the *while* loop, *lc* is tested against the lower and upper bounds by the RTL for the exit test lying between the *T* and *E* nodes.

The *for* loop representation is slightly more elaborate than that of the *while* loop. Figure 5.10a illustrates this by showing the Spring-C code for the *for* loop and the (almost) equivalent *while*. The *while* is not exactly equivalent to the *for* since a *continue* statement in *S3* would execute *S2* in the *for* but not in the *while*. Figure 5.10b shows how the statements specified within the *for* loop framework are supported by the RTL. The *LE-1* and *LE-2* nodes mark the boundaries of the loop body and the expression *S2* executed at the end of each iteration. This is important later, when we consider programs containing *continue* statements, and is discussed in Section 6.2.

This completes the discussion of the RTL structures produced for Spring-C loops, but it is important to note how the *break* and *continue* statements are handled. The *break* within a loop generates a jump to its end, and represents an alternative path through the body of the code for the *last* iteration. It is important to know this when calculating the WCET of the loop, so a structural node representing the *break* is added after the RTL for the jump implementing it. Note that in Spring-C, the jump will go to the beginning of the exit code testing the loop counter against the iteration bounds.

The *continue* statement also generates a jump, but its destination is the beginning of the *while* loop's condition statement, and and the beginning of *S2* in the *for* loop of Figure 5.10. The *continue* thus represents and alternative path through the body of the loop for *every* iteration. A structural node marking the *continue* statement is added following the RTL for the jump implementing it. The structural nodes marking the *break* and *continue* statements are used during subgraph reduction of the TG, as described in Chapter 6.

5.1.3.1 Procedure Definitions and Calls

Previous sections have discussed all the Spring-C statements that can be used to described a computation within a single procedure. Here we discuss the RTL used to represent a procedure as a whole, and to represent calls to it. Figure 5.11a illustrates the declaration of a simple subroutine, and the corresponding RTL representing it. The structural nodes P and E are endpoints for the RTL of the procedure, and the P node holds its name. Each procedure has a small amount of entry code which precedes its body, and a small amount of exit code that follows the body. The entry and exit code are charged with setting up and taking down the call frame associated

with the procedure on the stack, and for saving and restoring registers as required, since the Spring-C compiler uses the *callee-saving* convention.

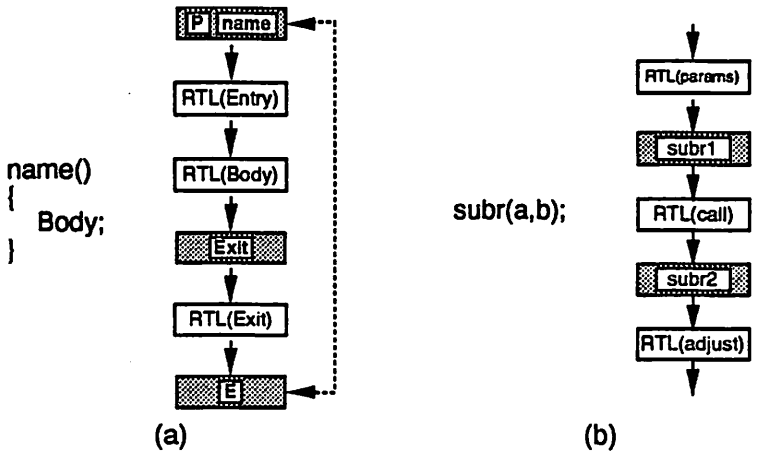


Figure 5.11. Enhanced RTL for Simple Procedure Definition and Call

The *Exit* structural node marks the beginning of the exit code. This is important when considering the behavior described by the *return* statement, since it specifies a jump from where the *return* appears in the body of the procedure directly to the exit code. Since return statements have this property, their locations are marked with a *return* structural node, just as the locations of *break* and *continue* statements are marked. The implications of the *return* for WCET calculation are discussed in Section 6.2.

The RTL for the body of the procedure can be arbitrarily complicated, and may include calls to other procedures. Figure 5.11b illustrates the structural nodes used to mark a procedure call, as already discussed in Section 5.1.1. The RTL for preparing the parameters on the stack and then calling the routine is generated normally. A set of structural nodes surrounds the call instruction, and any code required to restore the stack to its state prior to pushing the parameters follows.

Recursive procedures must, however, be handled differently with respect to both their definition and the procedure call which begins the recursion. The *originating* call to a recursive routine specifies the recursion depth bounds, as discussed in Chapter 4 and illustrated in Figure 5.12a. The code for the call actually produced assumes additional depth bound pointer arguments, and is equivalent to the code illustrated in Figure 5.12b. Of particular interest are the creation of the *lb* and *ub* variables, pointers to which are passed to the recursive routine. They will be used to monitor how many times the routine is called during recursion. Modifying the code emitted for the originating *call* in this fashion is simple, but requires a corresponding change to the code emitted for the recursive procedure.

```
name(a,b)(LB,UB);
```

(a)

```
{
    int ub,lb;
    lb = LB;
    ub = UB;
    name(a,b,&lb,&ub);
}
```

(b)

Figure 5.12. Recursive Procedure Call

Figure 5.13a illustrates the definition of a recursive procedure as written in Spring-C, while Figure 5.13b illustrates what the code produced would look like in source form. The situation is analogous to that already discussed for loops; the compiler would *not* transform the Spring-C source code as illustrated, and then compile it, rather, the RTL produced by Figure 5.13a would be logically equivalent to Figure 5.13b, which presents several points of interest.

First, the routine has the *lb* and *ub* parameters added, which are used to track the number of times the routine is called recursively, and enforce the bounds associated with the originating call as illustrated in Figure 5.12b. When the procedure is entered,

```
recursive ret-type
name(t_1 a, t_2 b)
{

    BODY;

}
```

(a)

```
ret-type
name(t_1 a,t_2 b, int *lb, int *ub)
{
     (*lb)--;
     (*ub)--;
     if ( *ub < 0 ) {
         bound_violation();
     }
     BODY;
Exit:
     if ( *lb > 0 ) {
         bound_violation();
     }
}
```

(b)

Figure 5.13. Recursive Procedure Definition

the upper and lower bound counters are decremented, since a new recursive call is beginning. Then the upper bound is checked. If it has decremented past zero the call frame just entered violates the upper bound, and the system is notified of the violation. If the upper bound is not violated, then the execution of this recursive call may proceed. Note that the use of counters in this way tracks the *total* number of times the routine is called. When a routine calls itself only *once* in each recursive case, this is the same as the recursion *depth*. However, when the routine calls itself more than once in the recursive case, then this method still properly enforces the limit on the total number of calls. Although we commonly refer to the limits as "recursion depth bounds", it may be more accurate to call them recursion *iteration* bounds.

As already discussed, the *return* statement translates into a jump instruction that transfers control to the *exit* code at the end of the routine. The exit code is responsible for restoring registers, unlinking the call frame, and then executing the return from subroutine (rts) instruction. In the case of a recursive procedure we add a check of

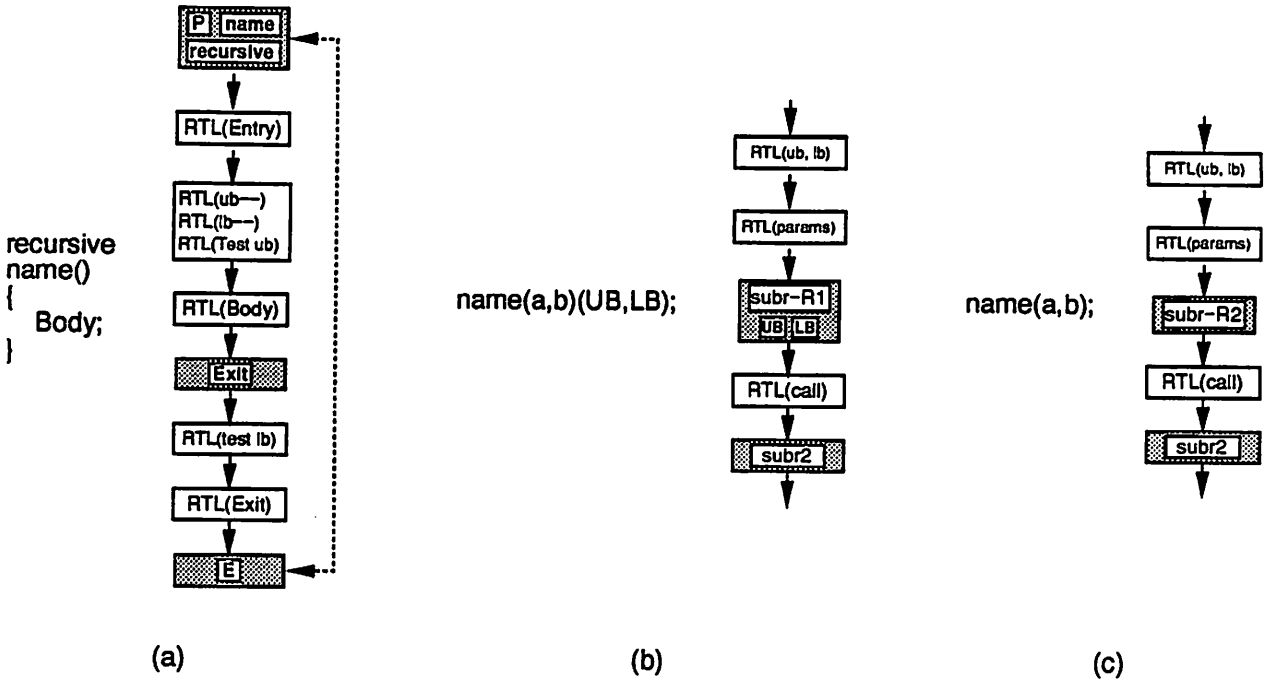


Figure 5.14. Enhanced RTL for Recursive Procedures

the lower bound, as illustrated in Figure 5.13b, following the *Exit* label but prior to register restoration.

Figure 5.14 illustrates the enhanced RTL produced for various Spring-C constructs associated with recursive procedures. Figure 5.14a illustrates the RTL for the procedure definition, showing several additions to the template for non-recursive procedures illustrated in Figure 5.11. The RTL generated for the entry code is the same as before, with the addition of decrement of the *lb* and *ub* counters and the upper bound test. The exit code is supplemented with the lower bound test prior to its previously discussed actions.

Figure 5.14b shows that the original call to the recursive routine generates RTL that sets its upper and lower recursion bound variables, and pushes their addresses on the stack as additional parameters to the recursive routine, prior to doing so for the original parameters. The first structural node of the pair surrounding the *jsr*

instruction call the routine also records the upper and lower bounds. These are used during subgraph reduction to calculate the WCET represented by a call to a recursive routine.

In contrast, Figure 5.14c illustrates the RTL generated for the recursive calls. These are the calls to the procedure made from within the procedure itself. Note that the syntax for the recursive calls within the procedure do not provide recursion depth bounds, since they represent calls in the middle of the recursion rather than at its origin. However, the RTL generated for such calls *is* different from a conventional procedure call in that the pointers to the upper and lower recursion bounds are pushed on the stack as parameters of the next recursion level. The compiler can easily identify these recursive calls and the need to push the extra parameters, since the name of the called procedure matches that of the procedure being compiled.

5.2 Time Graph Construction

The *time graph* (TG) is a representation of the control structure of a program which abstracts information required to predict the target behaviors of a procedure, while discarding everything else. Recall that the behaviors include each execution episode's WCET, resource use, explicit delays, and synchronous communication acts. The TG is constructed by a simple procedure which begins with the enhanced RTL representation constructed by the compiler, as discussed in Section 5.1. The TG is constructed just prior to final code emission, and so the RTL used reflects any the changes made by various compiler optimization phases. It is important to note that the optimization phases are currently disabled, because they have not yet been modified to handle the new structural and suspension point nodes added to the RTL. For some optimizations modification will be trivial, while others will require more significant effort. However,

optimized or not, the essential point is that the TG reflects the properties of the code which is *actually emitted.*

Building the original TG is fairly simple. Structural and suspension point nodes in the TG are essentially copies of those in the enhanced RTL. The TG versions of these nodes are, however, slightly more complex since they contain pointers to the RTL nodes they represent, as well as recording the source line numbers at which the structures they represent appear. Figure 5.15 gives the type definition for a TG structural node containing the *rtl_node* and *src_line* fields. The use of this information is discussed in Chapter 6.

```
typedef struct {
    int                type;     /* type of structural node */
    struct _tg_node *next;
    rtx                rtl_node; /* node in RTL represented */
    int                src_line;
} tg_struct_node_t;
```

Figure 5.15. Time Graph Structural Node Definition

The set of RTL nodes comprising a basic block of the procedure is represented in the TG by a *time node* giving the WCET of the corresponding sequence of assembler instructions, as calculated by the target machine execution time model, or machine model, described shortly. Each basic block is a sequence of assembler instructions that are executed in order, and which may or may not terminate in a *jump* instruction. Each time node thus represents a *subgraph* of the original RTL representation of the procedure, and so contains fields pointing to the first and last RTL nodes of the subgraph. This is important when additions to the original RTL graph are required in response to situations identified during subgraph reduction. When such additions are required, the TG nodes' RTL subgraph pointers identify where the additions must

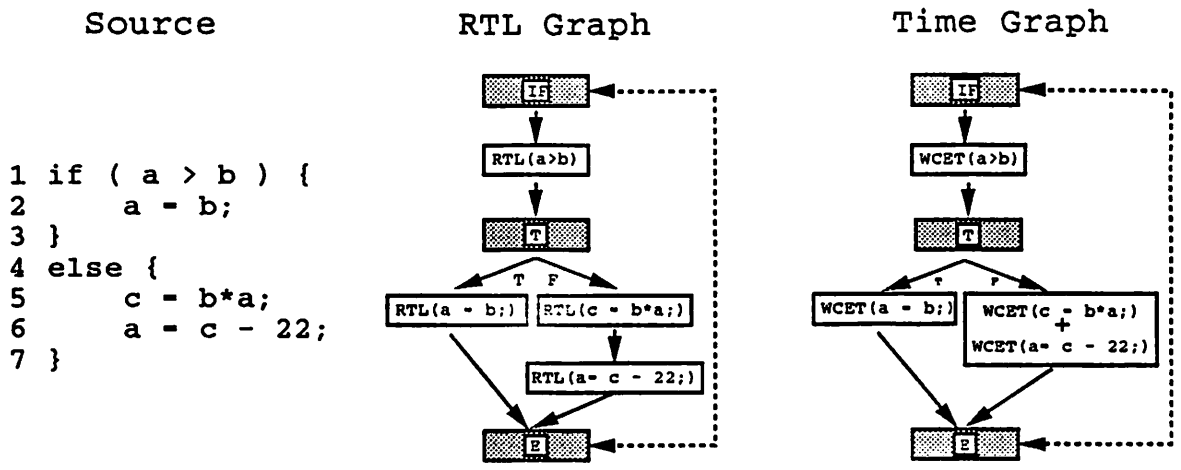


Figure 5.16. Source, RTL, and Time Graph Representations

be made. The time node also notes the source lines corresponding to the beginning and end of the subgraph it represents.

Figure 5.16 illustrates the source, RTL, and TG representations of a simple conditional statement. Note that the structural nodes which define the overall structure of the conditional are the same in the enhanced RTL and in the TG, while the RTL sequences representing the conditional expression, true clause, and false clause are each replaced with a time node giving their WCET. The time node representing the false clause would, for example, hold pointers to the first RTL node in the sequence representing source line 5 and the last RTL node in the sequence representing source line 6, since these delimit the RTL subgraph for the false clause's basic block. The time node also, of course, records the beginning and ending source line numbers of the subgraph, 5 and 6 respectively.

The TG representation of the conditional statement preserves the full range of its target behaviors, in this case only WCET, while eliminating its semantics. Since the example is simple, there are only two possible behaviors, corresponding to the two

possible paths through the *if* block. The TG correctly represents the target behaviors of both paths, discarding all extraneous information.

The procedure building the TG, *do_time_graph* must produce a WCET estimate for each basic block in the RTL graph, which clearly requires that the assembler instruction sequence comprising each basic block be known. We enabled the compiler to produce this information by the simple tactic of modifying its code emission routine, *final*, to optionally produce structural node notations, as well as the assembler, into a *sequence file.* The structural notations clearly define the assembler sequences corresponding to each basic block, since each is bounded by a specific pair of structural nodes. Note that the assembler sequences are emitted into the sequence file in the order they are required during TG construction.

Our approach to generating the assembler sequence for each basic block has several good properties. First, it was simple to implement, since it involved a relatively minor change to the existing routine *final.* Second, it is reasonably efficient, since it takes advantage of the work already done by the compiler in preparing the RTL. Third, it *guarantees* that the WCET prediction for each basic block is based on *exactly* the assembler instruction sequence actually executed by the program since *final* is also used to emit the code that will be assembled and linked.

Other methods for predicting execution time of programs predict the assembler that will be emitted for source level schema in a separate operation from actual compilation[65]. Since, however, the assembler and its behavior is predicted for each source level schema separately, it is difficult to see how such a tool can predict the effects of compiler optimizations which cross schema boundaries without either reproducing the optimization phases of the compiler, or integrating the prediction method

into the compiler proper, as Spring-C has done. For example, the elimination of common subexpressions is a standard compiler optimization. It searches for expressions which are components of two or more separate expressions, and restructures the emitted code so that the common expression is only computed once and the result stored. The other places where the common expression appeared are rewritten to use the stored result rather than computing the expression. If, however, the common subexpression is present in different source schemata, then the prediction method using source level timing schema would not take the effects of this optimization into account, since the optimization crosses schemata boundaries but the prediction method considers schemata individually.

Note however, that the Spring-C compiler *spr-cc* does not yet perform optimizations since each of the current optimization methods must be examined, and possibly modified, to ensure that they take the structural and suspension point nodes into account properly. For example, consider the optimization that moves invariant expressions outside the body of a loop. In the original compiler's RTL, where structural nodes did not exist, this optimization removes the invariant expression from the loop body and inserts it before the label marking the beginning of the loop. With the introduction of structural nodes, however, this label follows the *LH* structural node. Unmodified, the optimization would thus place the loop invariant *between* the *LH* node and the label at the beginning of the loop. This would leave it, as far as the behavioral prediction method is concerned, as part of the conditional expression.

The behavioral analysis would thus add the invariant's execution time into every loop iteration, producing a needlessly over estimated WCET. The optimization should, instead, move the invariant expression *before* the *LH* node, so that it becomes part of the loop initialization. Some optimization methods will be easy to modify, but

others may be more difficult. The fact that there are many independent optimization methods implemented in the compiler will, however, simplify the job of creating optimizations for Spring-C. As each optimization method is examined and modified as required, its effects will automatically be taken into account by the behavioral prediction method described in this dissertation as a natural consequence of when the predictive calculation is performed.

Other WCET calculation methods which work with the assembler output [1], or by disassembling the executable [29], clearly consider the instructions that are actually executed, but largely sacrifice the ability to transform the code in response to intermediate analysis results. The reason for this is that the emitted assembler does not contain the structural and semantic information required to perform the analysis, but which is available during compilation. Significant analysis and transformation of the assembler is theoretically possible, but it would required either emitting or reconstructing the information available during compilation. However, since the compiler would have to modified to *emit* the necessary information into the assembler, it seems simpler to also perform the analysis and transformation during compilation as Spring-C does.

Transformation of the emitted code in response to intermediate analysis is an indispensable part of the method for constructing a valid task group representation as discussed in Chapter 6. The need to balance two conflicting issues, the desire to work with *precisely* the assembler sequences which will be executed, and to preserve the ability to transform the program, is why we chose to fully integrate Spring's behavioral analysis into the compiler.

When the first call to *final* has made the assembler sequences for every basic block available, *do_tg_graph* then calls *tg_build* which makes another pass through the RTL,

using a recursive descent strategy similar to that used for subgraph reduction to build the TG. The basic idea is quite simple, and depends on the fact that the body of the procedure is a series of simple structures, corresponding to the various statements in Spring-C, within which other instances of these structures may be nested.

As *tg_build* scans through the RTL for the procedure, it encounters a series of structural nodes. Those at the main level of the procedure delimit the procedure's basic blocks. As was illustrated in Figure 5.11, the body of a very simple procedure will consist of a single basic block delimited by the *P* and *Exit* structural nodes in the RTL for the procedure. More complex routines will obviously contain nested blocks, which *tg_build* handles by calling a routine that knows how to build the portion of the TG for that block.

So, for example, consider the case where the conditional statement illustrated in Figure 5.16 appears in the body of a procedure. As *tg_build* scans the RTL for the procedure body, it encounters the *if* structural node, knows that it is the beginning of an embedded conditional block, and calls the routine handling conditional blocks. That routine copies the structural nodes as indicated in the figure, and produces a time node for each basic block. The basic blocks are delimited by structural nodes, which make it simple to find the proper assembler sequence in the sequence file. Note, however, that no *search* of the sequence file is required, since the sequences are encountered during TG construction in the order in which they are stored in the sequence file.

The WCET for each instruction in the sequence is found by submitting it to the target machine execution time model, or machine model. This tool parses the instruction, identifying it and its addressing modes, and returns its estimate of the WCET. We originally used the machine model described in [42], which was a version of the

tool described in [1] modified for the 68020. However, that model had several practical drawbacks, including a lack of support for floating point. Instead we constructed a table driven tool using times for every instruction and every addressing mode which were measured by direct experiment. The accuracy of the predictions produced by the current techniques, architectural features affecting the results, as well as several ways in which the accuracy of the predictions might be further improved are discussed in Section 6.6, in the context of the predictions produced for whole procedures.

When *tg_build* completes its pass through the RTL for the procedure, the TG representing its target behaviors is complete, and ready for analysis. This is performed by subgraph reduction as explained in Chapter 6.

CHAPTER 6

BEHAVIORAL PREDICTION

The predictions describing a process's execution episodes and their *target behaviors* provide the information required to build the task group representation of the process required by the Spring scheduler. We use the term "target behaviors" since only a subset of all possible process execution time behaviors are represented by the task group, as discussed in Chapter 5. Recall that the basic idea is for each task in the group to represent an execution episode of the process, and so when each task in the group is scheduled according to its WCET, resource use, and precedence constraints, each episode of the process's execution is guaranteed to have every system resource it requires to execute properly. The target behaviors of a process being predicted are thus the number of execution episodes, and the WCET and resource requirements of each episode. The execution episodes are defined by suspension points within the program, such as delay, resource use, and synchronous communication. This chapter discusses the methods by which we make the predictions about the target behaviors exhibited by each execution episode of a process.

Predictions about the execution time behavior of a program must, obviously, be based on information about the executable code of the program, as well as information about constraints on its execution. Section 4.1 described the syntax changes made to conventional C to produce the Spring-C language. The new syntax explicitly represents execution constraints which include loop and recursion iteration bounds,

as well as explicitly denoting all the places in the source code where the program can suspend its execution.

Section 5.1 described how the information represented by the unique aspects of Spring-C's syntax was recorded by the enhanced version of the compiler's intermediate representation (RTL). That section also discussed how other additions to the RTL, called structural nodes, preserved the block structure of the Spring-C source code. Section 5.2 then discussed how the *time graph* (TG) of a process is constructed from the information contained in the enhanced RTL just prior to assembler code emission. The TG represents the target behaviors of a process in a fairly compact form, since it discards extraneous application semantics represented by the RTL. The TG is the basis for the behavioral prediction method described in this chapter.

Since each task in the group represents an execution episode of the process, the task must represent *all* the behaviors an episode might exhibit. One way to determine all possible behaviors of the execution episodes would be to examine *every* path through a process's TG. For programs of non-trivial size this would require the examination of an infeasible number of paths. We avoid this problem by simplifying the TG *before* the analysis determining the set of behaviors each episode can exhibit is performed. The TG simplification is performed by a technique called *subgraph reduction.* Subgraph reduction is based on a simple idea: that the block structure of the Spring-C language makes it possible to use a comparatively small set of simple rules to efficiently derive a description of the target behaviors of a process.

The reduction rules are simple because all extraneous details are discarded when the TG is constructed, leaving only those required to predict the target behaviors required to build the task group. Each reduction implements a rule about how to

simplify subgraphs of the TG corresponding to each type of block structured statement in the Spring-C language. A TG reduced to its most compact form is called the irreducible time graph (ITG). Each subgraph reduction simplifies the TG by replacing a subgraph through which there are two or more execution paths with a linear ITG representing the target behaviors of all the paths through the subgraph. As subgraphs of the TG are reduced the number of paths through the TG that must be considered when constructing the task group is reduced. When the TG is reduced to linear ITG form the number of paths through the TG is one, and deriving the target behaviors of each episode is simple.

Many of the Spring-C statements in a program will contain no suspension points, and can thus be reduced by rules concerned only with WCET. Section 6.1 discusses the set of reduction rules which replace subgraphs representing block structured statements in the Spring-C source with single time nodes giving the WCET of the block. The reductions described in this section are simple, but there are a few subtle points and they are included for completeness.

Since the reduction rules are based on the block structured language elements, it should be evident why the *goto* statement was eliminated from the Spring-C language. Using *goto* developers can create an extremely wide range of control flow structures which makes it difficult, or impossible, to construct a set of reduction rules capable of handling every possible subgraph that might arise. The advantage to the developer of using *goto* is thus modest in comparison to the cost in terms of the complexity it adds to the behavioral analysis. This is why we eliminated it from the Spring-C language. The *break*, *continue*, and *return* statements are, however, "structured" versions of the *goto* which specify alternatives to the normal control flow paths through Spring-C blocks, which we call *structured jumps*. They introduce significant, but manageable,

complexity into the analysis. Section 6.2 discusses the reduction rules for subgraphs containing structured jumps, but no suspension points.

More complex reduction rules are required for subgraphs containing suspension points, since they cannot be eliminated by subgraph reduction without discarding information about the process's execution episodes which is required to build a correct task group. However, we still want to have every subgraph reduction produce a linear ITG so the original TG can be simplified as much as possible before the analysis required to produced the task group is performed. Reducing subgraphs containing suspension points to linear ITG form requires the combination of suspension points. For example, consider an *if* block whose branches contain suspension points associated with the use of different resources. Reducing this TG to linear ITG form requires that the TGs representing the branches of the conditional be combined to form a single *composite* TG representing the worst case target behaviors of *all* the conditional's branches. The TGs representing the branches of the conditional must first be reduced to ITG form and then compared.

When a set of ITGs representing different branches of a conditional contain the same number and type of nodes in the same order, they are *balanced.* Combining balanced ITGs to form a new composite ITG is fairly simple. When the ITGs representing the conditional's branches are *not* balanced, they cannot be correctly combined because some branches exhibit a different number of execution episodes than others, and so no single ITG correctly representing the target behaviors of them all exists. Leaving the ITGs representing the conditional's branches unbalanced would mean leaving the conditional unreduced, which would complicate the analysis required to build the task group in a way we wish to avoid.

Transformation of the code represented by one or more of the ITGs is required to balance them. These transformations involve the insertion of code for the suspension points and time nodes required to make the number of execution episodes exhibited by every ITG match the number exhibited by the others with which it must be combined. The methods for reducing subgraphs containing SPs but no structured jumps, methods for balancing ITGs that must be combined, and their limitations, are discussed in Section 6.3. The sections described so far reflect the current state of the Spring-C compiler implementation.

Section 6.4 briefly considers how the methods for handling and resolving alternative paths through a block required for structured jumps, discussed in Section 6.2, and the methods for balancing and combining parallel paths required to handle blocks containing suspension points, discussed in Section 6.3, can be combined and generalized to handle blocks containing both structured jumps *and* suspension points. Section 6.5 then discusses the properties of subgraph reduction, showing among other things that the TGs of all Spring programs will reduce to linear ITGs.

Finally, Section 6.6 presents the several ways in which we evaluated the current implementation. The first level of evaluation employed test cases addressing each type of subgraph reduction. They were used to test the reduction rules and do not use the target machine timing model. This ensures that the reductions performed during each test are independent of changes to the target machine model. This is particularly important for tests addressing the more elaborate scenarios including: nested loops, switch statements with and without default cases, switch cases falling into others in various combinations, as well as those addressing the *return*, *continue*, and *break* structured jumps statements. The test cases for subgraphs containing SPs address a wide range of scenarios involving the combination of conditional branches, and

in particular those requiring the balancing of several sequences before combination. These test cases also ensure that the RTL and source subgraphs corresponding to each time node and SP in the TG are properly tracked as reduction proceeds. The RTL subgraph information is used during program transformation, while the source level information helps relate the behavior represented by the ITG back to specific portions of the source program.

The second type of evaluation is represented by the application software examples completed and in progress, as discussed in Section 4.3. They clearly demonstrate that the system gives the developer the ability to write the application code at the level of processes, and to have a task group representation of the application constructed automatically. The correctness and accuracy of that representation are addressed by the other two types of test cases, although we must obviously consider the results produced by the compiler with a critical eye, to detect any residual problems.

The third type of evaluation addressed the accuracy of the WCET predictions. Each test case concentrated on a different aspect of the target machine execution time model. For example, some addressed the accuracy of predictions about linear code sequences, while others considered loops, procedure calls, and conditional statements. The various experiments revealed the affects of many subtle factors on the predictions, and enabled us to improve prediction accuracy. The current results also serve to point out the need for a target machine model capable of taking every aspect of the target hardware into account, which is not true of most of the currently available tools [65, 1, 29].

6.1 Basic Subgraphs

This section begins by presenting the set of basic subgraph reductions which make it possible to reduce procedures containing no suspension points and no structured jumps to single time nodes giving the WCET of the procedure. With this set to use as examples, we then illustrate how the application of these rules is organized using a recursive descent strategy. The reductions described in this section are simple, as is the recursive descent control of their application. There are, however, a few subtle points, and the techniques described here form the foundation for more complex reductions described in later sections.

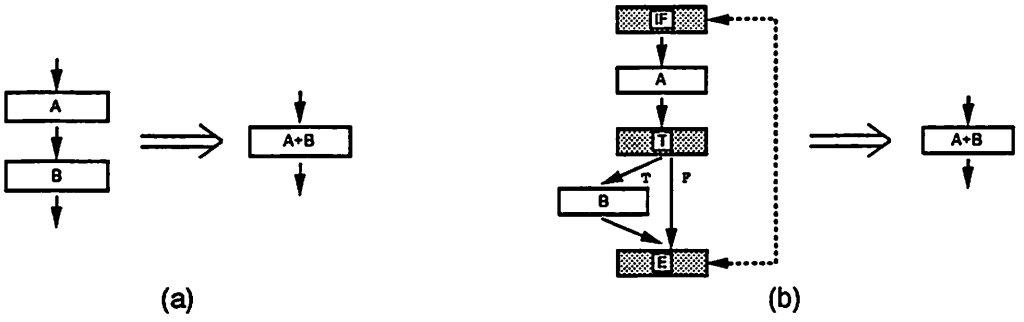


Figure 6.1. Linear and If-Then Subgraph Reductions

The simplest subgraph reduction is that for two consecutive time nodes, which can be be reduced to a single time node whose WCET is the sum of the WCETs of the original nodes, as illustrated in Figure 6.1a. The next simplest reduction is that for an *if-then* conditional statement, shown in Figure 6.1b. The entire subgraph, including the structural nodes, can be reduced to a single time node whose WCET is the sum of the WCETs of the time nodes representing the conditional expression and the true clause. However, it is important to note that the subgraph being reduced must *exactly* match the left side of the reduction rule. The conditional expression and true clause must each be single time nodes. If this is not true, then the subordinate

clauses must be reduced before the conditional reduction rule of Figure 6.1b can be applied. As will be discussed shortly, this is why a recursive descent strategy is used to control subgraph reduction.

The subgraph reduction for the *if-then-else* statement is almost as simple, but WCET of the time node it produces is the sum of the WCET of the conditional expression and the maximum of the WCETs of the true and false clauses. This is illustrated in Figure 6.2 and, as with all reduction rules, requires that all three subordinate clauses of the statement already be reduced to single time nodes.

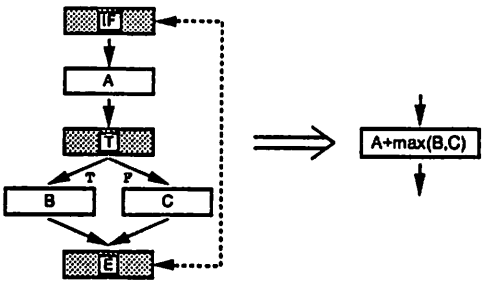


Figure 6.2. If-Then-Else Subgraph Reduction

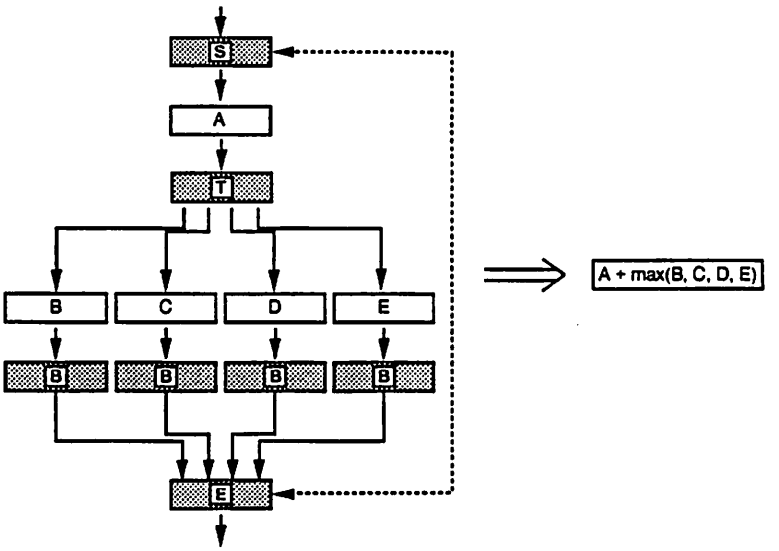


Figure 6.3. Switch Statement Subgraph Reduction

Figure 6.3 illustrates the reduction for the subgraph representing a *switch* statement, which is a natural extension of that for the *if-then-else*. When the TG for the expression generating the switch value has been reduced to a single time node, and when the same is true for each of the case statements, then the *switch* can be reduce to a single time node whose WCET is the sum of the time node for the switching expression and the maximum of all the cases. A slightly more complex problem is presented when one of the cases falls through into the next. In this case, even when the individual cases have been reduced to single time nodes, a subgraph *transformation* is required before the *switch* block can be reduced.

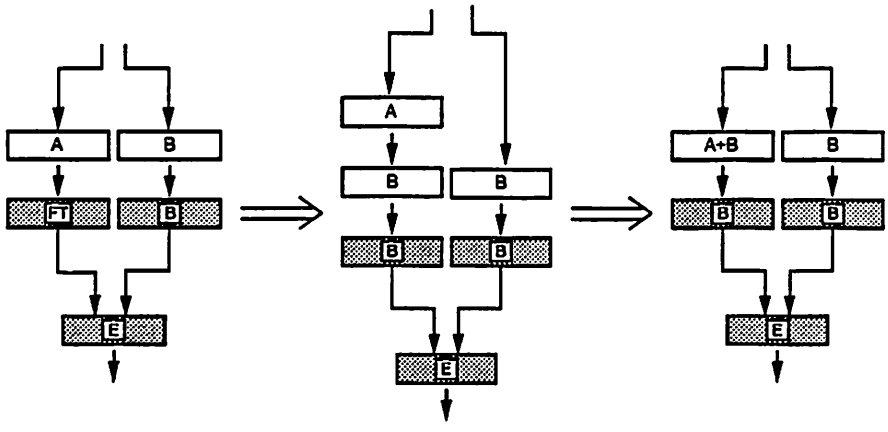


Figure 6.4. Switch Fall-Through Subgraph Transformation

Figure 6.4 illustrates the transformation, showing the TG nodes for the two *switch* cases involved. The transformation first duplicates the time node representing the case that is fallen into, appending it to the end of the sequence representing the case that is falling into the other. It can then simply replace the *FT* structural node with a *B* node. However, now the first case is not longer fully reduced as required by the *switch* reduction rule given in Figure 6.3. However, the reduction of Figure 6.1a can be applied to produce the fully reduced form which effectively replaces the time node of the case falling into the next with a node whose WCET is the sum of its WCET and that of the case into which it falls.

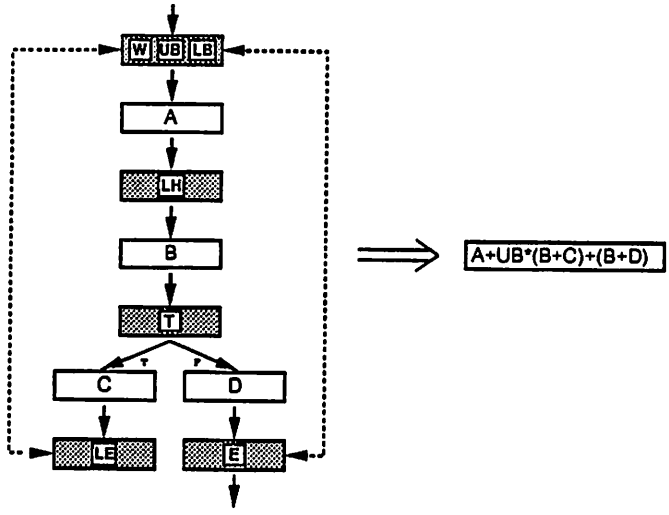


Figure 6.5. While Loop Subgraph Reduction

Figure 6.5 illustrates the reduction for the *while* loop. As should now be familiar, each clause of the subgraph being reduced must itself have been reduced to a single time node. In Figure 6.5, A represents the time for the loop initialization, B the time for the loop conditional, C the time for the body of the loop, and D the time for the loop exit code. The expression for the WCET of the time node produced by the reduction has three components: initialization, loop execution, and loop exit. The initialization time is obviously A. The time for a single iteration of the loop is the time to execute the loop conditional, plus the time for the body of the loop, $B + C$. The maximum time for loop execution is thus the product of the time to do one iteration and the upper bound on loop iteration, giving $UB * (B + C)$. Exiting the loop requires a final execution of the loop condition and then execution of the exit code, giving $B + D$.

The reduction of the *do-while* loop is quite similar to that of the *while*. Figure 6.6 illustrates the original TG and the resulting time node after reduction. In the figure, A is the time for loop initialization, B the time for the body of the loop, C the loop conditional test, and D the loop exit time. The expression for the WCET

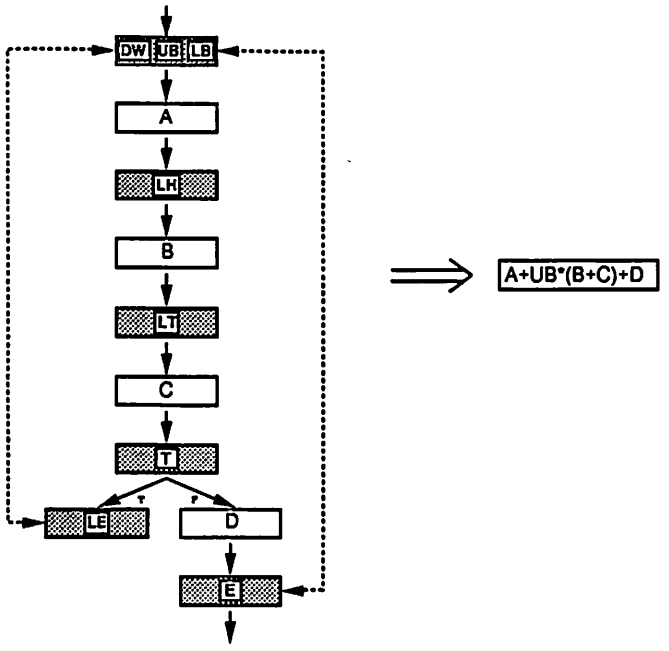


Figure 6.6. Do-While Loop Subgraph Reduction

of the resulting time node has three components as before, initialization, execution, and exit. As already discussed in Section 5.1.3, the *for* loop is an extended version of the *while*. Its reduction rule is thus similar to that of the *while* loop, extended to include the *for*'s initialization and loop iteration statements.

Note that while the discussion has addressed the WCET value of the time nodes involved, other information is also being manipulated during subgraph reduction. Of particular importance is the information describing what portion of the RTL for a process is prepresented by each node, which we call its RTL subgraph. As subgraph reduction takes place the nodes of the TG tend to represent ever larger RTL subgraphs. For example, consider a conditional statement that is reduced to a single node. Each RTL node is referenced using a unique number, so assume the *if* structural node of the conditional is numbered 893, and its *E* structural node is numbered 913. The reduction routine sets the RTL subgraph of the new node so that it begins with 893, and ends with the 913. The new time node thus states that it represents

the entire subgraph for the conditional, as it should. Each of the other reductions set the RTL subgraphs of the nodes produced in a similar manner.

Knowing the starting and ending nodes of the RTL subgraph represented by each TG node is important when new code must be inserted before or after the RTL represented by a given time node or suspension point. This is often required when blocks containing suspension points are being reduced, and ITGs representing components of the block being reduced must be balanced before combining them, as discussed in Section 6.3.1. Each TG node also notes the status of the RTL subgraphs endpoints, as well as the source lines it represents. The status information is used when ITGs are being combined, describing constraints on how the ITGs may be transformed to make combining them possible. The source line subgraphs are handled just as the RTL subgraphs are during subgraph reduction. This information helps the developer by showing what portions of the source program determine the properties of each task in the task group. Eventually this information could be used by debuggers and development environment tools.

The set of reductions discussed are all those required to fully reduce TGs representing Spring-C procedures containing no suspension points and no structured jumps. Applying these reductions to the TG for such a procedure until no further reduction is possible will reduce it to a single time node giving the procedure's WCET. Accounting for the effects of procedures called by the one whose TG is being reduced is simple, requiring only that the structural node representing the call to the procedure be replaced in the TG being reduced by the ITG of the procedure called.

While the subgraph reduction rules presented are sufficient to reduce the TGs of procedures without suspension points or structured jumps to single time nodes, *achieving* that goal requires the compiler's TG reduction phase to apply the proper

reductions to subgraphs of the original TG in the proper *order*. The Spring-C compiler uses a recursive descent strategy, very similar to that used by recursive descent compilers, to control the application of the reduction rules. The effectiveness of this approach depends on the fact that the original TG constructed for every Spring-C procedure always consists of a sequence of time nodes and suspension points with embedded block structures.

The recursive descent approach to subgraph reduction requires a reduction algorithm for each subgraph reduction rule. The reduction of the TG for a procedure is handled by the *reduce_proc* algorithm which begins at the structural node marking the beginning of the procedure. It proceeds through the body of the procedure's TG, reducing embedded structures by using the appropriate reduction algorithms, until it reaches the structural node marking the end of the procedure. There are various subordinate reduction routines addressing specific structures and situations. The *reduce_if*, for example, handles the reduction of embedded *if* blocks as they are encountered. The *reduce_item_sequence* routine is used to reduce a sequence of items to irreducible form.

The reduction algorithms for each block type all follow the same strategy: convert each section of the subgraph to irreducible form by using *reduce_item_sequence*, and then perform the reduction for the subgraph as a whole. During the reduction of each section of the subgraph, when a nested block is encountered the appropriate reduction algorithm for the block is used recursively. So, for example, the *reduce_if* algorithm applies *reduce_item_sequence* to its conditional, true, and false clauses. As each item in a sequence is considered, *reduce_item_sequence* would use *merge_time_nodes* to combine two consecutive time nodes, and the appropriate algorithm to reduce a nested block. The embedded block structures can, of course, be nested arbitrarily deep.

The property upon which everything rests is that when the reduction algorithm for a nested block completes, the nested block has been transformed into a linear ITG ready for further reduction as part of the subgraph within which it was nested. As is explained in subsequent sections, the structure of Spring-C and the syntactic features added to create it, ensure that the TGs of all Spring-C programs *will* reduce to linear ITGs, and that a task group representation can thus be constructed for every Spring-C program.

Appendix B presents the *reduce_proc* algorithm in detail, as well as some of the subordinate algorithms addressing the reduction of specific block structures. An example of the recursive descent application of subgraph reduction rules to a simple procedure is also given in Appendix B. It is important to note that these versions of the algorithms only handle those TGs which contain no structured jumps or suspension points. These basic versions will be extended in later sections to handle both.

6.2 Subgraphs with Structured Jumps

Section 6.1 discussed the basic subgraph reductions and how the recursive descent strategy is used to control the application of reduction rules to the TG. The rules discussed in that section assumed that the flow of control entered and left the nested TG blocks representing the nested Spring-C statements of the procedure in an orderly way. This well structured behavior simplifies several aspects of subgraph reduction, and it was thus reasonable to explain the basic approach in that context. There are, however, three statements in Spring-C which violate block structured control flow: *break*, *continue*, and *return*. Each of them specifies a transfer of control which can

jump across block boundaries, but where the *destination* of the transfer is still well defined by the statement's location within the procedure's nested block structure.

The word *structured* is used to distinguish these jumps from those transferring control to *arbitrary* locations within a procedure, such as those created by the *goto* statement. The subgraph reductions used to calculate program behaviors can take *structured* jumps into account fairly easily. The destination of each structured jump is defined by its type, and by its *context block*. The *break* statement specifies a jump to the *end* of its context block. The context block of the *break* statement is the most deeply nested *switch* or loop block containing it. For example, consider a *while* loop containing a set of *if* statements nested five levels deep, with a *break* statement along a branch of one of the conditionals. The *while* is the context block of the *break*, which specifies a jump to the end of the *while*. The level at which the *break* appears within the nested *if* statements does not affect the destination of the jump. However, if the nested *if* statements are part of a *switch* statement's case, which is in the body of a *while* loop, then the *switch* is the context block of the *break*, which specifies a jump to the end of the *switch*.

