Program Representation and Translation for Predictable Real-Time Systems*

Douglas Niehaus
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

As increasingly complex real-time systems are created, program representations must improve to meet the demands of more sophisticated programs. In real-time systems, *when* an answer is produced is part of its correctness. Proper run-time management in a real-time system thus depends on accurately *predicting* program execution behavior, which must be based on information either specified in the program source or derived from it during translation. This paper discusses our method for deriving behavioral predictions while translating between the programming and run-time representations used by the Spring system. A graph representation of the program is derived from the intermediate representation the compiler uses to emit code. This graph is then reduced and analyzed to make behavioral predictions. We present the basic translation method, and give examples of how it translates programming language constructs for critical sections and synchronous communication.

1 Introduction

As real-time systems become more complex, and are created for ever more dynamic environments, the methods used to represent programs will have to increase in sophistication to meet the new demands. Examples of such dynamic and demanding application environments include command-and-control systems, space shuttle and aircraft avionics, automated factories, process control systems, nuclear power plants, and sophisticated robots. In these 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 of the real-time system must thus include control of when computations are completed, which depends on the *predictability* of program behavior.

Predicting the behavior of programs with such time-dependent semantics is a challenging problem; and representing the predicted behavior will require descriptions that are substantially more complex than those for programs which are not time dependent. Predictions about process behavior depend on a number of factors, and unsatisfactory treatment of any of them renders the predictions less accurate or invalid. Coordination of the designs for different layers of a real-time system is required to ensure predictable behavior for the system as a whole. These issues cover the programming language, translation, operating system and hardware levels.

The programming representation is used by the developer to describe the properties of real-time computations, which include: timing constraints, resource requirements, importance levels, and a number of other aspects of the computation's properties and requirements. Typically, the familiar non-real-time process model is enhanced with representations of the temporal and system configuration information. The many real-time languages that are objects of current research differ from one another mostly in the details of how they express the temporal and configuration attributes of each process. These differences are partly stylistic, and partly a reflection of different assumptions about the properties of the run-time system.

Most real-time systems use a process representation at run-time as well as for programming, and schedule the processes using a priority driven scheduler. However, the Spring system scheduler is significantly different because it dynamically builds explicit execution plans, rather than being priority driven. To do this, it assumes that each process is represented as a group of precedence related non-blocking tasks with known execution time and resource use behaviors.

In this paper we concentrate on the translation between the programming and the run-time representations. The translation requires us to predict process behavior in a number of ways, and to represent these predictions in the task based form the Spring scheduler expects. The behaviors of the process we must predict include: its execution time, its use of resources shared with other processes, and its communication with other processes. The differences between the properties of the process based and task based representations require that we develop a translation method differing significantly from other efforts. This paper presents the basic translation method, and gives examples of how it treats programming language

*This work is part of the Spring Project at the University of Massachusetts funded in part by the Office of Naval research under contract N00014-85-K-0398 and by the National Science Foundation under grants DCR-8500332 and CDA-8922572.

constructs used to specify critical sections and synchronous communication. We address other aspects of constructing a predictable real-time system elsewhere [11, 9].

One of our goals is to establish a source language, Spring-C, that can serve as a target for many of the existing languages. The details of the language we have developed for describing the process properties and system configuration, and our modifications to C syntax are described in [10]. The purpose of Spring-C is to provide many existing real-time languages with access to a run-time system that can predictably produce the behavior they specify. The translation method presented here is one step toward this goal.

A survey and comparison of several existing real-time languages is given in [2]. In general, these efforts consider the source language structures required to describe real-time computations, while paying little attention to *how* the behavior can be predictably produced by the run-time system. They assume that the worst case execution time (WCET) of a process can be calculated, and that the system is capable of producing whatever behavior is specified. Some investigators into real-time languages have suggested the use of multiple implementations of an algorithm, one of which is selected at run-time according to execution time and other scheduling constraints [4, 8]. Klingerman and Stoyenko use a model where processes contend for access to shared resources, and so take worst case blocking time into account in the execution time [5]. In more recent work, Stoyenko and Marlowe use a graph representation to analyze program behavior, and control transformations [19]. However, their method assumes a system using a process based run-time representation, and so has significant differences from the work described here. Nirkhe describes program translation of an object oriented language with temporal extensions that has some similarities to the translation described here, but does not address how the behavior of the program can be accurately predicted, and then reliably produced by the underlying system [12]. This is also true of the object oriented language described in [3].