The *continue* statement is only applicable to loops, specifying a jump to the beginning of the loop for the next iteration. This has slightly different implications for reduction of the *continue* in the context of the *for*, *while*, and *do-while* loops, but the basic idea is the same. The *return* statement specifies a jump to the end of its context block, which is the procedure containing the statement, no matter how deeply nested the *return* is within the nested blocks of the procedure.

Given these specific definitions for the destination of each structured jump it is a reasonably simple matter to create subgraph reductions to handle them. Structured jumps which are *nontrivial* represent an *alternative path* through their context blocks,

in addition to the *main* paths accounted for by the subgraph reductions already described. A structured jump is *trivial* when it specifies a main path through the context block. For example, a *continue* at the end of a *while* loop body is trivial since it specifies precisely the same transfer of control as would occur if it were not there. Also, a *return* which appears as the last statement of a procedure is trivial in the sense it specifies a jump to the exit code of the procedure which the unmodified flow of control would immediately enter in any case.

The behavioral analysis for a block must consider *every* path through it. In the case of WCET, the need for this is clear because an alternative path might represent the largest execution time. The basic problem is how to retain information about alternative paths through the context block until the components of the block are fully reduced, and final reduction of the block can be done. At that time the behavior represented by the main path can be resolved with that of the alternative paths. The reason this is a non-trivial problem is that non-trivial structured jumps are always contained within a clause of some conditional statement nested within the context block, and are thus eliminated during reduction of the context block components. If they were *not* contained within a conditional, they would be trivial.

In the case of context blocks containing no suspension points, the alternative path can be represented by a time node giving the WCET from the beginning of the context block to the structured jump. For a *switch* statement this is the WCET from the beginning of the case, which is the end of the switching expression, to the *break* statement. For a loop, this is the WCET of the path from the beginning of the loop body to the *break* or *continue* statement. For a procedure, it is the WCET of the path from the beginning of the procedure to the *return* statement. The set of alternative paths is stored by the structural node starting each block type.

The rest of this section will discuss how the alternative paths are calculated and then resolved when the context blocks are ready for final reduction. Section 6.2.1 discusses how an alternative path is calculated and stored when a structured jump is encountered in the course of reducing a sequence. Section 6.2.2 will then discuss how the alternative paths are resolved with the other components of the context block when it is ready for final reduction.

6.2.1 Alternative Path Calculation

When a structured jump is encountered in the course of reducing a sequence of TG nodes, the *reduce_item_sequence* algorithm knows that it has reached the end of an alternative path through the structured jump's context block and must preserve information about the alternative path before proceeding. Since we are not yet considering blocks containing suspension points, the main aspect of the alternative path that must be noted is its WCET.

Essentially, three questions must be answered in order to handle structured jumps properly: (1) what is the context block of the structured jump, (2) how can we ensure that the main path's WCET is calculated properly without either ignoring relevant behavior or considering behavior associated only with the alternative path, and (3) what is the WCET of the alternative path ending at the structured jump. The WCET of the alternative path is denoted *path_length*(*context*, *sj-node*), where *context* identifies the context block, and *sj-node* specifies the structured jump being reduced. For a loop, for example, the context block is the loop, and the path length is calculated from the beginning of the loop body to the structured jump.

The information identifying the context block of a structured jump is easily maintained using a *context stack* for each structured jump type. The context of the *continue* structured jump, for example, is defined by the loop block within which it appears. When the reduction algorithm for a loop is about to use *reduce_item_sequence* to reduce the body of the loop it pushes a pointer to the structural node starting the loop block, the *root node* of the block, onto the continue context stack before proceeding and pops it off the stack when the loop body has been reduced. Thus, whenever a *continue* is encountered during reduction of the sequence, a pointer to the proper context block is on the top of the continue context stack.

Similarly, the context for a *break* statement is either a loop or a *switch* case. Whenever a loop reduction algorithm is about to reduce the body of the loop it pushes a pointer to the loop's root node onto the break context stack and pops it off after the body is reduced. The reduction algorithm for *switch* pushes and pops a pointer to the block's root node before and after the reduction of each case.

```
int proc1()
{
    int a,b,c;
    a = 0;
    while ( a < 22 )(1,22) {
        b = subr2(a);
        if ( b > 72 ) {
            c = huge_calculation(a,b);
            return(c);
        } else {
            a++;
        }
    }
    return(1);
}
```

Figure 6.7. Return Loop Context Example

The context for *return* is a little more complex. The procedure is obviously a context block, and a pointer to the procedure's root block is pushed on the *return* context stack at the beginning of *reduce_proc*. However, loops are also *return* context blocks. Consider the procedure which has a conditional inside a loop with a *return* in the true clause of Figure 6.7. The main path through the procedure, which will be accounted for by the subgraph reductions already described in Section 6.1, includes all 22 of the *while* loop's possible iterations, and ends at the `return(1)` statement.

The alternative path from the beginning of the procedure to the `return(c)` *also* includes 22 iterations of the loop, but enters the true clause of the conditional during the last. Calculating $path\_length(proc1, return)$, however, requires that we know the WCET for the conditional and body of the *while*. Yet, the `return(c)` is encountered in the middle of the loop body's reduction, so the WCET for the conditional and body of the loop are not yet known. The solution is to make the loop a context for *return* so $path\_length(while, return)$ can be saved as a *return* alternative path associated with the loop block until the other information becomes available. When the components of the*while* block have been reduced to ITG form, and the block is ready for final reduction, the information required to calculate $path\_length(proc1, return)$ is available. This information includes the WCET of the *while* conditional expression, the WCET of the *while* body, $path\_length(proc1, while)$, and $path\_length(while, return)$.

The second of the three questions, how to ensure the main path is properly calculated, also has a fairly simple solution. The alternative path diverges from the main path at the *TEST* node of the conditional statement. The main path through the body of the *while* in Figure 6.7 includes the conditional expression since the conditional expression is evaluated on every loop iteration. The true clause is *not*

part of the main path since *return* terminates the loop by jumping to the end of the procedure.

The third question, finding the alternative path length, is more subtle. Each reduction algorithm maintains a pair of pointers marking the boundary between the reduced and unreduced portions of the block it is reducing, called the *reduction boundary*. Behind the boundary each component of the TG of the block is in ITG form, while in front of the boundary is the portion of the TG that will be reduced next. Thus, whenever a structured jump is encountered *path_length*(*context*, *sj-node*) is the sum of the WCETs of all the ITG components from the beginning of the context block to the structured jump. Since, however, the recursive descent control method makes recursive calls to *reduce_block* for every nested block, the information required to calculate the alternative path length is distributed across several call frames. The problem is somewhat simplified by the fact that we are currently considering only TGs without suspension points, but the method described here can be extended to handle subgraphs containing both suspension points and structured jumps, as discussed in Section 6.4.

The alternative path length calculation is simplified by adding a new parameter to, and slightly extending the calculations done by, each reduction algorithm. Each reduction algorithm, starting with *reduce_proc*, maintains a running sum, *wcet_so_far*, of the WCET from the beginning of the procedure to the reduction boundary. The value of *wcet_so_far* is passed to *reduce_item_sequence* when a component of a block is being reduced, and to each algorithm reducing a specific block type within the sequence. The current value of *wcet_so_far* is pushed on the structured jump context stacks, along with the pointer to the block's root node, when the body of a context block is about to be reduced.

When a reduction routine encounters a structured jump, it determines the context block by checking the top of the proper context stack. Then it determines the WCET from the beginning of the context block to the structured jump, *path_length*(*context*, *sj-node*), by subtracting the *wcet_so_far* stored on the context stack, which is the time from the beginning of the procedure to the beginning of the context block, from the current *wcet_so_far* value. The alternative path is added to the list for the context block using the pointer to the context block structural node obtained from the context stack.

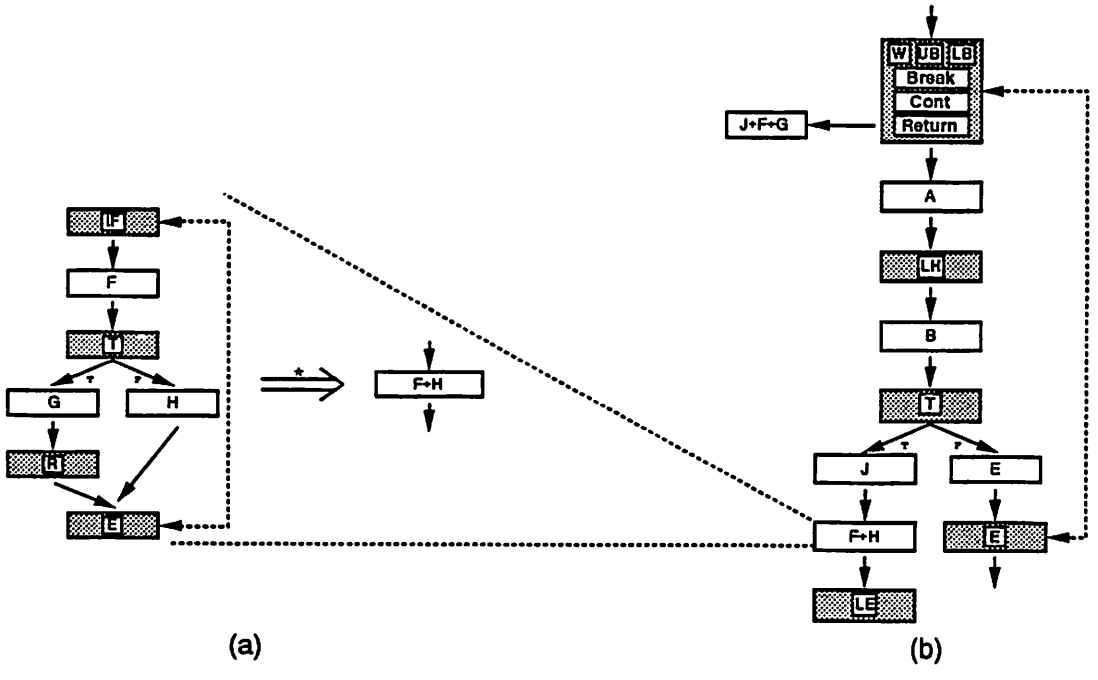


Figure 6.8. Alternative Path Length Calculation Example

Figure 6.8 illustrates the point for the *while* loop of Figure 6.7. Figure 6.8a shows the TG of the conditional statement within the *while* loop which has the *return* in its true clause. When the *reduce_while* algorithm was used to reduce the loop illustrated in Figure 6.8b, it was given a value for *wcet_so_far*, say X, which gave the time from the beginning of the procedure to the root node of the *while*. Then, when *reduce_while* was about to reduce the body of the loop, it pushed a pointer to the

while root node onto the stack as well as the current value of *wcet_so_far*, $X + A + B$. The *reduce_item_sequence* algorithm then shifted the reduction boundary across the node with WCET J, encountered the embedded *if* block, called *reduce_block* which classified the block and called *reduce_if*. The call to each algorithm passed the current value of *wcet_so_far*, $X + A + B + J$. When *reduce_if* calls *reduce_item_sequence* on the true clause, it passes $X + A + B + J + F$ as the value of *wcet_so_far*. When the *STRUCT_RETURN* structured jump is encountered *wcet_so_far* is $X + A + B + J + F + G$.

The structured jump processing begins by fetching the pointer to the root node of the context block and the value of *wcet_so_far* at that time, $X + A + B$, from the *return* context stack. Calculating $path\_length(while, return)$:

$$(X + A + B + J + F + G) - (X + A + B) = J + F + G$$

This is the WCET used when constructing the alternative path added to the *return* alternative list of the context block, as illustrated in Figure 6.8b. The true clause of the conditional is then deleted, and after the several subsequent reduction steps for the *if* block, produces the single time node representing the main path through the conditional as shown in Figure 6.8a. Figure 6.8b shows that this is the value used when reducing the *while*.

When the components of the block have been fully reduced, the worst case path through the block is determined by analyzing the main path, and its alternatives. For a loop this requires us to consider both the *continue* alternative paths and *break* alternative paths. In this example, there is only the *return* alternative. The loop's contribution to this alternative path is:

$$A + (UB - 1) * (B + J + F + H) + (B + (J + F + G))$$

This sum represents the initialization of the loop, A, one less than the upper bound iterations of the loop, $(UB - 1) * (B + J + F + H)$, before executing the *return* on the last iteration. The time for the last iteration is the sum of the loop conditional, B, and the *return* alternative path, $J + F + G$. This value is used to calculate $path\_length(proc1, return)$ by adding the value of *wcet_so_far* at the beginning of the loop, X, and subtracting the value obtained from the *return* context stack. The *return* context of the loop is *proc1*, which pushed a value of 0. The alternative path added to the *return* alternative list of *proc1* is thus:

$$X + (A + (UB - 1) * (B + J + F + H) + (B + (J + F + G)))$$

Finally, it is important to note how the RTL subgraph of the time node representing the alternative path is set. The begin point of the RTL subgraph is set to the RTL begin point of the relevant portion of the context block. In the case of a loop, this is the RTL beginning the loop body. The endpoint is simply the RTL structural node representing the structured jump. The status attributes of the RTL subgraph endpoints are also important during alternative path processing. When a time node is created during TG construction, it always represents code at the main level of the program and the status of the RTL subgraph endpoints is *MAIN_LEVEL*. During reduction, however, new status values can arise.

We have already seen one, although it was not mentioned at the time. When the ITG of a procedure is substituted for a call to the procedure, the status of the RTL subgraph endpoints in the nodes of the procedure ITG are set to *WITHIN_PROC*. The reason for this is clear when the insertion of nodes into an ITG during balancing

prior to combination, and thus into the RTL graph, is considered. Since the nodes of the ITG represent behavior within a procedure which has already been compiled, and which may be a library routine the current compilation has no authority to transform in any case, the node status notes that the compiler should avoid trying to perform an impossible code transformation.

Similarly, when the compiler detects an alternative path, and creates a time node to represent it, it must note that the begin point of that time node is *SHARED* with another path. This will not become crucial until we consider how to handle blocks containing both structured jumps and suspension points, but is worth mentioning. The basic idea is that code transformations affecting *SHARED* nodes should be done only if the transformation was appropriate to *all* the paths affected.

This section, and this example, have shown how the reduction algorithms can be extended to keep track of the information required to calculate and record the alternative paths represented by structured jumps. The next section discusses how this set of alternative paths are resolved with the main path derived for each block when the block is ready for final reduction.

6.2.2 Alternative Path Resolution

When components of a block have been reduced to ITG form, the block itself is ready for reduction. When no suspension points are present, as we assume in this section, every component reduces to a single time node. When no structured jumps exist, then the reductions presented in Section 6.1 are sufficient. When structured jumps *are* present, the reduced components of the block represent the behavior of the "main" path, the one not executing any structured jumps. This must be resolved with the

behavior represented by the alternative paths before a correct representation of the block's worst case behavior can be constructed.

The details of doing this can differ slightly with the type of structured jump and the context block, but the basic approach is the same for all. First, the various alternative paths are combined with one another, producing a single composite node for each alternative path type. This is done using the same method as is used to combine the sequences representing the true and false clauses of an *if* block during its final reduction. Since we are assuming that none of the blocks being reduced contain suspension points, the combining operation only has to find the alternative path with the largest WCET. The problem is obviously more complex when suspension points are involved. How the routine handles this more general form of the alternative path resolution problem is discussed in Section 6.3.1.

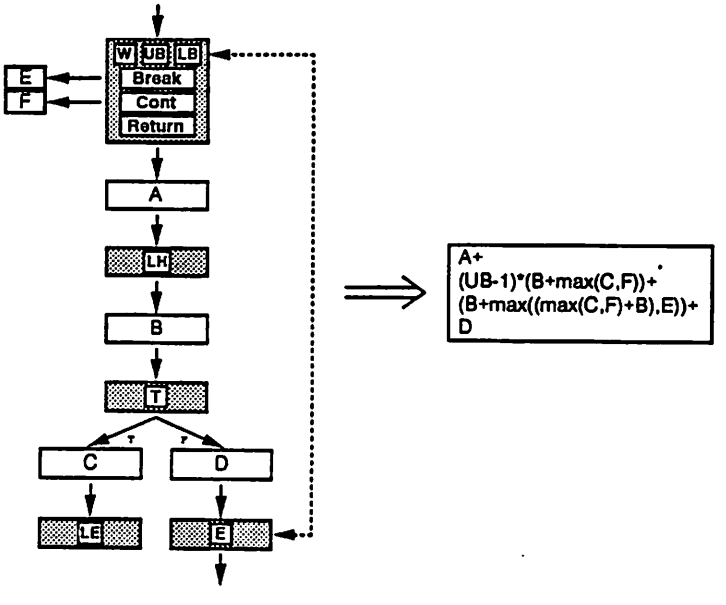


Figure 6.9. While Reduction Handling Alternative Paths

When a single, possibly composite, alternative of each type is established the block is ready for final reduction. This must, however, be done by extended versions of the reductions discussed in Section 6.1. Figure 6.9 illustrates the extended version

of the *while* block reduction. The example illustrates how the *break* and *continue* alternatives are considered when performing the final reduction of the block. How the *return* alternatives are handled was discussed in Section 6.2.1.

The WCET of the time node representing the behavior of the entire loop is given by the expression shown, which has four major sections corresponding to four segments of the worst case execution path through the *while* loop. The first segment, with the value A, accounts for the loop initialization which is done only once. The next expression accounts for all loop iterations but the last. During each of these, the worst case path through the body of the loop could be either the main path or the *continue* alternative, which is the reason for the $max(C, F)$ component of that term. During the last iteration, however, there are three possible paths: the main path, the *continue* alternative, and the *break* alternative. Further, at the end of the last iteration if the main path or *continue* alternative are taken, the conditional expression, represented by the node with WCET B, is executed one last time before the exit code is executed. If the *break* alternative is taken it transfers control directly to the exit code, without executing the conditional expression a final time. The third segment of the expression, while a bit complicated, reflects these possibilities. The last segment of the expression accounts for the exit code execution time D. The other loop reductions handle the final block reduction in the presence of alternative paths in a similar way, modified slightly due to the differing semantics of the loops.

The *switch* is a context block for the *break* statement. Normally *break* statements are used only at the end of each case. They can, however, also be nested within conditional statements and thus represent alternative paths through a case. Such situations may represent dubious programming practice, but are handled easily using methods very similar to those for loops. As each case is reduced, each *break* statement

encountered creates an alternative path. When the body of the case is fully reduced a set of alternative paths may thus be present. They are combined with the body using the methods already discussed. The final result represents the worst case path through the case. Reduction of other cases and of the *switch* block then proceeds normally.

The reductions discussed in this section extended those of Section 6.1 to ensure that all TGs containing structured jumps, but no suspension points, will reduce to linear ITGs. This preserves the property required by the task group construction, discussed in Chapter 7. The next logical extension of the basic reductions is how they can handle blocks containing suspension points, but no structured jumps. Section 6.3 addresses that question, while Section 6.4 considers how blocks containing both would be handled.

6.3 Subgraphs with Suspension Points

This section considers how the reduction methods are extended to handle suspension points. Suspension points cannot be eliminated without discarding information required to determine the number and properties of the process's execution episodes, and thus discarding information required to build a task group correctly representing the process's run-time behavior. Recall that each task represents an execution episode of the process, and that it must represent *all* the behaviors an episode might exhibit. One way to determine all possible behaviors of the execution episodes is to examine *every* path through a process's TG, but this is infeasible for programs of non-trivial size.

The purpose of subgraph reduction is to simplify the analysis required to build the task group by reducing the number of paths through the TG. Each reduction reduces

the number of paths by replacing a subgraph through which there are two or more paths with a linear ITG representing the target behaviors of *all* the paths through the subgraph. Reducing subgraphs containing suspension points to linear ITG form often requires the combination of suspension points. For example, consider an *if* block whose branches contain suspension points. Reducing the TG of such a conditional block to linear ITG form requires that the ITGs representing the branches of the conditional be *combined* to form a single composite ITG representing all the target behaviors of the original sequences. The method for combining ITGs is discussed in Section 6.3.1.

Once the method for combining ITGs is established, minor but significant changes to the reductions for the conditional blocks *if* and *switch* are required, which are discussed in Section 6.3.2. Loops can also contain suspensions points, and reducing such loops requires a new kind of reduction which effectively "unrolls" the behavior of the loop to create a linear ITG. Section 6.3.3 describes how this is done. Finally, it is important to note that these new reductions ensure that all TGs with blocks containing suspension points, but no structured jumps, will reduce to linear ITG form.

6.3.1 ITG Combination

Reducing conditional blocks containing suspension points is clearly a new type of problem since the number and type of suspension points in the branches of the conditional may differ, yet the reduction routine must produce a single ITG representing all the possible target behaviors of the conditional block as a whole. The crucial step in doing this, discussed in this section, is constructing an ITG which correctly represents *all* the target behaviors of the conditional block's branches.

The *combine_tg_sequences* algorithm performs this operation, and takes the set of ITGs which must be combined as input. It has already been discussed implicitly since it is used when reducing conditional blocks which do not contain suspension points, as discussed in Section 6.1. For those reductions, the ITGs given as input to *combine_tg_sequences* are the time nodes representing the branches of the conditional block. The algorithm is also used to combine the set of ITGs representing alternative paths through a block created by structured jumps, as discussed in Section 6.2. Simplifying the discussion in these sections by not explicitly mentioning *combine_tg_sequences* was reasonable since the new ITG it produced by combining the input ITGs was always a single time node which was immediately eliminated by further reduction of the block. However, when the conditional block contains suspension points, then the block will *not* reduce to a single time node, and the properties of the ITG produced by *combine_tg_sequences* are important.

Constructing a new sequence to represent a set of input sequences requires that the input sequences be in ITG form and that they be *balanced.* Two or more sequences in ITG form are balanced when they have the same *number* and *type* of TG nodes *in the same order.* When a set of sequences is balanced, they can be combined to form a new *composite* sequence which represents all the target behaviors of every input sequence. Each time node in the composite sequence has the maximum WCET of the time nodes in the input sequences it represents. Each suspension point reflects the properties of the suspension points it represents in the original sequences: maximum delay, union of resource use, and union of the synchronous communication acts. Both time nodes and suspension points in the composite sequence retain the union of the RTL and source subgraphs for the nodes they represent.

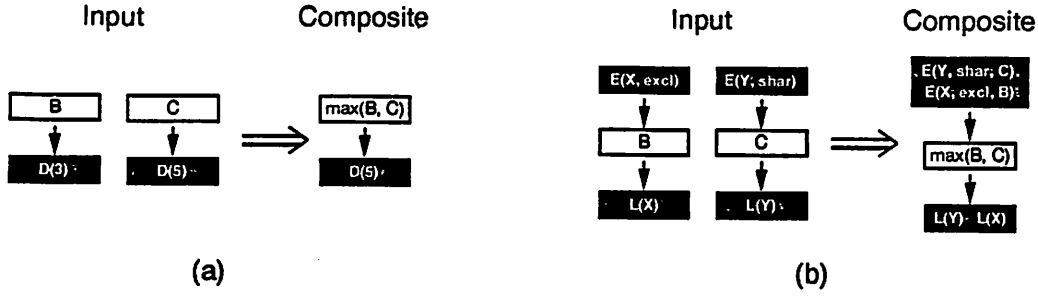


Figure 6.10. Balanced Sequence Combination Examples

Figure 6.10 illustrates two simple examples of combining balanced sequences to form a composite. The sequences shown as input to *combine_tg_sequences* might be the *true* and *false* clauses of an *if* block, or the clauses of a *switch* block with only two cases. Figure 6.10a shows how the WCET of the composite time node is the maximum of the input nodes it represents, while the *delay* value of the composite suspension point is the maximum of the input nodes' *delay* values. Figure 6.10b shows how the union of resource use is accumulated when suspension points are combined. Recall that the E suspension points represent *entering* a region where the resource is used in the mode specified, while the L suspension point notes where the region of resource use ends.

The two sequences in Figure 6.10b thus describe regions using resource X in exclusive mode and resource Y in shared mode. If these are the clauses of an *if* block, then the process will use one resource if it executes the true branch, and the other if it executes the false branch. The suspension points of the composite sequence representing the behavior of *both* input sequences specify the use of *both* resources since the ITG produced must represent *all* the possible target behaviors of the conditional. Note, however, that the suspension point representing the beginning of resource use retains the WCETs of the time nodes representing the body of each resource use block. This information will be reflected in the resource use of the tasks

in the group used to represent the process, and enables the Spring scheduler to avoid holding each resource for longer than necessary.

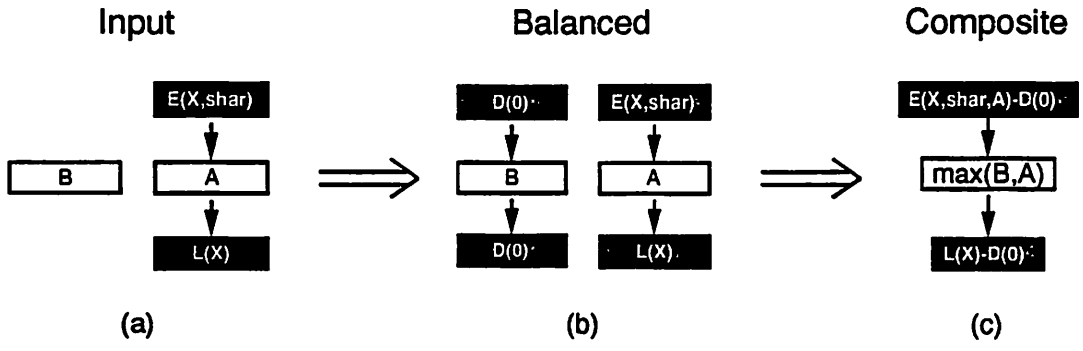


Figure 6.11. Simple Balancing Example

Figure 6.11a presents two input sequences which must be balanced before they can be combined. In the figure, the sequence consisting of a single time node with WCET B is balanced with the other by inserting vacuous suspension points matching the E and L suspension points to produce the balanced sequence illustrated in Figure 6.11b. The insertion of the vacuous suspension points is extremely important because it ensures that each of the input sequences will exhibit the same number of execution episodes. This is required to ensure that an ITG can be produced which correctly represents all the target behaviors of the block within which the input sequences appear.

Inserting a vacuous node before or after an existing TG node N implies the insertion of new RTL nodes before or after the subgraph(s) in the procedure's RTL represented by N. Note that node N may be *composite* as the result of a previous use of *combine_tg_sequences*. If so, then any insertion in the TG before or after N implies a corresponding insertion with respect to *every* RTL subgraph N represents. A vacuous time node represents RTL placeholder nodes which generate no assembler instructions. A vacuous suspension point represents RTL nodes which generate

assembler instructions calling the *suspend* system call, but imposes no minimum interval during which execution must be suspended. For this reason, vacuous suspension points are described as *delay*(0) suspension points. The balancing operation transforming Figure 6.11a into Figure 6.11b, for example, inserts a *delay*(0) suspension point before and after the subgraph represented by the time node with WCET B.

The balanced sequences are combined to produce the composite sequence of Figure 6.11c. The composite sequence notes the use of the resource X, and that the resource is required for a WCET of C. It also records the existence and RTL subgraph of the vacuous suspension points inserted in all of the input sequences to balance them. Including the vacuous nodes in the composite node's set of RTL subgraph information is crucial when subsequent balancing operations must insert other vacuous nodes before or after the composite suspension points in the sequence of Figure 6.11c. The reason is that any such insertion implies a corresponding insertion before or after *every* RTL subgraph represented by the suspension point, including the vacuous suspension points just inserted. The time node's WCET is the maximum of the two input time nodes it represents. The preservation of the resource use time C is important when $B > C$, since that information enables the scheduler to hold the resource for less than the WCET of the execution episode.

Figure 6.12 summarizes the *balance_sequences* algorithm. The first step analyzes the set of input sequences to determine the structure of the composite sequence capable of representing all of them. The composite sequence will be at least as long as the longest of the input sequences, and may be longer for reasons that are discussed shortly. The next step is to choose how the nodes of each of the input sequences should be mapped to those of the composite template. The search for a mapping in step 2 can currently either minimize the total WCET of the composite sequence's

```
balance_sequences (tg_node_t **sequences, int number, int *terms)
1. set up composite sequence template
2. select the best mapping of the input sequences to the composite
3. insert vacuous nodes in input sequences
4. allocate composite nodes
5. merge nodes of input sequences into composite
6. link composite as a sequence
7. free nodes of input sequences
end balance_sequences
```

Figure 6.12. Sequence Balancing Algorithm

execution episodes, or minimize the composite sequence's total elapsed time. The best mapping selected is thus the one that minimizes the selected value.

Selecting the mapping of the input sequence nodes to the nodes of the composite sequence is the most computationally expensive step, but it does not have an unreasonable impact on total compilation time for several reasons. First, the number of suspension points in a real-time program is comparatively small, and so the number and length of the sequences that must be balanced are usually modest. Another reason is that the algorithm searching for the best mapping employs a branch and bound strategy which is often able to avoid large sections of the search space. The number of mappings that must be evaluated is further constrained by properties of the input sequences. For example, a mapping which requires the insertion of a vacuous node within a section of an input sequence representing the behavior of a called procedure is not legal, since the transformation would require modifying the code of the called procedure. Other properties of the input sequences further constrain the mappings that must be considered. The search for the best mapping is, however, exhaustive and

several hundred mappings may be evaluated if the input sequences are particularly troublesome.

When the best mapping has been selected, vacuous nodes are added to the input sequences as required to balance them with the composite, and thus make them ready for combination with it. The nodes of the composite sequence are allocated at line 4, and then combined with the transformed input sequences at line 5. The resulting composite nodes are then linked into a sequence at line 6, and the nodes of the input sequences are discarded at line 7.

This completes the overview of the balancing algorithm. The rest of this section will illustrate each major step of the balancing algorithm of Figure 6.12 in greater detail using a running example which requires non-trivial processing at each step of the balancing algorithm. Figure 6.13 illustrates a set of four input sequences, which might be the clauses of a *switch* block. The reduction of the *switch* to linear ITG form requires that a composite sequence correctly representing all four input sequences be constructed.

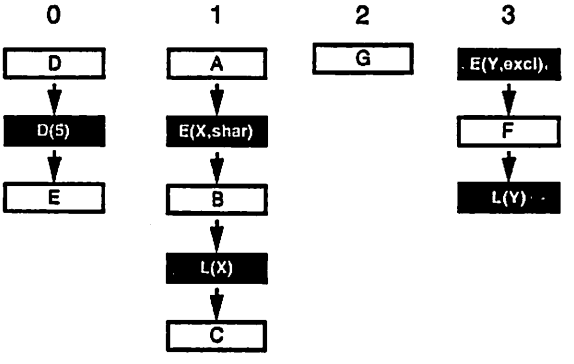


Figure 6.13. Sequence Combination Running Example

6.3.1.1 Setting Up the Composite Sequence Template

Figure 6.14 specifies the main steps in determining what the structure of the composite sequence which will represent the input sequences should be. The first step in building the composite sequence is establishing its structure, which is determined by the properties of the input sequences. The first three steps of *setup_templates* accomplish this. Step 1 links the nodes of the input sequences into a set of tables which makes them easily accessible during subsequent processing, and derives information about each node that constrains how it can be mapped to the composite template. Step 2 sorts the sequences in the table by length.

Figure 6.15 shows the sequence table produced. Note that the table is a two dimensional array, with each entry containing two elements. The first element is a pointer to the input node it represents, and is represented in the figure by the image of the node referenced. The second element is the *status* of the node, which includes whether a node may be mapped to the composite template independently of its predecessor or it is *DEPENDENT* (D) and *must* follow its successor directly, whether the node ends in a jump (J), and whether it is impossible to insert nodes into the sequence after the node. The *DEPENDENT* attribute constrains the mapping of a sequence's nodes to the composite template, as will be seen shortly. The *ENDS_IN_JUMP* and *CANNOT_INSERT_AFTER* attributes affect how vacuous nodes can be inserted into a sequence when it is being balanced, after the best mapping to the composite has been found.

In the example, the nodes marked as *DEPENDENT* have that attribute because resource use is currently constrained to not span task boundaries. This constraint arises from assumptions we currently make about the system scheduler, and could be changed as the scheduler evolves. If the mapping of the input sequence to the

setup_templates
1. allocate and initialize sequence tables
2. sort sequences by length
3. determine composite template from longest sequence(s)
4. allocate best and current composite mappings

end setup_templates

Figure 6.14. Composite Template Construction

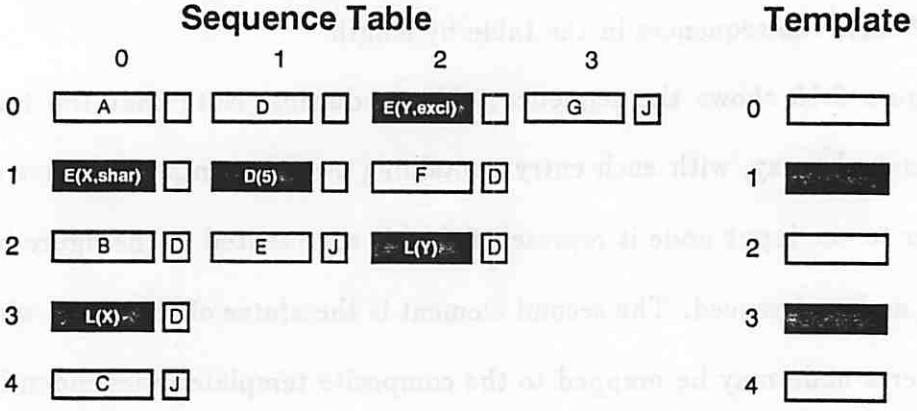


Figure 6.15. Sequence Table and Template Structure

composite required insertion of suspension points within the resource use block, that would mean the use of the resource spanned execution episodes and thus tasks, contradicting our assumption that resource use does not span task boundaries. Since we assume that the input sequences are cases of a *switch* block, all but one is marked as *ENDS_IN_JUMP*. Sequence 2 in the sorted table does *not* end in a jump under the assumption that it is the last case, and thus falls into the statement following the *switch* block.

Step 3 of *setup_templates* derives the structure of the composite template from the longest sequence or sequences in the input set. Figure 6.15 illustrates this with the

template which shows the *type* of node at each position, but where the node attributes are as yet unspecified. When the best mapping of the input sequences' nodes to the template is found the attributes of the composite template's nodes will be determined by the input nodes mapped to it. The longest input sequence generally defines the structure of the composite, but not always. The composite may be longer if there are two input sequences of the same length which are not already balanced. In that case, the composite must be one node longer than either to ensure that both may be mapped to it.

The composite may also be longer when the longest input sequence ends in a suspension point, but some other input sequence ends in a jump. Sequences generally end with jump instructions when they represent the bodies of loops or conditional clauses. This constrains composite template structure and how input nodes are mapped to it since it is meaningless to insert a node in an input sequence, and thus new nodes in the procedure's RTL, *after* a jump. Doing so would create dead code which is never executed. So, if the longest sequence ends in a suspension point, but another input sequence ends in a jump, the composite must be one longer and end in a time node to which time nodes representing jumps can be mapped.

After the composite template is established, working data structures are allocated for use during the search for the best mapping of input nodes to the nodes of the composite template. These data structures are not illustrated, as they simply list which nodes in each sequence are mapped to each node in the composite template. Two sets are allocated, one to record the best mapping encountered up to the current point in the search, and one specifying the mapping currently being evaluated.

6.3.1.2 Finding the Best Mapping

The best mapping of the input sequences to the composite template is found by searching the space of all possible mappings. The best mapping is the one producing the composite sequence with the minimum WCET or minimum elapsed time, depending on which metric is chosen to control the search. The WCET of the sequence is the sum of the WCETs of the time nodes in the sequence. The elapsed time adds the delays associated with suspension points to the WCET of the sequence. The compiler currently uses the WCET of the sequence by default but total elapsed time can be used by specifying a compiler command line option. The search is conducted by *iterate_sequence_mappings*, a simple branch and bound algorithm. The value of a time node in the template is the maximum WCET of all the input nodes mapped to it, while the value of a suspension point is the maximum delay associated with all suspension points mapped to it.

The search space is generated by permuting each sequence's mapping in the order in which the sequences appear in the sequence table. Figure 6.16 illustrates the search space generated for the four sequences in our running example. There are only 12 possible mappings because only sequences 1 and 3 have more than one mapping. The generation of the search space begins with mapping $(0,0)$ and proceeds across each row. After each mapping is evaluated, the *next_mapping* algorithm is used to determine the next mapping that requires evaluation.

Recall that the value of each node in the composite is determined by the nodes mapped to it. A mapping is thus evaluated by examining the input nodes mapped to each composite node, determining the maximum WCET of the time nodes, the maximum delay of the suspension points, and summing these values as required to

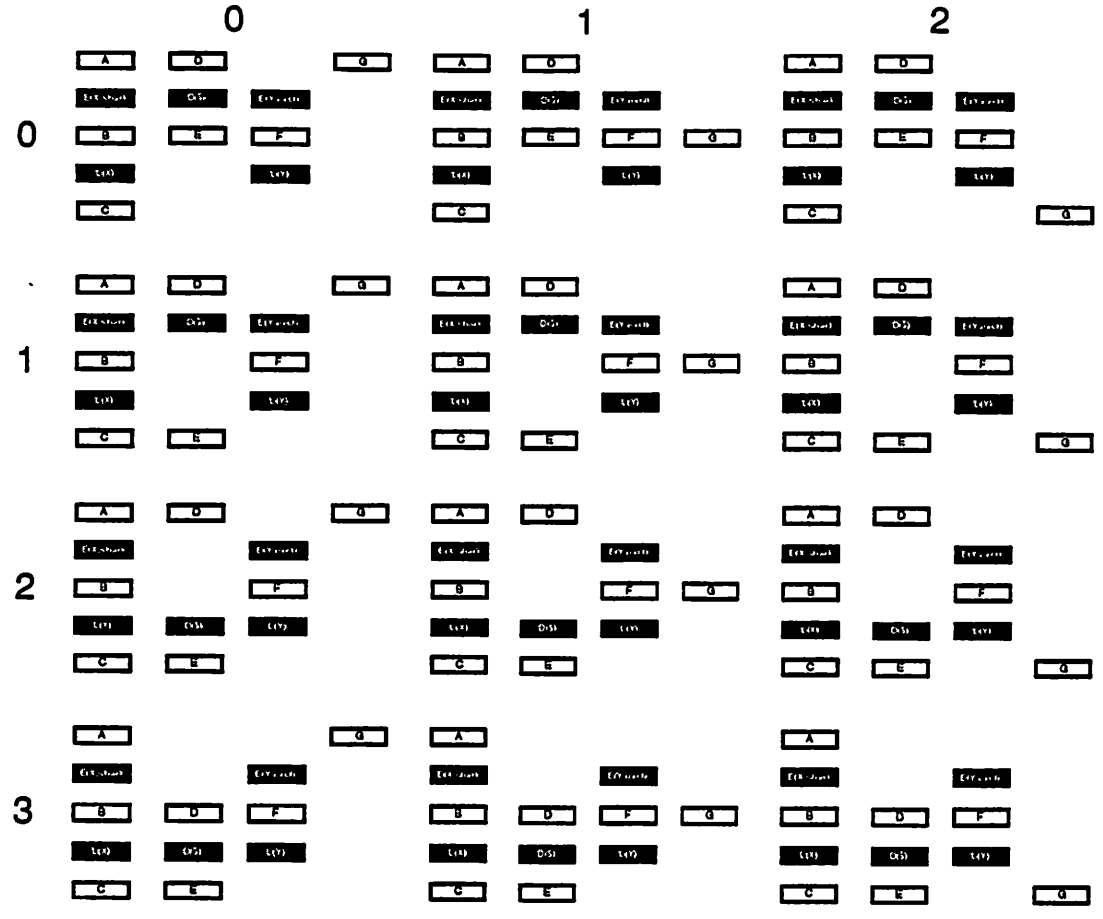


Figure 6.16. Sequence Mapping Search Space Example

determine the total WCET or the total delay. While not onerous, the cost of evaluating a mapping is not trivial, and searching the set of all possible mappings for the one with the best value is potentially time consuming. There are, however, two factors constraining the number of mappings which must be evaluated.

First, recall that when *setup_templates* created the sequence table and determined the structure of the composite sequence template, it also noted those with the *DEPENDENT* property, which states that a node *must* immediately follow the node before it in any mapping to the composite. A node has this property, for example, when it and the node before it represent part of the behavior of a procedure called by the one whose TG is being reduced. The node is *DEPENDENT* because a mapping

that separates such a node from the one preceding it implies the insertion of a vacuous node into the body of the called procedure, which would change its behavior for all other procedures calling it as well.

So, for example, if sequence 1 in Figure 6.15 represents the behavior of a called procedure, then each node would be *DEPENDENT*. When the evaluation of mapping $(0,2)$ was complete *next_mapping* would consider the mapping of sequence 1 to the template represented by rows 1 and 2 of Figure 6.16, but realize that they are illegal because they imply a transformation of the called procedure. It would instead move directly to mapping $(3,0)$. The *DEPENDENT* constraint on nodes 2 and 3 of sequence 1 would thus eliminate half of the search space.

The other factor considered by *next_mapping* which can prune significant portions of the search space arises when the change in position of a time node from the current mapping to the next is *certain* to have no effect on the value of the mapping. For example, assume that E is less than both B and C. In that case, when *next_mapping* considers moving from mapping $(0,2)$ to $(1,0)$ it would notice that changing the mapping of E, node number 2 in sequence 1, would not change the value of either node 2 or node 4 in the composite. As a result, it would move directly to mapping $(2,0)$ and avoid having to evaluate the mappings of row 1.

When *next_mapping* determines that no additional legal or significant mappings exist, then the best mapping found so far is the the best possible.

6.3.1.3 Vacuous Node Insertion

When the best mapping has been found, vacuous nodes must be inserted into each input sequence where the mapping of its nodes to the composite template has left gaps. It is important to note that when a vacuous node is inserted before or after a

node N in an input sequence, corresponding insertions are made before or after *every* RTL subgraph in the procedure's RTL represented by N. When the input sequences are the product of several previous reductions each insertion in the TG can thus engender the insertion of several vacuous RTL nodes. When the insertions required are complete, each input sequence matches the length and structure of the composite, and is ready to be merged with it.

For example, assume that mapping (1,2) of Figure 6.16 is selected as the best mapping and consider the TG and RTL insertions this implies for sequence 1. Nodes 0 and 1 of sequence 1 are mapped to positions 0 and 1 of the composite, but node 2 is mapped to position 4 of the composite. This leaves gaps at positions 2 and 3 of sequence 1 which must be filled with vacuous nodes. A vacuous time node is inserted at position 2 and a vacuous suspension point at position 3. If the nodes of sequence 1 each represent a single RTL subgraph, then it should be easy to see that RTL nodes implementing a vacuous time node and a vacuous suspension point are inserted between the RTL subgraph represented by the nodes 2 and three of the input sequence. When the insertions are complete sequence 1 matches the structure of the composite and is ready to be merged with it.

Nodes can be inserted either before or after existing nodes, with some restrictions, as noted by the *status* element of each entry in the sequence table. How the *DEPENDENT* property constrains where an input node may be mapped into the composite sequence was discussed in the previous section. The two other properties constrain how vacuous nodes may be inserted in the input sequence. The *CANNOT_INSERT_AFTER* (CIA) property indicates that it is not permitted to insert a node after the one which has the property. It is, however, always possible to insert *before* the next node. The CIA constraint applies, for example, to nodes representing

the conditional block expressions of the *if* and loop blocks, and the value expression for the *switch* block.

Conditional expressions in Spring-C, as in C, can have a number of clauses which are evaluated from left to right. Depending on the structure of the conditional expression its value, zero or non-zero, may be known before all clauses of the conditional are evaluated. In such cases the RTL emitted for conditional expressions jumps directly to the true or false clause of an *if* without passing through the end of the conditional expression. Since the code emitted for the conditional can transfer control directly to one of the conditional's clauses, code represented by nodes inserted after the node representing the conditional will not always be executed. Since the node is inserted to balance a sequence by *executing* the inserted code, the reason for the CIA constraint should be clear.

Similarly, when the value used to choose a *switch* case is known a decision tree may be used to select the appropriate *case*, jumping directly to it without passing through the end of the value expression. As a result, inserting *after* the TG node representing the conditional or value expression is useless, since the inserted nodes would not be executed. However, inserting *before* the node representing the destinations of the jumps is always possible, because the inserted RTL is guaranteed to be placed *after* the labels which are the destinations of the jumps. It is also important to note that the CIA property only constrains vacuous node insertion when the *if* or *switch* block does not reduce to a single time node.

Another property of a time node considered during the insertion of vacuous nodes is *ENDS_IN_JUMP*. This indicates that the time node represents an RTL subgraph ending in an unconditional jump, and that inserting after the node would thus be useless. However, the balancing operation might have selected a mapping which

appears to require such an insertion. For example, if mapping $(0,0)$ of Figure 6.16 was chosen, this would imply the insertion of several nodes after the original time node of sequence 3 which is mapped to position 0 of the composite. This time node does, since by our assumption when starting the running example it represents a *switch* case, so inserting a sequence of nodes after it is useless because they will not be executed. On the other hand, requiring that such a node *always* be mapped to the last node of the composite sequence is unnecessarily restrictive.