The WCET is an important aspect of real-time process behavior, though it is by no means the only aspect of behavior that must be predicted. At the simplest level, any method of calculating WCET for the code of a process, or a section of it, will require summation of the execution times for individual machine instructions specified by the code emitted by the compiler. The difficult aspects of the problem are how to organize the summation, and how to phrase the description of process behavior. Park and Shaw address the derivation of WCET using source level timing schema and assume a process based run-time model [13]. This method has the advantage of being partly independent of the target architecture, but it has difficulty predicting the actual code emitted by the compiler due to the effects of optimizations performed by the compiler that cross the boundaries of their source-level schemata. Since they work with *approximations* to the actual emitted code, they also would have difficulty accurately accounting for the effects of hardware features such as caches. Amerasinghe derives WCET from the assembler code produced by compilation [1]. An extension of this work for the Spring project was recently completed [6]. Since this method works at the assembly language level, it can more easily consider how specific hardware features will affect execution time. However, since it works solely with the assembler output of the compiler, it is very difficult to perform any transformations on the program.

The translation method described in this paper compromises between the source and assembler extremes by working with the *compiler's* intermediate representation of the code. This permits us to work with the machine instructions actually emitted by the compiler, while preserving the ability to transform portions of the program. Program transformations enable us adjust the number of tasks in the process's run-time representation. This lets us balance the benefit of a finer grain description of a process's behavior using more tasks, against its cost in terms of increased scheduling load resulting from the larger number of tasks. In addition to deriving the WCET of tasks in the group representing the process at run-time, our translation method also derives task resource requirements, and the precedence relations between tasks arising from synchronous communication.

Section 2 of this paper will discuss two different scheduling paradigms and how their properties and assumptions require different run-time representations. Section 3 describes the translation method we have developed to bridge the gap between the programming representation using processes, and the run-time representation using a group of well defined, non-preemptable tasks to represent a process. Section 4 will then describe the current status of our work and how we plan to develop it in the future.

2 Scheduling Paradigm Assumptions

A distinction must be drawn between the great body of scheduling research that exists, and the scheduling paradigms used at run-time in real-time systems. We are concerned with the run-time scheduling paradigms and how their assumptions affect the representation of a process. One of the significant approaches to run-time scheduling for real-time systems is the explicit construction of task execution plans. This research generally assumes that non-blocking tasks with known WCETs and resource requirements are the entities being scheduled [15]. This explicit planning paradigm *avoids* resource contention by taking the tasks' resource use into account when constructing the execution plan. In this context a *resource* is used to represent any element in the system for which concurrent access is restricted. The access restriction creates a constraint that the scheduler must take into account when constructing the execution plan. Compare this with a *run-time* representation using processes that run freely, contending for resources as they require them, and blocking until they are available. The explicit planning approach must consider tasks that represent a *portion* of the process's execution to achieve adequate levels of efficiency

and schedulability. The run-time representation of a process under the explicit scheduling paradigm will thus be a *group* of non-blocking tasks each representing a portion of a process instance's execution. Precedence constraints among the tasks in a group ensure that the scheduler considers the process's execution episodes in the correct order. This is one reason why a non-trivial translation between the programming and run-time representations is required for a system using explicit plan scheduling. Current research is addressing the problem of explicit plan construction for groups of tasks with precedence constraints [20].

In contrast, systems using the rate monotonic approach to run-time scheduling use preemptable periodic processes for both the programming and run-time models. The basic rate monotonic approach was described by Liu and Layland [7] assuming no resource contention. Under this scheduling paradigm the WCETs of the processes are assumed to be known, and processes are assigned priorities proportional to the frequency with which they must execute. The run-time scheduling decision is simple since the system always runs the highest priority ready task. Schedulability analysis can determine if all processes will meet their deadlines.

A number of extensions to the basic paradigm have been developed to handle blocking due to resource contention [14] and aperiodic process execution in response to environmental events [16]. However, the rate monotonic approach has significant limitations, as well. First, the rate monotonic approach has not, as yet, been extended to handle groups of processes with precedence constraints, or to consider processes which engage in synchronous communication. The handling of aperiodic events is limited to statically allocated execution time for periodic servers, which makes it more difficult for the system to handle dynamic environments. Also, the blocking time for resource contention can become very large since schedulability analysis must always assume the worst possible blocking behavior.

Both scheduling paradigms depend on the accurate prediction of the process's behavior including WCET and resource use, but they approach the problem quite differently. The rate monotonic approach uses the same representation for programming and at run-time, which is an initially attractive feature. The explicit planning paradigm uses the task group representation of a process at run-time creating a gap that must be bridged with a translation step. This may at first appear as a disadvantage, but the explicit planning paradigm has several attractive features which cause us to select it for the Spring system, and to accept the problem of translating between the programming and run-time representations.

First, the handling of aperiodic process execution is handled much more directly, by inclusion in the execution plan, rather than through servicing by a periodic server. The explicit planning approach gives more a detailed view and finer control of the system state under overload conditions. Resource contention is explicitly represented, and so the blocking time need not be included in the WCET, but is treated separately. This is useful for building a system where the real-time processes are divided into the *critical*, which absolutely *must* meet their deadlines and have resources statically allocated, and *essential* which are necessary to the operation of the system, but will not cause a catastrophe if they are not finished on time. We believe that real-time systems of the future will have a large number of essential processes, and that the construction of explicit execution plans will enable the system to better handle the tradeoff among conflicting essential tasks. The details of these and other arguments supporting this approach to scheduling are given in [18].

3 Program Translation

Programs are translated from the representation used by the developer, into the representation used by the system at run-time. We use the conventional programming model based on processes, which execute until they finish, or until they are suspended for one of several reasons. A *task* is a non-blocking episode of execution with known WCET and resource use. A *task group* is a set of tasks among which precedence relations hold, which express constraints on the order in which the tasks may be executed. We have developed a method for translating from the process to the task group representation. Each task in the group represents an episode of execution for the process, and the process execution is certain to be complete when each task in the group has been executed. The task groups are thus used by the scheduler to build an execution plan for the processes. To perform the translation, we must consider all the places in the code where the process can suspend its execution.

In conventional systems, processes suspend for four major reasons: preemption, access to a critical section, explicit delay statements, and synchronous communication. Since we will only consider nonpreemptive task execution at this time, preemption is not an issue. This leaves critical sections, delay statements, and synchronous communication calls; which we explicitly represent during the translation. Under the explicit plan scheduling paradigm, these points of potential process suspension create task boundaries. We call them *scheduling points* since they delimit the episodes of execution for which the scheduler constructs a plan.

We translate from the process to task representation by predicting the process's worst case resource requirements and duration of its execution episodes. An *execution episode* begins and ends with process suspension. The execution episodes, as predicted at compile time, are then presented to the scheduler as "tasks" for which it constructs an execution plan. The translation method has three phases:

- time graph construction,
- subgraph reduction, and
- task group construction.

Time graph construction builds a representation of the program's control structure which preserves its

temporal behavior while discarding its semantics. Calculating the worst case behavior of the process requires us to consider every possible path through the process code. Subgraph reduction replaces sections of the time graph with single nodes giving the worst case time through the subgraph being replaced. This is a form of preprocessing since the single nodes are equivalent to the original subgraph, for WCET calculation purposes, but using them reduces the number of possible paths through the time graph. Task group construction uses the fully reduced form of the time graph to calculate the worst case behavior of the process by enumerating all possible paths through the reduced graph, and finding the worst possible behavior for each execution episode of the process.

The rest of this section discusses each phase of the translation method, describes scheduling points in greater detail, and gives examples of some program transformations that can be done.

3.1 Time Graph Construction

The *time graph* (TG) is a representation of the control structure of a process which abstracts its temporal and resource use behavior, discarding everything else. The TG is constructed from the intermediate representation of a process used by the compiler for code emission, called "register transfer language" (RTL) [17]. The TG thus reflects the temporal behavior of the code which is *actually emitted*. Each node in the intermediate representation is represented in the TG by a node giving the WCET of the machine instructions it represents, as calculated by the tool described in [6]. The original time graph is thus as complex as the program's intermediate representation at code emission time. Figure 1 illustrates this by showing a simple conditional statement and the corresponding RTL graph and TG.

The conditional statement is converted into the corresponding RTL control flow graph by the compiler. We produce an enhanced RTL graph, with nodes added which preserve the original block structure of the source. In the figure these are the grey nodes marking the beginning and end of the conditional statement. The dotted arrow connecting them illustrates that the two structural nodes refer to one another. Similar structural nodes would be emitted for loops, switch statements, and to bound the procedure itself. Structural nodes are required to provide context for some of the reductions related to loops, switches, and procedures.