Instead, when vacuous nodes are being inserted after a time node with the *END_IN_JUMP* property, the time node is *split* by taking the RTL representing the jump from the end and creating a new time node representing just the jump. The new time node is mapped to the time node terminating the composite sequence, and insertion of vacuous nodes continues for other positions in the sequence requiring it. The rational for splitting the time node is that the change in the WCET of the node losing the jump is minimal, and thus unlikely to invalidate the status of the mapping chosen as best. If it does, then the difference will be minimal. Similarly, the addition of the node representing the jump to the set mapped to the last time node in the composite is unlikely to change the value of the mapping, but if it does the change will be minimal.

When vacuous nodes have been inserted in each input sequence as required by the mapping chosen to make them match the composite, then the modified input sequences are ready for merging into the composite.

6.3.1.4 Merging Input Nodes into the Composite

Merging the input sequences into the composite is comparatively simple once they have been balanced. Figure 6.17a illustrates the set of input sequences from Figure

6.15 balanced according to mapping (1,1) of Figure 6.16. Note that vacuous time nodes and suspension points have been inserted as required by the mapping, and the original time node of sequence 3 has been split into nodes with WCET of G' and G'' respectively. The node with WCET G'' represents the jump, and the node with WCET G' represents the rest of the original time node.

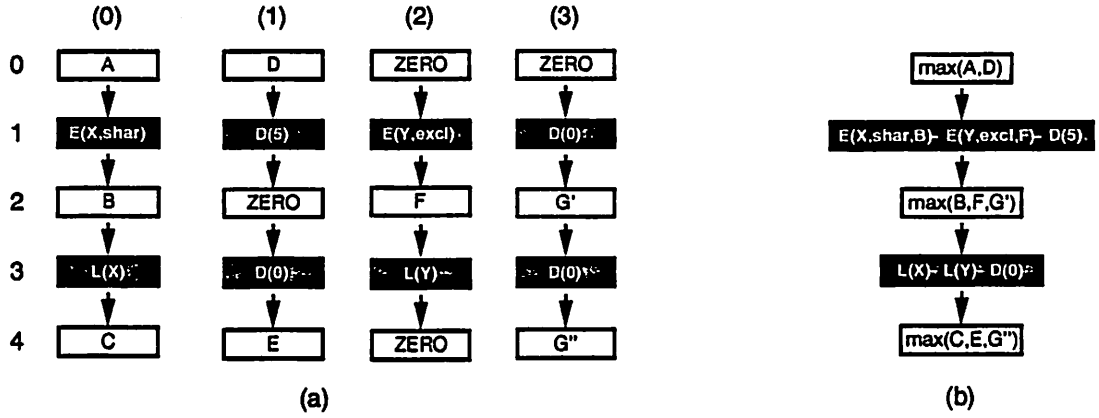


Figure 6.17. Merging Balanced Sequences

Each of the nodes in the transformed input sequences are merged into the corresponding node of the composite sequence. Figure 6.17b presents the composite sequence produced. Each of the composite time nodes has a WCET equal to the maximum WCET of the input nodes it represents, and the composite suspension points have the union of the properties of the input nodes they represent. So, for example, the second node in the composite sequence represents entry into both of the resource use regions, as well as the *delay*(5) and the vacuous *delay*(0). The *delay*(5) subsumes the *delay*(0) in the composite node.

The other important aspect of composite nodes is that their RTL subgraphs represent the union of the RTL subgraphs of the input nodes mapped to it. This is significant when the newly constructed composite sequence becomes an input sequence in a subsequent balancing operation, and a vacuous node must be inserted before

or after one of its composite nodes. The RTL subgraph information is then used to perform the insertion required with respect to every subgraph in the procedure's RTL represented by the composite node.

The merging of balanced input sequences into a composite sequence completes the combining operation, and makes the composite available for use in the reduction of the block within which the input sequences appeared.

6.3.2 Conditional Subgraph Reduction

The basic rules for reducing the *if* and *switch* conditional blocks were presented in Section 6.1. Extending these rules to handle conditional blocks containing suspension points is easy, but subject to some restrictions. The basic approach is simple: the components of the conditional are reduced to ITG form, the set of parallel clauses are combined using the methods just described, the ITG of the conditional or value expression is concatenated with the composite sequence, and the reduction completed by reducing the concatenated sequence.

For example, Figure 6.18 illustrates the extended method with two *if* blocks which reduce to significantly different ITGs. Note that the time node representing the conditional expression has the CIA property when created. As discussed earlier, the reason for this is that the code implementing the conditional may contain several jump statements transferring control to one of the conditional's clauses and that nodes inserted after this node would not always be executed. Figure 6.18a illustrates an *if* block for which the CIA property of the time node with WCET A becomes irrelevant when it is merged with the time node at the beginning of the composite sequence. Any subsequent balancing operations will be able to insert before or after the merged time node without restriction. Figure 6.18b, however, shows that the

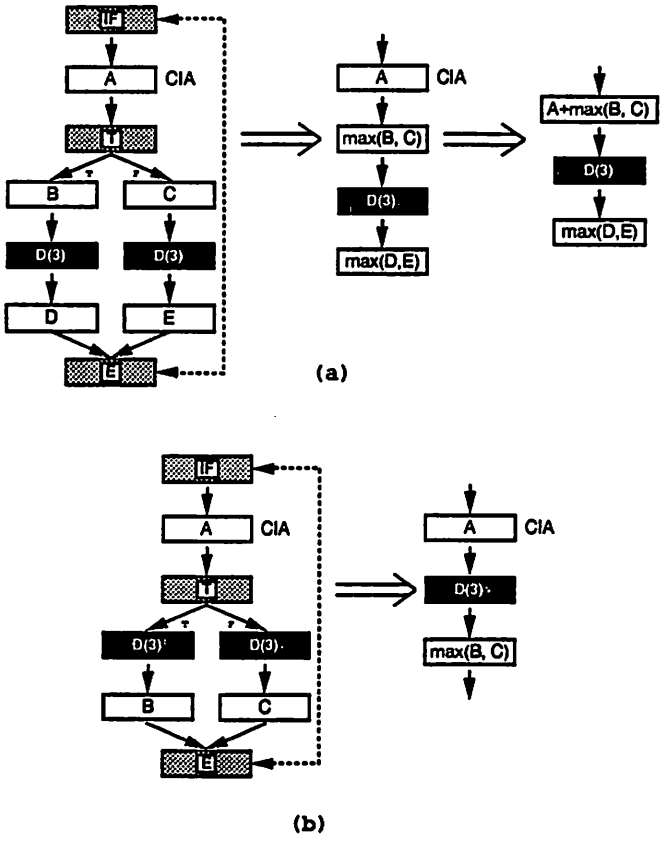


Figure 6.18. If Block Reduction Examples

CIA property is not always eliminated. When the composite sequence produced by the clauses of the conditional begins with a suspension point, then the time node with the CIA property is present in the ITG produced by the conditional block. Subsequent balancing operations wishing to insert vacuous nodes between the time node with WCET A and the suspension point with $Delay(3)$ will have to insert *before* the suspension point, rather than after the time node.

This explains the essential aspects of reducing conditional blocks containing suspension points. The current implementation, however, imposes two restrictions on where suspension points may appear that should be noted. The first is that neither the conditional expression of the *if* block nor the value expression of the *switch* block

may contain suspension points. The compiler notes any violation and emits an appropriate warning. In the case of the *if*, the need for the restriction arises because each clause of the conditional is evaluated from left to right. If the value of the expression as a whole is determined before every clause has been evaluated, the emitted code jumps directly to the true or false clause as appropriate, creating more than one path through the conditional expression. If one or more clauses of the conditional contain suspension points, then each of the possible paths through the conditional must be balanced. However, the RTL for the conditional is not currently emitted in a form supporting the balancing operation. The problem with the *switch* block's value expression is similar, in that the code emitted does not lend itself to balancing and thus proper reduction.

The other restriction on the *switch* statement is slightly less trivial: suspension points may not appear in cases which fall into others, nor in cases which are fallen into. The compiler enforces this, emitting a warning when it is violated. The reason for the restriction is that the code implementing the case which is fallen into is *shared* with that of the case falling into it. That is the whole purpose of permitting the practice in conventional programs, but it creates a contradiction when balancing is required. For example, consider two *switch* cases, A and B, where A falls into B and both cases contain a suspension point. The case A is thus represented by a sequence containing two suspension points, while B is represented by a sequence containing one. Assume, that case A is the longest sequence and establishes the structure of the composite template. When the sequence representing case B is balanced against the composite, a suspension point will be inserted. However, since the code for case B is shared with case A, this also adds a suspension point to the code for case A which now *actually* contains 3 suspension points, although the sequence representing it still

contains only 2. The physical sharing of the code for case B with case A makes it impossible to balance the clauses of the *switch*.

In general these restrictions impose only a modest cost on the developer, since the Spring-C source can always be rewritten, with minimal effort, to avoid them. In light of their moderate cost, we have thus far deferred modifying the compiler to eliminate these restrictions in favor of other implementation tasks. It is, however, clearly possible to do so.

6.3.3 Loop Subgraph Reduction

Under the loop reductions discussed in Sections 6.1 and 6.2 a loop executing fewer iterations than its upper bound specified required no special consideration. Fewer iterations cause the execution episode containing the loop to use less than its WCET, but this can occur for any number of reasons. When the loop contains a suspension point, however, variations in the number of iterations result in variations in the number of execution episodes exhibited by the procedure.

Since we make the basic assumption that each execution episode is represented as a task, a new type of reduction is required to ensure that all possible behaviors of a loop containing a suspension point are properly represented and handled. The basic idea is that the *behavior* of the loop is "unrolled" by concatenating copies of an ITG representing each iteration. The number of copies concatenated is determined by the loop's upper iteration bound. Since the loop may execute for fewer than the upper bound iterations we must also add code to the exit portion of the loop, the "SKIP" code, to handle this.

A simple example illustrates the problem. Consider a loop containing a single suspension point which means that each iteration of the loop creates a new execution

episode. Further assume that the loop is immediately followed by a resource use block using resource X in exclusive mode. Finally, assume that the loop executes one iteration less than its upper bound. Then, if no measures are taken to enable the system to account for this situation, when the dispatcher sees that it is time for the task representing the last iteration of the loop to execute, and changes context back to the process, the process will *actually* be executing the block of code exiting the loop. Then when the dispatcher changes context to the process to execute the exit from the loop, it will be executing the code block using resource X in exclusive mode. Yet, it is the *next* task in the execution plan, not the current one, to which the use of resource X has been allocated. This obviously violates the assumptions made by the scheduler when it constructed the execution plan, and invalidates every aspect of the guarantees provided by the scheduler.

```
if ( lc < UB ) {
  int suspensions_skipped;

  suspensions_skipped = ((UB - lc)*SUSPENSIONS_PER_ITERATION)-1;
  task_skip(suspensions_skipped);
}
```

Figure 6.19. Additional Loop Exit Code

The solution is to have the exit code of the loop check to to see how many iterations less than the worst case a particular execution of the loop has taken. This is used to calculate the number of execution episodes, and thus tasks, the loop has *skipped* in the representation used to schedule its execution. Recall that the normal loop exit code, as discussed in Section 6.1, checked the loop iteration counter *lc* against the upper and lower iteration bounds, calling *bound_violation* if the iteration count fell outside the proper range. Since a loop containing a suspension point represents a

variable number of execution episodes, an additional test on *lc* as illustrated in Figure 6.19 is required to maintain the correspondence between execution episodes and tasks assumed by the scheduler when constructing the execution plan.

The number of execution episodes, and thus tasks, skipped by a particular execution of the loop is calculated from the number of suspension points in the body of the loop and the number of iterations skipped. The result is decremented to account for the suspension that the call to *task_skip* will impose on the process P currently executing. The argument to *task_skip* tells the dispatcher how many tasks associated with process P to skip. This ensures that when the dispatcher next transfers control to the context of P, it will do so at the time allotted for the task representing the execution episode following the last iteration of the loop, and not before.

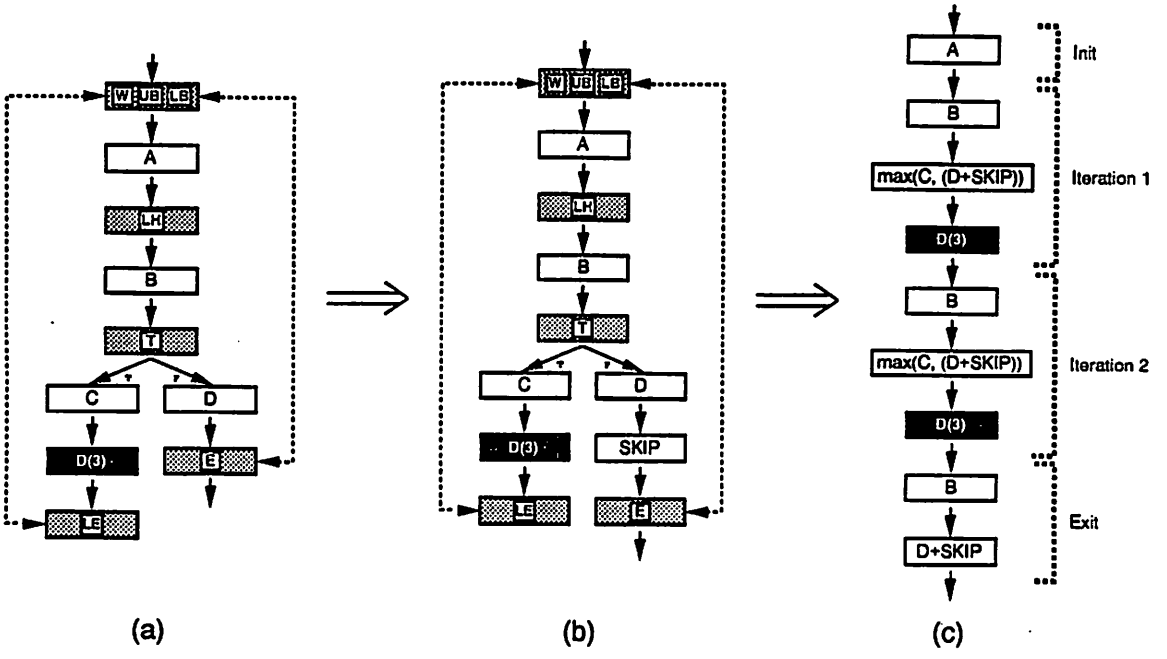


Figure 6.20. Reduction of a While Loop with Suspension Points

Figure 6.20 gives an example of the method. Figure 6.20a illustrates a *while* loop containing a suspension point after the *reduce_while* routine has reduced each of its components to ITG form. The reductions for *do-while* and *for* loops containing

suspension points are similar. When the presence of a suspension point in the body of the loop is detected the "SKIP" code discussed earlier is added to the exit code of the loop, as shown in Figure 6.20b. The WCET of the exit code will thus become $D + SKIP$. The behavior of the loop is "unrolled" by creating a sequence of nodes representing its behavior when it executes the upper bound number of iterations, as illustrated in Figure 6.20c assuming an upper iteration bound of 2. Note that the sequence begins with the node representing loop initialization which is executed only once. This is followed by the set of sequences representing each iteration, and is terminated with the sequence representing the last execution of the conditional expression and the exit code. The reduction of the loop block would be completed by applying *reduce_item_sequence* to the sequence, thus merging all consecutive time nodes.

Skipping tasks creates the opportunity for a new form of resource reclaiming since the skipped tasks could be marked idle in the schedule. When the dispatcher encountered such an idle task, it could use the task's allotted time to execute non-real-time processes, or it might choose to act as if the task had already completed, and reclaim the resources[77]. The fact that tasks can be marked as idle before they execute also makes it possible for the *scheduler* to reclaim the resources allotted to them by creating a new schedule which eliminates the idle tasks.

6.4 Extending the Current Implementation

The current implementation cannot reduce blocks that contain *both* structured jumps and suspension points. This restriction is, however, a result of the incremental strategy adopted for Spring-C compiler development rather than a restriction inherent in the analysis and compilation methods described in this chapter. On the contrary,

the methods for reducing blocks containing structured jumps and those for reducing blocks containing suspension points were carefully designed to permit graceful extension to include the reduction of blocks containing both. This section describes the nature of those extensions.

To understand the extensions we must first review the most important aspects of the current reduction algorithms handling blocks with structured jumps and suspension points. Recall that, as described in Section 6.2, a structured jump creates an alternative path through its context block. So, for example, a *continue* statement represents an alternative path through its context block, the body of the loop within which it appears. When the body of the loop is being reduced the *continue*, no matter how deeply nested, is encountered and the required information about the alternative path it represents is recorded.

Recall also that, since the context blocks of structured jumps are currently allowed to contain no suspension points, the only information required to describe the alternative path is its WCET. This is easily calculated using the *wcet_so_far* parameter added to all the block reduction algorithms when extending the basic reduction rules, as described in Section 6.1. Each alternative path is thus characterized by a single time node with a value reflecting the WCET from the beginning of the context block, the loop body in this example, to the structured jump. Finally, it is also important to recall that the beginning of the RTL subgraph represented by the alternative path's time node is shared with the main path through the context block, as well as with all the other alternative paths through the loop, since every path through the loop body begins with the first instruction of the loop body. This fact is recorded by giving the begin point of the RTL subgraph of the alternative path's time node *SHARED* status.

When the components of the loop are reduced to ITG form the alternative paths created by all the structured jumps must be resolved with the ITG derived from the body of the loop before the loop as a whole can be reduced. The *combine_tg_sequences* algorithm is used for this, but without suspension points, the loop body and alternative paths are all single time nodes, which is a simple special case. A single composite time node is produced which represents at least two RTL subgraphs. Each subgraph has a common begin point, the shared beginning of the loop body, but many end points, as defined by the end of the loop body and all the structured jumps. Resolving the alternative paths with each other and with the body of the loop was discussed in Section 6.2.2.

When suspension points appear in a block, they represent information that cannot be eliminated during subgraph reduction. Section 6.3 described in detail the issues arising from the reduction of blocks containing suspension points. Some particularly important points concerning how ITGs are combined, as described in Section 6.3.1, should be emphasized. Recall that ITGs being combined often have to be balanced before they can be combined. This requires that the structure of the composite sequence be determined, and that a legal mapping of every node in the existing sequences to a node in the composite sequence be found.

It is particularly important to remember that several properties of the nodes in an ITG constrain the set of legal mappings. For example, recall that some nodes' RTL subgraph endpoints can have the *WITHIN_PROC* status indicating they represent behavior of a called procedure. Mappings for the ITGs containing these nodes must satisfy the constraint that no vacuous nodes may be inserted between nodes representing behavior inside the procedure, since that would imply modifying the procedure's RTL, which is not available. When the best legal mapping is found, each

ITG is balanced with the composite. This requires the insertion of vacuous time and suspension point nodes in the ITGs being balanced, with corresponding insertion of new code in the proper places in the RTL of the procedure, as specified by the RTL subgraphs of the nodes in the ITG before or after which vacuous nodes are inserted.

The extensions required to handle blocks containing both structured jumps and suspension points are most easily understood as an application of the methods for combining ITGs, as described in Section 6.3.1, to the problem of alternative path resolution discussed in Section 6.2.2. The presence of suspension points also complicates the calculation of the alternative paths which will be combined, requiring us to extend the methods for calculating alternative paths described in Section 6.2.1.

The extension of alternative path calculation to handle suspension points is fairly simple, when viewed as a more general version of the extension of the reductions described in Section 6.1 to create those described in Section 6.2 which handle structured jumps. Recall that the alternative path's WCET was calculated using the *wcet_so_far* values. Each reduction routine used *wcet_so_far* to track the WCET from the beginning of the procedure to the reduction boundary, passing the current value on to reduction routines it called to handle nested blocks. As each block providing context for a structured jump was entered, the current value of *wcet_so_far* was pushed on the appropriate context stack or stacks.

Then, when a structured jump was encountered, the WCET of the alternative path was simply the difference between the current value of *wcet_so_far* and the value that was on the top of the structured jump's context stack. An integer was sufficient to track the information required since no suspension points were permitted to appear between the beginning of the context block and the structured jump. Now that we must extend the method so that suspension points *may* appear along the path between

the beginning of the context block and the structured jump, an integer is no longer sufficient.

Instead, the reduction routines must be modified to maintain a sequence of TG nodes describing the path from the beginning of the procedure to the reduction boundary. A stack will serve the purpose, which we will call the *alternative path stack*, or AP-stack. Each reduction routine will push a node on the AP-stack as the reduction boundary is moved forward. Then, as each reduction routine enters a structured jump context block, instead of pushing the current value of *wcet_so_far* on the context stack, it pushes a pointer to the top of the AP-stack. Some of the details become somewhat involved, particularly those concerning nodes that must be popped from the AP-stack as different components of a block are reduced, as well as when the final reduction of a block is made, but they present no great difficulty.

The important point is that when a structured jump is encountered, an ITG describing the alternative path from the beginning of the context block to the structured jump must and can be constructed. The sequence of nodes on the AP-stack which lie between the pointer pushed on the structured jump context stack and its current top provide the information required to do this. That section of the AP-stack is not always in ITG form because each reduction routine pushes nodes onto it independently, and it can thus contain two or more consecutive time nodes. It is easy, however, to construct the proper ITG from the available information.

Now consider the RTL subgraph status of the nodes in the newly constructed ITG. The new ITG clearly begins at a point which is *shared* with the ITG being constructed for the body of the context block, as it is shared with every other alternative path. The RTL subgraph begin point of the new ITG will clearly have *SHARED* status. It is equally clear that the last node in the ITG represents an RTL subgraph

that ends with the structured jump, and which is thus unique to the alternative path being constructed. This status is called *MAIN_LEVEL* since the RTL subgraph represents the main level of the procedure code being reduced, as opposed to being *WITHIN_PROC*, for example. Somewhere between the beginning and the end of the new ITG representing the alternative path there will clearly be a point at which the status changes from *SHARED* to *MAIN_LEVEL*.

The point where the status changes is fairly easily defined, when the structure of the source code and execution paths through it are considered. For example, take a loop containing a single *continue*. While it is possible for the *continue* to appear at the main level of the loop body, this is a trivial case. The only structure that makes sense is for it to appear within the clause of a conditional, however deeply nested. The ITG produced for the body of the loop will take *every* path through the code of the body into account *except* for the one ending with the *continue*. Further, the path from the beginning of the loop body to the *continue* clearly has a prefix which is shared with at least one path through the body which will be represented by the ITG produced for the loop body. This prefix ends at the beginning of the conditional block clause containing the *continue*. The ITG representing the alternative path is added to the list of alternatives associated with the context block.

It is slightly more difficult to ensure that the nodes in the ITG produced for the body of the loop which correspond to the shared prefix of the alternative path will also have the *SHARED* status, but it can be done by using the nodes on the AP-stack. When the shared prefix of the alternative path is identified, the corresponding nodes on the AP-Stack must be marked as shared. This information is then used by routines reducing the blocks within which which the *continue* was nested to properly set the RTL subgraph status of the nodes in the ITGs they produce. This is important

because of the constraint on ITG balancing represented by the *SHARED* status. It is similar to *WITHIN_PROC* in placing a restriction on whether vacuous nodes may be inserted between nodes that are shared. The reason being that if such an insertion is made in *one* ITG, it implies a similar insertion in *all* the ITGs sharing those nodes. The constraint is slightly different from *WITHIN_PROC* since such an insertion *is* permitted if *all* the sequences sharing the prefix are similarly transformed.

When the context block is reduced, its ITG is *combined* with those of the alternative paths, using the methods already described in Section 6.3.1, but slightly modified to handle the mapping constraints implied the *SHARED* status. Note that when balancing is required, it will always be possible since every ITG will have a suffix which is not *SHARED*. Balancing can always be done by inserting nodes in the suffix.

While significant work obviously remains before the reductions handling blocks containing both suspension points and structured jumps are implemented, it should be obvious after this discussion that such reductions are possible, and that the methods already implemented provide an excellent foundation for extension to include the new case.

6.5 Properties of Subgraph Reduction

This section briefly discusses why all Spring-C programs, written according to practices and restrictions specified, will reduce to the standard linear ITG form of alternating time nodes and suspension points. It also presents an argument that the computational complexity of the reduction phase which transforms the original TG of a procedure to linear ITG form is, in general, no greater than that of compilation. Recall that the compiler performs the reduction phase to limit the number of paths

through the TG that must be considered when building the task group representation of the process's behavior as described in Chapter 7.

The reductions described in Sections 6.1, 6.2, and 6.3 specify the set of simplifying operations currently available, whose application is controlled by the recursive descent strategy introduced in Section 6.1, and described in detail in Appendix B. The set of reductions must be complete, in the sense of handling all the structures a Spring-C program can present, to ensure that the TG of every Spring-C program will reduce to linear ITG form, as required.

The completeness of the reductions can be seen by considering the correspondence between the reduction rules and the Spring-C block constructs. Spring-C programs are built by specifying a series of block structured statements, which may have other blocks nested within them. For programs not containing any suspension points, the set of basic reductions contains a rule for every block type in Spring-C, so all blocks will reduce to single time nodes. This assertion is supported by the compiler reduction test suite, discussed in Section 6.6.1, which includes examples of every Spring-C statement and block type.

Each reduction rule simplifies each of its block's components before performing the final reduction. During the reduction of each component, the recursive descent aspect of *reduce_item_sequence* will find and reduce the nested blocks. The depth of the recursive descent will always be finite since it is limited by the nesting depth of the source of the procedure whose TG is being reduced. Procedures called within the code of a procedure do not add to the nesting depth since their ITGs, which are always linear, are substituted for the calls. Spring-C programs without suspension points or structured jumps will thus reduce to a linear ITG, a single time node in this

case, in a finite number of subgraph reductions. Specifically, one reduction for each block structure in the procedure's source code.

Structured jumps within the procedure do not change the number of reductions performed, but create additional processing associated with each structured jump in the procedure. When a structured jump is discovered, a record of the alternative path it represents is added to the appropriate list associated with the context block. When the components of the context block have been reduced, the alternative paths are resolved with the main path by the *combine_tg_sequences* algorithm, and the context block as a whole is reduced. Since blocks containing structured jumps will *not* contain suspension points under the current programming restrictions, resolving the alternative paths is simply a matter of finding the maximum WCET, and accumulating a list of the RTL subgraphs of the time nodes being combined. The result is always a single time node, and all blocks containing structured jumps will thus reduce to a linear ITG.

The computational complexity of subgraph reduction for procedures without suspension points is easy to see. Each reduction rule reduces the components of its block to ITG form before reducing the block as a whole. When the block contains no nested blocks, the number of operations required to reduce the components is zero, since each component of the block consists of a single time node. Reducing the block as a whole requires $O(1)$ operations, since it combines a constant number of block components. When nested blocks are present, each component requires $O(n)$ operations, where n is the number of nested blocks. Here n is clearly less than the number of Spring-C statements used to describe the components of the block, since even in the worst case every Spring-C statement in the source cannot be a nested block.

When structured jumps, but not suspension points, are present the generation of an alternative path by each structured jump statement requires $O(1)$ operations. When the context block as a whole is reduced, reconciling the alternative paths with the body of the block requires $O(1)$ additional steps for each structured jump. Reducing a block containing structured jumps thus requires no more than $O(n)$ steps, where n is the number of Spring-C statements contained within the block, since every Spring-C statement cannot be a structured jump. Reducing programs without suspension points thus requires $O(n)$ operations, where n is the number of Spring-C statements in the program source.

The presence of suspension points within the procedure does not change the number of block reductions performed, but requires additional processing when the components of the block containing a suspension point have been reduced to ITG form. When the block is a conditional, an *if* or *switch*, then *combine_tg_sequences* is used to create a composite ITG representing the input sequences. The structure of the composite is determined, the best mapping of the input sequences to the composite is found, the input sequences balanced with it by inserting vacuous time and suspension point nodes, and then the input sequences are combined to form the composite. Once the composite is available, it is trivial to produce an ITG for the conditional.

This procedure produces a linear ITG representing the conditional block. The search for the best mapping may be lengthy, but is clearly finite and will succeed since there is at least one legal mapping for every input sequence. It is also important to note that the compiler does not *have* to search for the *best* set of sequence mappings. It could select a *legal* set without having to search at all.

When the mapping of each input sequence to the composite is established, insertion of nodes into the input sequences may have to be done *before* some nodes rather

than *after* others, but it is always possible to insert the required nodes. Combining the input sequences into the composite thus cannot fail to produce a linear ITG. When available, the composite ITG is appended to the ITG of the *if* conditional expression or *switch* value expression, which it then reduces using *reduce_item_sequence*. This is also certain to produce a linear ITG.

In the case of loops, the components of the loop are reduced to linear ITGs and then the behavior of the loop is "unrolled" into a linear structure. The *reduce_item_sequence* algorithm reduces this to the standard linear ITG.

All procedures containing suspension points, subject to the restriction that the suspension points not appear within the context block of a structured jump, will thus reduce to the standard linear ITG form required for task group construction, as described in Chapter 7. When the reduction methods are extended as described in Section 6.4, then all procedures will reduce to linear form, without restriction.

The computational complexity of reducing conditional statements containing suspension points is determined by the complexity of the *combine_tg_sequences* operation. Recall that the combining operation has several phases. Determining the composite template length sorts the input sequences by length, which requires $O(k \log k)$ operations where k is the number of input sequences. It also requires $O(l)$ operations to form the composite, where l is the number of nodes in the longest sequence.

Finding the best mapping requires an exponential ($O(2^m)$) number of operations since, in the worst case, it considers every possible mapping of each sequence to the composite template. The exact number of possible mappings is a complex function of the number of input sequences and the number of possible mappings of each sequence to the composite. The number of mappings actually examined is usually far fewer, in practice, given the constraints on legal mappings and the use of branch and bound

techniques discussed in Section 6.3.1.2. The combining operation requires $O(n)$ operations, where n is the total number of nodes in the sequences being combined, and is thus less than the number of Spring-C statements in the source for the conditional being reduced.

As a whole, then, the current implementation requires an exponential number of operations to reduce a conditional containing suspension points, because of the exhaustive search for the best mapping of the input sequences to the composite. This is not surprising since many, if not most, optimization algorithms are exponential. It is important to note, however, that searching for the *best* mapping is not required. A *legal* mapping can be created in $O(n)$ operations, where n is the total number of nodes in the input sequences. The number of operations devoted to the search for a better mapping can be as large or small as desired. The complexity of a legal reduction of a conditional statement is thus $O(n \log n)$ in the number of input sequences, the number of clauses in the conditional statement, and $O(n)$ in the number of Spring-C source statements it contains. The latter will predominate for most conditionals.

The complexity of reducing loops which contain suspension points is determined by the need to "unroll" its behavior, once the components of the loop have been reduced to linear ITG form. The unrolling operation requires $O(n * m)$ operations, where n is the number of nodes in the ITG of the loop conditional and loop body, and m is the upper iteration bound of the loop. This should be clear, since the unrolling operation creates new copies of the ITG representing each iteration.

In summary, the complexity of reducing the TG of a procedure is $O(n)$, where n is the number of Spring-C statements in the source for the procedure, subject to several assumptions. First, that when balancing is required the search for a better mapping of input sequences to the composite is limited. Second, that the $O(n*m)$ term generated

by unrolling loops does not predominate. While programs violating this assumption can certainly be written, most Spring-C programs generating a task group which can be scheduled will not. Third, that the $O(n \log n)$ operations required to sort the input sequences of a conditional for balancing does not predominate. Again, while it is probably possible to construct a Spring-C program violating this assumption, most will not.

If these reasonable assumptions are fulfilled, the $O(n)$ value demonstrates that the subgraph reduction phase is of no greater complexity than the compilation of the program, since n is proportional to the number of Spring-C statements in the program source.

6.6 Evaluation

The current implementation of the Spring-C compiler reflects the categories of subgraph reduction described in Sections 6.1, 6.2, and 6.3. It handles all Spring-C programs without structured jumps or suspension points, those with blocks containing structured jumps, and those with blocks containing suspension points. It is not yet able to reduce blocks which contain *both* suspension points and structured jumps, nor is it able to reduce recursive procedures. These are reasonably straightforward extensions of the methods already discussed. Section 6.4 discusses how the current implementation will be extended to handle blocks containing *both* suspension points and structured jumps.

This section discusses how the current implementation has been evaluated. The evaluations performed fall into two categories: reduction tests and prediction accuracy tests. The reduction tests determine whether the compiler can successfully reduce the TGs representing a wide range of programming scenarios and software structures to

ITG form. Some reduction tests were explicitly written to test the reduction of TGs arising from particular software structures, while others test the compiler's ability to process real application code. The prediction accuracy tests are small programs specifically written to test the accuracy with which the compiler predicts the WCET of specific language features, particularly loops and procedure calls. It is important to note that several of these tests include code from robotics application programs.

6.6.1 Reduction Test Suite

The first level of reduction testing is done by a test suite in which each test focuses on the reduction of a particular Spring-C structure, block type, or programming scenario. For example, the set of tests addressing the reduction of *if* blocks begins with one reducing a single statement without suspension points or structured jumps and progresses through tests reducing sets of deeply nested conditional blocks, blocks containing structured jumps, and those containing suspension points which require the TGs representing the conditional's branches be balanced before final reduction can take place. Some tests use complex software structures to create specific ITG balancing problems.

The reduction test suite currently consists of 110 procedures which each present the compiler with one or more specific reduction problems, and provide a reasonably complete test of the compiler's ability to reduce TGs of Spring-C programs to ITG form. It is important to note that the actions taken by many subgraph reductions are influenced by the relative magnitude of the WCETs of time nodes in the subgraphs being reduced. Some of the reduction test cases would, for example, perform two significantly different sets of TG and RTL manipulations if the relative magnitude of the WCETs of two time nodes were reversed. The WCET of each time node is

normally determined by the target machine timing model, and changes to the target machine timing model might thus change behavior of some reduction tests by changing the relative magnitude of some time nodes' WCETs.

However, the reduction test suite is meant to test the validity of the reduction rules, and should not depend on the properties of the target machine timing model. The WCETs of the time nodes are, instead, assigned arbitrary values when the reduction tests are run. This renders the predicted WCETs meaningless with respect to the behavior of the procedures if executed, but ensures that the tests do not change their behavior when the target machine timing model is modified. The arithmetic of WCET calculation for each subgraph reduction was checked, the correctness of the RTL subgraph and source subgraph endpoints after each reduction verified, and the correct insertion of vacuous nodes examined in detail. When verified as correct, the output of each test was stored as part of the reduction test suite reference output. The reduction test suite was run fairly regularly as modification of the compiler proceeded, and the results were compared to the reference output. The effort to ensure the reduction test output did not vary with the timing model, enabling an automatic comparison to stored output, proved its worth by detecting several otherwise unsuspected problems created by specific compiler modifications.

While the reduction test suite provides a valuable way to test the correctness and completeness of the subgraph reductions at one level, more realistic tests using plausible application code are also clearly required, and we have conducted several tests of this type. The successful compilation of the programming model test cases described in Section 4.3 are good examples of this type of test. The first test case was an implementation of the programming example used to illustrate the use of the SDL in Section 4.2.7. The second application oriented test we performed was the

compilation of some reactive level robotics control code. The third programming trial implemented a simulation of a robotics assembly application, and was done as part of a student's Masters project [6]. The fourth test is an on-going project involving the implementation of another robotics manufacturing application with several more demanding characteristics [94].

The compiler has successfully compiled and run the code for the first two tests. Most of the code for the third has been compiled, and some of it run, but the student is continuing to work. The work on the fourth test is still in the design and testing phases where only small test programs are written to verify that the compiler's treatment of the code is appropriate to the needs of the application. In each of these test cases the full Spring scheduler, capable of building execution plans for groups of precedence related tasks, was not available. Thus, when the code was run, we had to modify things so that using the simple round-robin dispatcher on each application processor would be satisfactory. However, the scheduler's first full implementation is nearing completion and so more realistic tests of the application code should be possible soon.

The ability of the compiler to pass the reduction test suite and to compile all of the application code indicate that the reduction rules are correctly implemented and that they are capable of reducing Spring-C programs which follow the current set of programming conventions and restrictions. These tests also give us confidence that the behavioral analysis performed is reasonable, but they only verify that the analysis corresponds to our theories of how WCET should be calculated. They do not test the accuracy of the predictions made. That question was addressed by the prediction accuracy test suite discussed in the next section.

6.6.2 Prediction Accuracy Test Suite

The prediction accuracy test suite is a set of programs designed to gauge the accuracy of the WCET predictions made by the TG construction and subgraph reduction phases of the Spring-C compiler. It is important to remember, however, that the requirement is for *reliable* WCET predictions, which means that they should *never* underestimate the actual WCET. The results of this section show that this is harder than it appears and that fulfilling this stringent requirement will probably mean accepting a degree of overestimation in return for reliability.

The accuracy of the predictions depends on two independent factors: the accuracy of the WCETs assigned to time nodes in the original TG, and the correctness of the calculations performed during subgraph reduction. The WCET of each time node in the original TG is the WCET predicted by the target machine timing model for the assembler code sequence represented by the time node. We currently have three timing models, each taking a different approach to prediction execution time, and present results in this section which help compare and contrast the accuracy and reliability of the three models.

Each of the accuracy tests focuses on a specific subset of target machine instructions and Spring-C source constructs. For example, one test is a single sequence of floating point operations implementing a forward kinematic solution taken from a robotics application, while another implements a 50x50 matrix multiplication using integers, another does the same matrix multiplication using double precision floating point numbers, and another makes a series of procedure calls within a loop. The results indicate that the predictions we can currently make are reasonably accurate, but that there are clearly ways in which both the target machine timing models and

Table 6.1. WCET Prediction Accuracy Test Suite Description - Part 1

Test	description
01	Null proc_exec
02	Single sequence (forward kinematic solution)
03	Sequence with math library subroutine calls (reverse kinematic solution)
04	Single sequence (jacobian)
05	Sequence with one math library subroutine call (quadratic equation)
06	Sequence of 10 conditionals
07	Sequence of 5 switch blocks with 4 cases each
08	loop w/short body (loop is the $x * 128$ function)
09	$x * 128$ function w/ loop unrolled by a factor of 2
10	$x * 128$ function w/ loop unrolled by a factor of 4
11	$x * 128$ function w/ loop unrolled by a factor of 8
12	$x * 128$ function w/ loop unrolled by a factor of 16
13	$x * 128$ function w/ loop unrolled by a factor of 32
14	$x * 128$ function w/ loop unrolled by a factor of 64
15	$x * 128$ function w/ loop unrolled completely
16	Loop w/longer body (vector of quadratic equations), iteration count: 10
17	Loop w/longer body (vector of quadratic equations), iteration count: 100
18	Loop w/longer body (vector of quadratic equations), iteration count: 1000
19	Loop w/longer body (vector of quadratic equations), iteration count: 10000
20	Nested loop: 50x50 matrix addition (integers)
21	Nested loop: 50x50 matrix addition (doubles)
22	Nested loop: 50x50 matrix multiplication (integers)
23	Nested loop: 50x50 matrix multiplication (doubles)

the subgraph reduction analysis technique can be improved. For example, the timing models currently seem to overestimate the WCET of both loop overhead and procedure calls, while the subgraph reductions currently overestimate loop WCET by including certain error processing overhead in the loop time. These and other improvements will be made as development proceeds, and are discussed at greater length in the context of the tests which highlight the need for a particular change.

Tables 6.1 and 6.2 list the members of the prediction accuracy test suite. Note that the source code of every test is written in a way that ensures its execution will

Table 6.2. WCET Prediction Accuracy Test Suite Description - Part 2

Test	description
24	Loop w/single null procedure call, iteration count: 50
25	Loop w/5 null procedure calls, iteration count: 10
26	Loop w/25 null procedure calls, iteration count: 2
27	Sequence of 50 null procedure calls
28	Loop w/25 null procedure calls, iteration count: 128
29	Loop w/25 null procedure calls, iteration count: 512
30	Loop w/25 null procedure calls, iteration count: 2048
31	Loop w/25 null procedure calls, iteration count: 8192
32	Loop w/25 doubly nested null procedure call, iteration count: 128
33	Loop w/25 doubly nested null procedure call, iteration count: 512
34	Loop w/25 doubly nested null procedure call, iteration count: 2048
35	Loop w/25 doubly nested null procedure call, iteration count: 8192
36	Loop w/25 triplely nested null procedure calls, iteration count: 128
37	Loop w/25 triplely nested null procedure calls, iteration count: 512
38	Loop w/25 triplely nested null procedure calls, iteration count: 2048
39	Loop w/25 triplely nested null procedure calls, iteration count: 8192
40	Loop w/25 procedure calls (int vector addition), iteration count: 128
41	Loop w/25 procedure calls (int vector addition), iteration count: 512
42	Loop w/25 procedure calls (int vector addition), iteration count: 2048
43	Loop w/25 procedure calls (int vector addition), iteration count: 8192

follow the path producing the WCET. Test 1 measures the time for a null process, one executing no instructions apart from the *proc_exec* procedure's prologue and exit code. This helps assess the accuracy and a lower limit on the resolution of our measurement method. Tests 2 through 5 measure the accuracy with which the WCET for some representative mathematical computations are predicted. These tests are more realistic than they might first appear, since some of the instruction sequences were extracted from robotics control code which will be part of future Spring applications.

Tests 6 and 7 checks the predictions for conditional blocks. The clauses of the conditionals are sequences of simple arithmetic operations. Tests 8 through 15 all implement the same mathematical calculation ($x*128$), but use different combinations

of iteration and sequential code. This gives us a view of the difference in prediction accuracy for straight line sequences and for loop iteration overhead. Tests 16 through 19 are a variation of this using a constant loop body while varying only the loop iteration count. Tests 20 through 23 use loops to implement matrix addition and multiplication using both integers and double precision floating point numbers, which gives an idea of the difference in accuracy for integer and floating point calculations.

Tests 24 through 43, described in Table 6.2, concentrate on what seem to be two of the most important sources of predictive inaccuracy, loop overhead and procedure calls. Test 24 through 27 keeps the number of procedure calls constant while decreasing the loop iterations and increasing the length of the sequence of consecutive calls. Tests 28 through 31, 32 through 35, and 36 through 39, vary the composition of the loop body and the iteration count. Each set of four tests holds the loop body constant at 25 procedure calls, but increases the loop iteration count by a factor of 4. The loop body differs across the sets in the nesting depth of the procedure calls. The first set's loop body is a sequence of calls to a null procedure. The second set's loop body is a sequence of calls to a procedure calling a null procedure, and so on. Tests 40 through 43 vary the iteration count of a loop whose body contains a series of 25 calls to a procedure performing vector addition.

Four WCET values were generated for each member of the prediction accuracy test suite: actual, old, new, and sequence. The first is the *actual* execution time recorded. Recall that the source for each test is written in a way that guarantees it will follow its worst case path, so the execution times measured are the actual WCETs.

The second WCET is that predicted by the *old* timing tool, which was derived from an existing tool [42]. This was derived, in turn, from a similar tool for the 68000 [1].

We include its predictions in the results reported here because it implements the most obvious approach to building a target machine timing model. It is a table driven tool which essentially counts the worst case number of machine cycles that each instruction should take as specified by the 68020 processor manual [54]. However, the obvious approach, using the timing information published by the manufacturers, produces very poor results. It's accuracy is limited by the fact that the cycle counts specified in the manual are not necessarily accurate or correct [65, 54]. Most of the time the WCETs provided by the old tool are pessimistic, but a significant part of the time they WCETs *underestimate* the true value, which is clearly wrong. It is important to note, however, that while it is clear that the old tool often performs badly, it is not always as easy to see why. The results from some of the performance tests give a fairly clear idea of the instructions for which the old tool is in error, but others do not.

There were also some practical ways in which the old tool was not fully suitable for use by the Spring-C compiler, the most obvious being that it did not handle floating point instructions or several addressing modes generated by the compiler. Since many of the test suite members generated these instructions and addressing modes, the old tool was modified to return times predicted by the new tool when it encounters instructions unknown to it. Having the old tool use the new tool's times for such gaps seemed reasonable since the only alternative was to extend the tables of the old tool. Given the generally inferior performance of the old tool compared to the new, extending the old tool seemed to entail a fairly substantial cost in return for a dubious benefit. As a result, however, the predictions produced by the old tool for tests containing a large number of these instructions are close to the quality of those

produced by the new tool. The reason for this is that most of the times predicted by the old tool are actually new tool predictions.

The third WCET was predicted by the *new* tool which also uses a table driven approach, but in this case the table contains WCETs measured by experiment instead of counting cycles specified by the manual. The distinction between the old and new tool is thus that between using the published and directly measured execution times for each instruction. The table contains a WCET for every possible combination of addressing modes for every instruction. This method has the advantage of avoiding inaccuracies in the manual by measuring the actual behavior of the target machine, but is limited by the resolution of the clock used to measure elapsed times. The current target machine's timer chip has .5μsec resolution, but the execution time of the 68020 instructions ranges from less than .5μsec (.23μsec for *nop*) to 10μsec and above (67μsec for *fmovem* of all 27 FP registers).