For clarity we have illustrated the RTL graph with a single node for each C statement, but there will generally be a single TG node for each basic block in the source procedure. This representation of the conditional statement preserves the full range of its execution behavior, while eliminating its semantics. In this case the only possible behaviors are the paths following the true and false branches, but this representation permits us to calculate the time for each. Note that code *will* eventually be emitted for the procedure, but we do execution behavior analysis and task group construction first.

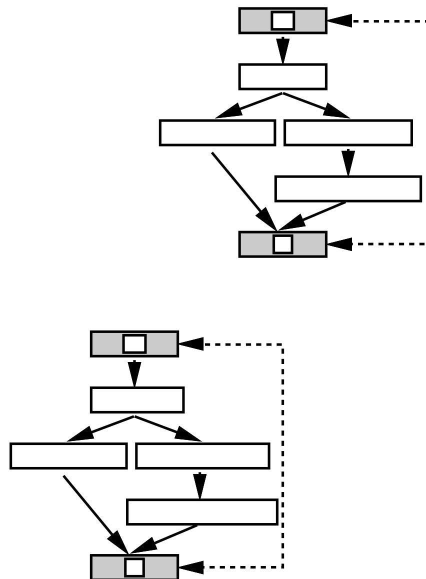

Figure 1: Time Graph Construction

The RTL nodes in the figure are far simpler than those actually used. For example, consider the RTL and time graph nodes representing the conditional test "$a > b$". The actual RTL statements and the equivalent assembler instructions are:

```
           RTL                            Assembler
(insn 24 4 6 (set (reg:SI 0)              movel _a,d0
   (mem:SI (symbol_ref:SI ("a")))) 
     -1 (nil) (nil))
(insn:QI 6 24 7 (set (cc0)                cmpl _b,d0
   (compare (reg:SI 0)
     (mem:SI (symbol_ref:SI ("b")))))
       11 (nil) (nil))
(jump_insn 7 6 8 (set (pc)                jle L2
   (if_then_else (gt (cc0)
     (const_int 0)) (pc)
       (label_ref 13)))
         227 (nil) (nil))
```

The triplet of integers in each statement give the statement number, its predecessor, and successor in the RTL graph, respectively. The execution time for the three assembler instructions would be added together, and this would be the value of the TG node, "$WCET(a > b)$".

Procedure calls in the TG are simply replaced with the TG of the procedure being called. If the TG for the procedure is not currently known, a place holding node can be inserted. TGs can be stored in an intermediate form, pending the analysis of all referenced procedures. The TG for a process is the TG for its main level procedure, where the TGs for all the procedures it calls are properly substituted.

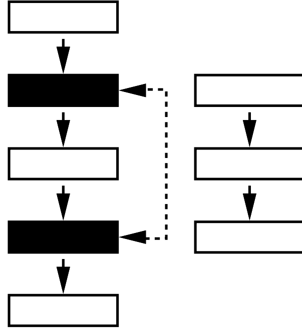

Figure 2: Time Graph Code with Critical Section

3.2 Scheduling Points

Scheduling points mark places in the process's TG and ITG where suspension could occur. This discussion emphasizes the suspension that comes from blocking for access to shared resources and synchronous communication, but a scheduling point will exist wherever suspension can occur, for whatever reason. We could, for instance, *insert* scheduling points to enforce process suspension at specific places. How scheduling points affect code expansion is subject to a number of factors which are discussed in Sections 3.4 and 3.5. In this section we discuss scheduling points arising from critical sections, explicit delay statements, and synchronous communication calls.

A critical section exists, by definition, to control access to a resource that is shared among concurrently executing processes. Under explicit plan scheduling we *avoid* resource conflict by scheduling computations using the same shared resource so they do not run concurrently [15]. Figure 2 illustrates code containing a critical section, the corresponding TG, and the task group that would represent the code. In this example arbitrary C code is represented by the statements "S1;", "S2;", and "S3;". The critical section is denoted by the **with** statement, whose argument gives the critical section name.

One part of translating from the process to the task group representation is to present the code *inside* the critical section as a separate task, so that its use of the resource represented by the critical section can be noted, and the task may be scheduled to avoid conflicts. We *could* declare that the process uses the resource protected by the critical section for its entire WCET. This would enable us to represent the process with a single task, but would also, in general, mean reserving the resource for much longer than the WCET of the code actually contained in the critical section. Reserving the resource for longer than necessary will needlessly constrain concurrency, and so we represent the critical section by a separate task using the critical section's resource. We use the **with** construct rather than simple **P** and **V** semaphore calls to emphasize that this discussion applies *only* to critical sections, and not to other applications of **P** and **V**. On the other hand, in the RTL graph and TG we use the **P** and **V** nodes at the critical section boundaries because in some situations these nodes will be expanded into actual **P** and **V** subroutine calls when code is emitted.

While full analysis can only be done in the context of the process's ITG, it should be fairly easy to see that the **P** and **V** scheduling points are where the system will suspend process execution, marking the boundary between tasks in the run-time representation. In conventional systems, of course, processes may not suspend at the **V** call, but this is clearly required by our task representation to mark the end of the task using the resource. The three tasks will correspond to the three C statement nodes in the ITG, with precedence constraints between them to ensure they are executed in the correct order. It is important to note that a consequence of our method of representing critical sections as tasks using resources is that critical sections *do not nest*. The WCET of a task representing a set of nested critical sections will be equal to the WCET of the whole set, and the resource use of the task will be the union of resources representing the critical sections in the nested set. However, while we must acquire all the resources at the beginning of the task, it is possible to release them before the end of the task, moderating the increase in resource holding time for the nested critical sections.

Synchronous communication calls and explicit delay statements will be translated into single scheduling points. The **delay** statement defines a boundary between two tasks with the first ending at the entry to the **delay** and the second beginning at the exit from it. The statement's argument gives the interval that must elapse between the end of the first task and the beginning of the second in the execution plan constructed by the scheduler. A synchronous communication point will create a task boundary and a set of precedence relations between tasks in the sending and receiving process representations. The details of synchronous communication translation are given in Section 3.5, when sufficient background understanding of the issues has been established.

3.3 Subgraph Reductions

A naive approach to calculating the worst case behavior of the process would examine every possible execution path through the original TG, but this is unnecessary. We *reduce* the TG to minimal size to simplify the calculations required during task group construction. The method is based on a set of rules for reducing sections of the original TG, also called subgraphs, which are of a specific structure to single nodes giving the WCET for the subgraph. The new node is, generally, equivalent to the original subgraph for the purpose of calculating the worst case behavior of the process. Detecting cases where this is *not* true requires the use of constraints arising from the semantics of the program, which we do not attempt at this time.

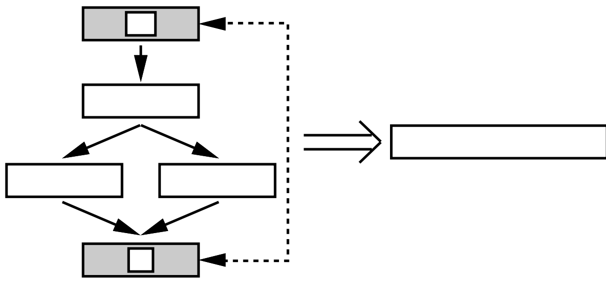

Figure 3: Conditional Subgraph Reduction

Scheduling points cannot be eliminated without discarding information about how the process can block. As subgraph reductions are applied the TG will reach a point beyond which, because of the scheduling points, it cannot be reduced further. This form of the TG is called the *irreducible time graph* (ITG). A TG without scheduling points will reduce to a single node giving the WCET of the process.

The basic idea is illustrated in Figure 3, which shows the reduction of the graph for the conditional statement considered in Figure 1. Note that the basic block in the false branch of the conditional is shown as a single node. The worst case time value for this subgraph is easily understood as the sum of the time to evaluate the condition and the maximum time of the true and false clauses. Similar rules exist for all other programming constructs, but space limitations prevent us from specifying them. The structural nodes are required for those situations where the worst case calculation requires the program context of the subgraph. An example of such a situation is the **break** statement in C. It may appear in either a **switch** statement or a loop, but reducing the subgraph containing it requires us to update information in the **switch** or loop block within which it appears.

The ITG is a condensation of the RTL graph for the program, and so a path through the ITG represents a *set* of paths through the process code. When a subgraph is reduced, a single node replaces two or more nodes. In the case of the conditional in Figure 3, the node giving the WCET represents two paths through the original subgraph. A path through the ITG represents all the paths through the original TG than can be constructed by replacing each single node along the ITG path with any of the paths through the subgraph from which it was produced. Since subgraphs are nested in precisely the way programs are nested, a single path through the ITG can represent a large number of paths through the original TG. This is particularly true when loops containing conditionals are reduced.