We considered using the WCET measured for single instructions, but the comparatively coarse granularity of the clock resulted in significant overestimation for most instructions, and gross overestimation of the WCET of a test process since the overestimate contributed by every instruction was added together during subgraph reduction. Instead we made 10,000 separate measurements of the execution time for a sequence of 50 copies of the instruction in question, took the largest time observed, and divided by 50. This produced a floating point number giving μsec per instruction, which is stored in a table indexed by the instruction type and addressing modes of the operands. While these times are arguably not the *worst* case times, they are the best approximations we could obtain using available instruments and produce, as will be seen, reasonably accurate predictions of WCET for the test suite members.

The fourth WCET value was generated by the *sequence* method, which attempted to avoid limitations characteristic of the table driven approach used by both the old and new tools. For example, some overestimation is present in the predicted WCET for each instruction, since we were always careful to prefer an error that overestimated rather than one that underestimated. When instruction times are added together to calculate the WCET of a sequence, so are the overestimations. Another limitation is that table driven methods do not take the *interactions* among instructions into account.

The most obvious source of interaction among instructions is pipelining. The presence of even modest pipelines can create a significant difference between the sum of the WCETs of the instructions in a sequence, where the WCET for each instruction was measured in isolation, and the measured WCET of the *sequence*. The reason for this is the interactions among the instructions in the sequence. The WCET measure for the instruction in isolation cannot take such interactions into account, while measuring the WCET of the sequence does so. The available documentation on the 68020 only peripherally mentions the existence of internal pipelines, but it certainly performs instruction prefetching which should also lower the WCET of a sequence compared to the sum of the WCETs for the instructions in the sequence.

In an effort to assess how the pipelines and cumulative measurement error affects the accuracy of the WCET predictions, we modified the compiler to support a method of directly measuring the execution time of each *sequence* of assembler instructions represented by a time node. The method requires a two phase approach to compilation. In the first phase the compiler generates the assembler sequences for the basic blocks, which will be represented by time nodes in the TG, in a form that can be used to build a test program measuring the WCET of the each sequence. Each test

program places the sequence in a context as close as possible to that in the procedure being analyzed. Some hand crafting of initialization code is required for sequences using one of the more context sensitive instructions or addressing modes, but the processing required to prepare most sequence tests is automated. The WCET of each sequence is measured in the same way the WCET of individual instructions were measured to produce the new tool's tables. A sequence is concatenated 50 times to create a new sequence whose WCET is sampled 10,000 times. The largest execution time observed is then divided by 50, giving the WCET estimate for the original sequence.

When the tests measuring the WCET for each sequence are complete, the source file is compiled a second time. A flag tells the compiler to use the measured WCETs for each sequence as the WCETs of the corresponding time nodes when building the original TG. The advantage of this method is that it can account for the effects of pipelining, concurrent execution by the floating point coprocessor, the effects of other context sensitive aspects of processor performance, and decreases the cumulative measurement error. The method should thus improve the accuracy of the predicted times. There are, however, several limitations and subtle points worth discussing.

One of the more subtle contextual factors is the *alignment* of the sequence. We observed small but significant variation in the WCET of sequences when they are aligned on 16 and 32 bit boundaries. Some execute faster when aligned on one boundary, and some execute faster when aligned on the other. It is important to note in this context that 68020 instructions are of variable length, so the alignment of various instructions in a sequence will often differ from that of the first instruction. It is also important to note that the alignment of sequences within the executable image is essentially random. It can, however, be determined from the executable.

Table 6.3. WCET Prediction Accuracy Results - Part 1

Test	actual(μsec)	old(μsec) (P/A)	new(μsec) (P/A)	sequence(μsec) (P/A)
01	5.5	6.1 (1.11)	6.9 (1.25)	7.4 (1.34)
02	149.0	157.7 (1.06)	158.6 (1.06)	152.9 (1.03)
03	676.0	666.3 (0.99)	667.0 (0.99)	678.6 (1.00)
04	348.5	358.8 (1.03)	361.4 (1.04)	352.4 (1.01)
05	125.0	134.8 (1.08)	135.5 (1.08)	138.0 (1.10)
06	187.5	212.3 (1.13)	208.4 (1.11)	200.0 (1.07)
07	112.5	127.7 (1.13)	137.0 (1.22)	123.9 (1.10)
08	814.5	921.1 (1.13)	1202.7 (1.48)	1033.1 (1.27)
09	431.0	514.6 (1.19)	621.8 (1.44)	551.9 (1.28)
10	239.0	291.9 (1.22)	341.5 (1.43)	304.2 (1.27)
11	143.0	180.6 (1.26)	200.4 (1.40)	180.4 (1.26)
12	95.0	124.9 (1.31)	129.8 (1.37)	119.0 (1.25)
13	71.0	97.1 (1.37)	94.5 (1.33)	87.8 (1.24)
14	59.5	83.1 (1.40)	76.9 (1.29)	73.5 (1.23)
15	39.0	55.2 (1.42)	42.8 (1.10)	41.2 (1.06)
16	1782.0	1981.7 (1.11)	1932.5 (1.08)	1888.7 (1.06)
17	17527.5	19567.3 (1.12)	19030.9 (1.09)	18435.9 (1.05)
18	176008.0	194787.0 (1.11)	190232.7 (1.08)	184718.8 (1.05)
19	1646657.5	1947591.9 (1.18)	1901998.4 (1.16)	1846805.5 (1.12)
20	61658.0	80354.6 (1.30)	66612.5 (1.08)	66174.2 (1.07)
21	92918.5	116053.9 (1.25)	102011.8 (1.10)	100364.9 (1.08)
22	4185256.0	5427597.7 (1.30)	4403025.8 (1.05)	4418337.2 (1.06)
23	5908621.0	7760051.0 (1.31)	6750478.8 (1.14)	6285491.2 (1.06)

When the 50 copies of the sequence are concatenated to form the sequence whose execution is measured, we insert one or more *nop* instructions between the copies as required to ensure each sequence begins with the alignment it has in the actual executable. Ensuring the proper alignment of each sequence made a small but significant improvement in the accuracy of the sequence times. We also shield the concatenated sequence from the influence of the code testing it by surrounding it with *nop* instructions to force any pending memory references to complete, and to flush the processor's

Table 6.4. WCET Prediction Accuracy Results - Part 2

Test	actual(μsec)	old(μsec) (P/A)	new(μsec) (P/A)	sequence(μsec) (P/A)
24	521.5	557.3 (1.07)	695.4 (1.33)	686.0 (1.32)
25	285.5	278.9 (0.98)	345.0 (1.21)	370.4 (1.30)
26	238.0	223.2 (0.94)	275.0 (1.16)	307.3 (1.29)
27	220.0	196.2 (0.89)	242.1 (1.10)	277.3 (1.26)
28	16356.5	12968.0 (0.79)	16136.8 (0.99)	19323.0 (1.18)
29	57879.5	51658.8 (0.89)	64535.3 (1.11)	72829.0 (1.26)
30	231463.5	206576.6 (0.89)	258067.4 (1.11)	291366.8 (1.26)
31	925775.0	826248.1 (0.89)	1032195.9 (1.11)	1164903.1 (1.26)
32	30737.5	25063.8 (0.82)	31112.5 (1.01)	36994.5 (1.20)
33	117409.5	100041.8 (0.85)	124438.1 (1.06)	144128.3 (1.23)
34	469573.0	400108.8 (0.85)	497678.6 (1.06)	576319.4 (1.23)
35	1878218.5	1600376.6 (0.85)	1990640.8 (1.06)	2305697.9 (1.23)
36	45583.5	37159.5 (0.82)	46088.2 (1.01)	54842.2 (1.20)
37	174780.0	148424.8 (0.85)	184340.9 (1.05)	214906.2 (1.23)
38	699070.5	593640.9 (0.85)	737289.8 (1.05)	859552.4 (1.23)
39	2796224.5	2374505.1 (0.85)	2949085.6 (1.05)	3438629.2 (1.23)
40	538406.5	650587.2 (1.21)	654524.0 (1.22)	628089.7 (1.17)
41	2123000.0	2602135.8 (1.23)	2618084.2 (1.23)	2445501.8 (1.15)
42	8491223.0	10408484.6 (1.23)	10472263.1 (1.23)	9781936.2 (1.15)
43	33964147.5	41633879.9 (1.23)	41888978.8 (1.23)	39115386.0 (1.15)

pipe, before the sequence begins. The *nop* following each sequence ensures that memory references associated with the last few instructions in the sequence are taken into account. This is a conservative approach, and clearly overestimates the WCET of some sequences. This is, however, preferable to underestimating the WCET of the sequence by failing to take these points into account.

Tables 6.3 and 6.4 present the WCETs produced for each of the prediction accuracy tests. The columns give the actual, old, new, and sequence times in μsec. Each of the columns reporting the output of one of the timing tools also gives the ratio of the predicted time to the actual, (P/A). For example, the actual WCET for test

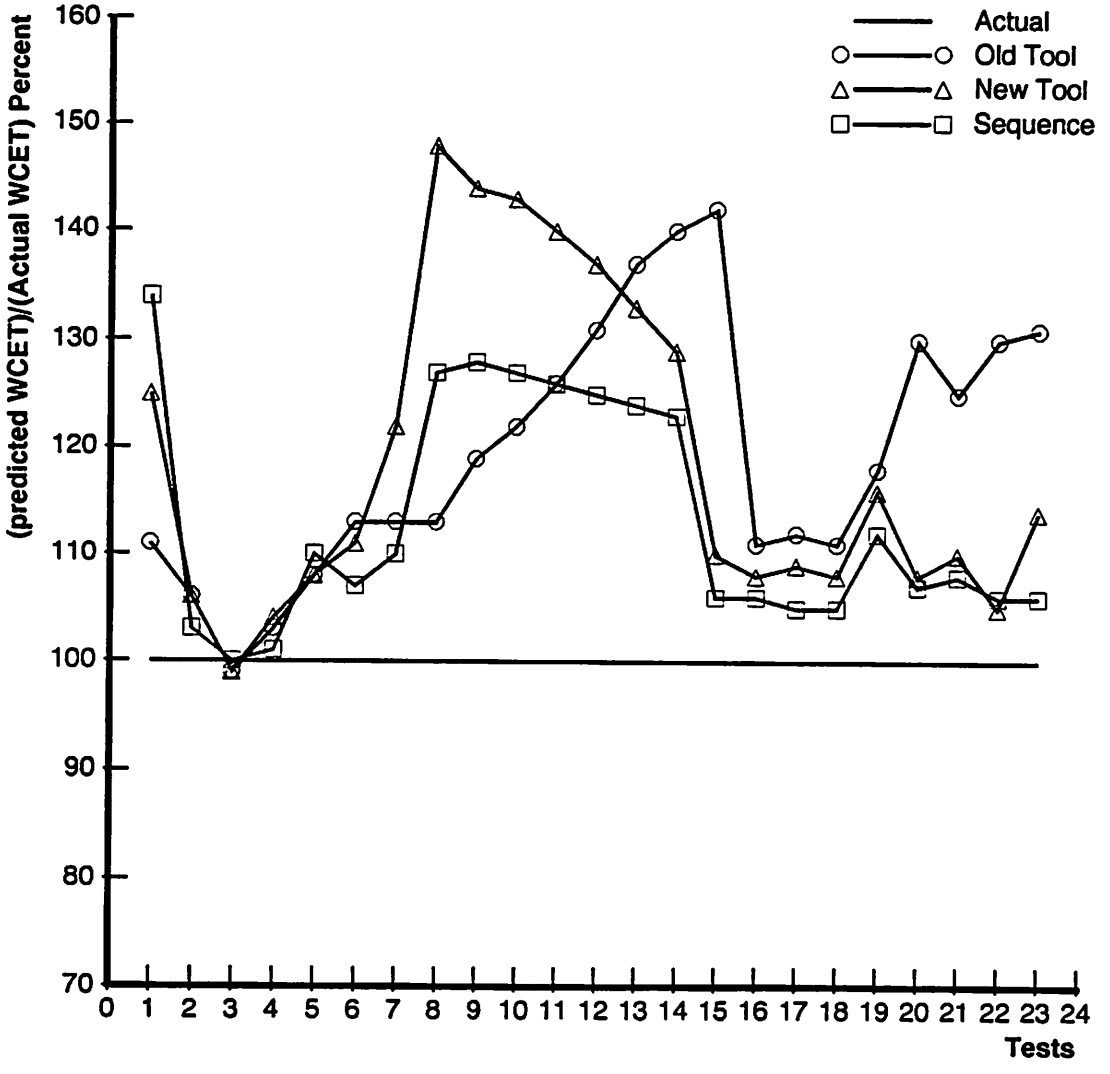


Figure 6.21. Prediction Accuracy Profile - Part 1

number 6 was 187.5 μsec, while the old tool's prediction was 212.3 μsec, giving a P/A ratio of 1.13. Of particular interest are those tests for which the WCET predictions are *less* than the actual value. It is easiest to find these cases by looking at the P/A ratio. In test 3, for example, both the old and new tools underestimate the actual time, as does the old tool for tests 25 through 39. Since an underestimation of the WCET is not acceptable, such results are of particular concern. They represent errors in the design of the target machine timing model which must be corrected for the tool to be of use.

Figures 6.21 and 6.22 present, in graphic form, the results of Tables 6.3 and 6.3, respectively. The graphic presentation makes it easier to see several points of interest for specific tests and sets of tests. Note that the vertical axis shows the predicted WCETs as a *percentage* of the actual execution time measured, and that the actual execution times of the various test cases differ from one another by several orders of magnitude. The use of *percentage* of actual WCET normalizes the results from different tests. The horizontal line at 100 thus marks the actual execution time of each test, and establishes a hard lower bound for WCET estimates since *underestimating* the WCET is clearly an error. The horizontal axis of each plot gives the test number.

The lines in the figure thus serve to describe an *accuracy* profile for each timing tool across the test suite. There is a limit, however, on how much can be interpreted from the shape of the profiles. Within a set of related tests the shape of the profile reveals important trends by showing how accuracy varies as some aspect of a test is changed. Portions of the profile which cross the boundaries between unrelated sets of tests only reveal the relative accuracy of the WCET for the two sets of tests, and nothing more. Ideally the accuracy profile would be roughly parallel to the *actual* line, and lie as close as possible to it. Such a profile would indicate that all the predictions made using a given timing tool were of roughly the same accuracy. None of the timing tools exhibits such an admirable accuracy profile, but there are several points worth considering.

The most obvious problem is that old tool underestimated the WCET for several tests, while the new tool did so by roughly 1% for tests 3 and 28. The sequence tool did not underestimate the WCET for any test. The new tool generally does better than the old with one obvious exception that is discussed shortly. The sequence times are usually better than those of the new tool, but not always. The new tool

did slightly better on tests 1, 5, and 22, while it did substantially better on tests 25 through 39. Explanations of these observations are discussed in the context of the tests involved. Sequence times *should* provide better predictions for programs with long instruction sequences whose WCET is significantly affected by pipelining or concurrent coprocessor execution. For programs without long sequences, this advantage is limited and the sequence times should converge with those of the new tool. All of the timing tools seem to exhibit some problems, however, and further work will be required to improve WCET predictions.

The results presented here show that while each method for constructing a target machine timing model has some problems, in general the sequence tool does best, followed by the new tool, while the old tool did worst. The sequence tool does best because it *never* underestimates the true WCET, and generally produced better results than the new tool. The old tool was worst because it underestimated the WCET for a significant number of test cases, and generally produced more pessimistic WCETs than the other tools for the remaining test cases. The new tool's performance was mixed. It occasionally underestimated the WCET, and was usually a little more pessimistic than the sequence tool.

The most likely explanation for the sequence tool's generally superior performance is its ability to take the effects of pipelining and interactions between instructions into account, in contrast to the other tools inability to do so. The usual effect of pipelining is to decrease the execution time of a sequence of instructions. This would account for the WCET predictions made by the sequence being lower than those of the other tools for many tests. However, when instructions interact the execution time of the *sequence* can actually be *greater* than the sum of the times for each instruction,

depending on the structure of the pipeline, and on how the instruction times were measured.

For example, assume a sequence where each instruction uses a register set by its predecessor, and sets a register used by its successor. Whether this fact is important or not depends on the properties of the CPU pipeline. Some CPU designs ensure that the registers written by an instruction are *always* ready for use by the next instruction, but some do not. When this is *not* true, the pipeline stalls until the register is written and the result ready for use by the next instruction. For the instruction sequence in this example, every instruction experiences the stalling delay. Finally, if the individual instruction times were measured using instructions whose registers were always ready for use, then the WCET of the sequence experiencing the delays will be greater than the sum of the measure instruction times.

Such interaction effects could account, in part, for those tests where the new and old tools underestimated the WCET, since neither measuring the WCET nor adding up the cycles predicted by the manual take such interactions into account. However, since the old tool so grossly underestimates the WCET of several tests, we have concluded that either some of the cycle counts specified in the manual are wrong, or some of the table entries in the tool were incorrectly transcribed. We turn now to a discussion of some of these problems, and ways that they can be minimized or solved, by discussing the individual tests.

Test 1 is an empty process, thus having the smallest possible WCET of any Spring-C program, and serves to reveal limits on the resolution of the prediction methods which measure execution time, as well as limits on how precisely the actual and predicted times can be compared. The old tool performs the best on this test, followed by the new and sequence tools, but while the percentage overestimation is quite large,

this is partly a result of the fact that the actual execution times are quite small, as shown in Table 6.3.

There are several points of interest here. First, limits on the resolution of measuring the actual time and comparing it to the predictions arise in several ways. The .5μsec resolution of the clock means that different samples of the actual time may differ by a full μsec. The actual WCET reported is the maximum observed over 100 separate executions of the test, which seems reasonable since the goal is to evaluate WCET estimates that must support *every* process execution. The WCET predictions made by the new and sequence tools are, in contrast, produced by methods discussed earlier which *average* the execution time of either several copies of an instruction, or several copies of a sequence of instructions. This is why, for example, the *actual* times in Tables 6.3 and 6.4 are all multiples of the .5μsec clock resolution while the WCET predictions produced by the timing tools are not.

Another limit on the resolution of the measurement method arises from subtracting the overhead of the instructions which capture the start and end times of the code from measured WCETs, since we subtract the *smallest* value from the actual times measured, to be conservative. This tends to overestimate the measured times, but applies to the actual, new, and sequence times in slightly different ways since the overhead is amortized across the times being measured in different ways. Another source of error is that the actual times include the *jsr* to *proc_exec* which is not considered by its predicted time. All of these sources of error are rather small, but clearly significant in relation to the 5.5μsec actual execution time of test 1. Efforts are continuing to refine the accuracy of the sampling method.

Tests 2 through 5 consist of single sequences of various lengths which use different instructions. All of the WCET predictions were close to the actual value, within 10%,

with the sequence tool generally producing the best estimates. This was particularly true for test 3 whose WCET the other tools underestimated. However, the sequence tool did not do quite as well as the others for test 5. We have found no apparent reason for this, but the estimates differed from one another by only 3.2 μsec, which is roughly the magnitude of several of the limits on time resolution discussed earlier.

Tests 6 and 7 address the conditional statements *if* and *switch* respectively. The new tool and sequence method do better than the old tool on test 6, but perform slightly less well than they did on the previous tests. A partial explanation of this is that we made the *true* clauses of the *if* blocks the worst case path. When this is so, the conditional jump at the end of the *if* block's conditional expression is *not* taken. It is a property of conditional jumps that they take *more* time when taken than not taken. Both the new tool and sequence methods estimate the time for the conditional assuming that the branch is taken, thus overestimating for this test. The effect is fairly small, accounting for roughly 10% of the error. This is an example of error arising from the current implementation of subgraph reduction which will be corrected as the implementation is improved.

Another factor is that the *true* clause immediately follows the conditional expression which provides an advantage on architectures which prefetch and pipeline instructions, as the 68020 does. A similar situation exists at the end of the *false* clause, which falls directly into whatever instructions follow the *if* block. It is difficult to estimate what portion of the overestimation is caused by this effect, since it also depends on the sequences involved. However, it seems possible to at least take the effect of prefetching into account as the reduction phase is refined.

The new tool does not do as well on test 7, although the sequence tool's prediction is good. One source of error lies in the subgraph reduction calculation. The code

implementing the expression given as an argument to *switch* produces a value which is then used to select which case to execute. When the number of cases is more than 5, a dispatch table is built, and the *dispatch* instruction used. The *dispatch* instruction is, however, the one instruction which none of the timing tools can yet handle. Test 7 has 4 cases among which one is selected by a decision tree implemented as a set of nested *if* blocks. The reduction of the decision tree suffers the conditional jump error already discussed.

Tests 8 through 15 illustrate several interesting points. First, the old tool's accuracy decreased as the loop was unrolled, while the accuracy of the new and sequence tools improved. The old tool appears to estimate loop overhead more accurately than the new and sequence tools, while the 42% overestimation for the unrolled loop clearly indicates a significant error in estimating the time for the arithmetic component of the calculation. As the loop is unrolled the total execution time goes down, the percentage of the total time represented by the arithmetic component goes up, and the accuracy of the old tool's predictions thus deteriorates.

Another interesting point is that the accuracy of the sequence tool's estimates is always better than that of the new tool. An obvious explanation is that the sequence tool is accounting for pipelining effects. The fact that the accuracy of its predictions improves only slightly as the loop is unrolled indicates that this is probably a minor component, but the sudden in the accuracy of the predictions for the fully unrolled loop (test 15) indicates that it is probably a factor. The steady improvement in accuracy as the loop is unrolled, as well as the vast improvement when the loop is eliminated entirely, supports the view that the new and sequence tools estimate WCET of loop overhead less accurately than they do the arithmetic component fo the calculation. Further, the fact that the difference between the accuracy of the new

and sequence tools decreases as the loop is unrolled indicates that the sequence tool estimates loop overhead better than the new tool.

Tests 16 through 19 show the results when loop iterations are increased while the body of the loop, a moderately complex set of quadratic equations, is held constant. In this case the WCET of the loop body was a significantly larger than the loop overhead, which tends to explain why the accuracy of the predicted WCETs for all methods is almost constant, although it suffers an unexplained decrease of approximately 5% for test 19. Further, the relative accuracy of the methods supports the idea that the new tool gives better times for arithmetic operations than the old, but that the sequence tool does even better by accounting for some pipelining effects.

Tests 20 through 23 show that the sequence method is good at predicting WCET of both integer and floating point arithmetic operations. The new tool is roughly as good for integers, but has a higher error for the floating point operations, particularly multiplication. This is to be expected since the CPU and FPU can execute *concurrently* during the substantial time required for the floating point multiply. The new tool cannot account for this, but the sequence tool can and thus produces a more accurate WCET prediction. The old tool, in contrast, did much worse for every test in this set.

Figure 6.22 presents in graphic form the numeric results for tests 24 through 43 given in Table 6.4. These tests concentrate on what seem to be two of the most important sources of predictive inaccuracy, loops and procedure calls. Tests 24 through 27 keeps the number of procedure calls constant while decreasing the loop iterations and increasing the length of the loop body. The most obvious result of these tests is the old tool's gross underestimation of the WCET. Another interesting point is that while they produce equally good predictions for test 24, the new tool produces better

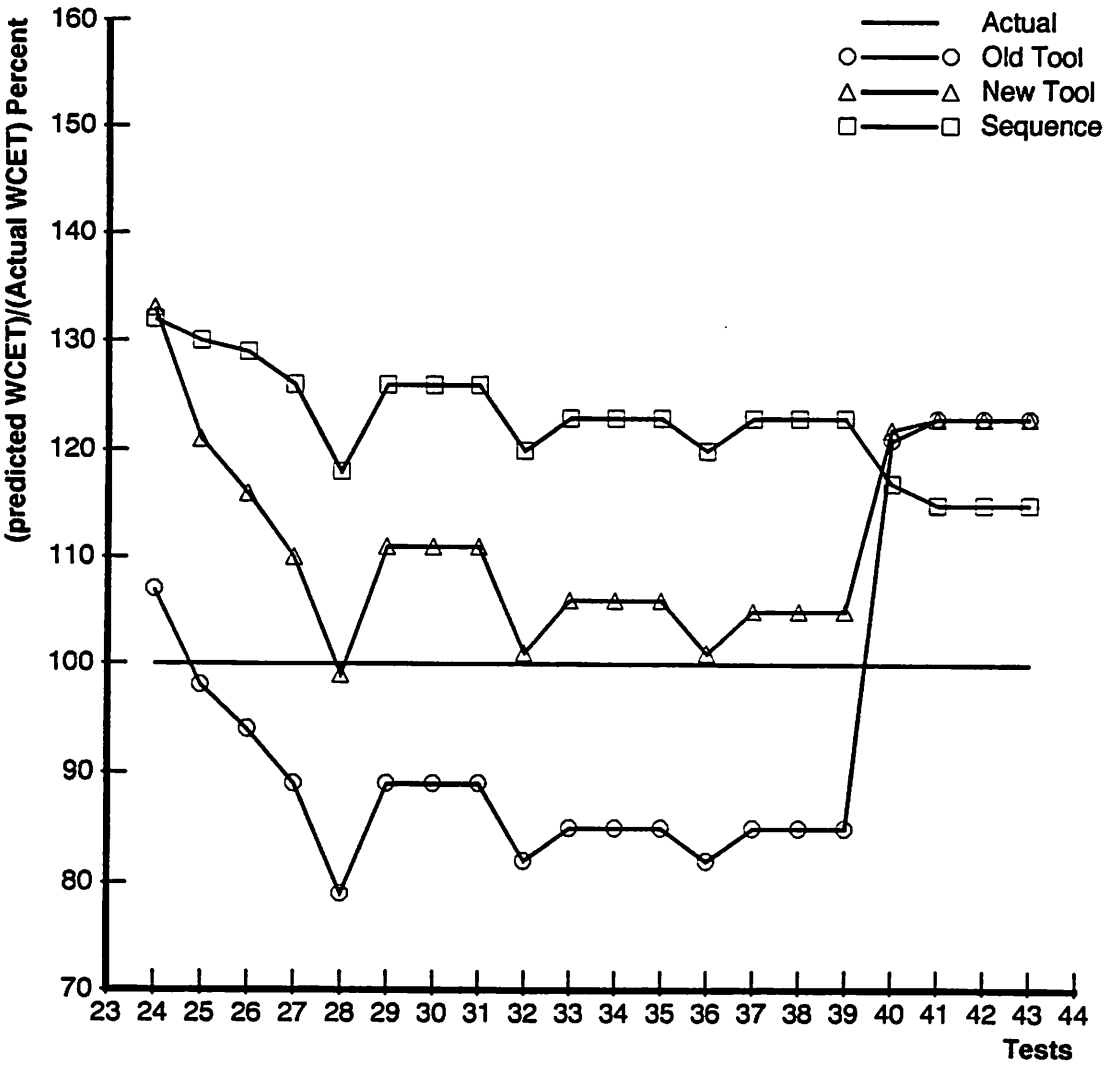


Figure 6.22. Prediction Accuracy Profile - Part 2

predictions than the sequence tool for all the others. Since these tests contain no sequences of appreciable length, and certainly not any with appreciable pipelining or concurrency, this is not too surprising.

The new tool's prediction for the WCET of the loop overhead is greater than that of the sequence method, but the predicted time for a procedure call produced by the sequence method is twice that of the new tool. This appears to be the result of using different alignments when measuring the *jsr* instruction time for the new tool's table entry and measuring the "sequence" time for the instruction. Test 24 has

the most iterations, and the error in the loop overhead prediction by the new tool roughly equals the advantage of the lower error in its procedure call prediction. The alignment of the *jsr* in the test program probably matches that used for the table time, and not that of the sequence test.

As the number of iterations decrease the inaccuracy contributed by the loop overhead decreases as well, and the overall prediction accuracy improves. From the fact that the accuracy of the sequence method does not improve as quickly as the new tool's when iterations decrease, we conclude that the accuracy of the loop overhead prediction is only slightly less than that for a procedure call. Thus, as loop overhead is eliminated from the sequence tool's predictions, the overall accuracy improves only slightly.

Tests 28 through 31, 32 through 35, and 36 through 39, vary the composition of the loop body and the iteration count. Each set of four tests holds the loop body constant at 25 procedure calls, but increases the loop iteration count by a factor of 4. The loop body differs across the sets in the nesting depth of the procedure calls. The first set's loop body is a sequence of calls to a null procedure. The second set's loop body is a sequence of calls to a procedure calling a null procedure, and so on. The relative constancy of the prediction error for loops with the same body but iteration counts differing by orders of magnitude indicates that the subgraph reductions for the loop are correct, and the inaccuracy arises from the WCETs predicted by the timing tools which are used to build the TG.

The old tool obviously underestimates the WCET of procedure calls. Test 28 indicates that the new tool probably also underestimates the WCET of a procedure call slightly, since it underestimated the WCET of the test. The slight improvement as the nesting level and thus the total number of procedure calls increases supports

the view that the accuracy of the WCET of a procedure call is slightly better than that of the loop overhead. It is not clear why the accuracy of predictions for the loops with an iteration count of 128 are better than the others in every group of tests.

Tests 40 through 43 are similar to 28 through 31 in that they vary the iteration count of a loop containing a sequence of 25 procedure calls. However, in this case the procedure performs integer vector addition, and the advantage of the sequence tool reasserts itself since the calculations benefit from pipelining. The old tool does not underestimate since its overestimation of the arithmetic operations compensates for its underestimation of the procedure call time.

There are some sources of inaccuracy for loops arising from properties of the subgraph reduction method. We have already discussed the overestimate resulting from the conditional jump at the end of the conditional expression, as well as that arising from not considering the prefetching advantage. However, these factors also apply to the test of the loop counter *lc* against the upper bound at the beginning of the loop body. It ends with a conditional jump not taken unless the bound is violated. The loop overhead is thus overestimated by the WCET of the jump on every iteration.

Another source of inaccuracy for loops lies in the exit code. Recall that a test is made of the loop counter *lc* against the upper and lower bounds. If violated, the routine *bounds_violation* is called. The current reductions treat this just like any other conditional and take the WCET of the *bounds_violation* routine into account. However, we currently tell the system that this routine takes very little time. Further, this is a constant overestimation not varying with loop iteration count. While prediction accuracy will improve as the reductions are extended to take these effects into

account, they do not appear to account for more than a small part of the observed inaccuracy.

Each set of tests suggests a set of questions about why the results were not better. The answers can lead to further tests, modifications of the WCET estimation methods, or both. The process is similar in some ways to optimizing the performance of an application program. The best tactic is to concentrate on the parts of the program that take the most time. Similarly, when trying to improve the accuracy of the WCET predictions, we look for what seem to be the largest sources of inaccuracy at any given point. Unfortunately, the number of possible sources is quite large, and the process of refinement consequently long.

The experiments assessing prediction accuracy are still fairly crude, and leave a number of instruction types untouched, but they tend to support several important points. First, the method of making WCET predictions using subgraph reduction works well, when accurate WCET predictions for basic blocks are used to build the original TG. Creating an accurate target machine timing model is the subject of several research efforts [1, 29, 68, 65], although of these only [29] attempts to account for pipelining effects. In our opinion the timing model must be a detailed simulation of the target machine capable of noting interactions among instructions in a sequence, and of accounting for the effects of the pipeline. Such models should be supplied by manufacturers of processors intended for real-time application, or sufficient information should be supplied about the processor to build such a model. The exposure of RISC processor's internal pipelines offers some hope that accurate models might be built for them.

Supported by an accurate model of the target machine, the code analysis and task group construction methods described here seem to perform well. Improvements in

the timing model, expansion of the test suite, and further use of Spring-C for real applications will be required to know or sure. However, it is important to remember that the requirement is for reliable WCET predictions, which means that they should *never* underestimate the actual WCET. The results of this section show that this is harder than it appears and that fulfilling this stringent requirement will probably mean accepting a degree of overestimation in return for reliability.

CHAPTER 7

TASK GROUP CONSTRUCTION

The term "target behaviors" has been used in this dissertation when referring to the aspects of a computation's behavior about which a system requires reliable predictions. Real-time systems use predictions about the target behaviors to ensure that computations satisfy various constraints while they execute, including: deadlines, restrictions on concurrent use of resources shared with other computations, minimum delays between specific actions taken by a computation, and restrictions on the order in which sections of a computation are executed.

While systems differ somewhat in the set of target behavior predictions they require, and the form in which the predictions are expressed, all such predictions tend to *segment* the run-time behavior of a computation into a set of execution episodes. For example, one episode might be distinguished by its use of a shared resource from the episodes before and after it which do not use the resource. The boundary between two other episodes might be created by an explicit delay of N time units required for proper control of an external device. Systems also differ in the extent to which their designs facilitate reliable prediction of execution time behavior, and thus the difficulty of the segmentation problem.

Each system uses the behavioral predictions, taking the different characteristics of each execution episode into account, when managing the execution of all the computations sharing the machine so that they all satisfy their execution constraints.

The method of producing the set of information required by the system to manage computations, the target behavior predictions, can thus be viewed as *segmenting* the computation's behavior into a set of execution episodes. The problem of managing the computation at run-time can be viewed as scheduling its execution episodes in a way which satisfies their constraints.

This chapter describes how the segmented description of a computation's target behaviors required by the Spring scheduler at run-time can be built from the ITG produced by the methods described in Chapter 6. The Spring scheduler expects the behavior of each *computation* to be represented as a group of tasks with known WCET and resource use. The execution order of the tasks within the group is constrained by precedence relations. The method for program translation described in this dissertation represents each execution episode of a process using a separate task.

This approach creates the requirement that for each thread of control we know the maximum number of execution episodes, the WCET and resource use of each execution episode, and the order in which they occur. Recall that Spring currently restricts each process to a single thread of control. As discussed in Chapter 4, concurrency within a computation or shared address space is implemented using a group of processes. The group structure determines what processes, and thus threads, can execute concurrently.

Chapter 5 gave an overview of the translation process, described how the Spring-C source is compiled into the RTL based intermediate representation, and how the original TG describing the target behaviors of a procedure is built from its RTL representation. Since each task in the group represents an execution episode of the process, the task must represent *all* the behaviors an episode might exhibit. One way to determine all possible behaviors of every execution episode would be to examine

every path through a process's TG. For programs of non-trivial size this would require the examination of an infeasible number of paths.

We simplify the problem by simplifying the TG. Chapter 6 discussed how the original TG is simplified using subgraph reduction, which simplifies the TG by replacing a subgraph through which there are two or more execution paths with a linear ITG representing the target behaviors of all the paths through the subgraph. As subgraphs of the TG are reduced the number of paths through the TG that must be considered when constructing the task group is reduced. When the TG of the process is reduced to linear ITG form the number of paths through the TG is one, and deriving the target behaviors of the process's execution episodes from it is simple, which simplifies constructing the task group.

Spring's functional partitioning, first discussed in Section 3.2, plays an important role in the analysis as does the non-preemptive aspect of the execution plan built by the scheduler. Functional partitioning enables the system to shield the application processors from external interrupts, thus eliminating one source of suspension for application processes. The non-preemptive scheduling policy eliminates another source. A process, once the dispatcher has changed context to it, is thus left to run until it suspends itself, or until it exceeds the execution time allocated by the system for the current task. These aspects of Spring's design were chosen precisely because they eliminate all external reasons for process suspension, leaving only the explicit suspension points within the program for consideration.

These design decisions, as well as others, were carefully chosen to support the prediction of the target behaviors of a process's execution episodes. Other important aspects of the system design include those specifying what information should be represented by the enhanced RTL, the information stored in the new structural and

suspension point nodes, and those defining the analysis performed on the original TG by subgraph reduction. The restrictions on preemption and external interrupts probably can, however, be relaxed as the accuracy and subtlety of the analysis are increased. The coordination of design decisions at different layers of the system is an example of the *vertical slice* approach to system design discussed in Chapter 3.

This chapter first discusses how the task group describing a *process* is built using the information about the process's execution episodes provided by its ITG, and then considers how the task group describing a *computation* is built from the task groups describing the processes which implement the computation. Recall from the discussion in Chapter 4 that the ITG of a *process* is the ITG of its *proc_exec* procedure since that is, by convention, the entry point the system uses to execute a process. Building a task graph from the ITG must take several system level issues into account, in addition to the behavior of the process represented by the ITG. These include the time required to switch context to and from a process when beginning and ending each execution episode, the worst case time to enter and leave the system calls implementing the suspension points which create the task boundaries, and communication delays associated with virtual circuits supporting synchronous communication.

It is easy to build the task group representing a computation which is implemented as a single process. It is only slightly harder to build the task group representing a computation implemented by a process group, where the processes in the group do *not* engage in synchronous communication. Section 7.1 discusses how task groups representing these classes of computations are built. The use of synchronous communication among processes in a group requires an additional level of processing. The task groups representing each communicating process are produced by the method

described in Section 7.1, but this algorithm does not have enough information to create the precedence relations between tasks representing different processes required to enforce the semantics of synchronous communication. Instead, it leaves unresolved precedence relations between tasks in the groups representing the communicating processes, which are handled by a separate algorithm described in Section 7.2.

7.1 Processes Without Synchronous Communication

The construction of a task group from the ITG of a process is straightforward because the analysis producing the ITG of the process was explicitly designed to ensure that it represents the boundaries and properties of the process's execution episodes. The suspension points in the ITG mark the episode, and thus task, boundaries. They also provide the information required to determine resource use of each task, and the attributes of the precedence relations between tasks in the group. The WCET of each task representing a particular episode is determined by the WCET of the time node between the suspension points bounding the episode.

The Spring linking/loader *spr-ld* builds the executable file containing the information required to execute each process. This includes the task group description of the process's execution time behavior. The position of the linking step within the information flow of the software generation system was discussed in Section 3.1, and illustrated in Figure 3.3. As the figure shows, when the executable image of the process has been constructed the SDL information in the *full.db* file, which describes all the information gathered or derived when the source files were compiled, is available for *spr-ld* to use. In addition, it also has the set of SDL information collected from the files used to construct the executable.

Spr_ld builds the executable file for each process in the normal manner. It accumulates code and data from each object file and object library in the order specified on the command line, and as required to resolve references to code and data. It is important to note, however, that as it accumulates the conventional information in the conventional manner, it also accumulates the SDL information from each of the input files. When the executable file for a process is complete in the conventional sense, *spr-ld* uses the accumulated SDL information, as well as the more general information available in the *full.db* file, to construct a task group describing the process's behavior. The task group description is then included in the SDL section of the Spring executable produced. The algorithm, *build_task_group1*, is described in detail in Appendix C, but an illustrative example should make its essential features clear. The actions of the algorithm are described using the notation defined in Section 4.2.1.

The *build_task_group1* algorithm uses the ITG of the process P's *proc_exec* procedure, which *spr_ld* collected when accumulating the information from the SDL sections of the object files used to build the process's executable. The algorithm scans through P's ITG, accumulating the properties of each task in the group $TG(P)$ as it goes. The ITG is the primary source of information for the algorithm, but other parts of the accumulated SDL information are also used. Note that we are not considering processes engaging in synchronous communication at this time. The extensions to *build_task_group1* required for synchronous communication are discussed in Section 7.2.

Figure 7.1 illustrates the reduction of a conditional statement containing suspension points to a linear ITG according to the methods described in Section 6.3. Each clause of the conditional block contains a resource use block, the true clause using resource X in exclusive mode, and the false clause using resource Y in shared mode.

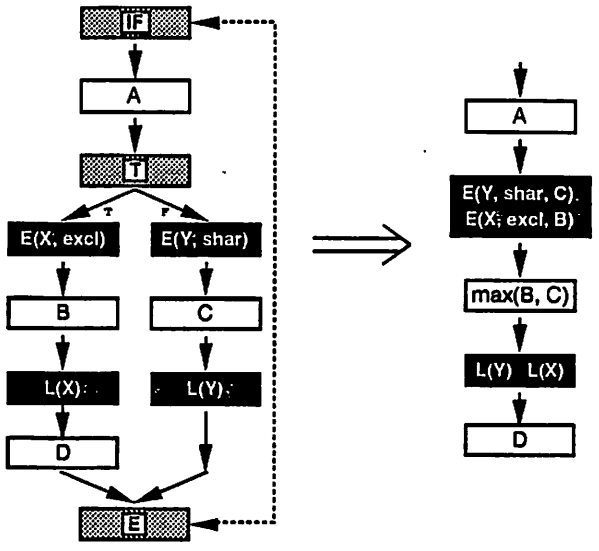


Figure 7.1. If Block Reduction Combining Suspension Points

Assume, for simplicity, that this is the only part of a process P's code which contains suspension points, and that it appears at the main level of P's *proc_exec* procedure, ensuring that every execution path will execute it.

Since every other part of P's code contains no suspension points, its *proc_exec* procedure will reduce to an ITG with the structure illustrated in Figure 7.2a. The time node with the WCET G represents the execution episode from the beginning of the process up to and including the conditional expression, the node with WCET A, of the *if* block containing the suspension points in Figure 7.1. The time node with WCET C represents the execution episode which uses *either* resource X or Y, depending on the execution path taken at run-time. In this example we assume that $C > B$, and so the time node has a WCET of C. The last time node, with WCET H, represents the execution episode starting when the process leaves the resource use block to the end.

Applying the *build_task_group1* algorithm to this ITG, we obtain the task group illustrated in Figure 7.2b. The general idea is that the algorithm scans through the

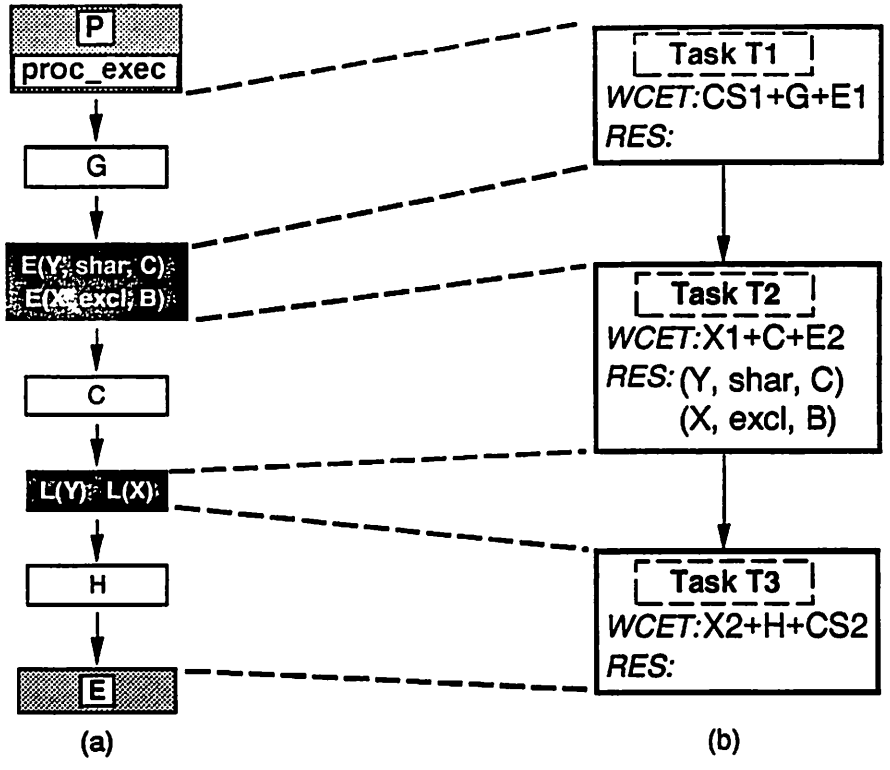


Figure 7.2. Procedure ITG and Corresponding Task Group

ITG, and assigns the WCET, resource use, and precedence relations of each task according to the WCET of each time node and of the properties of the suspension points surrounding it. In practice, there are some fairly subtle details that must be handled. First, there is a non-zero time that must be added to account for switching context between the dispatcher and the process at the beginning and the end of process execution. The time for switching to the process when its execution begins is added to the WCET of the first task, and the time for switching away from it when execution is complete is added to the WCET of the last task. Similarly, each suspension point represents a system call with non-zero *entry* and *exit* times. The entry time is the WCET from the *jsr* starting the system call to when process execution is suspended by changing context to the dispatcher, while the exit time is the WCET between the

dispatcher switching context back to the process to when it returns from the system call.

These issues are reflected in the WCETs assigned the tasks in Figure 7.2b. The *build_task_group1* algorithm begins by observing the time node with value G, and the suspension point which follows. The WCET of the first task, *T1*, is the sum of three components. The first is the time to change context to P the first time, *CS1*, since this is the first task. The second component is the WCET of the first execution episode, G. The third component is the time required to enter the suspension point, *E1*. The value *E1* is the maximum entry time required by any of the system calls represented by the suspension point. The maximum must be considered because, for example, the system call implementing synchronous communication has different entry and exit times than the system call implementing *delay*. The task *T1* is added to the task set $T(P)$ and precedence precedence relation $(Begin, T1, 0)$ which states that task *T1* is the successor of the *Begin* node, is added to the precedence relation set $PR(P)$. The use of the *Begin* node was discussed in Section 4.2.1.

When *build_task_group1* moves on to the second time node, it creates the second task *T2*, whose WCET also has three components: *X1*, the exit time for the first suspension point, C, the WCET of the time node, and *E2*, the entry time of the next suspension point. The suspension point marking the beginning of this episode represents the beginning of two resource use blocks, for resources X and Y. Consequently, task *T2* uses these resources in the modes and for the durations specified. The precedence relation $(T1, T2, 0)$ noting that *T1* must finish executing before *T2* may begin is added to the precedence relation set $PR(P)$. The third task, *T3*, is created in a similar fashion. Its WCET includes the time to exit from the second suspension point *X2*, the WCET H of the time node, and the time required to change

context back to the dispatcher after P has finished execution, *CS2*. The precedence relation $(T2, T3, 0)$ is added to $PR(P)$ to complete the task group.

The derivation of the task group illustrated in Figure 7.2b from the ITG of Figure 7.2a demonstrates the essential aspects of how the task group for a single process is constructed from its ITG. When a computation is implemented using a group of processes, then the task group representing the computation *as a whole* must be assembled from the task groups produced by *build_task_group1* which represent each process. The process group description, as discussed in Section 4.2.1.2, specifies the precedence relations between processes using a successor list, just as the task group description does.

It is often convenient to say that a task T *represents* a process P when it is a member of the group constructed from P's ITG, that is, when $T \in TG(P)$. Using this terminology, we can divide the precedence relations, $PR(TG(C))$, among tasks in a group describing a *computation* C, implemented by more than one process, into two classes: those between tasks representing the same process, and those between tasks representing different processes. The precedence relations between tasks representing the same process are all derived when the task group representing each process is created. The precedence relations between tasks representing different processes are those derived from the process level precedence relations specifying the process group structure.

Consider a computation implemented by a process group C consisting of two processes P_1 and P_2, where P_1 must complete before P_2 begins execution. Assume further that a delay of 3 time units is required between the completion of P_1 and the start of P_2. The precedence relation $(P_1, P_2, 3)$ expresses that constraint and must be translated into precedence relations between tasks representing P_1 and P_2. These

relations will hold between tasks in the *end* set of P_1 and those in the *begin* set of P_2. The *begin* set of the task group representing a process P is already represented by the successor list describing the task group structure:

Definition 8: The *begin set*, $Begin(Tasks(P))$, of the task group $TG(P)$ representing a process P is the set of tasks:

$$Begin(Tasks(P)) = \{t \mid t \in Tasks(P) \wedge (Begin, t, *) \in PR(P)\}$$

where $(Begin, t, *)$ matches any precedence relation stating that t is a successor of *Begin*, regardless of the delay value.

These are the tasks in the group that are initially eligible to run. The *end* set is not explicitly represented by the successor list, but is easily derived from it.

Definition 9: The *end set*, $End(Tasks(P))$, of the task group $TG(P)$ representing a process P is the set of tasks:

$$End(Tasks(P)) = \{t \mid t \in Tasks(P) \wedge \nexists x \in Tasks(P) s.t. (t, x, *) \in PR(P)\}$$

where $(t, x, *)$ matches any precedence relation stating that the t must complete before x may begin regardless of the delay value.

The end set thus comprises the elements of $Tasks(P)$ which have no successors. When every element of the end set has completed its execution the computation it represents is complete.

The set of precedence relations $PR(TG(C))$ describing the structure of the task group $TG(C)$ representing the process group C is the union of several sets of precedence relations. $PR(TG(C))$ obviously includes the precedence relations among tasks representing the same process, $PR(TG(P_1))$ and $PR(TG(P_2))$, but it also includes

those derived from the process level precedence relations $PR(C)$ describing the structure of the process group, PR'. $PR(TG(C))$ for the example is thus:

$$PR(TG(C)) = PR(TG(P_1)) \cup PR(TG(P_2)) \cup PR'$$

The members of PR' are generated from the process level precedence relation $(P_1, P_2, 3)$, giving:

$$PR' = \{((P_1, e), (P_2, b), 3) \mid \begin{array}{l} e \in End(Tasks(P_1)), \\ b \in Begin(Tasks(P_2))\} \end{array}$$

Note that the members of PR' are specified using the tuple notation to identify tasks: (p, t) denotes the task t within the group representing the process p. More generally, the precedence relations $PR(TG(C))$ describing the structure of the task group $TG(C)$ representing a process group C is:

$$PR(TG(C)) = \left(\bigcup_{P \in Procs(C)} PR(TG(P)) \right) \cup PR' \qquad (7.1)$$

Equation 7.1 states that the set of task level precedence relations specifying the structure of the task group representing the computation is the union of the precedence relation sets of the task groups representing each process in the group, as well as PR'. The elements of PR' are the task level precedence relations generated by the process level precedence relations in $PR(C)$ describing the structure of the process group:

$$PR' = \{((P_1, e), (P_2, b), d) \mid \begin{array}{l} (P_1, P_2, d) \in PR(C), \\ e \in End(Tasks(P_1)), \\ b \in Begin(Tasks(P_2))\} \end{array} \quad (7.2)$$

Equations 7.1 and 7.2 specify, abstractly, how to create the task group describing the computation as a whole from the task groups describing each process. How we chose to *implement* the operations described was influenced by practical considerations. The calculations must obviously be performed after the executables, and thus the task group descriptions, for each process are produced by *spr_ld*. The calculations could be done as a final compilation step, but we found it convenient to perform them while downloading the process executables during system initialization. In the current implementation of Spring, this means that the final construction of the task group is done while the Spring debugger *sbug* downloads the system and application software onto the Spring node, as illustrated in Figure 3.3.

One reason for this approach arises from the nature of the run-time data structures required by the scheduler, which are built at system initialization time. The scheduler requires a set of data structures describing the task group for each computation which it can use efficiently. Precedence relations are expressed by *spr_ld* in terms of task and process names, which is how they are stored in compiled SDL form. This is reasonable since when a developer wishes to examine the information it is in understandable form. Relations expressed in this form are not, however, efficient at run-time.

Instead, precedence relations expressed in terms of process names and task names in the SDL are translated into pointers between data structures during system initialization. Since the scheduler specific run-time data structures must be built at initialization time in any case, and it is simple to build them directly from the SDL information about the process group structure, and from the task group information

in each executable, it seemed pointless to add another processing step. Another reason arises from when transmission delay times for virtual circuits can be known, as discussed in Section 7.2.

The Spring debugger *sbug* controls Spring system booting and initialization, which includes process activation. It begins by reading the SDL *layout* section, described in Section 4.2.6, from the *full.db* file. The layout describes the set of processes and process groups that must be loaded onto each processor of the Spring node, as discussed in Section 4.2.6. *Sbug* begins by downloading the system executable image onto each target node processor, and starting their execution. Each system executable takes care of initializing itself, allocating resource pools, and sundry other actions required to prepare the system to activate the set of processes specified by the *layout.*

Sbug then downloads each process and process group onto the system, which we call the *process activation* phase. The portion of the Spring system which communicates with *sbug* during process activation also runs a system process which handles many of the operations required to activate an application process, the Process Activation Process (PAP). The PAP includes a scheduler specific initialization phase which builds and initializes the run-time data structures for each task in the group representing a computation, thus implementing Equations 7.1 and 7.2. The PAP uses the SDL information supplied by *sbug* and obtained from the processes' executable files.

7.2 Processes with Synchronous Communication

Section 7.1 first discussed how the task group representing a single process is constructed. It then considered how the task group representing a computation implemented as a group of processes is assembled from the task groups representing its

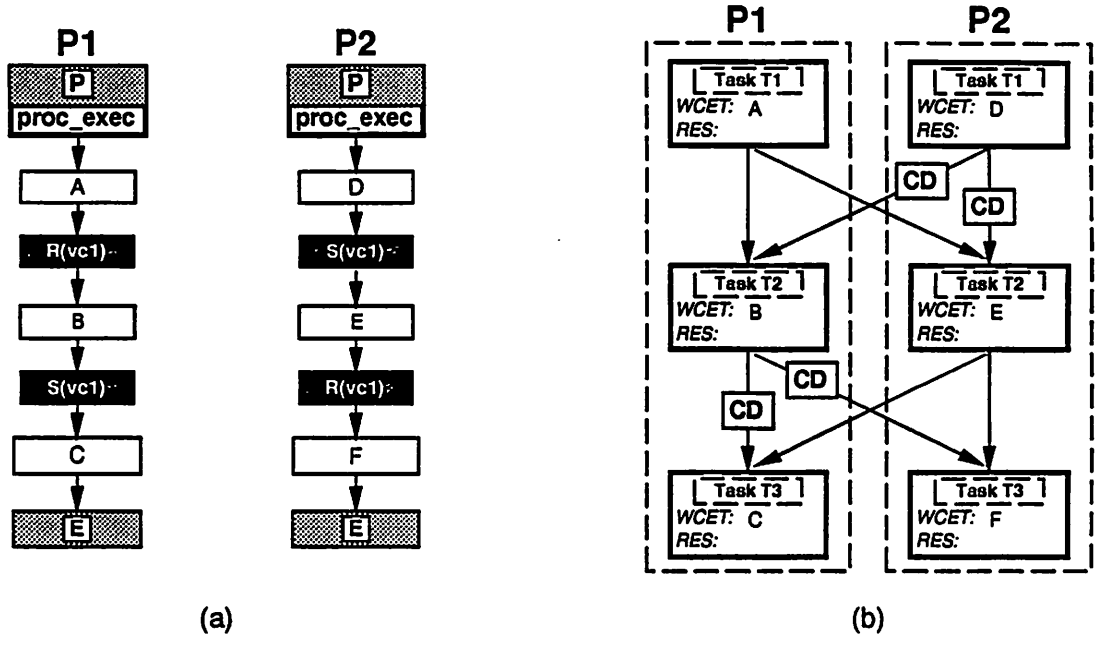


Figure 7.3. Processes Using Synchronous IPC and Corresponding Task Group

component processes, using the additional information provided by the precedence relations describing the process group structure. Synchronous communication statements within a process's code imply additional precedence relations that cross process boundaries to enforce the communication semantics. This section shows how these additional precedence relations are created.

The synchronous communication commands in the Spring-C language, discussed in Section 4.1, permit the developer to specify the synchronous sending and receiving of messages across named virtual circuits. The Spring-C compiler translates these statements into send and receive suspension points in the RTL intermediate representation of the procedure, as discussed in Section 5.1.1. The suspension points are reflected in the original time graph, as discussed in Section 5.2, and are combined with other suspension points during subgraph reduction, as discussed in Section 6.3. Procedures containing synchronous communication suspension points are treated no

differently, and behave no differently, than procedures containing other types of suspension points throughout the phases of compilation ending with the production of the ITG for a process.

Figure 7.3a illustrates the ITGs of two processes *P1* and *P2* which engage in synchronous communication. Both processes perform some initial processing, and then *P2* sends a synchronous message to *P1*. This might be a request for service from the client, *P2*, to the server, *P1*, with the second send-receive pair returning the results from *P1* to *P2*. The semantics of synchronous communication under Spring requires that both *P1* and *P2* must have *entered* their respective calls before either may leave. Further, enough time must elapse between when the sending process transmits the message and when it resumes execution to guarantee that the message has been delivered to the receiving side. We currently assume that no messages are lost.

Recall from the discussion in Section 7.1 that suspension points mark the boundaries of the process's execution episodes, and that the WCET of the task has three components. The three components are the worst case exit time of the suspension point preceding the episode, the WCET of the time node representing the episode, and the worst case entry time for the suspension point marking the end of the task. In Figure 7.3b, for example, the WCET of task $(P1, T1)$ includes the process's entry into the *sync_recv* system call, while the WCET of task $(P1, T2)$ includes the process's exit from it. The corresponding task boundary in *P2* represents the process's entry to and exit from the *sync_send* system call.

Figure 7.3b shows the task group constructed to represent the behavior of this process group. The precedence relations between tasks representing the same process were already discussed in Section 7.1. Those between tasks representing different

processes are the ones added by the methods described in this section. The precedence relation from the first task in *P1* to the second task in *P2*, $((P1,T1),(P2,T2),0)$, implements the semantic restriction that *P1* must enter its *sync_receive* statement before *P2* may leave its *sync_send* statement.

The precedence relation $((P2,T1),(P1,T2),(CD\ vc1))$ implements the restriction that *P2* must enter its send statement before *P1* leaves its receive. The communication delay $(CD\ vc1)$ in this relation and in $((P2,T1),(P2,T2),(CD\ vc1))$ ensures that neither of the processes may return from their respective synchronous communication calls until enough time has passed, determined by the communication delay of the virtual circuit $vc1$, to guarantee the message has reached the receiving process. The delay is expressed symbolically because it is a property of the virtual circuit established during the process activation phase of system initialization, and is thus not known at compile time. The PAP determines the numeric value of the delay for any precedence relation associated with synchronous communication when the process is activated at boot time and the delay of the virtual circuit known.

In Section 7.1 we discussed how the *build_task_group1* algorithm is used to build the task group representing the process, using the process's ITG as input. That algorithm was, however, limited to processes containing no synchronous communication suspension points. The *build_task_group2* algorithm includes the extensions required to handle precedence relations arising from synchronous communication suspension points. It is described in detail in Appendix C, but an example should be sufficient to understand its essential features.

Figure 7.4 illustrates the derivation of the task groups for two processes. Figure 7.4a shows the ITGs of the processes which exchange messages synchronously, and Figure 7.4b shows the task groups constructed for each process with the unresolved

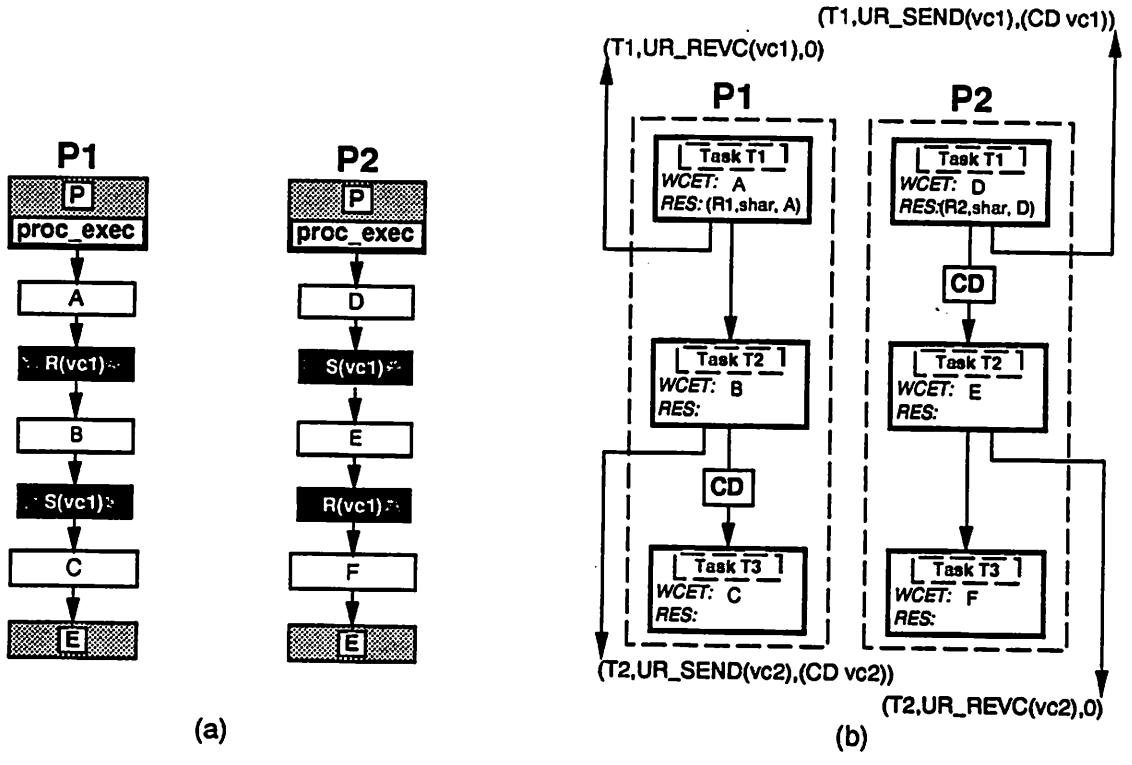


Figure 7.4. Unresolved Precedence Relation Derivation Example

precedence relations generated by the synchronous communication suspension points. The *build_task_group2* scans through the ITG of each process as before, creating tasks and noting their resource use from the suspension points that surround each time node. However, it also generates a new type of precedence relation when a suspension point represents a synchronous communication act.

For example, when it creates the first task for process *P1*, the fact that the suspension point represents a synchronous receive causes it to generate the unresolved precedence constraint *(T1,UR_RECV(vc1),0)*, which states that *T1* is the predecessor of a task representing the process on the other side of the virtual circuit *vc1*, but that the name of that task is not yet known. The unresolved relation notes that no delay is associated with it, and that it represents a receive operation on the virtual circuit *vc1*.

The unresolved precedence relation will, when resolved, become the relation between task *T1* representing process *P1*, and task *T2* representing process *P2* in Figure 7.4b.

As *build_task_group2* continues, it encounters the second suspension point which represents a synchronous send. Its actions are almost the same as for the receive, with the exception that the synchronous send implies a delay, as illustrated in Figure 7.4b. The unresolved precedence relation *(T2,UR_SEND(vc2),CD(vc2))* notes that it holds between task *T2* and a task representing the process on the other side of virtual circuit *vc2* whose name is not known, and that the relation has a delay whose magnitude is determined by the properties of the virtual circuit. Note that the same delay is applied to the precedence relation *(T2,T3,CD(vc2))*, which is a relation between two tasks prepresenting *P1*. The algorithm performs similar operations on the ITG of *P2*, producing the task group illustrated in Figure 7.4b.

These are the task group descriptions which are produced as part of the executable files for each process by *spr_ld*. Recall that the SDL information in the executables of a computation's component processes is used by the PAP during system initialization to construct the task group for the computation as a whole. When synchronous communication is involved, the task group description in the process executable, as *spr_ld* produces it, is not sufficient to build the run-time data structures required. The unresolved precedence relations must be handled before the processing of the executable files is complete.

The *resolve_sync_comm* algorithm, discussed in detail in Appendix C, handles the precedence relation resolution problem and is illustrated here with several examples. Figure 7.5a illustrates the task group descriptions produced by the *build_task_group2* algorithm for the two procedures of Figure 7.3a, including the unresolved precedence relations generated by the synchronous communication suspension points. Figure 7.5b

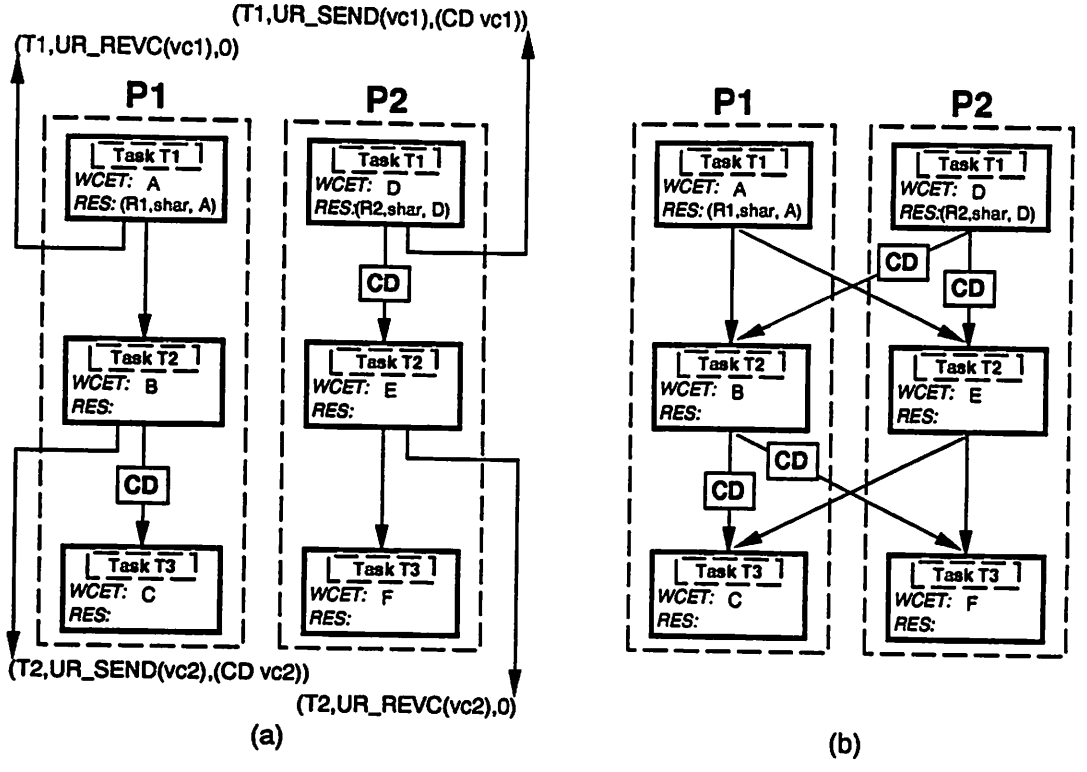


Figure 7.5. Unresolved Precedence Relation Resolution Example

shows the task group, matching that of Figure 7.3b, produced by *resolve_sync_comm* after it resolves the necessary precedence relations.

The basic approach is to scan from the beginning to the end of the task group representing each process looking for complementary (send-receive) pairs of unresolved precedence relations. The success of the method rests on the fact that each synchronous communication suspension point is labeled with the name of the virtual circuit it uses. Recall that this was one of the additions to the Spring-C syntax discussed in Section 4.1. Under the current implementation each virtual circuit is currently limited to a single sender and receiver, and each connection is thus half-duplex. It should not be hard to extend *resolve_sync_comm* to support full-duplex connections and to support multiple receivers, thus supporting broadcasting, but that will be done as part of future work. Since there is only a single sender and receiver on each virtual

circuit, the only problem is making sure that the *sequence* of complementary unresolved precedence relations associated with the send and receive actions are resolved in the proper order.

When processing the task groups of Figure 7.5a, the algorithm begins by reading the SDL sections of the two process's executable files. The algorithm then scans the successor lists of the processes' task group descriptions, and constructs the two *ordered* sets $UR1$ and $UR2$ containing the unresolved precedence relations of each process *in the order that they appear within the successor lists.* The ordering is important since the algorithm resolves *pairs* of unresolved precedence relations, and the order in which they are paired has crucial implications for the semantics of the synchronous communication.

The elements of an unresolved relation set are of the form $(type, vc, proc, task)$, where *type* indicates whether the unresolved reference arises from a send or receive operation, *vc* names the virtual circuit involved, and *proc* and *task* identify the task following the suspension point. This is the task to which the complementary unresolved precedence relation will be directed. For the example of Figure 7.5a, the set of unresolved precedence relations for process *P1*, *UR1*, contains two elements. The first element is associated with the receive suspension point, and the second element is associated with the send suspension point. The set of unresolved references for process *P2* also has two elements, giving:

$$\begin{array}{lcl} \text{UR1} & = & \{\ \text{(R,vc1,P1,T2)}\ \ \text{(S,vc2,P2,T3)}\ \ \} \\ \text{UR2} & = & \{\ \text{(S,vc1,P2,T2)}\ \ \text{(R,vc2,P2,T3)}\ \ \} \end{array}$$

The *resolve_sync_comm* algorithm maintains a pointer for each set of unresolved references which identifies the next element awaiting resolution in each set. We call the set of relations referenced by these pointers the *current set*, since they are the

relations currently under consideration. The algorithm then iterates, checking the current set for complementary pairs on each iteration. The example is simple, so there is only one pair of unresolved relations in the current set to check, but it should be clear that they *are* complementary, and are thus used to resolve one another.

The missing process and task fields of the unresolved precedence relation arising from the receive are taken from its complement, and are set to *P2* and *T2*, respectively. Similarly, the missing process and task fields of the unresolved relation arising from the send are taken from the information supplied by the receive, and are set to *P1* and *T2*, respectively. Each of the pointers into the unresolved relation sets are advanced, changing the members of the current set, and the algorithm begins the next iteration. The next pair are also complementary, and are resolved in a similar way. When the last reference is fully resolved, the task group is complete, as illustrated in Figure 7.5.

This demonstrates that *resolve_sync_comm* correctly resolves the precedence relations for the fairly simple case illustrated. Real application programs might present more complex situations, but the algorithm will still construct a correct task group, as long as the unresolved relations appear in an order which is compatible with their representation by precedence relations within a task group.

Programs can, however, be written that specify contradictory communication behavior. It is fairly easy to illustrate how this can happen, and how *resolve_sync_comm* detects the error, but the discussion will be more compact if we use a simplified notation for an unresolved reference, *XN*, where *X* is either *R* for receive or *S* for send, and *N* is the virtual circuit. So, for example, *R2* represents an unresolve precedence relation generated by a receive on virtual circuit 2. Subscripts are used to distinguish multiple instances of the same operation on a virtual circuit. For example, $R2_1$ and $R2_2$ represent the first and second receive operations on virtual circuit 2, respectively.

Now consider a set of three processes which all communicate with one another. *P1* receives messages from *P2* through virtual circuit 1, and sends messages to *P2* through virtual circuit 2. *P1* also sends messages to *P3* through virtual circuit 3. If the processes' synchronous communication is properly ordered, they might generate the following sets of unresolved precedence relations:

$$\begin{array}{lcl} \text{UR1} & = & \{\ R1_1 \quad S3_1 \quad S2 \quad R1_2 \quad S3_2 \quad R1_3\ \} \\ \text{UR2} & = & \{\ S1_1 \quad\quad R2 \quad S1_2 \quad\quad S1_3\ \} \\ \text{UR3} & = & \{\ \quad R3_1 \quad\quad\quad R3_2 \quad\quad\ \} \end{array}$$

Within the initial current set, $\{R1_1 S1_1 R3_1\}$, the send and receive on virtual circuit 1 are complementary. When the pointers into the current set are updated, the next iteration scans $\{S3_1 R2 R3_1\}$ for a complementary pair. As *resolve_sync_comm* iterates, it always finds a complementary pair among the current set, and eventually every member of the unresolved sets is resolved and the task group for the computation is complete.

If, however, the synchronous communication pattern is contradictory, then the *resolve_sync_comm* algorithm will detect the problem by being unable to find a complementary pair of unresolved precedence relations while the unresolved sets are non null. A contradictory pattern of behavior in this example would occur, for example, if the order of the $S2$ and $R1_2$ synchronous communication actions of *P1* were reversed, giving:

$$\begin{array}{lcl} \text{UR1} & = & \{\ R1_1 \quad S3_1 \quad R1_2 \quad S2 \quad S3_2 \quad R1_3\ \} \\ \text{UR2} & = & \{\ S1_1 \quad\quad R2 \quad S1_2 \quad\quad S1_3\ \} \\ \text{UR3} & = & \{\ \quad R3_1 \quad\quad\quad R3_2 \quad\quad\ \} \end{array}$$

This pattern is contradictory because it states that *P1* should receive a second message on virtual circuit 1 *before* it sends a message on virtual circuit 2, but it also

states that *P2* will receive the message on virtual circuit 2 before it sends the second message on virtual circuit 1. When *resolve_sync_comm* begins execution the current set is $\{R1_1 S1_1 R3_1\}$, within which $R1_1$ and $S1_1$ are a complementary pair. The current set then becomes $\{S3_1 R2 R3_1\}$, within which $S3_1$ and $R3_1$ are a complementary pair. In the next iteration the current set is $\{R1_2 R2 R3_2\}$ within which no complementary pair exists, because of the contradictory behavior of *P1* and *P2*. At this point *resolve_sync_comm* realizes there is a contradiction, and announces the error.

This example shows that *resolve_sync_comm* can also handle more complex communication patterns among members of a process group, and that the algorithm can detect contradictory patterns of communication. The algorithm will always succeed for sets of processes with non-contradictory communication patterns because there will always be at least one complementary pair of unresolved relations in the current set, and resolution will continue until no relations are left unresolved. The algorithm will always detect contradictory communication patterns because the contradictory relations will never be resolved, and the algorithm will thus reach a point where the current set is non-null, but contains no complementary pairs. The description of the *resolve_sync_comm* algorithm completes the set of algorithms required to produce a task group description for computations implemented as a process or group of processes written in the Spring-C language.

All Spring-C programs which follow the documented programming conventions and restrictions, whether they use synchronous communication or not, will be successfully compiled by the set of algorithms described in this dissertation. The conventional aspects of compilation produce an executable file which is used in the normal way to create an executable image of a process. The novel aspects of the compilation, the analysis and prediction methods described in this dissertation, add the task group

description of the process's execution time target behaviors required by the scheduler. The task group descriptions from the executables for each process in a group, as well as the process level descriptive information, are used to build a task group describing the behavior of computations implemented by groups of processes.

Building the task groups in this way means that the system has a *static* view of the process group implementing a computation. This is significantly different from a conventional system's view, where a group of processes implementing a computation is usually created *dynamically*, and where the group's structure can be influenced by properties of the input data. Such a flexible software architecture does confer some advantages, but it does not support sufficient predictability for real-time systems providing the notion of guaranteed execution. As discussed in this dissertation, the structure of the task group, and thus of the process group, must be known when the computation is scheduled. As the program analysis and task group construction methods improve, it may be possible to make the task group structure less static. One possible way to do this is to make the group structure a function of specific input parameters. However, this is part of possible future work, and is not considered further in this dissertation.

7.3 Evaluation

This chapter has presented the methods used to construct the task group representation of a process from its ITG. The evaluation of these methods falls into two major categories: test cases and application programs. The construction methods described are simple, and the testing is correspondingly less complex than that for other aspects of the SGS. For the simplest case of a computation implemented by a single process,

the only variations come with the number of tasks and differences in combinations of suspension point types represented in the ITG.

The test cases are complete, in the sense that the processing of each suspension point type (explicit delay, resource use, and synchronous communication) is tested. The implementation of the methods described in this chapter correctly assigned the WCET of the tasks, chose the proper delay associated with each precedence relation, and correctly noted all resource use. The construction of the task group representing a process group from the task groups representing the component processes has also been demonstrated by a set of thirteen test cases implementing different process group structures. The program implementing the *resolve_sync_comm* algorithm, *spr-sync-comm*, resolved precedence relations of processes performing synchronous communication, and produced a correct task group. The debugger *sbug* successfully read the layout and process group information from the *full.db* file and used it to download the processes' executable files. The PAP noted the start of each process group, and constructed the run-time data structures describing the task group for the computation as the individual processes in the group were activated. The PAP also noted the presence of precedence relations representing synchronous communication and handled them properly.

The test cases do not, however, test every combination of suspension point properties, and as they are used in the future on a variety of application software some minor problems may be revealed. However, the simplicity of the various algorithms gives us confidence that since each basic situation is successfully handled, any problems with the current implementation will be easily corrected. One limitation is that the current implementation of *spr-sync-comm* only handles two processes at a time.

Rewriting it to handle any number of processes should not be difficult, and is part of our future work.

The use of *spr-ld* to produce task group descriptions for actual application programs is still limited because development of real application software is just beginning. A few simple processes in the application simulating the robotic assembly cell for circuit boards, discussed in Section 4.3, have been compiled. These included process groups using process level precedence constraints and engaging in synchronous communication. The full implementation of this application is not, however, yet complete. Some of the programs for the integrated manufacturing application, also discussed in Section 4.3, are currently being implemented. This application also involves a number of process groups using process level precedence constraints and engaging in synchronous communication. Work on this software is still preliminary, and does not yet make significnat demands on the implementation of the task group construction methods.

The software implemented so far has all been simple and relatively undemanding, but the success of the test cases gives us confidence that the approach described here is viable. Problems may arise as more and more applications are developed, but the simplicity of the methods required to build task groups indicates that any problems should be easy to resolve.

CHAPTER 8

SYSTEM ISSUES

This chapter considers how real-time requirements affect the operating system structure and other aspects of the system's design and implementation. Chapter 3 discussed how the design of the different levels of a real-time system had to be more closely coordinated than those of a non-real-time system because of the stringent requirement that the system support predictable execution of real-time computations. Chapters 5 through 7 described how Spring-C programs are analyzed during compilation, how predictions about their worst case execution time target behaviors are made, and how these predictions are used to build a task group describing the behavior of a computation. These behavioral predictions are based, however, on several assumptions about the underlying system support. This chapter focuses on how the design and implementation of the system are affected by the need to satisfy these assumptions, and how the implementation of Spring exemplifies the issues involved.

The real-time operating system must support and manage the execution of the application processes *predictably*, which in this context means that the properties of the system must make it possible to produce valid behavioral predictions for the computations executing on the system. This includes all aspects of the system: scheduling, memory management, context switching, system calls, and interprocess communication. The designs of each must be compatible with that of the other system layers, and with the overall goal of predictability, as discussed in Chapter 3.

While each of these issues offers points of interest, and subtle pitfalls, some of the most illustrative examples are presented by our approaches to: system support for accurate behavioral prediction, providing support for logical address spaces, and extending the original scheduler implementation to handle groups of precedence related tasks. Even the features of the hardware vary in their suitability for implementing a predictable system. The properties of the MMU, for example, affect the ability of the memory management implementation to support logical address spaces which can be used predictably. In another area of the system, the design of the CPU affects the predictability of instruction sequence WCETs, as does the method used by the system to handle both internal and external interrupts.

The rest of this chapter illustrates the need for an integrated approach to real-time system design, the vertical slice approach introduced in Chapter 3, by discussing how the design of the system as a whole should be oriented toward support for predictable execution, and then considering more specific aspects of our approach to memory management and scheduling.

8.1 System Support for Predictability

In this section we discuss how several aspects of the system and the underlying hardware might be explicitly designed to support the methods for behavioral prediction described in earlier parts of this dissertation. Many of the ideas in this section were developed as part of SpringNet, an architecture designed to support high performance computing of critical, distributed real-time systems [86]. The architecture consists of a multidimensional network of the multiprocessor Spring nodes discussed in Chapter 3, and is one part of an integrated system-wide solution. In this section we consider the SpringNet architecture at three levels of detail: system, functional,

and component. The system and functional level discussions reflect the completed, current, or planned development of the system. The component level discussion is often speculative, proposing architectures which do not yet exist, but is guided by our experiences with the current target hardware and its support for the current system implementation.

8.1.1 System Level

SpringNet is a physically distributed system composed of a network of multiprocessors each running the Spring Kernel. Each multiprocessor, as discussed in Section 3.2 and illustrated in Figure 3.4, contains one (or more) application processors(AP), one (or more) system processors(SP), and an I/O subsystem. APs execute previously guaranteed tasks as specified in the execution plan constructed by the scheduler executing on one or more SPs. SPs offload the scheduling algorithm and other OS overhead from the APs both for speed, and *so that external interrupts and OS overhead do not cause uncertainty in executing guaranteed processes on the APs.* The I/O subsystem is partitioned from the Spring Kernel, handling non-critical I/O, slow I/O devices, and fast sensors.

Each node contains 4 processors with 4Mb of local memory each, which is also visible to every other processor in the node using the system bus. Currently there are 3 multiprocessor nodes connected via two networks; an ethernet to support non real-time traffic and for downloading the system from the development platform, and a fiber optic register insertion ring connecting 2 Mb replicated memory boards in each node, corresponding to a single vertical ring in Figure 8.1 which illustrates the SpringNet network architecture. The replicated memory provides a shared memory

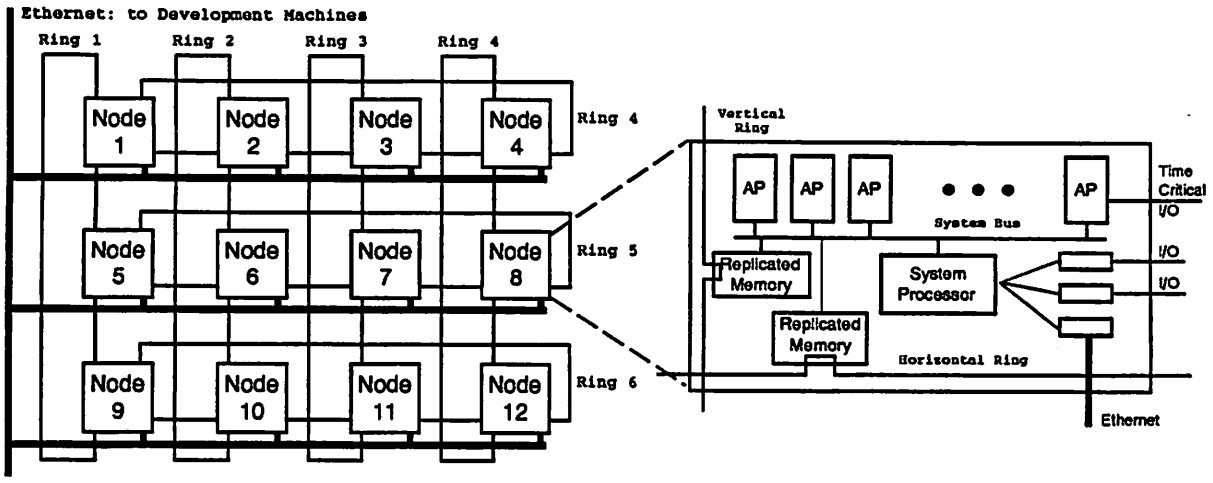


Figure 8.1. SpringNet Network and Node Architecture

model for its 2 Mb (of physically distributed but logically centralized memory), and is implemented via the off-the-shelf product called Scramnet [90].

Small scale helps achieve predictability within a single node, while predictability across physically distributed nodes is supported by the replicated memory and the higher level software for end-to-end scheduling and real-time virtual circuits. The replicated memory can also be used to support shared memory across node boundaries and for fault tolerance when more than one ring is used to connect a set of Spring nodes in a grid. Important data structures written to the replicated memory are automatically reflected in the other boards on the ring. The duplicate information is useful in recovering from several classes of node failure faults. It is possible to enhance fault tolerance by adding *parallel* replicated memory rings. The SpringNet architecture can scale by connecting rings of replicated memory in an n-dimensional grid. Figure 8.1 illustrates a 2-dimensional grid with one replicated memory ring for each row and column. Even though the SpringNet architecture resembles a multicomputer, it is important to note that the SpringNet architecture can be physically

distributed, limited only by the maximum fiber optic ring size. The current implementation contains three SpringNet nodes connected by a single reflective memory ring.

8.1.2 Functional Level

The system architecture described above facilitates several important aspects of the Spring paradigm. Among the most important is *functional partitioning.* For example, a node might contain one conventional system processor, and a special coprocessor addressing all or part of the problem of constructing a schedule [9, 60], as discussed in Section 8.3.2. Functional partitioning provides other benefits, including the ability to separately manage different classes of computations, currently termed critical, essential and non-essential. The system also enhances predictability by shielding application processes from interrupts. Environmental interrupts directly affect the system processor and I/O front ends, indirectly affecting applications by generating work whose execution must be added to the execution plan maintained by the on-line scheduler.

The protection of application processes from external interrupts, along with the scheduler providing guaranteed execution, allows the construction of a more *macroscopic* view of a predictable system. Context switches are reduced, predicting computation behavior is made simpler, which simplifies guaranteeing that a task will complete by its deadline. The SpringNet strategy also partitions the real-time processes into those that require static resource allocation, those requiring a dynamic scheduling algorithm in the front-end, and those, which typically have higher levels of functionality and greater latency, handled by the dynamic on-line guarantee routine.

Those requiring static allocation are typically fast I/O device drivers and critical processes. Slow I/O devices can be multiplexed through a front-end processor, which might use a cyclic or rate monotonic scheduler. The APs support the higher level application processes given dynamic on-line guarantees.

The *microscopic* predictability of the system is addressed by the methods described in this dissertation for segmenting the worst case behavior of every process, including their use of operating system primitives, into a group of *time and resource bounded* tasks. The Spring-C language described in Chapter 4 supports the implementation of computations as a set of processes and process groups. The translation methods described in Chapters 5, 6, and 7, describe how the software generation system derives a description of the execution time behavior of those computations as a group of tasks, whose execution is guaranteed by the scheduler.

Many current real-time scheduling algorithms schedule process execution independently of other resource use, leaving the processes to block at arbitrary points due to resource contention. The Spring approach uses the representation of processes by task groups, and the task's predicted WCET and resource use to integrate CPU scheduling and resource allocation, thus *controlling* resource contention so that tasks, and processes, execute as predicted. The run time information about processes and process groups, as discussed in Section 4.2, also includes: deadlines, importance level, precedence constraints, and fault tolerance requirements. Kernel primitives to read all of this information, and alter some of it, enhance system flexibility.

Most *application* computations comprise a set of smaller dispatchable units, but most current real-time kernels do not support a set of items with a single deadline. The Spring Kernel supports task and process groups, where the execution order of group members is constrained by precedence relations, leading to the notion of

end-to-end scheduling within and across node boundaries. The Spring Kernel and scheduler also support the notion of a real-time virtual circuit. For example, a periodic process group might consist of a producer and consumer process exchanging data through the IPC facilities. The virtual circuit is set up during system boot in the current implementation, including allocation of processing resources required to support it, and supports bounded transmission time thereafter. The scheduler takes the precedence relations at the process and task level into account when scheduling the task group representing the computation implemented as a process group. Note that the replicated memory is a very efficient way to support predictable real-time communication since there are no complicated layers of communication software. In the future we want to extend virtual circuit support to communication media with other properties, and to schedule the processing required to support the virtual circuits more dynamically.

There are two other aspects of the Spring paradigm which are important, but which we mention only briefly here since they do not directly bear on the subject of this dissertation. These and other aspects of the Spring paradigm are considered in more detail elsewhere [87]. First, the *planning* aspect of the scheduler enables it to *anticipate* deadline violations, and consider the overall system state when a process cannot be guaranteed. Other real-time schedulers have a myopic view, only knowing *which process to run next*, without regard to the feasibility of its deadline. The other is that when the Spring scheduler fails to guarantee an essential processes, the *separation of deadline and importance* in our paradigm enables it to shed load selectively by eliminating the least important processes[7].

8.1.3 Component Level

One of the most significant points made by several sections of this dissertation is that for real-time systems *predictable worst case behavior* is the relevant design criterion, rather than average case performance which is appropriate for conventional systems, and that the use of a new criterion leads to significantly different design decisions. WCET is the most obvious behavior for which predictions are required, and the ability to produce them, as discussed in Chapters 5, 6, and 7, is affected by the architecture at all levels of the system, including: process execution management, interrupt handling, caches, and pipelines.

Complex designs for deeply pipelined machines with large caches, and shared memory accessed across a system bus in a multiprocessor node can make it difficult, or impossible, to predict WCETs for sequences of instructions without being extremely pessimistic. It would be better to use an architecture with execution times that are both *predictable* and have *low variance.* Our experiences in developing SpringNet, and those described in this dissertation, have confirmed the importance of two design principles: each component must be as predictable as possible, and slower but predictable is generally preferable to faster but less predictable. The RISC approach tends toward predictable designs, but as designers strive for better average performance some problematic features of CISC designs are being re-introduced. The rest of this section will discuss important features of the CPUs and several coprocessors.

8.1.3.1 The Application Processors

The CPU design must support reasonably accurate and simple WCET prediction. For example, pipelining is a popular way to increase throughput, but some of the more complex schemes can actually *increase* the WCET. Those using dynamic instruction

scheduling and requiring complex scoreboards can have highly variable fill and drain times, while many pipelines handling interrupts and cache misses require redoing some stages. Pipelined or not, an accurate model of the machine *must* be available at compile time to support the prediction of WCET for instruction sequences. Chapters 5 and 6 discussed the use of the target machine timing model by the translation method presented in this dissertation.

Our experience with the current target CPU, the 68020, is illustrative. Section 6.6.2 discussed the relative merits and prediction accuracy of three timing models, called the old, new, and sequence tools. One of the most obvious points is that the CPU manual was never intended for *exact* WCET prediction and does not describe the pipeline and other internal features affecting WCET. The old tool, which uses the information from the manual, performs badly as a result, and most significantly, produces WCET predictions that are too low in many cases. The new tool attempted to overcome the limitations of the old by *measuring* the WCET of each instruction. This tool still overestimated the WCET since the WCET for each instruction were measured separately and it is not able to take any interactions between instructions in a sequence into account. The sequence tool addressed this problem by measuring the WCET for sequences of instructions. While the sequence tool generally performed best, the results clearly showed that significant room for improvement in the accuracy of the target machine timing model remains.

The example of the 68020 indicates many of the potential problems, affected by adopting predictability of WCET as a design criterion. In theory, the load and store architecture, simple instructions, and uniform instruction size and execution time of a simple RISC design make pipelining simple, fast, and predictable, although in practice, branches, interrupts, and long instructions (like multiply) complicate

the picture. The simplest approach to real-time CPU design is a non-pipelined, non-cached machine where all instructions require a well defined number of cycles. The Harris Semiconductor RTX 2000, for example, has no pipeline or cache, and all basic instructions execute in one cycle, while those accessing memory require two. If such a machine is *fast enough* then nothing else is required.

If not, we could add other CPUs for applications with available parallelism, or we might add a simple pipeline. If the structure of the pipeline is well documented, then it should be easy to build a timing tool which would be used by the compiler as described in Chapter 5, and which would accurately reflect the pipeline's influence on the WCET of instruction sequences. In essence, the tools would predict the WCET by simulating the progress of the instructions in the sequence through the pipe. The functional partitioning of the Spring node shields the APs from interrupts, and context switching happens only at well defined task boundaries, so the pipeline executes undisturbed. Note, however, that adding a cache and pipeline to a simple design so the machine *seems to be fast enough* is not the correct approach, only the *predictable* affects are any benefit.