The ITG is minimal because it contains no reducible subgraphs. A subgraph is not reducible if it contains a scheduling point. ITG transformations can make further reduction possible, as discussed in Section 3.5. We believe that every process will have a unique ITG, though we do not have room to offer a proof of this. However, it is fairly easy to see since the TG of a process consists of nested loops and conditionals, and the structure of the ITG is not sensitive to the order in which reductions are performed. The complexity of the reduction procedure will also be reasonable, since it is directly analogous to the compilation of the program. A depth first traversal of the TG reducing subgraphs as we find them will visit each node at most twice, indicating that the complexity will be proportional to the number of nodes in the graph. Compare this to the number of times a node would be visited during a depth-first enumeration of all possible execution paths through the original TG.

While we currently employ simple reduction techniques, the subgraph reduction approach is quite flexible, and can be enhanced to consider some of the effects of caching or to create symbolic expressions for WCET. In the caching case, we concentrate on instruction caching, and each block in the TG holds both cached and uncached times. The reductions become more complex, especially for loops, as we must take the possibility of following different paths through the loop body on different iterations into account. The presence of caches will also require us to modify how the executable code for a process is linked, so that we can control the overlap of the cache footprint of subroutines and the code that is calling them. This will enable us to predict a substantially smaller WCET if loops containing subroutine calls can coexist in the cache with the code of the subroutines they call. A preliminary discussion of how we handle the effects of instruction caching is given in [11].

Symbolic expressions are produced by subgraph reductions that construct expressions for the execution time instead of merely accumulating numerical values. This will make it possible to construct expressions for the WCET of a task, which could depend on the number of input data items or on the location of certain data structures in a memory hierarchy with non-uniform memory access times.

3.4 Task Group Construction

The worst case behavior for the process is calculated by enumerating all possible paths through the ITG. As the paths are enumerated, the WCET and resource use is determined for each *execution episode* along that path. An execution episode is defined, in the context of the ITG, as the segment between two scheduling points along any execution path. The predicted behavior for an episode of a given ordinality is the maximum WCET and the union of the resource use for all episodes with that ordinality along any execution path. Since we exhaustively enumerate all possible paths through the ITG, these are the worst case behaviors. The episodes are then presented to the scheduler as tasks, with the WCET and resource use calculated for each episode.

ITGs, and so the processes they describe, can be divided into three major classes: singular, linear, and

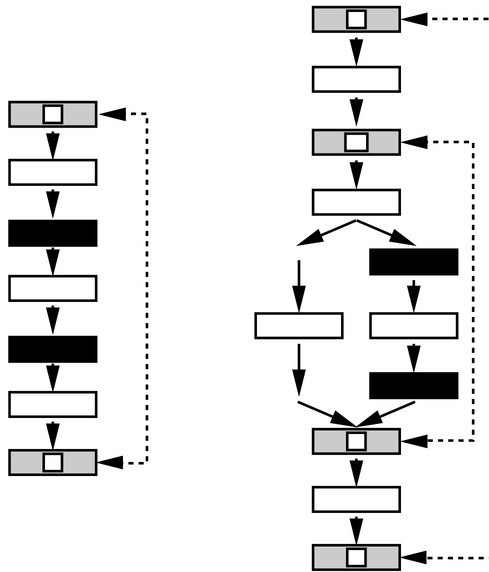

Figure 4: Examples of Irreducible Time Graphs

non-linear. Singular ITGs arise from processes containing no scheduling points, an are thus represented by a single node giving the process's WCET. Linear ITGs arise from processes that only have scheduling points in sections of the code that are visited by all execution paths. Non-linear ITGs (NL-ITG) arise from processes that have scheduling points in sections of the code that are visited by some execution paths, but not others. The number of tasks required to represent a path through the ITG is defined by the number of scheduling points it crosses, since each scheduling point represents a suspension of the process. Examples of linear and non-linear ITGs are given in Figure 4. In each of the ITGs the grey structural nodes mark the beginning and end of the process's main level procedure.

Construction of the task group representation of a process is done by enumerating all possible execution paths through the ITG, and accumulating the worst case resource use and execution time for each execution episode of a given ordinality. The enumeration of all possible paths is easily implemented as a depth-first traversal of the ITG. The ordinality of a given execution episode is defined by the number of scheduling points crossed along the execution path up to the beginning of the episode. We can easily maintain a count of the scheduling points crossed by the current path in the course of a depth-first traversal of the ITG. It is equally simple to accumulate the maximum execution time and union of resource use for each episode of a given ordinality as the depth-first traversal progresses.