If the design is still not fast enough, then it might be appropriate to consider methods for using a cache to *predictably* lower the WCET, an extension of the reduction methods discussed in Chapter 6 [59]. The lack of interrupts and the predictability of context switches in Spring enables permits the use of a direct mapped logically addressed instruction cache. Data caching is not currently considered, though it will be part of future work. Note, however, that only part of the cache's effects can be predicted, and its presence generally lowers the WCET while increasing the execution time variance. For example, the extended reduction rules currently assume that

either the entire body of a loop, including subroutines called, fits into the cache without conflict, or that none of it does. The current analysis thus fails to predict the significant speedup seen where cache conflict exists but is minimal. The subtlety of this and other aspects of the analysis is indicated by the fact that the execution path producing the worst case behavior *changes with the cache architecture.* This analysis interacts strongly with our work on predictably supporting logical address spaces, discussed in Section 8.2.

8.1.3.2 Special Purpose Coprocessors

A common technique used to increase the speed of performance critical operations in conventional systems is to implement them in hardware using special purpose coprocessors. Examples of such coprocessors include floating point accelerators, memory management units, device controllers, DMA, graphics accelerators, and VLSI implementations of application specific computations. As with other aspects of real-time systems, the difference between using the average case and predictable worst case performance as the coprocessor design criterion is crucial.

For example, many real-time applications systems using conventional hardware do not use floating point co-processors, usually called floating point units (FPU), because most designs achieve very fast average case execution at the expense of high WCET, and because the CPU pipeline and the FPU often operate asynchronously, creating the *precise interrupt problem* [79]. However, there are ways to support floating point predictably. One approach is to implement all floating point operations in software, but this will usually be too slow. A better solution is to add a FPU, but synchronize the CPU pipeline and the FPU.

When a floating point instruction appears, it is executed by the FPU at a much faster rate than if it were emulated in software, but still *in order*. Given the current level of chip densities it seems possible to put the FPU on-chip. The next problem is that the WCET of each FPU instruction must be known. However, the FPU need not be pipelined, since it executes each operation to completion, so this is simpler than for conventional designs. More subtle designs permitting concurrent execution of the FPU and CPU pipeline stages might improve preformance further, but would always have to be evaluated with respect to their predictable effect on the WCET.

We believe that next generation real-time systems will require the ability to predictably support logical address spaces, as distinguished from virtual memory under which it is difficult or impossible to derive a usable WCET because of paging delays. However, it *is* feasible to predictably support a logical address space which always resides in physical memory. Spring supports separate address spaces for each user process, with a single address space for all system mode execution, thus protecting each address space from unintentional modification by threads executing other spaces. In systems using the physical address space, such as VRTX [72], all threads can modify all of memory, leaving the system vulnerable to a single addressing error. Spring allows *controlled* address space overlap using the shared memory segments described in Section 4.2.3. The critical influence of address mapping on system performance requires the use of a coprocessor. Section 8.2 describes how a conventional memory management coprocessor is used in unconventional ways to provide predictable support for logical address spaces, discusses features that are desirable in an MMU designed explicitly for real-time systems, and considers how the features of several conventional MMU designs compare and contrast to the desired set.

Since the problem of creating an execution plan satisfying process deadlines is fundamental to the performance of the Spring system, it is reasonable to consider decreasing the planning time by providing hardware support. Such a coprocessor would have many advantages including: it could perform the scheduling operation faster, it could be continuously active to reduce scheduling latency, and it could perform various optimizations and preprocessing when not attempting to schedule a new process including: reordering the schedule, finding holes in which processes might be easily scheduled, and other optimizations. The SP's processing capacity could then be devoted to other system duties including interrupt handling. A possible disadvantage is that this new special purpose processor could be an example of a single point of failure if not backed up by redundant hardware, or the ability to run the scheduling algorithm on the system processor at need. A scheduling co-processor which performs the scheduling algorithm on a set of tasks has recently been designed and implemented [9, 60]. While this design is not a fully independent and continuously active co-processor, when validated it could become the core of a co-processor performing a wider range of operations on a larger set of information. Section 8.3.2 discusses the coprocessor design in greater detail.

8.2 Memory Management for Predictable Systems

This section begins by briefly discussing the sources of unpredictability in traditional memory management, then considers how address mapping can be made predictable. The discussion then turns to the implications of several specific aspects of the implementation, gives performance data, and concludes with a discussion of some properties MMUs designed specifically for real-time systems should, in our opinion, exhibit.

8.2.1 Problems with Traditional Memory Management

The 68851 MMU chip on each 68020 CPU board used in the current Spring implementation is an excellent context within which to consider a typical *virtual* memory access. This MMU chip was designed to provide address translation and memory protection for processes. The process's address space is divided into fixed size pages which are mapped to physical pages when in use. Unused pages in a process address space are not mapped. The MMU has a translation look-aside buffer (TLB) which caches 64 page mappings. When an address is submitted to the MMU, the TLB is checked first. If the relevant mapping is found, the address is translated without further delay.

If the relevant mapping is not found, the TLB flushes an entry as needed, and the MMU *automatically* loads the required entry into the TLB from the memory map. If the required virtual page is not currently mapped to a physical page, a page fault exception occurs. When the MMU is being used to do memory mapping, each memory reference is thus subject to a delay of three possible magnitudes: page fault, TLB loading, and TLB translation. If a real-time system implemented support for address mapping in this conventional way, then to compute the WCET it would be forced to assume a page fault delay on every memory reference. This would produce ludicrously large worst-case execution times, and be *extremely* pessimistic since most memory references do not actually create page faults.

8.2.2 Design of Predictable Memory Management

Many designers note that using virtual memory in a predictable real-time system is probably impossible and conclude that using physical memory is the only feasible solution, but using address mapping to support a logical address space confers benefits

beyond those of virtual memory. Predictably managing a logical address space is a different problem, and one with a reasonable solution. The solution used in the current Spring implementation combines two simple ideas:

1. Avoid page faults by preallocating, at process creation time, a physical page for every used page in a program's address space, and loading that page in memory.
2. Explicitly manage the contents of the TLB to ensure that *all* memory references experience TLB hits.

The first point eliminates unpredictability due to page faults. We use the term *logical address space* to distinguish this approach from virtual memory, since no demand paging occurs while the program is running. Of course, this scheme limits the number of pages used by the program to the number of physical pages the system has available to support the process, but this is generally acceptable in most application areas of real-time systems. While this approach is extremely simple, it is more flexible than many other real-time operating systems that force all pages to be memory resident but use physical addressing [72]. Physical addressing has limited protection, is less flexible when dynamically loading new programs, and makes it harder to support shared segments of the type described in Section 4.2.3.

The second point eliminates unpredictability due to TLB misses. Under this scheme, when a context switch occurs, the mappings for *all* the used pages in the logical space of a process are loaded into the TLB. This is a reasonable approach, under the assumption that processes are not preempted or interrupted at arbitrary points, since the TLB is loaded once when a task begins execution. One drawback of this approach is that it limits the number of pages a process can use to the number of TLB entries available. The TLB *always* contains the mappings for the portion

of the operating system space required to support process execution; they are never flushed. One goal of this solution is that the system need not make use of the process's memory maps during execution. Upon loading the TLB during context switch, the system marks the process's memory map *invalid.* Then any TLB miss, and thus any violation of the assumption that TLB misses do not occur during execution, is easily detected and handled as an exception.

One of the most obvious implications of this approach, used in the context of the existing 68851 MMU, is that size constraints abound. The TLB must hold the page mappings for the pages used by the current process and for the portion of the system space required to support program execution. Given the tradeoff between system and process space entries in the TLB, the current implementation tries to minimize the number of system pages used. The page size is 8K, permitting a fairly large system space to be mapped while leaving a reasonable number of TLB entries for the current process. One drawback of the relatively large page size is that physical memory suffers internal fragmentation.

Each process requires a minimum of four pages to support its text, data, user stack, and system stack segments. Some processes actually use only a small fraction of the minimum 32K consumed. There are a number of ways the system could save space, but they also compromise protection. For example, the system could put both stacks on the same page, but this would permit one to overflow into the other. The compiler could handle this by producing code that checks for overflow, but this would impose the significant performance penalty of always performing the check. For the current system we chose to accept the internal memory fragmentation resulting from using a larger page size since it made the implementation easier, and avoids any limitations on protection.

Another interesting issue is that the 68851 requires the same structure for the maps of the system and process spaces, which was not completely satisfactory. It would have been better, in many ways, to use two different map structures, but the hardware was not capable of handling this. Another interesting compromise was required when setting the address space size. The fact that the TLB must hold mappings for all pages a program uses argued for an address space containing no more pages than the TLB can map. On the other hand, when considering support for shared segments, a larger logical space provides flexibility when assigning a shared segment's logical base address. This is important because a shared segment containing pointers referencing other portions of the segment must appear at the same base address in all the logical spaces containing it. This is the reason for the *Matching* attribute discussed in Section 4.2.3. Since processes can use more than one shared segment, the flexibility in assigning base addresses provided by a larger address space is helpful.

The two level memory map used in the current implementation of Spring describes a logical space of 64 pages, giving 512K when using the current 8K page size. The first level has four entries, each of which point to a table containing 16 page descriptors. A specific logical page may or may not be mapped to a physical page, but the total number of logical pages mapped to physical pages must not exceed the number of TLB entries available for mapping the process space, or the assumption of every memory reference experiencing a TLB hit will be violated. An advantage of the two level structure is that the map can be marked valid or invalid by changing the four first level descriptors, while a single level map would require changing all 64 page descriptors. The process cannot use every page of this space since it would fill the TLB, leaving no room for the system entries.

A minor feature of the MMU provides an excellent example how hardware designed for conventional applications is not always well suited for real-time systems. In conventional systems, the modified (M) bit in the tag field of each TLB descriptor and in each page descriptor in the memory map provides vital support for virtual memory. The MMU is designed to automatically maintain the M bits in both the TLB entries and in the memory maps' page descriptors. This conflicts with the fact that the map is marked invalid during execution, to enforce the system's assumption that TLB misses do not occur, since any attempt by the MMU to modify the M bit generates an exception. Such exceptions could occur at an arbitrary point during program execution and thus make the program behavior less predictable. The solution was to create the maps with the M bits always set, so the MMU will not try to update it in the map. This was reasonable since the system does not need the M bit in any case. This and other issues illustrate the care with which the system design had to manipulate the features of the 68851 MMU, created to support conventional systems, in unconventional ways when implementing support for a predictable system.

The decision to explicitly manage the TLB contents also provides a good example of how the design of a real-time system is much more highly integrated than that of conventional systems. The viability of the approach, explicitly managing the TLB contents at context switch time, depends on carefully coordinating properties of the system that would normally be considered as belonging to separate design domains. It depends most obviously on the fact that the system scheduler manages process execution using a representation of process behavior as a set of non-preempted segments. The non-preempted aspect makes it possible to assert that the TLB contents need only change on a context switch, that legal memory references will always experience TLB hits, and that the overhead of TLB maintenance will occur at predictable

times with predictable cost. The non-preempted property depends, in turn, on the functional partitioning in the design of the system as discussed in Section 8.1.2. The functional partitioning shields processes from both external and internal interrupts, and makes it possible to completely dedicate processors to executing the segments of application processes without interference.

8.2.3 Implementation and Performance

Since every memory reference experiences a TLB hit, the MMU influence on memory reference time is predictable. We measured the TLB translation delay by comparing the execution time of an instruction writing to memory from a register with and without memory mapping. Writes without memory mapping take 0.85 μsec, while writes with memory mapping, require 1 μsec, indicating a delay of around a 150 nanoseconds for a TLB hit. A delay of this magnitude has an effect on the execution time of a program, but we consider the advantages of logical memory for writing programs and managing their execution to be worth the cost.

Explicitly managing the TLB contents gives the context switch more to do, and the time required to do it must also be predictable. Context switching in Spring requires four basic steps:

- saving the registers and user stack pointer of the old process,
- switching to the new system stack,
- switching to the new process address space map in the TLB, and
- restoring the registers and user stack pointer of the new process.

Table 8.1 gives the total time required for a context switch under the current implementation and for several of its component operations. The total time given is

Table 8.1. Context Switch Time Components

Component	Time (μsec)
Save Registers	15
Validate Page	21
Map Page	50
Flush Entry	10
Load Entry	10
Invalidate Page	17
Flush Map	8
Load Map	420
Restore Regs	16
Miscellaneous	75
Total	626

for a process using the minimum of four pages. The graph shows how the switch time varies with the number of TLB entries loaded. It is important to note that while the context switch times for other systems are substantially lower, they do not include time for loading the TLB. Conventional systems incur this cost, just as Spring does, but they do it as each TLB miss occurs during execution.

Saving and restoring registers is done in the obvious way, but switching system stack pages requires extra steps because the system keeps the memory maps marked invalid to enforce the assumption about TLB hits. The system stack's entry in the system memory map is first made valid, then modified to reference the physical page holding the new process's system stack. The old stack's TLB entry is then flushed and the new entry loaded. Note that we *must* explicitly flush the TLB cell holding the old entry to ensure it is available for use. Finally, the system stack's map entry is made invalid to guard against unexpected references to the map by the MMU.

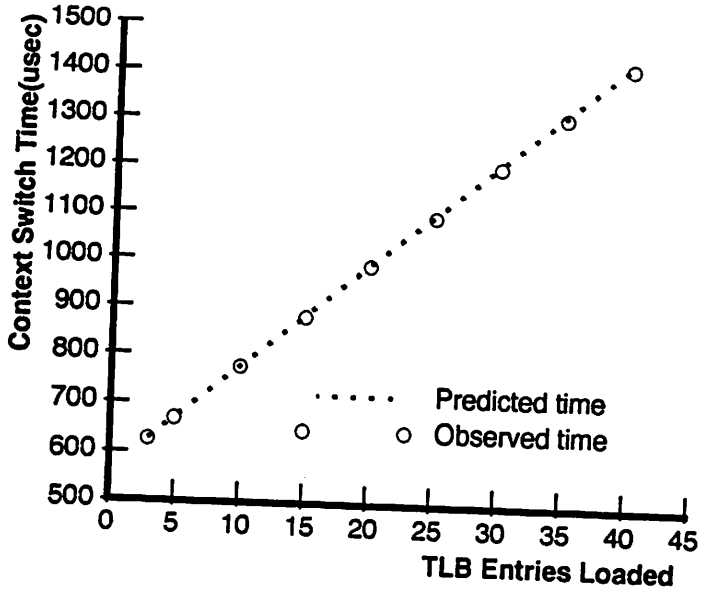


Figure 8.2. Context Switch Timing (μsec)

The TLB entries for the address space of the old process are then flushed from the TLB, using a single instruction. The MMU can do this quickly because each TLB entry contains a 3-bit address space ID field. The MMU maintains a table that maps the address of a space's memory map to its ID value. All entries associated with a memory map address can be flushed simultaneously, but the map of the new process must be loaded entry by entry.

Loading the TLB entries required by the new process is the most time consuming step. The current implementation scans linearly through the 64 entries of the memory map's second level tables, which are allocated contiguously to permit this. This has the advantage of being simple, but the disadvantage of having to examine all 64 entries to locate those used. A substantial portion of this overhead could be eliminated by explicitly tracking the pages used, and thus permitting the proper page descriptors to be loaded without searching the table at context switch time. This modification is planned.

While there is still room for improvement, and clear ways to achieve it, *the current* implementation is predictable. All times cited in Table 8.1 are the maximum values over 10000 context switches, but the minimum values were always within 3 μsec of the maximum, and often fell within the 1 μsec resolution of our measurement method. For example, the maximum time for context switching a process with the minimum number of pages exceeded the minimum time by 3 μsec, or .3%. The graph in Figure 8.2 shows how the context switch time varies with the number of page mappings loaded into the TLB from the process's map. The circles show the measured times, while the dashed line shows the times predicted by the following formula, where *cst* is the context switch time and p is the number of page mappings loaded:

$$cst = 562 + \lceil 21.5 * p \rceil$$

A process using the minimum of three pages in the process space required 626 μsec, as shown in the table, while a process using 10 pages required 778 μsec. The predicted times were *always* larger than the maximum observed times, but never exceeded them by more than 9 μsec, or .7%. The modification to explicitly track the pages used would move the line in the graph down substantially, without changing its slope, since the overhead of finding the used pages would decrease but the time required to load the descriptors for the used pages would remain the same.

This section has shown that a real-time system can be designed which provides support for logical address spaces while ensuring predictable execution behavior. Our experiences in designing and implementing the systems demonstrated a number of ways in which the design of such a real-time system must be highly integrated, carefully coordinating design decisions in areas considered essentially disjoint under conventional designs. The implementation also demonstrated a number of ways in which conventional hardware designs make assumptions and provide features which are not

fully compatible with the need for predictability or convenient when implementing a predictable system.

8.2.4 Desirable MMU Features

This section considers features we believe would be desirable in an MMU designed to support real-time systems, motivated by our experiences in implementing the Spring system. One of the most basic requirements would be support for variable page/segment size. Describing a logical space as a set of fixed sized pages is sensible for conventional systems, since it is in keeping with the needs of virtual memory. However, managing a memory resident logical space as a set of variable sized segments would be more convenient and efficient, since the paging demands of virtual memory are not present. It is easy to imagine an MMU whose TLB specifies mappings for regions of variable size, while providing support for protection in familiar ways. Each TLB entry would specify a logical base address for the segment, the base address of the physical memory assigned to it, and the segment's size.

The properties of the TLB have an important effect on the real-time system in three areas: address translation time, maximum logical space size, and context switch time. The memory access time is affected by how quickly the TLB can identify the proper entry, and translate the address. If segments are permitted to begin at arbitrary addresses and be of unconstrained length, then the TLB would have to subtract the logical base from the address, and add the result to the physical base. Performing two arithmetic operations is likely to produce a relatively slow translation time. If the system constrains segment sizes to be powers of two, and constrains the logical and physical base addresses to be zero modulo the segment size, then address translation can be implemented by masking and combining operations on sections of

the logical and physical addresses. This is analogous to what current MMUs do, and should provide comparable address translation times.

The maximum amount of memory a process can *use* is constrained by the assumption that the TLB contains mappings for the parts of its logical space used by the executing program. Since the TLB must also contain the mappings for the portion of the system space required to support program execution, the number of TLB entries is a significant limiting factor on system and program size. However, if the MMU supports variable segment sizes then a single TLB entry could represent a large memory area, the entire text section of the operating system for example, and constraints on address space size would be much less onerous than in the current implementation. Physical memory would be managed in reasonably large blocks of several sizes, but would be allocated and deallocated only during process activation and deactivation. Physical memory fragmentation would thus still be an issue, but would arise far less quickly than in a system engaged in paging to support virtual memory.

Context switch time, as seen earlier, is significantly affected by the operations required to manage the TLB contents. A context switch flushes TLB entries for the old process to free them, and then loads the entries for the new process. The TLB should provide an instruction for flushing all entries associated with a specific address space, as well as to load and flush individual entries. Each TLB entry describing a segment of the new process's logical space is loaded individually. Selection of a free TLB entry could be done either by hardware or software. Describing an address space generally requires fewer segments than it does pages, so the number of entries loaded would be lower than the current implementation. Using segments instead of pages should thus decrease context switch time. Note also that the context switch using such an MMU would be just as predictable as the current scheme was shown to be.

Several contemporary MMUs provide aspects of the support desired. The R4000's MMU supports variable page sizes ranging from 4K to 16 MB in multiples of 4 (i.e. 4K, 16K, 64K, 256K, ...) [66]. The TLB contains 48 entries which each map a pair of logical pages, and TLB entries are individually addressed, flushed, and loaded under software control. This approach is reasonably close to general support for segments, but has some drawbacks. First, support for page sizes in multiples of 2 with a smaller minimum size (i.e. 2K, 4K, 8K, 16K, ...) would be helpful. In the R4000 design, large sizes were intended for representing objects such as video frame buffers, and smaller sizes were not considered useful. The use of the 48 TLB entries mapping pairs of logical pages is less useful than 96 separate entries would be, since one descriptor is enough to describe a segment. The MMU also only supports manipulation of individual TLB entries, rather than a single instruction invalidating all entries associated with a specific address space. While not crucial, such an instruction would clearly decrease the context switch time.

The Motorola 88110 processor's MMU provides separate data and code TLB's which each contain 32 descriptors for 4K pages, and 8 descriptors for segments ranging from 512K to 64 MB [56]. The limited number of small fixed size pages imposes size limits even more restrictive than Spring's current hardware, while the segment descriptors have a minimum size far too large for application to most real-time programs. This is a good example of hardware designed for conventional systems which is inadequate for predictable real-time system implementation, rather than just being problematic. Another example is the 68030, the successor to Spring's current CPU, which is preferred over the 68020 for many conventional systems. It's MMU was moved onto the chip, but the number of TLB entries was reduced, probably due to

chip real-estate constraints [55]. The 68030 is thus *less* desirable for Spring support, even though it is generally considered a faster and better processor.

It is important to note that describing a process's logical space as a set of regions or segments is *not* the same as that used by processors such as the i486 [32]. Spring describes a set of occupied segments within a continuous logical address space. The process code is generated for this flat address space and emitted machine instructions have no notion of segments. The i486, in contrast, makes segmentation part of the machine instruction semantics. This would have some of the advantages we seek, since the portion of context switching related to managing the address space mappings would simply switch the segment descriptors. The i486 has only 6 segment description registers, which must be explicitly loaded during program execution if more segments are required. It would have no advantage over the kind of MMU we desire, and we prefer a flat address space approach that does not require loading segment descriptors as part of program execution.

8.3 Scheduling

The properties of scheduling algorithms used by real-time systems is not a focus of this dissertation, but cannot be ignored since the properties of the scheduling algorithm have a significant influence on the design of the system, and on the translation scheme which is the main concern of this dissertation. Section 8.3.1 first discusses the Spring scheduling algorithm and its extension to handle groups of precedence related tasks. Section 8.3.2 then considers the design and use of the Spring scheduling coprocessor (SSCoP), a custom VLSI design implementing the extended scheduling algorithm.

8.3.1 Extending the Spring Scheduling Algorithm

Many current real–time scheduling algorithms schedule the CPU independently of other resources. For example, consider a typical real–time scheduling algorithm, earliest deadline first. Scheduling a task which has the earliest deadline does no good if it subsequently blocks because a resource which it requires is unavailable. The approach taken by the Spring system integrates CPU scheduling and resource allocation so that this blocking never occurs. By integrating CPU scheduling and resource allocation at run time, the system is able to understand (at each point in time) the current resource contention and completely control it so that task performance with respect to deadlines is predictable, rather than letting resource contention occur in a random pattern that could result in an unpredictable system.

To support the predictable execution of tasks, Spring uses *guarantees*, a notion fundamental to predictable scheduling. A task is guaranteed by constructing a plan for task execution whereby all guaranteed tasks meet their timing constraints. Specifically, if a set S of tasks has been previously guaranteed and a new task T arrives, T is guaranteed if and only if a feasible schedule can be found for tasks in the set $S \cup \{T\}$. The Spring scheduling algorithm has been extensively studied, and discussed at length elsewhere [97, 70, 77]. These versions of the algorithm, as well as the first implementation [52], all assumed that every task was independent, while as the discussion in this dissertation has shown, many computations require a *group* of tasks to represent their execution behavior to the scheduler. Recent work has extended this research to include groups of precedence related tasks [98, 95].

A task is guaranteed subject to a set of assumptions, for example, about its WCET, resource use, and the nature of faults in the system. If these assumptions hold, once a task is guaranteed it *will* satisfy its temporal constraints. The need for

such detailed information about the execution time behavior of computations motivated the work described in this dissertation. It is important to remember that while the translation methods described here have been designed to satisfy the specific needs of the Spring scheduler, the methods and the predictions they produce are relevant to other real-time systems since many assume that similar detailed information about a computation's behavior is available.

When the system implementation progressed to the point where the real scheduling algorithm was required, the previous implementation had to be extended to handle computations represented by groups of precedence related tasks. In principle, the modification is simple. The precedence relations are enforced using the idea of the *eligible set*, where a task is eligible for scheduling when its precedence relations have been satisfied. Since it assumed independent tasks, the original algorithm always considered the whole set when selecting the next task for addition to the schedule. The extended algorithm must ensure that no task is scheduled to execute before all of its predecessors have finished execution, and thus cannot choose among all of the unscheduled tasks. It must instead choose among the eligible tasks.

The extended algorithm attempts to guarantee non-preemptable tasks given their arrival time T_A, deadline T_D or period T_P, worst case computation time T_C, resource requirements $\{T_R\}$, and predecessor set $\{T_{Pred}\}$. A task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes. A group of tasks with precedence constraints, where each task has its own resource requirements, is generated for each process during compilation and run-time data structures built which describe each computation to the scheduler, as described in Chapter 7. The predecessor set $\{T_{Pred}\}$ is derived by the process activation process (PAP), from the SDL successor list produced during compilation which describes the

```
S = task set to be scheduled;
partial schedule = empty;

repeat:
if S is empty Halt with "Success";

given a partial schedule
    T_est calculation:
      for each task T in S Compute T_est
    H value generation:
      for each task T in S Compute H;
    Task selection:
      determine task min_T with lowest H value, among eligible tasks;
    Update partial schedule or backtrack:
       if (partial schedule || min_T) is feasible and strongly feasible
         {partial schedule = (partial schedule || min_T);
          S = S - min_T;
          go to repeat}
        else if backtracking is allowed and possible
               backtrack to a previous partial schedule;
             else Halt with "failure"
```

Figure 8.3. Extended Scheduling Algorithm

task group structure, during the process activation phase. It is important to note that the PAP also performs scheduler specific preprocessing. For example, some scheduling algorithms assign intermediate task deadlines as a function of the computation (task group) deadline and task parameters [95], while others do not [98].

Spring uses a heuristic scheduling algorithm, illustrated in Figure 8.3, which tries to find a full feasible schedule for a set of tasks. It is important to recall that a task represents a non-preemptable portion of a process's execution generated by the compiler. The scheduler starts at the root of the search tree, an empty schedule, and tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, the system uses a heuristic function, H, (see details below) which synthesizes various

characteristics of tasks affecting real-time scheduling decisions to actively direct the search along a plausible path. H is applied to the eligible tasks that remain to be scheduled at each level of the search tree. The eligible task with the smallest H-value is used to extend the current schedule. A task is eligible when it has not yet been scheduled but all tasks in its predecessor set T_{Pred} have been scheduled. A task without predecessors is always eligible.

While extending the partial schedule at each level of search, the algorithm determines if the resulting partial schedule is *strongly-feasible* or not. A partial schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending it with any one of the remaining tasks are also feasible. Thus, if a partial schedule is found not to be *strongly-feasible* because, say, task T misses its deadline when the current schedule is extended by T, then it is appropriate to stop the search since no schedule extending the current schedule will ever satisfy T's deadline. In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a full feasible schedule. The earliest start time, T_{est}, at which task T can begin execution is used when calculating feasibility. T_{est} accounts for resource contention among tasks, and is a fundamental part of the guarantee calculation.

It is possible to continue the search even after a non-strongly-feasible schedule is encountered by *backtracking*. Backtracking discards the current partial schedule, returning to a previous partial schedule, and extending it using a different task. The task chosen is the one with the smallest H value, *among those which were not already chosen during a previous attempt to extend* this *partial schedule*. Backtracking overhead can be restricted either by limiting the maximum number of backtracks or

Table 8.2. Example Heuristic Functions

Minimum deadline first (Min_D):	$H(T) = T_D$
Minimum processing time first (Min_C):	$H(T) = T_C$
Minimum earliest start time first (Min_S):	$H(T) = T_{est}$
Minimum laxity first (Min_L):	$H(T) = T_D$ - $(T_{est}+T_C)$
Min_D + Min_C:	$H(T) = T_D + W_1 * T_C$
Min_D + Min_S:	$H(T) = T_D + W_1 * T_{est}$

by restricting the total number of evaluations of the H function. We use the former scheme in the implementation discussed here.

The heuristic function H can be constructed by simple or integrated heuristics. Table 8.2 shows examples of both simple and integrated heuristic functions. T_{est} denotes the earliest time at which a task that is yet to be scheduled can begin execution, and is given by:

$$T_{est} = Max(\text{T's arrival time}, \{EAT_i^u \mid (R_i, u) \in \{T_R\}\}) \tag{8.1}$$

where EAT_i^u, for a given partial schedule, is the earliest time resource R_i is available in mode u, where u is *shared* or *exclusive*.

The first four heuristics in Table 8.2 are *simple* because they treat only one dimension at a time, e.g., only deadlines, or only resource requirements, while the last two are *integrated* heuristics. W_1 is a weight used to combine two simple heuristics. Min_L and Min_S need not be combined because the heuristic Min_L contains the information in Min_D and Min_S. SSCoP supports several other integrated heuristics, some of which use a second weight value W_2. Some of these other heuristics consider the task's value density(task-value/task-computation-time) [98].

Extensive simulation studies of the algorithm for uniprocessor and multiprocessors show that the simple heuristics do not work well, but that the integrated heuristic

```
extended_scheduler(new_work, curr_sched)
{
   num_of_tasks = num_tasks(new_work) +
                  num_tasks(current_sched);
   sched_time   = estimate_sched_time(num_of_tasks);
   cutoff_line  = do_cutoff_and_eat(current_sched, sched_time);
   sched_set    = form_sched_set(new_work, current_sched,
                                 cutoff_line);

   result    = build_sched(cutoff_line);

   if ( schedule_succeeded(result) ) {
      put_sched_into_dsp_queues(current_sched, sched_set);
   } else {
      revert_to_previous_sched(current_sched);
   }
}
```

Figure 8.4. Extended Scheduler

(Min_D + Min_S) works very well and has the best performance among all the above possibilities as well as over many other heuristics tested [70]. For example, combinations of three heuristics were shown not to improve performance over the (Min_D + Min_S) heuristic. Hence, this is the heuristic that is implemented in the current Spring system.

The extended version of the scheduling algorithm in Figure 8.3 is fairly easy to understand, but the extension of the scheduling algorithm represents the easiest part of the *implementation* problem. The reason for the distinction is that the algorithm of Figure 8.3 assumes the set of tasks that must be scheduled, the scheduling set S, as input. Forming the scheduling set, and properly initializing the information describing its members, presents nontrivial problems. Figure 8.4 presents the essential steps of the extended scheduler implementation. The *build_sched* routine represents the implementation of the algorithm of Figure 8.3.

The current schedule, or system task table (STT), consists of a "dispatch queue" for each AP in the system. Dispatch queues are currently implemented as linked lists of STT entries, each describing a task. Each STT entry specifies relevant task attributes, including scheduled start and finish times, deadline, and resource use. STT entries are placed on their dispatch queue in increasing scheduled start time order. In Spring, dispatching of previously guaranteed tasks continues while new tasks are guaranteed. To ensure maximum flexibility when guaranteeing the new task, those tasks that will not begin execution until the guarantee is completed are considered for rescheduling. To identify these tasks, the following procedure is used.

The scheduler begins by finding the total number of tasks in the current schedule and in the new work requested, and then uses it to calculate the time required for the *complete* scheduling operation, including all pre and post processing. Note that the new work specifies computations that should be added to the schedule, each of which is represented by a group of tasks. Next, the scheduler examines the current schedule's dispatch queues, "cutting" each at the point corresponding to the current time plus the scheduling time, known as the "cutoff line".

The system must leave those tasks on each dispatch queue slated to begin execution before the cutoff line undisturbed to ensure their "guarantees" remain valid. This is easy to see for tasks whose deadlines fall before the cutoff line, since their deadlines will pass before the system can reschedule them, but is required for all such tasks regardless of their deadline. Those STT entries scheduled to start after the cutoff line are rescheduled. These, plus the STT entry structures allocated to represent the new work, comprise the *scheduling set* processed by the scheduling algorithm. The *do_cutoff_and_eat* routine also calculates the initial *EAT* values for every resource. This calculation is slightly more complicated than might be expected, since the cutoff

line can fall within the scheduled execution of a task, requiring the initial *EAT* values for the resources it uses to be the task's scheduled finish time.

Each of the STT entries in the scheduling set must have its attributes properly initialized before the *build_sched* routine can be called. The attributes include: deadline, arrival time, WCET, resource use, and precedence relations. This is where the complications arise from the extension to task groups, although initialization of many attributes is unaffected. Deadline and resource use, for example, are calculated from the values in the system's run-time data structures, and initialized by the PAP, just as they were for independent tasks. The complications arise in forming the initial eligible task set, and in initializing the predecessor information and arrival time for each task. Each STT entry maintains a count of predecessors that have not yet been scheduled. This count is initialized from the run-time data structures constructed by the PAP, but some situations require subtle adjustment of the predecessor count.

Forming the initial eligible task set is complicated because the cutoff line can fall within a group boundary. When that happens, the information about the group as a whole, stored by the system's run-time data structures, is not sufficient to determine what tasks are eligible. The scheduler has to take the tasks in the group falling above the cutoff line into account. For example, consider a computation in the current schedule represented by two tasks, $T1$ and $T2$, where $T1$ is the predecessor of $T2$. If both fall below the cutoff line, then $T1$ is in the initial eligible set, since it is a successor of the *Begin* node, but $T2$ is not since its predecessor $T1$ is as yet unscheduled. However, if the cutoff line falls between the two, then $T2$ *is* in the initial eligible set.

Initializing the predecessor counts in each STT entry is difficult for similar reasons. An STT entry contains pointers to each of its successors. When a task is added to the schedule, these pointers are used to decrement the predecessor count of its successors.

When the predecessor count of an STT entry is decremented to zero, the task it represents has become eligible, and is added to the eligible set. Each STT must thus be given to the scheduling algorithm implemented by *build_sched* with its predecessor count reflecting the number of its predecessors *in the scheduling set.* When the entire task group is in the scheduling set, then the predecessor count can be initialized from the system's run-time data structures. When the cutoff line falls within the task group, the proper number must be calculated. The run-time data structures built by the PAP during process activation contain the information required to do this, but the details are too complicated to address here. The important point is that the information provided by the compiler and that stored in the run-time data structures by the PAP was explicitly designed to make the calculation possible. This is another example of how the design and implementation of a real-time system must be carefully coordinated.

The arrival time is used to note the earliest time at which a task may be scheduled to begin execution. In the algorithm handling independent tasks its value was determined by the maximum EAT value among all the resources the task used. When the algorithm is extended to handle precedence relations, the delay values associated with the precedence relations must also be considered. Recall the previous example of a process represented by two tasks, and assume that the precedence relation has a delay of 5 seconds associated with it. If the cutoff line falls between $T1$ and $T2$, and $T2$ uses no resources other than the AP, then the $T2_{est}$ is determined by the EAT of the AP resource, EAT^{e}_{AP}, and scheduled finish time of $T1$, $T1_{sft}$. The delay associated with the precedence relation must be set to:

$$Max(EAT^{e}_{AP}, (T1_{sft} + 5))$$

When all the STT entries' attributes have been initialized, the scheduling set is given to the *build_sched* routine, and the search for a feasible schedule begins. If the scheduling algorithm succeeds, then the new schedule is used to construct the dispatch queues for each AP. If it fails, the system reverts to the previous schedule, as specified by the dispatch queues before the scheduling operation began.

This section has described many of the extensions to the original scheduling algorithm required to produce a working implementation that handles groups of precedence related tasks. The extended scheduling algorithm has been implemented and tested by a set of 13 scheduling scenarios. The difficult problems were largely practical, as opposed to conceptual, but of vital importance to the viability of the system because they directly impact the performance of the scheduling algorithm, and thus the scheduling latency of the system. Since the scheduling latency defines a lower bound on the deadline granularity of the tasks to which the algorithm may be applied, such practical considerations have considerable impact on the applications in which Spring can be used. For this reason any improvement in performance is desirable, and the implementation of a considerable portion of the algorithm in custom VLSI was considered worthwhile, as discussed in the next section.

8.3.2 Spring Scheduling Coprocessor

The performance of the scheduling operation is key to the performance of the system as a whole, and to the range of applications it can support. Since the scheduling performance is so important, it is reasonable to devote a significant amount of effort toward improving it. Traditionally, one way to improve performance of a critical

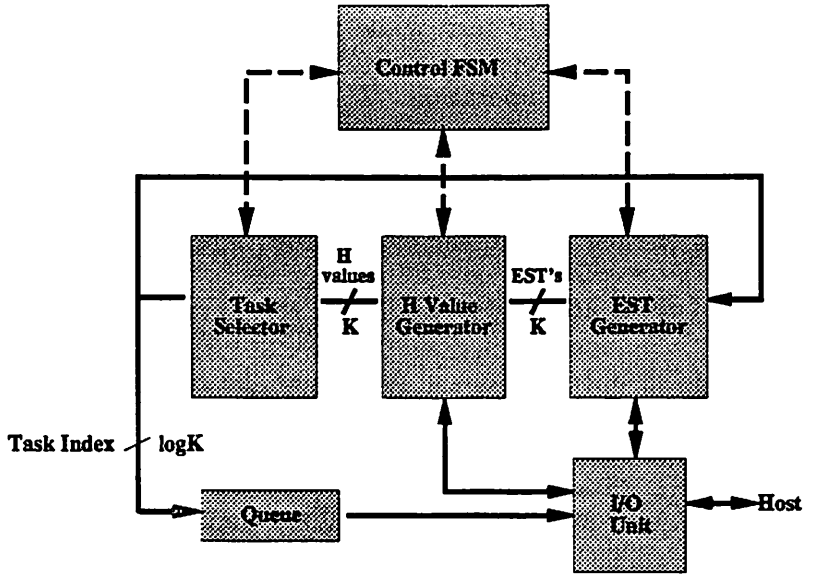


Figure 8.5. High Level Block Architecture

operation is to implement it in hardware as a coprocessor. The work described in this section, which is the result of a significant collaborative effort, is the first step in that direction [9, 60].

Specifically, this section describes the design, use, and expected performance of the Spring scheduling co-processor (SSCoP), a VLSI scheduling co-processor which accelerates the "core" of the Spring scheduling algorithms, and thus increases the range, in terms of deadline granularity, of applicability of the dynamic scheduling approach used by Spring. Subsequent implementations will expand the portion of the algorithms implemented in hardware, and thus continue to decrease the time required to complete a scheduling operation. The first SSCoP test chips, intended for testing the component implementations, have been fabricated and tested.

The high level architecture of SSCoP, which implements the algorithm given in Figure 8.3, is shown in Figure 8.5 [9]. The EST generator, H-value generator, and Task Selector in the figure correspond to the T_{est} calculation, H value generation,

and task selection steps in the pseudo-code, respectively. The H-value generator is also responsible for the feasibility test, while the Task Selector handles the "update partial schedule or backtrack" operation. The Queue holds the index of each task in the order they were added to the schedule, while the EST generator holds the scheduled start time of each task in the element of its register array specified by the task index. The host reads and writes all SSCoP registers through the I/O unit. The execution of the scheduling algorithm, in particular the complexities of backtracking, are controlled by a finite state machine (Control FSM).

In many ways SSCoP looks like a smart, albeit small, memory. All registers in SSCoP appear in the address space of the host, thus simplifying the programming interface. It behaves as a memory chip into which the host writes task attributes prior to initiating the scheduling operation, and from which a complete and feasible schedule can be read back, if the guarantee algorithm succeeds. The set of task attributes is held in the SSCoP *task table*, which is presented to the host as a set of parallel register arrays. Each task has a unique ID used to index the register arrays. Each task's entry in the SSCoP task table thus consists of the set of elements in each array at the position specified by the task's ID, or *index*.

Figure 8.6 shows the detailed block diagram of SSCoP, illustrating the parallel access shift register banks holding all task attributes required for the scheduling computation: deadlines (DReg), worst case computation time (CReg), arrival time (AReg), value density (VDReg), resource's earliest available times (EATReg), resource requirement (RReg), and predecessor matrix (PReg) representing each task's predecessor set. The H-value generator uses the T_{est} (ESTReg), deadline (DReg), worst case computation time (CReg), value density (VDREg), and weights W_1 and

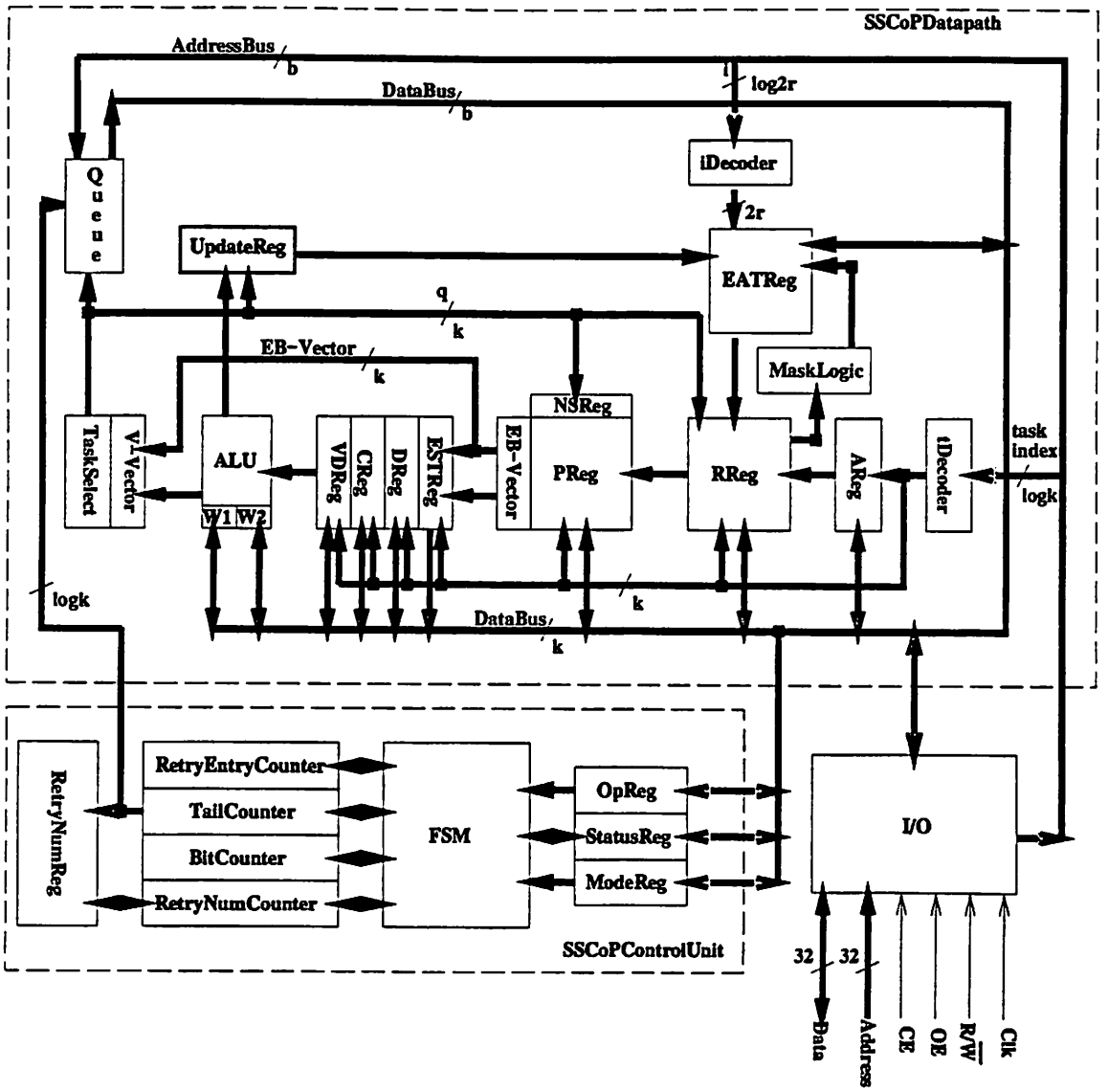


Figure 8.6. Functional Block Architecture

W_2 as required to compute the desired heuristic function, H, for each task. It is important to note that the SSCoP registers can hold any numeric data. The contents of the DReg need not, for example, be the deadline of a task.

8.3.2.1 SSCoP-System Interface

A critical aspect of SSCoP design, and that of any co-processor, is its interface to the host. Since the goal is to minimize the *total* scheduling time, how the SSCoP

```
sscop_scheduler(new_work, current_sched)
{
   num_of_tasks = num_tasks(new_work) +
                  num_tasks(current_sched);
   sched_time   = estimate_sched_time(num_of_tasks);
   cutoff_line  = do_cutoff_and_eat(current_sched,
                                    sched_time);
   sched_set    = form_sched_set(new_work,
                                 current_sched,
                                 cutoff_line);

   init_sscop_input(sched_set, cutoff_line);

   result = build_sched(cutoff_line);

   if ( sscop_succeeded(result) )
      put_sched_into_dsp_queues(current_sched,
                                sched_set);
   } else {
      revert_to_previous_sched(current_sched);
      eval_sscop_error(sched_set,result);
   }
}
```

Figure 8.7. Scheduling Algorithm

design influences the preprocessing required to set up a scheduling operation, and the postprocessing required to transform the resulting schedule into a form the application processors can use, is extremely important.

Figure 8.7 shows the SSCoP version of the scheduler from Figure 8.4. As discussed in Section 8.3.1, the essential data structures of Spring's scheduler include the current schedule, or system task table (STT), consisting of a "dispatch queue" for each AP in the system. The SSCoP version of the scheduler executes essentially the same operations as the previous version, through the formation of the scheduling set.

It finds the total number of tasks in the current schedule and in the new work requested, uses it to calculate the scheduling time, and then implements the cutoff

line and calculates the *EAT* values using the cutoff line. Note that the scheduling time calculation produces a cutoff line substantially closer to the current time since the SSCoP is being used to accelerate the scheduling calculation. The routine then forms the scheduling set from the tasks in the current schedule which fall after the cutoff line and those tasks representing the new work the system is trying to schedule. Note, however, that when the total number of tasks in the system exceeds the SSCoP task table size, k, the system cannot permanently assign every task a SSCoP task slot. Instead, the system allocates SSCoP task slots to scheduling set members during preprocessing by the *init_sscop_input* routine. Each task's attributes are then loaded into the appropriate entries within the assigned slot.

Initializing each task's temporal attributes, deadline, execution, and arrival time, is done by writing the proper values to memory mapped SSCoP registers. However, the precedence relations require further nontrivial preprocessing because at this point precedence relations are represented by relations between STT entries, while SSCoP's PReg represents precedence relations between SSCoP task slots. For example, a precedence relation $A \rightarrow B$ (A precedes B) is represented by setting the bit in task B's PReg entry at the position corresponding to A's task slot. The PReg entries for each task are initialized after all the tasks have been assigned to slots, and the relations between tasks can be translated into relations between slots.

Resource use requires nontrivial preprocessing because, just as the total number of tasks in the system can exceed k, so too can the total number of resources in the system exceed the resource table size, r. The solution is the same; resources used by the tasks in the scheduling set are assigned SSCoP resource slots as they are encountered during task slot assignment. Task resource use is represented by setting bits in the task's RReg entry.

That completes the preprocessing required to prepare SSCoP to calculate a schedule. The heuristic function and its weights are specified by setting SSCoP command registers. These generally remain unchanged for long periods, although it is simple to select a different H-function for every scheduling operation. There is, however, a fairly subtle aspect of preprocessing that has not been addressed. The task deadline, execution, and arrival times are initialized by writing them into the proper SSCoP registers, but *what* values are written depends on whether SSCoP uses *absolute* or *relative* times. Absolute times are easy to understand, and require no preprocessing, but SSCoP currently uses relative times.

The major reason for this is that H-functions were validated by simulations, which *always started at time zero.* Since Spring uses a 32-bit clock, as its value increases beyond the range validated by those simulations (generally somewhere above 10^5), the performance of some H-functions degrades because the time component gradually predominates, distorting the H-value. For example, the H-function $Min\_D + Min\_C$, which performs well, adds the deadline to the computation time. If absolute times are used this will converge with Min_D, which has poor performance, as the value of the absolute time becomes ever greater than the computation time. However, further research will be required to evaluate the sensitivity of specific H-functions to increasing time values, and to investigate efficient methods for handling the problem.

Another important reason to use relative times is that SSCoP arithmetic operations are bit-serial. Using relative times of length b saves $3(32 - b)N$ cycles when scheduling N tasks, as well as reducing SSCoP size, since all word length registers are correspondingly smaller. The word size for the final implementation has not been chosen, but will probably fall between 16 and 24. Our ability to so easily defer, or change, this decision is an important benefit of SSCoP's scalable design.

The scheduling operation is started by *build_sched* setting a bit in the SSCoP control register. It can monitor SSCoP status during execution by reading the status register. *Build_sched* checks for completion by polling the status register, and can interrupt SSCoP at any time by setting the abort bit in the control register. *Build_sched* also enforces the cutoff line by watching the system clock while waiting for SSCoP execution to finish. When complete, the output queue holds the indices of the scheduled tasks, in the order they were added to the schedule. Their starting times are held in the corresponding ESTReg entries.

The post processing is comparatively simple. Each task index is mapped to the corresponding STT entry using a table filled in as task slots were allocated during preprocessing, and the *absolute* scheduled start and finish times are calculated from the *relative* start time in the ESTReg by adding the cutoff line. The STT entry is then added to the end of the appropriate AP's dispatch queue, beginning with where each was truncated by the cutoff line. The STT entries in the resulting dispatch queues are always in increasing start time order, since they all use the same processor resource, and a task added to the SSCoP output queue after another using the same processor *cannot* have an T_{est} earlier than the scheduled finish time of the previous task.

8.3.2.2 Testing and Performance Results

This section discusses the several ways the correctness and performance of SSCoP have been tested. The initial test version ($k = 4$ tasks, $r = 4$ resources, $b = 8$ wordlength, $w = 4$ the length of the weights) has recently returned from fabrication, and has passed preliminary tests. Two previous test chips, separately implementing the data path and FSM controller, passed all tests. The SSCoP design has also been

tested at several levels through simulation. Finally, the pre and post processing code has been fully implemented, enabling us to directly measure its performance.

The SSCoP functional block diagram of Figure 8.6 was verified using a Verilog[11] behavioral simulation. A version of the Spring SSCoP scheduler capable of using this behavioral simulation was implemented on a UNIX workstation, enabling us to verify that SSCoP correctly handles benchmark scheduling problems, including the intricacies of backtracking. Verilog was also used to simulate the switch level behavior of the TSPC clocking scheme used by SSCoP latches and logic. The regular bit-slice nature of SSCoP permitted us to use MAGIC to layout cells which interconnect by abutment. IRSIM was used to simulate the switch level behavior of the layout and CAzM was used to simulate the electrical performance of circuits.

The scalability of the architecture allowed us to simulate small versions of SSCoP ($k = 4, r = 4, b = 8, w = 4$) initially to verify functionality without manipulating the large netlists which result from a realistic size SSCoP (eg. $k = 128, r = 32, b = 24, w = 4$). Testing the first fabricated version of SSCoP will be greatly simplified by the fact that *all* state information on the chip is readable and writable via the data bus. Unlike scan-path testing schemes, this allows on-the-fly checkpointing in addition to more conventional manufacturing chip tests.

We have, thus, tested the logical correctness of SSCoP design as thoroughly as we can prior to the first full-scale physical device becoming available. However, we have also evaluated our assumption that SSCoP will substantially improve the performance of the scheduling operation as a *whole*, which has three major components: preprocessing, SSCoP execution, and postprocessing.

Table 8.3. SSCoP Execution Time Prediction

Clock (MHz)	nsec/cycle	μsec/task	μsec/sched-op	
			$b = 24$	$b = 32$
50	20.00	2.10	134.40	165.12
60	16.67	1.75	112.00	137.60
70	14.29	1.50	96.00	117.94
80	12.50	1.31	83.84	103.00
90	11.11	1.17	74.88	113.02
100	10.00	1.05	67.20	82.56

We have considered the phases of SSCoP execution, each of which is executed once every time a task is added to the generated partial schedule. The cost per task is:

$b + w + 2$	$MAX(T_{est}, D - C)$
$b + 2w + 2$	H-value computation for all tasks in parallel
$b + w + \log_2 k + 3$	task selection
3	update
$3b + 4w + \log_2 k + 10$	Total for one task

and the total cost for a set of N tasks, without backtracking, is:

$$N * (3b + 4w + \log_2 k + 10) \tag{8.2}$$

If we assume a reasonably demanding scenario (eg. $N = 64, k = 128, b = (24, 32), w = 4$), then Table 8.3 gives the SSCoP execution times.

The performance of an earlier version of the scheduling algorithm which could only schedule sets of *independent* tasks, having no notion of precedence relations between tasks, was measured [52]. The measurements were made using a relatively coarse 10 millisecond clock resolution, but gave a scheduling time of 30 milliseconds for a scheduling set containing 10 tasks, or roughly 3 milliseconds per task since the

algorithm is of linear complexity. If we extrapolate this performance to 64 tasks, we get a *total* scheduling time of 192 milliseconds. The portion of the algorithm implemented by SSCoP represents roughly 95 percent of this time (182 milliseconds) in the original algorithm, as discussed below. When we compare this to the SSCoP execution time, we see that the current implementation should achieve a speedup of between 1300 and 2700 fold ($b = 24$), depending on the SSCoP clock rate, for the portion of the algorithm it implements. It is important to note that the first test chip was tested at a clock rate of 50 MHz, which was the fastest clock the testing equipment could supply. While this does not ensure that a full size implementation of SSCoP will run above 50 MHz, it creates a reasonable expectation that it will do so.

However, the total scheduling time must take the pre and post processing overheads into account, which significantly decreases the speed up. For example, for scheduling ten tasks, the sum of pre and post processing time is 1.335 milliseconds, which is less than 5 percent of the 30 milliseconds required by the original algorithm. However, since the other 95 percent has been sped up by three orders of magnitude, pre and post processing now dominate the total scheduling time, as illustrated in Figure 8.8.

The current results indicate a 30 fold speed up over the software scheduler, using the current pre and post processing code. It is easy to see that pre and post processing account for over 90 percent of the total scheduling time in the current configuration, since the SSCoP execution time curve is difficult to distinguish from the X-axis. It is also important to note that the software scheduler's execution time for scheduling a set of 10 tasks would appear at 30000, considerably off the vertical scale of Figure 8.8.

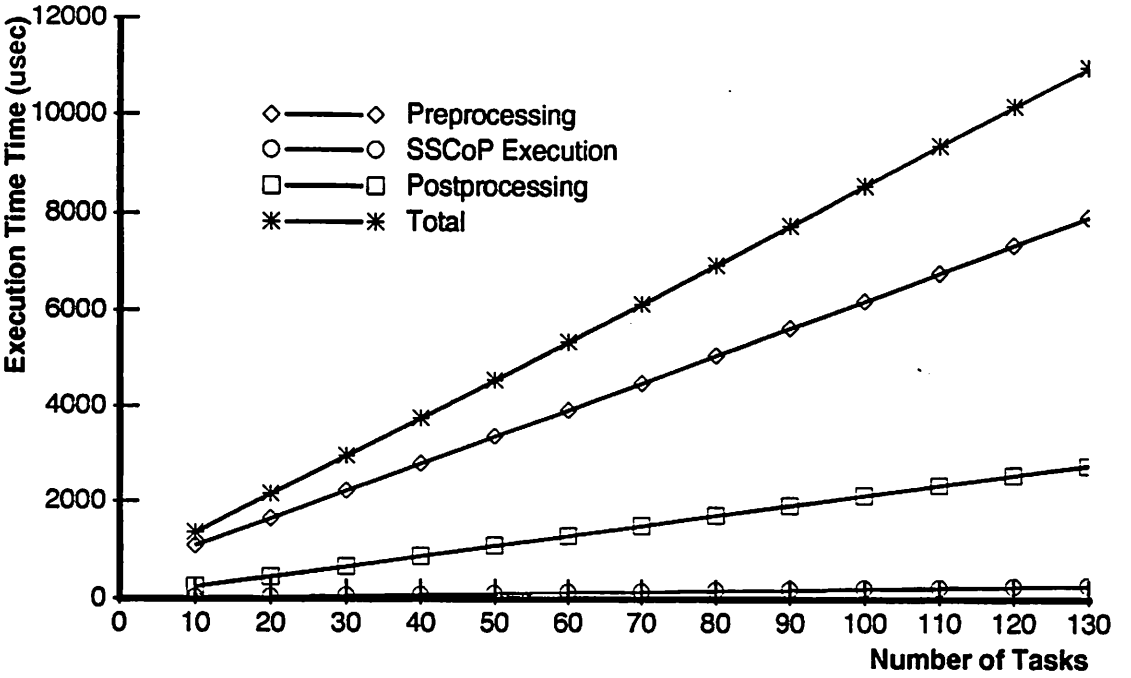


Figure 8.8. Comparative and Total Execution Times

The current target hardware (68020) is much slower than many current processors, and moving to a new architecture will clearly improve pre and post processing performance, but faster host hardware is not likely to change the fact that pre and post processing dominates total execution time. Optimizing total SSCoP scheduling time will clearly require careful attention to optimizing the pre and post processing code. However, the current preprocessing implementation is fairly simplistic, and there are many ways in which we should be able to improve its performance, including performing more functions in hardware.

8.4 Summary

This chapter has shown how the operating system structure and other aspects of the system's design and implementation are affected by real-time system requirements. The need to execute processes predictably leads to significantly different design decisions than those commonly found in conventional systems. This point was first

illustrated by a discussion of the new design considerations at the system, functional and component levels. These general principles were then illustrated in greater detail by a discussion of the way in which predictable support for logical address spaces was implemented in Spring, by a discussion of how the Spring scheduling algorithm was extended to handle sets of precedence related tasks, and by the description of the Spring scheduling coprocessor.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This dissertation has considered a range of problems associated with the design and implementation of predictable real-time systems. Many of the problems addressed arise from the fact that real-time systems require information about a computation's run-time behavior of a substantially greater range and at a greater level of detail than that required by conventional (non-real-time) systems. The need for so much information raises the questions of how the system can obtain it, and how the information is used when it is available. An important aspect of this information is that much of it must be *predictions* about computation behavior, since it is required *before the computation executes.*

Developing methods for predicting a computation's run-time behavior required us to confront the fundamental question of what information a system requires to support predictable execution of real-time computations. While the specific information required by each real-time system varies, the computation behaviors of most common concern are the deadline and the WCET. Other information of interest includes: resource use, value, communication behavior, and precedence relations. Making predictions about the behavior of a computation inevitably segments it by distinguishing episodes of its execution that have different characteristics, as discussed in Chapter 7. These issues, as addressed by this dissertation, were introduced by two questions:

1. How can we design and build a real-time system which represents a computation's behavior as a set of non-preemptable segments with known characteristics and which is capable of supporting complex real-time applications?

2. Will its design differ from that of conventional systems?

The first recognizes that making predictions about a computation's behavior inevitably segments it, and focuses on how the representation of the computation's behavior as a set of execution episodes can be created and used. The methods of compilation and analysis presented in this dissertation use tasks to represent a process's execution episodes, and thus use a group of tasks to represent the execution of a process. This satisfies the assumptions of the Spring system, which serves as an implementation example of the principles discussed throughout this dissertation [70, 87].

The second question focuses on the implications of the first for system design, asking how adopting predictability of behavior, particularly worst case behavior, as a primary system design criterion affects design decisions at all levels. One of the most important results presented by this dissertation is its demonstration that the problem of designing real-time systems *is* different from designing conventional systems, and that adopting predictability as a design criterion leads to significantly different design decisions.

One aspect of this is that the design of real-time systems must be highly integrated, using the vertical slice approach discussed in Chapter 3. The reason for this is that predictability of system behavior is a much more stringent performance criterion than optimizing average case performance. The predictability of a real-time system can be affected by the interactions between design decisions at different levels of the system which would not be considered in the context of a conventional system. For example,

the need to produce a task group representation of a process's behavior creates an interaction between the design of the programming language and the system level that is not present in a conventional system. The design and compilation of the source language must take the process's suspension points into account, and represent these to the system as task boundaries. In a conventional system the equivalent operations are handled by system calls, which decouples the design and compilation of the language from its support by the system at execution time.

The methods and results described by this dissertation address the needs of real-time systems as exemplified by the design and implementation of the Spring system [87]. The specific information about each segment of a computation thus included its WCET, value, resource use, and precedence relations with the other segments of the computation. It is important to note, however, that many of the results presented here are applicable to other real-time systems, since they require a similar level of detail and type of information. This introduces a third question:

3. What advantages do the methods presented in this dissertation have over other approaches to real-time systems?

One of the most obvious is that very few other efforts take the highly integrated approach to system design advocated here. Most other systems, as discussed in Chapter 2, concentrate on a more limited portion of the system, assuming conventional approaches for the remaining portions of the system. The integrated approach to design enabled us to handle the interactions between aspects of the system that are usually considered separately. The main advantage of the integrated approach to real-time system design is that it provides the designers with a clean slate, freeing them to choose the *best* solution to each problem.

Another advantage of the methods and results presented here is that they are scalable. The analysis methods, the algorithms for building task groups, the design and implementation of the operating system, and the principles for real-time hardware design are all as relevant to a large network of interacting nodes as they are to a single node system. Many other approaches to real-time are more appropriate for smaller systems supporting more static sets of computations. There are, of course, open questions related to scaling Spring and making it handle more dynamic computations, but it should be clear that the system was designed with these issues in mind, and that there are reasons to believe it will handle them successfully.

Finally, a significant advantage of the design and implementation methods described here are that they provide an excellent foundation for significant further development. The SDL is of particular importance in this regard, since it was specifically designed to support the specification, preservation, and derivation, of all information required by real-time systems at all phases of program development, compilation, and execution. It is easily extensible, and the environment for information exchange it provides is intended to support the development and use of many kinds of design, analysis and debugging tools. Another example of support for future extension in the current implementation is the maintenance of source line information during subgraph reduction, as described in Chapter 6. This was done with support for debugging in mind, since it is clearly required to enable the system to relate the tasks in a group back to the computation's source code.

More specifically, this thesis presented a number of contributions, of both theoretical and practical interest, at every level of system design and implementation, including:

- the Spring-C programming language which is process based and thus familiar, yet contains new features required for a predictable real-time system;
- program translation methods which predict the worst case run-time behavior of any Spring-C program and builds a task group representing that behavior to the system;
- an integrated software generation system including tools for compiling, linking, loading, analyzing, and debugging Spring system and application code;
- a predictable operating system implementation featuring an enhanced scheduler handling precedence constraints, memory management support of independent address spaces and shared segments, as well as other essential aspects of program execution;
- hardware design principles and suggestions adopting optimization of the predictable worst case execution time as a primary design criterion;
- the SSCoP, a collaborative design effort of several researchers [9], as a concrete example of hardware specifically designed to support real-time systems.

The source language developed, Spring-C, extends the C language in two significant ways, and was discussed in Chapter 4. First, several syntactic changes and use restrictions guarantee that reliable worst case behavioral predictions can be made for every program. Every Spring program is finite in the sense that it will either terminate, or the violation of its behavioral constraints will be detected and its execution aborted. Second, the SDL was developed to support specifying all information required to describe, compile, and predictably execute Spring applications. The SDL

is fully supported by the Spring-C compiler, permitting descriptive statements to be inserted into application source files as appropriate.

The other tools in the SGS use, modify, and produce information supported by various parts of the SDL as Spring-C programs are compiled, linked, downloaded, and executed. The relationship of these tools to each other and to the system as a whole was discussed in Section 3.1, which also considered how the SDL supports the preservation and exchange of information by which the various stages of SGS operation are coordinated and controlled. Chapter 7 showed how the SDL information describing the task group is included in the executable for each process and used by the process activation process (PAP) during system boot to build the run-time data structures used by the scheduler.

Chapters 5, 6, and 7, described the translation method by which the system makes predictions about the behavior of the processes implementing a computation, and build a representation of its behavior as a group of tasks with known WCET, resource use, and whose execution order is constrained by precedence relations. This is the segmented representation specified by the first of the two questions motivating the thesis. Section 6.6.2 presented the prediction accuracy results demonstrating that the methods presented in these chapters provide the system with usable and reasonably accurate predictions.

Chapter 8 considered how the design of the operating system, its use of the underlying hardware, and the design of the hardware itself affect its ability to support predictable execution of computations, and thus the system's ability to fulfill the predictions made by the translation methods. The design and implementation of the system was, of course, a collaborative effort among many people. Most aspects of the system's design and implementation were, however, strongly influenced by the results

presented in this dissertation. Of particular interest was the discussion of design features that we would consider desirable in hardware specifically designed to support real-time systems. The relevant points were illustrated by specific recommendations for CPU design, the design of memory management units for logical address space support, and the design of the Spring scheduling coprocessor. The latter was also a collaborative effort, but its programming interface and the methods for using it effectively as part of the system were specific contributions presented here.

While this dissertation presents a number of specific results, two important points are more general. The first is that predictable real-time systems *can* be designed and built. The second is that producing a well designed and built real-time system supporting predictable execution of application programs requires a fresh look at system design. The needs and performance criteria of real-time systems are sufficiently different from those of conventional systems to render many familiar designs and techniques inappropriate.

9.1 Future Work

The work presented in this dissertation was intentionally designed as a foundation for significant future development in all areas of system design, implementation, and analysis. The major areas of potential future work include: new programming language features, extensions to the translation and prediction methods, creation of integrated design and analysis tools, extending support for distributed and fault tolerant real-time computing, use of the system in real applications, implementing the operating system on more contemporary hardware, and further development of hardware designs specifically addressing the needs of real-time systems. Each of these areas contains a number of interesting research issues.

At the source language level, there are several ways in which the semantics of the Spring-C language could be enriched. The *delay* statement, for example, could be extended to take a second argument specifying the *maximum* interval that should elapse between the completion of one operation and the beginning of another. The vocabulary for describing resource use could also be expanded to permit the acquisition and release of resources as separate operations. This has implications that would require changes to the translation method, those for building task groups, and to the scheduler as well. These and other modifications would, however, be useful, for describing new classes of computations, and for providing effective support of real applications. As more ambitious real-time applications are implemented and tested the desire to express more complex ideas, operations, and constraints will suggest further extensions to the Spring-C language.

The translation method could be expanded in a number of ways. One of the most interesting is enabling it to consider the effects of caching on the WCET. While we have done some preliminary work in this area [59], significant questions remain, particularly how to consider the effects of data caches, and the reduction in WCET resulting from most but not all of a loop fitting into the instruction cache. Another interesting possibility is the modification of the subgraph reduction rules to derive *expressions* for WCET, rather than performing purely arithmetic calculations. This would enable the compiler to produce an expression for the WCET of a task parameterized by characteristics of its input, which might be the length of a list, or the value of an input variable. This expression could be compiled, and then used at scheduling time to determine the WCET which should be considered by the scheduler. Such expressions would make the system much more adaptable, since it would be able to take the affects of specific inputs on the behavior of the application into account.

The current implementation of the compiler should also be extended to use optimizations. The optimization phases of the compiler are currently disabled because the compatibility of each with the new RTL structural and suspension point nodes must be considered. Many will require only minor modification to handle the new structures correctly. Some optimization methods, however, have a significantly average case orientation, and may require more extensive adaptation to the new design criterion. Further, there are clearly new types of optimizations that should be considered. Optimizations which minimize WCET are the most obvious, but others minimizing the number of tasks required to represent a computation, or minimizing the utilization of a shared resource could be very useful.

The source language and translation methods might also be expanded in ways that support the creation of design and analysis tools. The SDL already provides a wealth of information which should be useful in this regard, and the reduction methods' maintenance of source line information provides important support for debugging and analysis tools. For example, it is possible to imagine a tool which would enable the developer to relate the structure of the task group representing a computation back to its source code. This would create a way for the developer to use feedback about the schedulability of a computation's task group to modify the source.

Using such a tool, the developer could consider various options for optimization of WCET, the number of tasks used to represent a computation, the utilization of specific shared resources, or a number other factors. Such a tool could be integrated with the compiler, enabling the developer to examine and interact with specific parts of the subgraph reduction phase. Tools could also be created which integrate the SGS with the simulation testbed more tightly. The advantage of this would be the ability to simulate an application, and compare the results of the simulation with

the application's actual behavior. Simulation results could also be used to get early indications of problems, or the need to optimize particular aspects of a computation.

Extending the support for distributed and fault tolerant computing would be another interesting area. The Spring-C language and the current system implementation provide some support for distributed computing, but this could be extended. For example, while the SDL provides ways to specify the layout of the application software on the target nodes, tools either helping determine an acceptable layout or automatically deriving one, would be useful. The operating system could provide more flexible support for moving computations, for dynamically activating and reconfiguring them, and for more varied forms of IPC. All of this would make the system more flexible, and capable of supporting a wider range of applications. Fault tolerance support would require related modifications, but would also involved extensions tot he existing SDL support, and to the scheduler.

The use of the system to develop and support real applications will provide more detailed feedback. Clearly, the system must be used for real development before we can be completely sure that it provides adequate support. Experience with real applications will also provide important information about what kinds of design, analysis, and debugging tools will be most useful.

At the system level, the implications of new target hardware, and of designing hardware specifically for the support of real-time systems are interesting. As discussed in Sections 8.1.3 and 8.2, the features of current hardware designed for conventional applications are not always appropriate for predictable real-time systems. Selecting new target hardware, and developing methods for using it predictably, is thus a nontrivial challenge. Extension of the SSCoP design to include more of the scheduling algorithm, and to support a wider range of system operations is one possible way

to extend our work in hardware designed for real-time. Designing a floating point coprocessor whose behavior could be more accurately predicted, and which could be used predictably, would be an excellent addition to a real-time system. The ultimate goal is to produce a node level design in which each component is specifically designed for real-time.

In summary, the need of real-time systems for design and implementation techniques untainted by the customs established by decades of conventional system design is clear, and much remains to be done. This is not meant to say that all, or even most, aspects of conventional design are inappropriate, only that the needs of real-time systems are sufficiently different from those of conventional systems that designers should begin with a clean slate, and be free to choose the *best* solution to the problem.

APPENDIX A

SDL GRAMMAR

This appendix presents the grammar of the System Description Language (SDL), as discussed in Chapter 4. The title of each section notes, in parentheses, the section of the dissertation in which that portion of the SDL is discussed.

System Description Language Top Level (4.2)

```
config_info  ::=  [conf_item]* system_layout [conf_item]* network_topology [conf_item]*
                | [conf_item]* network_topology [conf_item]* system_layout [conf_item]*
conf_item    ::=  node_desc
                | process_desc
                | process_group_desc
                | resource_desc
                | shared_seg_desc
                | task_group_desc
```

Successor List (4.2.1)

```
succ_list     ::=   succ_begin [succ_item]*
succ_begin    ::=   Begin: prec_list;
succ_item     ::=   name: prec_list;
prec_list     ::=   prec_item
                  | prec_list, prec_item
prec_item     ::=   list_item
                  | (list_item delay_val)
list_item     ::=   proc_group_name
                  | proc_name
                  | (proc_name task_name)
                  | task_name
delay_val     ::=   INTNUM
                  | (Comm_delay port_name_list)
                  | (Comm_delay INTNUM port_name_list)
```

Process Specification (4.2.1.1)

```
process_desc  ::=   Process(name) { [proc_attr]* } ;
proc_attr     ::=   comp_spec;
                  | exec_spec;
                  | sched_spec;

comp_spec     ::=   Comp_type comp_type;
                  | Importance INTNUM;
                  | Non_Periodic;
                  | Period INTNUM;
                  | Periodic;
                  | Separation INTNUM;
                  | Value INTNUM;

exec_spec     ::=   Code name
                  | Import name_list;
                  | Sharing name_list;
                  | Sync_ports port_list;

sched_spec    ::=   Deadline dist_spec;
                  | Deadline_type dln_type;
                  | Importance INTNUM;
                  | Laxity dist_spec;
                  | RT_type rt_type;

comp_type     ::=   Critical
                  | Essential
                  | Non_essential
dln_type      ::=   Hard
                  | Soft
                  | Non_real_time
port_list     ::=   port_item
                  | port_list, port_item
port_item     ::=   name
                  | (name use_type)
use_type      ::=   Receive
                  | Send
```

Process Group Specification (4.2.1.2)

```
proc_grp_desc   ::=   Process_group(name) { [pg_attr]* } ;
pg_attr         ::=   comp_spec;
                  |   Fault_tolerance { ft_spec };
                  |   Process_graph { succ_list } ;
                  |   sched_spec;

ft_spec         ::=   alternative_spec
                  |   copies_spec;
                  |   pb_spec;
                  |   voting_spec;

alternative_spec ::=  Alternative proc_name_list;

copies_spec     ::=   Copies proc_name (min_cp, max_cp);
                  |   Copies proc_name INTNUM;
min_cp          ::=   INTNUM
max_cp          ::=   INTNUM

pb_spec         ::=   primary_proc backup_proc
                  |   backup_proc primary_proc
primary_proc    ::=   Primary proc_name;
backup_proc     ::=   Backup proc_name;

voting_spec     ::=   Voting { voters arbiter };
voters          ::=   Voters proc_name_list;
arbiter         ::=   Arbiters proc_name_list;
```

Task Group Specification (4.2.1.3)

```
task_grp_desc  ::=  Task_group(proc_name) { tg_def } ;
tg_def         ::=  group_def [task]+
group_def      ::=  Group_list { succ_list } ;

task           ::=  Task(name) { [task_attr]* } ;
task_attr      ::=  M_time dist_spec;
                 |  Non_Preemptive;
                 |  Preemptive;
                 |  Resources [ru_spec]+;
                 |  sched_spec;
                 |  W_time dist_spec;

ru_spec        ::=  (name [use_attr]+)
use_attr       ::=  Start time_val;
                 |  End time_val;
                 |  Exclusive;
                 |  Sim_prob use_prob excl_prob;
                 |  Shared;
use_prob       ::=  FLOATNUM
excl_prob      ::=  FLOATNUM
```

Resource Description (4.2.2)

```
resource_desc   ::=   Resource(name) { [res_attr]* };
res_attr        ::=   Access access_type;
                    | Segment name;
                    | export_sym
                    | Instances INTNUM;
                    | Mode mode_type;
                    | Type res_type;

res_type        ::=   Read
                    | Write
                    | RW
access_type     ::=   Both
                    | Exclusive
                    | Shared
export_sym      ::=   Export name_list;
                    | Export export_type name_list;
export_type     ::=   Direct
                    | Indirect
mode_type       ::=   Appl
                    | Both
                    | Sys
```

Shared Segment Description (4.2.3)

```
shd_seg_desc    ::=   Shared_seg(name) { [seg_attr]* };

seg_attr        ::=   Code name;
                    | Logical_base HEXNUM;
                    | Matching;
                    | Memory_mapped;
                    | Mode mode_type;
                    | Physical_base HEXNUM;
                    | Predefined;
                    | Resources name_list;
                    | Size HEXNUM;
```

Node Structure Grammar (4.2.4)

```
node_desc          ::=   Node(name) { [node_attr]* };
node_attr          ::=   processor_desc
                       | mem_board_desc

processor_desc     ::=   Processor(name) { [processor_attr]* };
processor_attr     ::=   memory_area_spec
                       | Use processor_use;
mem_base           ::=   HEXNUM
mem_size           ::=   HEXNUM
processor_use      ::=   Appl;
                       | IO;
                       | Sys;
mem_board_desc     ::=   Mem_board(name) { [mem_board_attr]* };
mem_board_attr     ::=   memory_area_spec
memory_area_spec   ::=   Memory_area(mem_board_use, mem_base, mem_size);
mem_board_use      ::=   Data;
                       | Memory_mapped;
```

Network Topology Grammar (4.2.5)

```
network_topology   ::=   Network { [node_connections]* };
node_connections   ::=   Node (name) { connection_list };

connection_list    ::=   connection_spec
                       | connection_list, connection_spec
connection_spec    ::=   Connection (node_name) { [connection_attr]* } ;
connection_attr    ::=   Latency INTNUM;
                       | Speed INTNUM;
```

System Layout Grammar (4.2.6)

```
system_layout       ::=   Layout { [node_layout]* };

node_layout         ::=   Node_layout(name) { [node_layout_attr]* };
node_layout_attr    ::=   processor_layout
                        | mem_board_layout

processor_layout    ::=   Processor_layout(name) { [proc_layout_item]* };
proc_layout_item    ::=   Appl_set { [proc_load_item]* };
                        | System_set { [proc_load_item]* };

mem_board_layout    ::=   Mem_board_layout(name) { [mb_layout_item]* };
mb_layout_item      ::=   Appl_set { [mb_load_item]* };
                        | System_set { [mb_load_item]* };

proc_load_item      ::=   Boot_proc name;
                        | Process_set name_list;
                        | Resource_set name_list;
                        | Shared_seg_set name_list;

mb_load_item        ::=   Resource_set name_list;
                        | Shared_seg_set name_list;
```

Miscellaneous

dist_spec	::=	(C INTNUM)
	|	(E FLOATNUM)
	|	(E FLOATNUM)(INTNUM)
	|	(G FLOATNUM FLOATNUM)
	|	(G FLOATNUM FLOATNUM)(INTNUM)
	|	(N FLOATNUM FLOATNUM)
	|	(N FLOATNUM FLOATNUM)(INTNUM)
	|	(U INTNUM)
name_list	::=	name
	|	name_list ',' name
name	::=	$[a-zA-Z\_]^+[a-zA-z0-9\_.]^*$
time_val	::=	INTNUM
INTNUM	::=	$[0-9]^+$
	|	$-[0-9]^+$
HEXNUM	::=	$0x[0-9a-fA-F]^+$
FLOATNUM	::=	$[0-9]^+.[0-9]^+$
	|	$-[0-9]^+.[0-9]^+$

APPENDIX B

REDUCTION ALGORITHMS

This appendix presents the subgraph reduction algorithms discussed in Chapter 6 in greater detail. While this level of detail is not required to appreciate the results presented in this dissertation, it is likely to be useful to anyone trying to modify or extend the compiler implementation which embodies the results described here, or who wishes to implement these methods in the context of another software generation system. The title of each section notes the section of the dissertation to which it relates most strongly in the parentheses.

Basic Reduction Algorithms(6.1)

Figure B.1 gives the simplest form of the algorithm for reducing a procedure in pseudo-C code. A pointer (ptg) to the TG of a procedure, as constructed from its RTL graph, is given as input to the reduction algorithm which produces the irreducible TG, or ITG, of the procedure as output. The algorithm assumes the TG of a Spring-C procedure has the basic structure which was illustrated in Figure 5.11 in Section 6.1. As that figure showed, a procedure's intermediate representation, and thus the original TG constructed from it, consists of a sequence of nodes representing the prologue of the procedure, a sequence forming the body, followed by a sequence forming the exit code. The reduction procedure *Reduce-Proc1* reduces each sequence

```
Reduce-Proc1 (tg_proc_t *ptg)
1. curr = ptg → next;
2. terminators[0] = STRUCT_EXIT;
3. terminators[1] = STRUCT_RETURN;
4. terminators[2] = 0;
5. reduce_item_sequence(curr, &next, terminators);
6. if (type(next) == STRUCT_RETURN) {
7.         excise unnecessary nodes;
8. }
9. if (type(next)! = STRUCT_EXIT) {
10.        error;
11. }
12. curr = ptg → exit → next;
13. terminators[0] = STRUCT_END;
14. terminators[1] = 0;
15. reduce_item_sequence(curr, &next, terminators);
16. Excise EXIT node;
17. curr = ptg → next;
18. terminators[0] = STRUCT_END;
19. terminators[1] = 0;
20. reduce_item_sequence(curr, &next, terminators);
21. add_itg_to_table(ptg);
end Reduce-Proc1
```

Figure B.1. Reducing the TG of a Procedure - Simplest Version

in turn, by calling *reduce_item_sequence*. The *reduce_item_sequence* is presented after the explanation of *Reduce-Proc1*.

Lines 1 through 5 of *Reduce-Proc1* reduce the sequence representing the prologue and the body of the procedure. The *reduce_item_sequence* routine begins with *curr* pointing to the first node of the sequence being reduced, *terminators* containing a null terminated list of node types that terminate the sequence, and the value of *next* unspecified. It terminates with *curr* pointing to the last node of the reduced sequence and *next* pointing to the node that terminated it. Lines 6 through 8 state

that if the sequence terminated at a *return* structural node, then the algorithm excises the *RETURN* structural node that should be followed by the *EXIT* node. If not, an error is declared.

Lines 9 through 11 check that *next* now points to the *EXIT* structural node, whether or not a *RETURN* originally existed. If not, the procedure's structure is wrong, and an error is declared. Lines 12 through 15 reduce the sequence of nodes representing the exit code of the procedure, which lies between the *EXIT* and *END* structural nodes. When both component sequences are reduced, then the *EXIT* node is deleted and the two sequences concatenated at line 16. Lines 17 through 20 apply *reduce_item_sequence* to reduce the combined sequence, producing the irreducible TG, or ITG, of the procedure. Note that for procedures containing no suspension points, the ITG is always a single time node giving the WCET of the procedure. Line 21 stores the ITG for future use when the TGs of procedures calling the one whose ITG was just produced are being reduced.

Figure B.2 gives the algorithm for reducing a sequence of items. Lines 1 through 13 examine the first node in the sequence, and take appropriate action. The routine *classify_node* makes the algorithm, and the pseudo-code, more compact by combining several cases that are handled similarly. For example, it returns the value *STRUCT_BLOCK* whenever it sees an *if*, *while*, *switch*, *do-while* or *for* structural node since they indicate the start of a embedded block. Similarly, it returns *SUSPEND_NODE* when it encounters any of a number of suspension points, and returns *TIME_NODE* when it encounters a time node.

If the sequence begins with a time node, then no action is required. If the value returned by *classify_node* indicates that the sequence begins with an embedded block, then *reduce_block* is called to handle it. Like *classify_node*, this routine helps make

```
Reduce_Item_Sequence1 (tg_node_t sqp, tg_node_t *after,
                       int *terms)
1. curr = *sqp;
2. ret_val = classify_node(curr);
3. switch(ret_val) {
4. case TIME_NODE:
5.       break;
6. case STRUCT_BLOCK:
7.       reduce_block(&curr);
8.       break;
9. case STRUCT_PROC_CALL:
10.      substitute_proc_call(&curr);
11.      *sqp = curr;
12.      break;
13. }
14. next = curr → next;
15. while (next&&!(next ∈ terms)) {
16.      ret_val = classify_pair(curr, next);
17.      switch(ret_val) {
18.      case PAIR(TIME_NODE, TIME_NODE):
19.           merge_time_nodes(curr, next);
20.           next = curr → next;
21.           break;
22.      case PAIR(TIME_NODE, STRUCT_BLOCK):
23.           reduce_block(&next);
24.           curr → next = next;
25.           break;
26.      case PAIR(TIME_NODE, STRUCT_PROC_CALL):
27.           substitute_proc_call(&next);
28.           curr → next = next;
29.           break;
30. }
31. *after = next;
end Reduce-Item-Sequence1
```

Figure B.2. Reducing an Item Sequence - Simplest Version

the routine more compact by combining several cases. It looks at the type of block indicated by *curr*, and calls the appropriate reduction routine. For example, if *curr* points to a node of type *STRUCT_IF*, then *reduce_block* calls the *reduce_if* routine. If the sequence begins with a procedure call, the ITG of the called procedure is substituted for the structural nodes marking the call as was illustrated in Figure 5.11 in Section 6.1. Since every procedure begins with a non-null prologue, we know the ITG of every procedure begins with a time node. The time require for the instruction (jsr) calling the procedure is added to the WCET of the initial time node in the called procedure's ITG.

Lines 1 through 14 effectively "prime the pump" for the main part of the algorithm in lines 15 through 30, which follow a convention used throughout the other reduction routines; *curr* points at the last node in the reduced sequence, while *next* points to the first node in the portion of the TG still requiring reduction. The *curr* and *next* pair of pointers thus track the *reduction boundary* as it moves through the TG. Behind the boundary, the TG is in irreducible form, and thus represents the prefix of the ITG as it grows. In front of the boundary is the next item requiring reduction. This is illustrated by setting *next* at line 14 and the use of *classify_pair* which classifies the pair of nodes defining the reduction boundary, and returns a value encoding the type of each.

The *while* loop at line 15 continues to consider what reduction is required to move the reduction boundary forward so long as *next* is a non-null pointer, and the node it refers to is of a type terminating the sequence being reduced. The cases of the *switch* statement beginning at line 17 handle each possibility. If both nodes are time nodes, this corresponds to the reduction of Figure 6.1a in Section 6.1 which calls *merge_time_nodes* to combine the two consecutive time nodes. The WCET of the

node pointed to by *curr* becomes the sum of the WCETs of the two, while the RTL and source subgraphs are updated by setting the endpoints of *curr* to the endpoints of *next*. The next pointer of *curr* is set to that of *next* and the node pointed to by *next* is deallocated.

When *next* points to the beginning of an embedded block, then *reduce_block* is called. When it returns, the *next* pointer of *curr* is set to the first node in the sequence which is the ITG representing the block. Note that when *reduce_block* reduces a block it always ensures that the *next* pointer of the node at the end of the ITG produced points to whatever node originally followed the original block. The next iteration of the loop reclassifies the boundary, which for procedures containing no suspension points will now be a pair of time nodes. The third case substitutes the ITG of any called procedures, and then moves the reduction boundary forward on the next loop iteration.

Finally, when the terminating node is found, the *after* parameter, which is the *next* pointer of the routine calling this one, is set to point at the terminating node, at line 31. This ensures that when the *reduce_item_sequence* routine returns, the *next* pointer in the routine calling it will always point to the terminating node, since a pointer to *next* is always passed in as the *after* parameter.

Figure B.3 illustrates the essential elements of the algorithm used to reduce an *if* conditional block. Lines 1 through 7 reduce the nodes representing the conditional expression of the *if block*, lines 8 through 14 reduce the *THEN* clause, and lines 15 through 21 reduce the *ELSE* clause. These reductions ensure that the reduction of Figure 6.2 in Section 6.1 is applicable, since each component of the block is fully reduced to a single time node.

```
Reduce_If (tg_if_t *ifp)
1. curr = ifp → next;
2. terminators[0] = STRUCT_TEST;
3. terminators[1] = 0;
4. reduce_item_sequence(curr, &next, terminators);
5. if (type(next)! = STRUCT_TEST) {
6.         error;
7. }
8. curr = beginningofTHENclause;
9. terminators[0] = STRUCT_END;
10. terminators[1] = 0;
11. reduce_item_sequence(curr, &next, terminators);
12. if (type(next)! = STRUCT_END) {
13.         error;
14. }
15. curr = beginningofELSEclause;
16. terminators[0] = STRUCT_END;
17. terminators[1] = 0;
18. reduce_item_sequence(curr, &next, terminators);
19. if (type(next)! = STRUCT_END) {
20.         error;
21. }
22. curr = combine_tg_sequences(THEN, ELSE);
23. Delete STRUCT_TEST node;
24. curr = ifp → next;
25. terminators[0] = STRUCT_END;
26. terminators[1] = 0;
27. reduce_item_sequence(curr, &next, terminators);
28. Update time node RTL subgraph
29. Deallocate structural nodes
end Reduce-Item-Sequence1
```

Figure B.3. IF Block Reduction

Line 22 shows that the next step in the *if* block reduction is the combination of the ITGs representing the *THEN* and *ELSE* clauses. In the case we consider here, the combination is a special case of the more complex general problem considered later, since the ITGs being combined contain no suspension points.

The sequence resulting from combining two sequences which are single time nodes is a single time node having a WCET equal to the maximum of the two time nodes being combined. When combined, the *TEST* node is deleted, and the sequences for the conditional and the combined clauses are concatenated by line 23. Lines 24 through 27 reduce the concatenated sequence to a single node, which now gives the WCET for the whole *if* block. Line 28 illustrates that the RTL and source subgraph for the resulting time node is set to reflect the beginning and end of the whole *if* block which it now represents. The structural nodes which are no longer needed are deallocated by line 29.

This completes the description of several of the simpler reduction algorithms. Later sections of this appendix will present methods for extending these algorithms to handle first structured jumps and then suspension points. It is also important to note that not all of the reductions are explained in detail. The methods required for their implementation are, however, quite similar to those for the algorithms described, and should not pose a significant implementation problem.

Simple Reduction Example

The way in which the recursive descent strategy controls the application of specific subgraph reductions should be clarified by an example. Consider the reduction of the simple procedure *classify* given in Figure B.4. It contains an assignment statement, a nested *if*, and a final *return* statement. The *return* statement is not really a structured

```
int
classify()
{
    int a,b;
    a = Global_var;;
    if ( a < 0 ) {
        b = 1;
    } else {
        if ( a < 22 ) {
            b = 2;
        } else {
            b = 3;
        }
    }
    return((b+33)*47);
}
```

Figure B.4. A Simple Procedure for Reduction

jump when it comes at the end of the function, since the normal flow of control will take the program tot he destination of the jump in any case. While simple, this procedure is an excellent illustration of how recursive descent controls the application of subgraph reduction rules.

Figure B.5a illustrates the original time graph constructed for the *classify* routine. In the TG, the *A* and *B* time nodes represent the procedure's prologue code, and the assignment of *Global_var* to the local variable *a*, respectively. The subgraphs representing the nested *if* statements should be obvious. The *H* time node represents the expression calculating the value returned by the *return* statement, while the *I* time node represents the procedure's exit code.