It should now be obvious why subgraph reduction is used. The depth of the stack required to support the depth-first traversal of the ITG is determined by the length of the longest path through it, while the number of paths is determined by the number of branching points. Subgraph reduction eliminates unnecessary complexity in the path enumeration step, particularly that resulting from loops, by calculating the WCET for a reducible subgraph once. If the ITG we produce is still complex enough to contain a problematic number of possible execution paths, then it will also produce a task group representation too complex to be scheduled by any realistic scheduler. In such a case, we would provide feedback to the developer about which parts of the program were causing the problem.

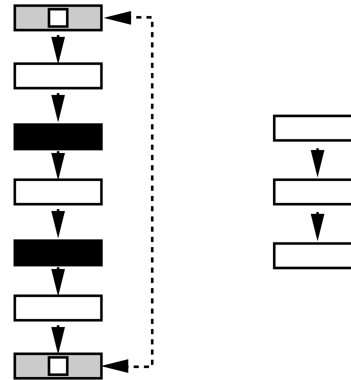

Figure 5: ITG and Corresponding Task Group

We first consider the analysis of ITGs containing scheduling points arising from critical sections, and show the task groups that we construct to represent them. We then discuss the more difficult case of synchronous communication scheduling points and their translation.

In part (a) of Figure 4, the ITG is linear and obviously exhibits a data independent number of execution episodes. In part (b), the scheduling points associated with the critical section lie along a data dependent subpath, and so the number of execution episodes will be data dependent. In (a) the critical section in the original TG was in a portion of the TG which is always visited regardless of the input data. Such TGs will *always* reduce to linear ITGs. In the second case, the conditional statement could not be reduced because one of its branches contained the scheduling points associated with the critical section.

The simplest linear ITG is a single node arising from a process containing no scheduling points. This ITG then maps directly onto a task group with a single task with the same WCET as the process. Non-trivial linear ITGs map onto isomorphic task groups, where scheduling points denote task boundaries, and critical sections specify which resources are used. Each task in the group corresponds to one of the nodes in the ITG, as with the single node case, and the tasks form a group with linear scheduling precedence constraints.

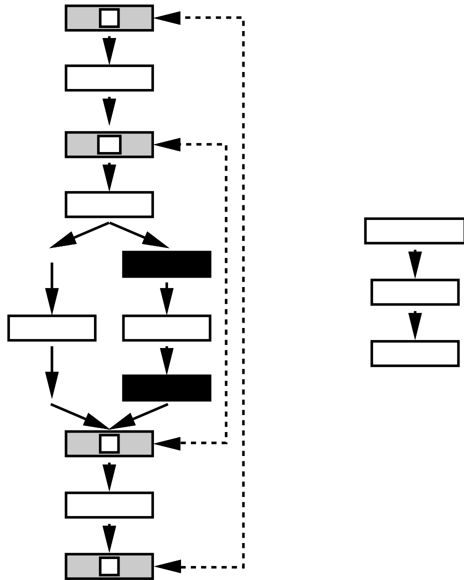

Figure 6: NL-ITG and Corresponding Task Group

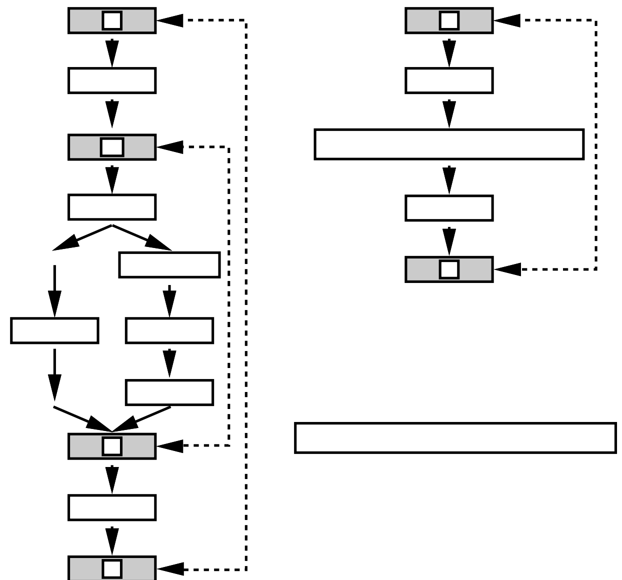

Figure 7: Reduction of a Transformed NL-ITG

Figure 5 illustrates this for the linear ITG from Figure 4. Note that the **P** and **V** scheduling points have been implemented as task boundaries. Exclusive access to the resource is ensured in the execution plan the scheduler will construct by noting that the second task in the group uses resource X.