The *reduce_proc* routine is passed the pointer to the *P* node of the TG. It calls *reduce_item_sequence* to reduce the body of the procedure. The *reduce_item_sequence* routine uses the *curr* pointer to keep track of the last node in the portion of a sequence which has already been reduced to irreducible form, and the *next* pointer to keep track

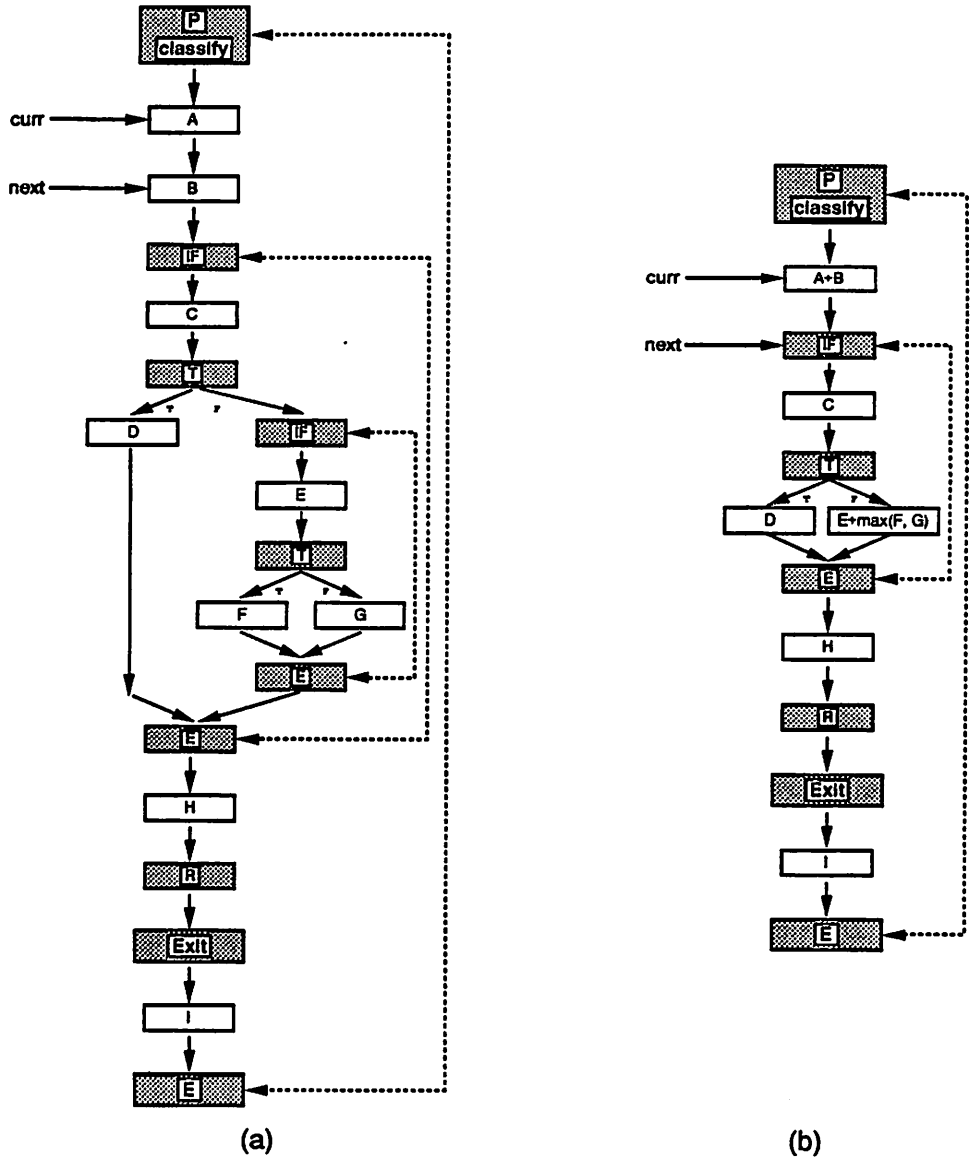


Figure B.5. Original TG and TG After Nested If Reduction

of the next item in the sequence that should be reduced to a form permitting it to be added to the irreducible portion. The *reduce_proc* routine will thus make the call to *reduce_item_sequence* to reduce the procedure body with *curr* pointing to the *A* time node, and *next* pointing to *B*. Its first action is to combine the *A* and *B* nodes using *merge_time_nodes*. The *next* pointer would then point to the first *IF* structural node. That will cause *reduce_item_sequence* to call *reduce_block*, which notes the block type and calls *reduce_if* on the outer conditional block. The first call to *reduce_if* will

reduce the sequence representing its conditional expression, already the single time node C, and then do the same for the *THEN* and *ELSE* clauses.

The *THEN* clause ($b = 1$) is already represented by the single time node D, but the call to *reduce_item_sequence* handling the *ELSE* clause will call *reduce_block* at line 7 in Figure B.2 to reduce the nested conditional, which will call *reduce_if* again in turn. Figure B.5b shows the TG after the return from the *reduce_block* call handling the inner *if*. The inner conditional has been reduced to a node with a WCET of $E + max(F, G)$.

Note that the *curr* and *next* pointers of the first call to *reduce_item_sequence* are unchanged, since the current reductions are taking place within the context of reduction routines called recursively, and we have not yet returned to the level of the original *reduce_item_sequence* handling the procedure body. We have, however, returned to the context of the *reduce_if* handling the outer *if* which is now reducible since each of its component sequences is in ITG form. The current frame handles the rest of the reduction, lines 22 through 29 in Figure B.3, producing a time node with the proper value, and returning to the *reduce_item_sequence* handling the body of the procedure.

Figure B.6a shows the TG at this point. The pointer *curr* points to the time node with the value of $A + B$, and next to the time node representing the nested conditionals. The *reduce_item_sequence* routine will apply the *merge_time_nodes* routine twice, merging the three consecutive time nodes. This leaves *next* pointing to the *RETURN* structural node, which terminates the sequence, and the *reduce_item_sequence* call-frame reducing the body of the procedure returns to *reduce_proc*. The *RETURN* is eliminated, since it signifies a jump to the current location in the procedure code, and the exit code is reduced by by another call to *reduce_item_sequence*. The *EXIT*

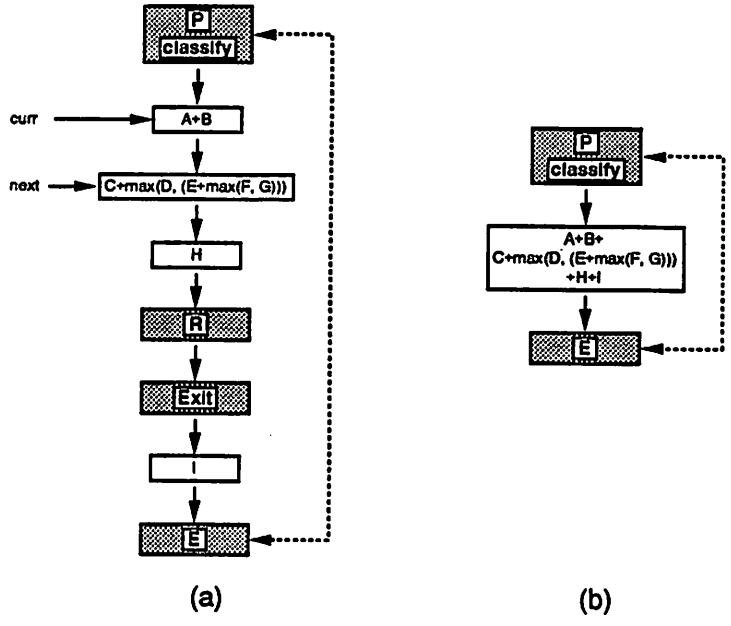


Figure B.6. TG After IF Reductions, and ITG of Procedure

node is eliminated, leaving the time node representing the body of the procedure and the time node representing the exit code ready for combination by *merge_time_nodes*. Figure B.6b illustrates the ITG thus produced, which is saved for substitution into the TG of the calling procedure whenever a call to *classify* is encountered.

Alternative Path Calculation(6.2.1)

The introduction of structured jumps into a Spring-C program makes it necessary to extend the methods of subgraph reduction as discussed in Section 6.2. A significant portion of that extension is the implementation of the alternative path calculation discussed in Section 6.2.1, and presented in detailed form here.

Structured jumps are correctly handled by extending the *reduce_item_sequence* algorithm, as illustrated in Figure B.7. The new case which handles the *return* structured jump, in lines 18 through 24, is analogous to similar cases for *break* and *continue*. The context block is found by reading the top of the *return* context

stack. Calculating the path length was discussed in Section 6.2.1, so assume that *path_length*(*context*, *return*) is known. This is then used to establish an alternative path which is added to the return alternative list associated with the context block. The last step is to delete the sequence being reduced from the TG. This is vital, since it is exactly this sequence which represents the unique portion of the alternative path which must be excluded from the calculation addressing the main path.

Reducing Time Graphs Containing Suspension Points(6.3)

Handling suspension points during the reduction of a TG node sequence requires only the straightforward changes to *reduce_item_sequence1* illustrated in Figure B.8, which also illustrates the use of *wcet_so_far* discussed in Section 6.2.1 with respect to its use in calculating *path_length*(*context*, $sj - node$). Line 3 represents lines 2 through 13 of *reduce_item_sequence1*, which handled the cases where the sequence begins with an embedded block or procedure call. The only change to this section of the original routine required to handle suspension points is a case doing nothing if the sequence begins with a suspension point.

The new cases in the *while* loop handle the presence of suspension points in a sequence being reduced. The case at lines 14 through 17 simply advances the reduction boundary to include the suspension point following a time node, while the case at lines 18 through 22 does the same when a time node follows a suspension point. Lines 23 through 26 handles the case where two suspension points directly follow one another with no time node between them. This is a comparatively rare situation, but merging the suspension points minimizes the number of tasks required to represent the process. How a suspension point can represent more than one reason for process suspension is discussed in Section 6.3.1. The other cases take no new actions,

but simply account for the fact that a nested block or procedure call might follow a suspension point.

If suspension points always appeared within the sequence representing the body of a procedure, and never in a nested block, then *reduce_item_sequence2* would be sufficient to reduce the original TG of every process to linear ITG form. Since suspension points are not eliminated by reduction, the ITG would be a sequence of alternating time and suspension point nodes. However, if the SP appears inside a conditional or a loop block, then the reduction of these blocks must be modified to handle the situation properly.

Recall that the reduction routines always begin by using *reduce_item_sequence* to reduce each of their components to ITG form before performing the final reduction of the block. The new version of each reduction routine will still do that, but they can no longer assume that the component sequences will reduce to single time nodes. For example, when a suspension point appears in one or more cases of a conditional block the *reduce_if* routine will proceed as illustrated in Figure B.3, until the clauses of the conditional must be combined. The call to *combine_tg_sequences* is unaltered, but the calculation it performs is substantially more complex, and may require transformations of the procedure's RTL representation. Section 6.3.1 discussed this more general problem of combining parallel sequences in some detail, and the next section of this Appendix presents more detailed information.

```
Reduce_Item_Sequence2 (tg_node_t *sqp,
                       tg_node_t *after, int *terms)
1. *** Lines 1 to 13 of reduce_item_sequence1 ***
2. next = curr → next;
3. while (next&&!(next ∈ terms)) {
4.         ret_val = classify_pair(curr, next);
5.         switch(ret_val) {
6.         case PAIR(TIME_NODE, TIME_NODE):
7.               merge_time_nodes(curr, next);
8.               next = curr → next;
9.               break;
10.        case PAIR(TIME_NODE, STRUCT_BLOCK):
11.              reduce_block(&next);
12.              curr → next = next;
13.              break;
14.        case PAIR(TIME_NODE, STRUCT_PROC_CALL):
15.              substitute_proc_call(&next);
16.              curr → next = next;
17.              break;
18.        case PAIR(TIME_NODE, STRUCT_RETURN):
19.              get context block from return stack;
20.              calculate path length;
21.              create alternative path structure;
22.              add it to the context block's return list;
23.              delete current sequence;
24.              break;
25. }
26. *after = next;
end Reduce-Item-Sequence1
```

Figure B.7. Reducing an Item Sequence - Structured Jump Extension

```
Reduce_Item_Sequence3 (tg_node_t sqp, tg_node_t *after,
                                    int *terms, int wcet_so_far)
1. curr = *sqp;
2. input_wcet = wcet_so_far;
3. *** lines 2 through 13 of reduce_item_sequence1 ***
4. wcet_so_far+ = wcet(curr);
5. next = curr → next;
6. while (next&&!(next ∈ terms)) {
7.         ret_val = classify_pair(curr, next);
8.         switch(ret_val) {
9.         case PAIR(TIME_NODE, TIME_NODE):
10.            wcet_so_far+ = wcet(next);
11.            merge_time_nodes(curr, next);
12.            next = curr → next;
13.            break;
14.        case PAIR(TIME_NODE, SUSPENSION_NODE):
15.            curr = next;
16.            next = curr → next;
17.            break;
18.        case PAIR(SUSPENSION_NODE, TIME_NODE):
19.            wcet_so_far+ = wcet(next);
20.            curr = next;
21.            next = curr → next;
22.            break;
23.        case PAIR(SUSPENSION_NODE, SUSPENSION_NODE):
24.            merge_suspension_nodes(curr, next);
25.            next = curr → next;
26.            break;
27.        case PAIR(TIME_NODE, STRUCT_BLOCK):
28.        case PAIR(SUSPENSION_NODE, STRUCT_BLOCK):
29.            reduce_block(&next, wcet_so_far);
30.            curr → next = next;
31.            break;
32.        case PAIR(SUSPENSION_NODE, STRUCT_PROC_CALL):
33.        case PAIR(TIME_NODE, STRUCT_PROC_CALL):
34.            substitute_proc_call(&next);
35.            curr → next = next;
36.            break;
37. }
38. *after = next;
end Reduce-Item-Sequence3
```

Figure B.8. Reducing an Item Sequence - Suspension Point Additions

ITG Combination (6.3.1)

Figure 6.14 specifies the main steps in determining what the structure of the composite sequence which will represent the input sequences should be. The first step, at line 1, is to make each of the input sequences easier to manipulate by building the *sequence table.* Figure B.10 illustrates the structure definition for a sequence table element. Each entry in the table is a structure noting the length of the sequence (length), a pointer to an array holding pointers to each node in the sequence (sq), and a parallel array noting any constraints on how nodes in the sequence may be mapped to the composite (depend). The *sq* array makes it possible to address each node in a sequence directly, rather than having to follow a chain of pointers from the beginning of the sequence. For example, the pointer to the third node in the fourth sequence would referenced by the expression, $Sq\_table[3] \rightarrow sq[2]$, since arrays in C are indexed from zero. The *depend* array notes whether a node may be mapped to the composite independently of its predecessor or *must* follow its successor directly, whether the node ends in a jump, and whether it is permissible to insert nodes into the sequence after the node in a given position. The dependency flags associated with the third node in the fourth sequence would, for example, be found by the expression, $Sq\_table[3] \rightarrow depend[2]$. Each of the dependency properties constrains the mapping of a sequence's nodes to the composite template, as will be seen shortly.

The second step in the algorithm sorts the sequences by length, with the longest sequence coming first in the table. The third step, which determines the structure of the composite sequence, begins by finding the longest sequence in the input set. Since the sequence table is sorted by length the longest sequence or sequences in the input set are at the beginning of the table. The longest input sequence generally defines the structure of the composite , but not always. The composite may be longer if there

```
setup_templates (tg_node_t **sequences, int number, int *terms)
1. allocate and initialize sequence tables
2. sort sequences by length
3. determine composite template from longest sequence(s)
4. allocate best and current composite templates
end setup_templates
```

Figure B.9. Composite Template Construction

```
typedef struct {
  int          length;
  tg_node_t **sq;
  int         *depend;
} sq_table_item_t;
sq_table_item_t *Sq_table;
```

Figure B.10. Sequence Table Structure Definition

are two input sequences of the same length which are not already balanced. In that case, the composite must be one node longer than either to ensure that both may be mapped to it.

The other reason the composite may be longer is when the longest input sequence ends in a suspension point, but some other input sequence ends in a jump. Sequences often end with jump instructions since they represent the bodies of loops or conditional clauses. This fact constrains the structure of the composite template and how these sequences are mapped to it because it is meaningless to insert a node in the TG, and thus new nodes into the RTL graph of the procedure, *after* a jump. Doing so creates dead code which is never executed. So, if the longest sequence ends in a suspension point, but another input sequence ends in a jump, the composite is one

longer and ends in a time node to which a time node representing the jump can be mapped.

```
typedef struct {
  int          node_type; /* suspension point or time node */
  int          value;     /* delay for SP, WCET for TN */
  tg_node_t **nodes;      /* nodes in each sq assigned here */
} template_item_t;
template_item_t *Best_tpl;
unsigned int     Best_value;
template_item_t *Current_tpl;
```

Figure B.11. Template Structure Definition

Figure B.11 shows the definition of a template item and of the *Best_tpl* and *Current_tpl* templates. *Best_tpl* is used to store the best mapping found so far, while *Current_tpl* stores the mapping currently being examined. Each item in the template represents a node in the composite sequence, noting its type (time node or suspension point) in the *node_type* field. The *nodes* field points to an array of node pointers indicating which nodes in the input sequences are mapped to this item in the template. For example, if we wish to examine the node in the third input sequence which is mapped to the fifth position in the *Best_tpl* template we would use the expression, $Best\_tpl[4] \rightarrow nodes[2]$. Note that when no node from a particular input sequence is mapped to a given position in the template, then the corresponding entry in the *nodes* array is null.

The *value* of the template item is determined by the values of the nodes in the input sequences that are mapped to it. The value of a time node in the template is the maximum WCET of all the nodes mapped to it, while the value of a suspension point in the template is the maximum delay associated with all suspension points mapped to it.

```
iterate_sequence_mappings (sequence_table)
1. establish first mapping in Current_tpl
2. Best_value = WORST_VALUE;
3. is_another = 1;
4. while (is_another) {
5.         curr_val = evaluate_mapping(Current_tpl);
6.         if (curr_val < Best_val) {
7.             copy_tpl(Best_tpl, Current_tpl);
8.             Best_value = curr_val;
9.         }
10.        is_another = next_mapping(Current_tpl);
11. }
end setup_templates
```

Figure B.12. Composite Template Construction

The *iterate_sequence_mappings* routine is illustrated in Figure B.12. Lines 1 through 3 of the algorithm establish the first mapping and prepares to enter the loop of lines 4 through 11 which steps through the set of all possible mappings. Each mapping is evaluated, as explained in Section 6.3.1.2, and its value compared to the best seen so far. The most interesting point is the *next_mapping* routine which selects the next mapping by permuting the mapping of each sequence in the order that sequences appear in the sequence table. Thus, when a particular mapping for the sequence at position n in the table is being considered, all possible mappings for sequences at positions greater than n are considered before the next mapping of the sequence at position n is considered.

When *next_mapping* determines that no other legal or significant mapping exists it returns 0, and the loop stepping through each mapping terminates. When that occurs, *Best_tpl* holds the mapping with the best value, and the *nodes* array at each

template position contains values which are either null to indicate that a node that must be inserted, or a pointer to the node from an original sequence that has been mapped to that node of the composite.

APPENDIX C

TASK GROUP CONSTRUCTION ALGORITHMS

This appendix provides a more detailed view of the algorithms discussed in Chapter 7, which are used to construct the task group representing a process from its ITG. It also presents the algorithms used to construct the task group for a computation implemented by a group of processes from the task groups describing each component process, and the information describing the structure of the process group. The title of each section notes the section of the dissertation to which it relates in the parentheses.

Task Group Construction(7.1)

When the executable file for a process is otherwise complete, *spr-ld* uses the algorithm illustrated in Figure C.1 to construct the task group. The operations of the algorithm are expressed in pseudo-C and in the notation defined in Section 4.2.1. The algorithm specifies precisely how the set of tasks in the group $TG(P)$ representing the process P are created, and the values of their attributes determined. Specifically, how their WCET and resource use are derived from the process's ITG. It also shows how the precedence relations describing the structure of the task group are derived from the ITG. The ITG is the primary source of information for the algorithm, but other data in the accumulated SDL information are also used. Note that we are not considering

processes engaging in synchronous communication at this time. The extensions to *build_task_group1* required for synchronous communication are discussed later in this appendix.

```
build_task_group1 (itg_t *proc_exec)
1. Find the process name: P
2. Create task group TG(P)
3 tc = 1
4. Tasks(P) = {T1}
5. WCET(T1) = SWITCH_TO_TIME
6. PR(P) = (Begin, T1, 0)
7. node = proc_exec → next;
8. while (node != proc_exec → end) {
9.         switch (node → type) {
10.        case TIME_NODE:
11.             WCET(Ttc)+ = WCET(node)
12.             break;
13.        case SUSPENSION_NODE:
14.             WCET(Ttc)+ = Entry_time(node)
15.             Tasks(P) = Tasks(P) ⋃ Ttc+1
16.             WCET(Ttc+1) = Exit_time(node)
17.             PR(P) = PR(P) ⋃ (Ttc, Ttc+1, delay(node))
18.             for (∀(res, mode, t) ∈ Using) {
19.                  RU(Ttc) = RU(Ttc) ⋃ (res, mode, 0, t)
20.             }
21.             Using = Using ⋃ Res_enter(node)
22.             Using = Using \ Res_leave(node)
23.             tc++;
24.             break;
25.        }
26.        node = node → next;
27. }
28. WCET(Ttc) = SWITCH_FROM_TIME
end build_task_group1
```

Figure C.1. Task Group Construction Algorithm

Line 1 finds the name of the process, P, for which we are constructing the task group representation. The name of the group must be the name of the process it represents, as described in Section 4.2.1. This is found by looking for a process description in the set of SDL information accumulated from the files supplying components of the executable. There should be one and only one present. There should be at least one because an SDL process description should be specified in the Spring-C source file containing the *proc_exec* real-time entry point of each procedure. The object file produced by compiling that source file must obviously be one of those providing components of the executable, and so the process descriptor will be present in the SDL information accumulated from executable components.

There should be only one because otherwise the set of files used to construct the executable included a second process description and another *proc_exec* real-time entry point. This is just as meaningless as linking a conventional C program using a set of files containing two *main* procedures. This error would, however, be detected prior to reaching the point of calling *build_task_group*.

Line 2 allocates the SDL task group structure for $TG(P)$ capable of holding the set of information described in Section 4.2.1.3. Line 3 initializes the task counter *tc*, while line 4 establishes T_1 as the initial member of $Tasks(P)$. Line 5 initializes the WCET of T_1 to the time required to switch context from the dispatcher to the *proc_exec* entry point of the process. Line 6 establishes the precedence relation $(Begin, T_1, 0)$, stating that T_1 is the successor of the *Begin* node and thus initially eligible to run, as the initial member of $PR(P)$. Line 7 initializes the time graph node pointer *node* to reference the first node of the ITG.

Lines 8 through 27 specify a loop that steps through the nodes of the ITG, adding members to the task and precedence relation sets as required, as well as setting the

attributes of existing set members. If *node* points to a time node, then at line 11 the WCET of the time node is added to that of the current task T_{tc}. In the case of the first task this is added to the time require to switch context from the dispatcher to the process's *proc_exec* entry point. For later tasks, this is added to the time required to return from the suspension point.

If the *node* points to a suspension point, several issues must be addressed. Suspension points represent system calls that suspend the execution time of a process. Up to this point it has been convenient to ignore the fact that these system calls require non-zero time to switch the processor context away from the process when called, and non-zero time to switch it back when the schedule specifies that the next task in the group should begin execution. The WCET of the time nodes in the ITG account for only the WCET of the application code and of system calls which do not suspend process execution.

Line 14 accounts for the time required to *enter* the suspension point, terminating the current task T_{tc}, by adding it to the WCET of the current task. Note that the *Entry_time* and *Exit_time* of suspension points in the ITG will vary with the set of operations they represent. All of the suspension points considered in this section have the same entry and exit times. However, those of synchronous send and receive differ. For this reason, *Entry_time*(*node*) is defined to return the maximum entry time of all the suspension point types *node* represents.

When the algorithm sees a suspension point it knows that there is at least one more time node, and thus one more task. The reason for this is that the ITG of a procedure always begins and ends with time node which represent, at least, the prologue and exit code of the *proc_exec* procedure. Line 15 allocates a new task structure for the next task T_{tc+1}, and adds it to the task set. Since the execution of

this task will begin by returning from the current suspension point, line 16 initializes the WCET of T_{tc+1} to the exit time of *node*.

Line 17 adds the precedence relation $(T_{tc}, T_{tc+1}, delay(node))$ to $PR(P)$. The relation states that the current task T_{tc} is the predecessor of the next task T_{tc+1}, and that the delay attribute should be set to the delay value of *node*.

Lines 18 through 22 derive the resource use of each task by maintaining the membership of the *Using* set, which is initially empty. When a suspension point represents the beginning of a resource use block, it notes the resource, the mode, and the length of time the resource is used by the execution episode. This was discussed in Section 6.3.1. A suspension point in the ITG may represent entry into or exit from more than one resource use block as a result of node combination performed during subgraph reduction.

Lines 18 through 20 add a member to the resource use set of T_{tc} for every member of the *Using* set, since the members of the set specify the resources in use. Lines 21 and 22 then update the *Using* set membership. Line 21 adds a triple denoting the resource use to the *Using* set for every entry into a resource use block represented by *node*, and remove a member for every exit from a resource use block that *node* represents. It is important that the update of *Using* set membership be done *after* using it to determine T_{tc} resource use because the changes in resource use represented by *node* apply to T_{tc+1} and not T_{tc}.

Line 23 increments the task counter *tc*, since the next task is now the current task. Line 26 moves the ITG pointer on to the next node in the sequence since by this point in the algorithm all the updates to $TG(P)$ required by the current ITG node are complete. Line 28 is the companion to line 5, since it adds the time required to switch context from the process to the dispatcher, after the *proc_exec* exit code is

finished to the last task in the group representing the process, T_{tc} at this point in the algorithm.

The *build_task_group1* algorithm fills in the WCET and resource use of each task, and specifies the set of precedence constraints defining the task group structure. There are a few other task attributes, discussed in Section 4.2.1.3 and illustrated in Figure 4.14, that deserve mention. As already discussed, the system currently treats all tasks as non-preemptive, so every task has this attribute. The only other attributes are the task's scheduling specification. The treatment of these attributes is the domain the of the scheduler, and so they are left unspecified during compilation.

Handling Synchronous Communication(7.2)

Section 7.1 discussed how the *build_task_group1* algorithm is used to build the task group representing the process using the process's ITG as input. That algorithm was, however, limited to processes containing no synchronous communication suspension points. Figure C.2 illustrates the *build_task_group2* algorithm which includes the extensions required for synchronous communication on lines 22-1 through 22-9. These extensions check each suspension point to see if it represents one or more receive actions at line 22-1. If *vc_list* is non-null, then it contains the list of virtual circuits from which the *sync_receive* commands represented by the current suspension point expect to receive messages.

Lines 22-2 and 22-3 add the precedence relations ensuring the receiving sides enter their *sync_receive* calls before the senders exit their *sync_send* calls. Note, however, that we do not yet know to what task in the other process each relation should apply since the algorithm is executed by *spr-ld* while linking a process executable. These are *unresolved* precedence relations which can be resolved only when the task groups

```
build_task_group2 (itg_t *proc_exec)
1. Find the process name: P
2. Create task group TG(P)
3 tc = 1
4. Tasks(P) = {T_1}
5. WCET(T_1) = SWITCH_TO_TIME
6. PR(P) = (Begin, T_1, 0)
7. node = proc_exec → next;
8. while (node != proc_exec → end) {
9.         switch (node → type) {
10.        case TIME_NODE:
11.             WCET(T_tc)+ = WCET(node)
12.             break;
13.        case SUSPENSION_NODE:
14.             WCET(T_tc)+ = Entry_time(node)
15.             Tasks(P) = Tasks(P) ⋃ T_tc+1
16.             WCET(T_tc+1) = Exit_time(node)
17.             PR(P) = PR(P) ⋃ (T_tc, T_tc+1, delay(node))
18.             for (∀(res, mode, t) ∈ Using) {
19.                  RU(T_tc) = RU(T_tc) ⋃ (res, mode, 0, t)
20.             }
21.             Using = Using ⋃ Res_enter(node)
22.             Using = Using \ Res_leave(node)
22-1.           if (vc_list = Recv_set(node)) {
22-2.                PR(P) = PR(P) ⋃
22-3.                {(T_tc, UR_REVC(vc), 0) | vc ∈ vc_list}
22-4.           }
22-5.           if (vc_list = Send_set(node)) {
22-6.                (T_tc, T_tc+1, delay(node)) =
22-7.                          (T_tc, T_tc+1, (CD delay(node) vc_list));
22-8.                PR(P) = PR(P) ⋃
22-9.                {(T_tc, UR_SEND(vc), (CD vc)) | vc ∈ vc_list}
22-10.          }
23.             tc++;
24.             break;
25.        }
26.        node = node → next;
27. }
28. WCET(T_tc) = SWITCH_FROM_TIME
end build_task_group2
```

Figure C.2. Task Group Construction Algorithm - Version 2

of both the sending and receiving processes are available. Until then $UR\_RECV(vc)$ notes that each relation is associated with a receive operation on the listed virtual circuit port.

Lines 22-5 through 22-10 handle suspension points which represent send operations. Lines 22-6 and 22-7 add the communication delay associated with the suspension point to the precedence relation between the current and next tasks which was created by line 17 of the algorithm. Note that the communication delay specification lists the original absolute delay as well as the list of ports upon which this suspension represents send operations. The PAP will use the maximum delay from all the sources listed for the run-time data structures when the communication delays are known. This ensures that the process will not return from the *synch_send* statement until enough time has passed to ensure that any of the messages it might have sent along a particular execution path is ready to be received by the receiving process. Lines 22-8 and 22-9 add the set of unresolved precedence relations that ensure the receiving processes cannot return from its *sync_receive* statement until the message is ready to be received. These precedence relations note that they are associated with a send operation on the listed ports.

When the executables, and thus the task groups, for the processes exchanging synchronous messages have been produced, it becomes possible to resolved the unresolved precedence relations associated with synchronous communication. The resolution method assumes that virtual circuit names are unique and their use is limited to a single sender and receiver. Extension of the method analysis and resolution method to cover multiple senders and/or receivers may be possible, but is not considered here. Also, such extensions must be considered in relation to the use of process groups to describe computations. It is not clear if more than point to point communication

is required to describe communication within a group implementing a computation. Even if multiple senders and receivers are desired, it is not clear how its use can be reconciled with the need to build a task group describing the worst case behaviors of the process group.

```
resolve_sync_comm (set of communicating processes
                                CP = {P1  P2 ...})
1. ∀Pi ∈ CP,  SDLi = SDL_section(Pi);
2. ∀Pi ∈ CP,  URi = Unresolved_pc(SDLi);
3. CURR = ⋃i first(URi);
4. while (CURR ≠ {}) {
5.        if (∃(ur1, ur2) ∈ CURR are complementary) {
6.              ur1 → pc → proc = ur2 → proc;
7.              ur1 → pc → task = ur2 → task;
8.              ur2 → pc → proc = ur1 → proc;
9.              ur2 → pc → task = ur1 → task;
10.             CURR = CURR − {ur1  ur2}
11.             CURR = CURR ∪ {ur1 → next  ur2 → next}
12.       } else {
13.             ERROR: contradictory communication
14.       }
15. }
16. ∀Pi ∈ CP Write SDL_section(Pi)
end resolve_sync_comm
```

Figure C.3. Resolution of Unresolved Precedence Relations

Figure C.3 presents the *resolve_sync_comm* algorithm used by the *spr-sync-comm* command to find and resolve unresolved precedence relations associated with synchronous communication among a set of processes. The algorithm is described for a set of executables which communicate only within the group. The algorithm begins at line 1 by reading the SDL sections of the processes' executable files. When the SDL information has been read, line 2 scans the successor lists of the task group descriptions of the processes, and constructs the *ordered* set UR_i containing the unresolved

precedence relations of each process *in the order that they appear within the successor lists.* We assume that the elements of each set are kept in order on a linked list using a *next* pointer.

The ordering is important since the algorithm resolves *pairs* of unresolved precedence relations, and the order in which they are paired has crucial implications for the semantics of the application. The elements of UR_i are of the form $(pc, type, port, proc, task)$, where *pc* is a pointer to the SDL successor list structure with the unresolved process and task name entries, *type* indicates if the unresolved reference arises from a send or receive operation, *port* names the virtual circuit port involved, and *proc* and *task* identify the task following the suspension point. This is the task to which the unresolved precedence relation paired with this one will be directed.

Line 3 then initializes the *current set CURR*, which is the set of unresolved precedence relations currently under consideration, with the first element from the unresolved set for each process. The loop in lines 4 through 15 then iterates until all unresolved relations have been resolved, or until a contradiction is detected. During each iteration the algorithm searches for a complementary pair of unresolved precedence relations. A pair of precedence relations are complementary if they use the same virtual circuit, with one sending and the other receiving. If such a pair exists, they are resolved by lines 6 through 9 which fill in the missing process and task names in each unresolved successor list element from the information supplied by the complementary relation, as discussed in Section 7.2.

Line 10 then removes the resolved relations from the *CURR* set, and adds the successors of each from the unresolved sets associated with each process. The next iteration of the loop at line 4 checks to see that there are still unresolved references. If

not, then the algorithm has succeeded, and the successor list describing the structure of the task group representing each process is now complete. Line 16 writes this information into the SDL section of each process's executable file.

Line 13 is executed when the *CURR* set is not empty, but no complementary pair of unresolved precedence relations exists. This occurs if the communication pattern of the processes is contradictory, and no task group structure can be built which correctly represents it.

BIBLIOGRAPHY

[1] Amerasinghe, P. An Interactive Timing Analysis Tool for the SARTOR Environment. Master's thesis, University of Texas at Austin, 1985.

[2] Baker, T. The Use of Ada for Real-Time Systems. *Real-Time Systems Newsletter*, 6(1):3–8, January 1990.

[3] Barbacci, M. and Wing, J. Specifying Functional and Timing Behavior for Real-Time Applications. Technical Report CMU-CS-86-177, Computer Science - Department Carnegie Mellon University, 1986.

[4] Barbacci, M. and Wing, J. Durra: A Task-Level Description Language. In *Proceedings of the International Conference on Parallel Processing*, pages 370–376. ACM, 1987.

[5] Beneviste, A. and Gerard, B. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[6] Bickford, C. A Robotics Application for the Spring Real-Time System. Master's thesis, University of Massachusetts, 1993.

[7] Biyabani, S., Stankovic, J., and Ramamritham, K. The Integration of Deadline and Criticalness in Hard Real-Time Scheduling. In *Proceedings of the Real-Time Systems Workshop*. IEEE, May 1988.

[8] Boussinot, F. and Simon, R. D. The ESTEREL Language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[9] Burleson, W., Ko, J., Niehaus, D., Ramamritham, K., Stankovic, J., Wallace, G., and Weems, C. The Spring Scheduling Co-Processor: A Scheduling Accelerator. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 140–144. IEEE, October 1993.

[10] Burns, A. and Wellings, A. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1989.

[11] Cadence Design Systems Inc. *Verilog-XL Reference Manual*, 1991.

[12] Callison, H. R. and Shaw, A. C. Building a Real-Time Kernel: First Steps in Validating a Pure Process/ADT Model. Tr 89-07-04, University of Washington, 1989.

[13] Chen, M. Timing Analysis Language - TAL Programmer's Manual. Technical report, University of Texas at Austin, 1985.

[14] Cogswell, B. and Segall, Z. MACS: A Predictable Architecture for Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 296–305. IEEE, December 1991.

[15] Damm, A., Reisinger, J., Schwabl, W., and Kopetz, H. The Real-Time Operating System of MARS. *Operating Systems Review*, 23(3):141–157, July 1989.

[16] Faustini, A. and Lewis, E. Toward a Real-Time Data Flow Language. *IEEE Software*, 3(1):29–35, January 1986.

[17] Furht, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., and McRoberts, M. *Real-Time UNIX Systems - Design and Application Guide.* Kluwer Academic Publishers, 1991.

[18] Garrett, W., Bianchini, R., Kontothanassis, L., McCallum, A., Thomas, J., Wisniewski, R., and Scott, M. Dynamic Sharing and Backward Compatibility on 64-bit Machines. Technical report, University of Rochester, 1992.

[19] Gehani, N. and Ramamritham, K. Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems. *Real-Time Systems Journal*, 3(4):377–406, December 1991.

[20] Gene, E. The Spring Simulation Testbed - V2
User's Guide. Technical report, Spring Project Documentation, 1990.

[21] Gopinath, P., Bihari, T., and Gupta, R. Compiler Support for Object Oriented Real-Time Software. *IEEE Software*, 9(5):45–50, September 1992.

[22] Gopinath, P. and Gupta, R. Compiler Assisted Adaptive Scheduling in Real-Time Systems. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pages 62–69. IEEE, May 1990.

[23] Gopinath, P. and Schwan, K. CHAOS: Why One Cannot Have Only an Operating System for Real-Time Applications. *Operating Systems Review*, 23(3):106–125, July 1989.

[24] Grimshaw, A. S., Silberman, A., and Liu, J. W. S. Real-Time Mentat Programming Language and Architecture. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pages 82–87. IEEE, May 1990.

[25] Halang, W. and Henn, R. Additional PEARL Language Structures for the Implementation of Reliable and Inherently Safe Real-Time Systems. In *Proceedings of the International Federation for Real-Time Control*, pages 35–42. IFAC, 1988.

[26] Halang, W. and Stoyenko, A. Comparative Evaluation of High-Level Real-Time Programming Languages. *Real-Time Systems Journal*, 2(3), 1990.

[27] Halang, W. and Stoyenko, A. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, 1991.

[28] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[29] Harmon, M., Baker, T., and Whalley, D. A Retargetable Technique for Predicting Execution Time. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 68–77. IEEE, December 1992.

[30] Holmes, V. and Harris, D. A Designer's Perspective of the Hawk Multiprocessor Operating System. *Operating Systems Review*, 23(3):158–172, July 1989.

[31] Hong, S. and Gerber, R. Compiling Real-Time Programs into Schedulable Code. In *Proceedings of the Conference on Programming Language design and Implementation*, pages 166–176. ACM, June 1993.

[32] Intel, Inc. *i486 Microporcessor Programmer's Reference Manual*, 1990.

[33] Ishikawa, Y., Tokuda, H., and Mercer, C. Object Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proceedings of OOPSLA/ECOOP*. ACM, October 1990.

[34] Kandlur, D., Kiskis, D., and Shin, K. HARTOS: A Distributed Real-Time Operating System. *Operating Systems Review*, 23(3):72–89, July 1989.

[35] Katcher, D., Arakawa, H., and Strosnider, J. Bridging the Gap Between Scheduling Theory and Practice. In *Proceedings of the Workshop on Architectural Supports for Real-Time Systems.* IEEE, December 1991.

[36] Kenney, K. and Lin, K. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5):70–78, May 1991.

[37] Kirk, D. SMART(Strategic Memory Allocation for Real-Time) Cache Design. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, December 1989.

[38] Kirk, D. *Predictable Cache Design for Real-Time Systems.* PhD thesis, Carnegie Mellon University, 1990.

[39] Kirk, D. and Strosnider, J. SMART(Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, December 1990.

[40] Kligerman, E. and Stoyenko, A. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, September 1986.

[41] Kuan, C. Spring Target Node Servers. Technical report, Spring Project Documentation, 1990.

[42] Lavoie, P. S. Tool to Analyze Timing on 68020 Processor. Master's thesis, University of Massachusetts, 1991.

[43] Lee, E. and Messerschmitt, D. Static Scheduling of Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987.

[44] Leinbaugh, D. and Yamini, M. Guaranteed Response Times in a Distributed Hard-Real-Time Environment. *IEEE Transactions on Software Engineering*, 12(12):1139–1144, December 1986.

[45] Levi, S., Tripathi, S., Carso, S., and Agrawala, A. The MARUTI Hard Real-Time Operating System. *Operating Systems Review*, 23(3):90–105, July 1989.

[46] Liu, C. and Layland, J. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *JACM*, pages 46–61, February 1973.

[47] Liu, J., Lin, K., Shih, W., Yu, A., Chung, J., and Zhao, W. Algorithms for Scheduling Imprecise Calculations. *IEEE Computer*, 24(5):58–68, May 1991.

[48] Magee, J., Kramer, J., and Sloman, M. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):1305–1320, June 1989.

[49] Marlin, C., Zhao, W., Doherty, G., and Bohonis, A. GARTL: A Real-Time Programming Language Based on Multi-Version Computation. In *Proc. International Conference On Computer Languages.* , 1990.

[50] Mercer, C. and Tokuda, H. The ARTS Real-Time Object Model. In *IEEE Real-Time Systems Symposium.* IEEE, December 1990.

[51] Mok, A. K. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proceedings of the Sixth IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80. IEEE, 1989.

[52] Molesky, L., Ramamritham, K., Shen, C., Stankovic, J., and Zlokapa, G. Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pages 20–26. IEEE, May 1990.

[53] Molesky, L. D., Shen, C., and Zlokapa, G. Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems. *Real-Time Systems*, 2(3), September 1990.

[54] Motorola, Inc. *MCOR Unit68020 32-bit Microprocessor User's Manual*, 1986.

[55] Motorola, Inc. *68030 32-bit Microporcessor User's Manual*, 1990.

[56] Motorola, Inc. *88110 RISC Microporcessor User's Manual*, 1992.

[57] Niehaus, D. Program Representation and Translation for Predictable Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 53–63. IEEE, December 1991.

[58] Niehaus, D., J.Stankovic, and Ramamritham, K. Spring System Description Language. Technical Report 93-01, Computer Science Department, University of Massachusetts, 1993.

[59] Niehaus, D., Nahum, E., and Stankovic, J. A. Predictable Real-Time Caching in the Spring System. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87. IEEE, May 1991.

[60] Niehaus, D., Ramamritham, K., Stankovic, J., Wallace, G., Weems, C., Burleson, W., and Ko, J. The Spring Scheduling Co-Processor: Design, Use, and Performance. In *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, pages 106–111. IEEE, December 1993.

[61] Niehaus, D., Kuan, C.-H., and Mao, D. Address Space and Resource Management in the Spring System. Technical report, Spring Project - in preparation, 1992.

[62] Nirkhe, V. and Pugh, W. A Partial Evaluator for the Maruti Hard Real-Time System. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, December 1991.

[63] Nirkhe, V., Tripathi, S., and Agrawala, A. Language Support for the Maruti Real-Time System. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, December 1990.

[64] Ostroff, J. A Verifier for Real-Time Properties. *Real-Time Systems Journal*, 4(1):5–36, March 1992.

[65] Park, C. and Shaw, A. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[66] Performance Semiconductor. *PR4000 Microporcessor User's Manual*, 1991.

[67] Pospischil, G., Puschner, P., Vrchoticky, A., and Zainlinger, R. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, 9(5):35–44, September 1992.

[68] Puschner, P. and Koza, C. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems Journal*, 1(2), 1990.

[69] Rajkumar, R., Sha, L., and Lehockzy, L. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, 1988.

[70] Ramamritham, K., Stankovic, J. A., and Shiah, P.-F. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[71] Ramamritham, K., Stankovic, J. A., and Zhao, W. Meta-Level Control in Distributed Real-Time Systems. In *Proceedings of the Conference on Distributed Computing Systems.* ACM, September 1987.

[72] Ready Systems. *VRTX 32/68020 User's Guide.*

[73] Schwan, K., Gheith, A., and Zhou, H. From CHAOS-base to CHAOS-arc: A Family of Real-time Kernels. In *IEEE Real-Time Systems Symposium*, pages 82–91. IEEE, December 1990.

[74] Sha, L. and Goodenough, J. Real-Time Scheduling Theory and ADA. Cmu/sei-88-tr-33, CMU, November 1988.

[75] Shaw, A. C. Reasoning About Time in Higher-Level Language Software. *IEEE Trans. on Software Engineering*, 15(7):875–889, July 1989.

[76] Shen, C. *An Integrated Approach to Dynamic Task and Resource Management in Multiprocessor Real-Time Systems.* PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Mass., September 1992.

[77] Shen, C., Ramamritham, K., and Stankovic, J. A. Resource Reclaiming in Multiprocessor Real-Time Systems . to appear Transactions on Parallel and Distributed Systems, 1993.

[78] Shin, K. HARTS: A Distributed Real-Time Architecture. In *Proceedings of the 3rd ONR Workshop on the Foundations of Real-Time Computing*, pages 225–254. ONR, October 1990.

[79] Smith, J. and Plezkun, A. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5), 1988.

[80] Sprunt, B., Sha, L., and Lehoczky, J. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[81] Stallman, R. Using and Porting GNU CC. Technical report, Free Software Foundation, May 1992.

[82] Stankovic, J. On the Reflective Nature of the Spring Kernel. In *Proceedings of Process Control Systems Conference.* Springer erlag, 1991.

[83] Stankovic, J. Reflective Real-Time Systems. Technical Report 93-56, University of Massachusetts, 1993.

[84] Stankovic, J. A. Misconceptions About Real-Time Computing. *IEEE Computer*, 21(10), October 1988.

[85] Stankovic, J. A. Real-Time Operating Systems: What's Wrong With Today's Systems and Research. *Real-Time Systems Newsletter*, 8(1/2):1–9, 1992.

[86] Stankovic, J. A., Niehaus, D., and Ramamritham, K. SpringNet: An Architecture for High Performance, Predictable, and Distributed Real-time Computing. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 176–180, May 1993.

[87] Stankovic, J. A. and Ramamritham, K. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

[88] Steusloff, H. Advanced Real-Time Languages for Distributed Industrial Process Control. *IEEE Computer*, pages 37–46, February 1984.

[89] Stoyenko, A. A Schedulability Analyzer for Real-Time Euclid. In *IEEE Real-Time Systems Symposium*. IEEE, December 1987.

[90] SYSTRAN Corporation, Dayton, Ohio. *Scramnet Network Reference Manual*, 1991.

[91] Tindell, K., Burns, A., and Wellings, A. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *Real-Time Systems Journal*, 4(2):145–166, June 1992.

[92] Tokuda, H., Nakajima, T., and Rao, P. Real-Time Mach: Towards a Predictable Real-Time System. In *USENIX Mach Workshop*. USENIX, 1990.

[93] van Tilborg, A. and Koob, G., editors. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.

[94] Various. CARTS Demo: A Time Driven Delivery System. Working document, 1993.

[95] Wang, F. *Issues Related to Dynamic Scheduling in Real-Time Systems*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Mass., sep 1993.

[96] Wolfe, V., Davidson, S., and Lee, I. RTC: Language Support for Real-Time Concurrency. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 43–52. IEEE, December 1991.

[97] Zhao, W., Ramamritham, K., and Stankovic, J. A. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transactions on Computers*, pages 949–960, August 1987.

[98] Zlokapa, G. *Real-Time Systems: Well Timed Scheduling and Scheduling with Precedence Constraints*. PhD thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Mass., feb 1993.