The creation of a task group representation for a non-linear ITG with two possible execution paths is illustrated in Figure 6. The execution path which follows the *true* branch of the conditional crosses no scheduling points and thus has a single execution episode with duration $(A+B+C+E)$. The execution path which follows the *false* branch of the conditional crosses two scheduling points, and so exhibits three execution episodes with durations $(A+B)$, (D), and (E), respectively. For the task group representation we take the maximum execution time and the union of resource use for every episode of a given ordinality along any path. In this case the first episode, with ordinality 1, is the only one with a representative along more than one path. No resources are used by the first episode along any path, so the worst case time is the only issue.

The first path has the maximum time for the first episode, so this value is used. The second and third episodes exist only along the second execution path, so their times are the maximum values. Resource X is used in the second episode along the second path, so the second task uses resource X. While this is a simple example, it shows how the episode attributes are accumulated.

A more complex situation arises when loops are involved. Note that the only loops that are important in this context are those containing scheduling points. Loops without scheduling points will be reduced in the normal course of subgraph reduction, and will not appear in the ITG. Loops that *do* contain scheduling points will be effectively "unrolled" by the path enumeration, and each iteration will then be represented by a separate task or set of tasks. While it is possible to imagine loops which would pose a processing problem during task construction, these same structures would generate task representations far too complex to schedule effectively. In such cases, feedback to the programmer is appropriate, pointing out the section of the program that is creating the problems, and suggesting a change.

3.5 Time Graph Transformations

The basic approach to constructing a task group representation from the ITG should now be clear, at least for critical sections. However, there are several interesting issues that arise around transformations to the original ITG that have different effects on the task group properties. In particular, one of these transformations is *required* to make the implementation of synchronous communication feasible. While there are several interesting transformations, due to space limitations we will only be able to present two of the most important ones. The first is an extension of the critical section example, and the second is a transformation required for synchronous communication.

The first transformation is of interest because some critical sections should not really be treated as separate tasks; the management of a FIFO queue accessed by several processes is a good example. Concurrent additions or deletions must be avoided, so access to the

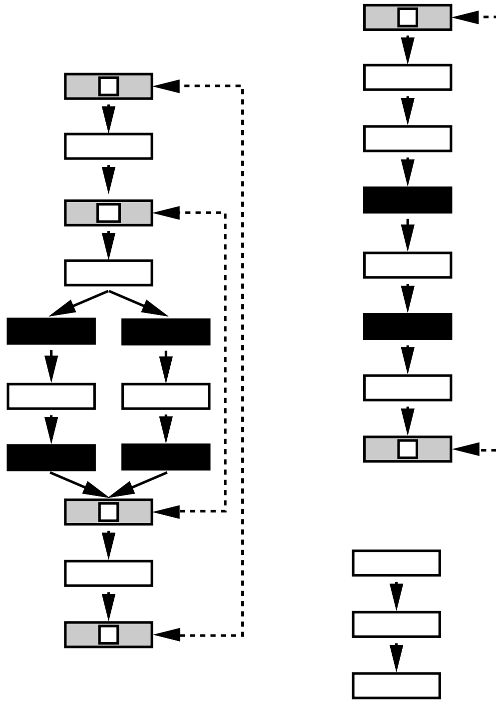

Figure 8: NL-ITG Transformation and Reduction

structure holding the list head and tail pointers is protected by a critical section. However, the time spent by each process to update the pointers is so small, it is difficult to justify the overhead of a separate task. In such a case the critical section scheduling points can be expanded into actual semaphore calls, as long as the worst case blocking time is taken into account. This is easy to calculate when all the uses of the resource in question are known, and their TGs are available. The scheduling points are replaced with nodes giving the worst case execution and blocking time, and the graph is then reduced further.

As illustrated in Figure 7, the **P** and **V** scheduling points of Figure 6 are replaced by nodes representing the execution and blocking time of actual semaphore calls. Changes are made to the RTL graph to insert the **P** and **V** subroutine calls. Note in particular that the node replacing the **P** scheduling point contains a term for the worst case blocking time on resource X, $BT(X)$, calculated using the WCET of the X critical sections in all processes using the resource. The rest of the figure illustrates the reduction and translation of the transformed ITG into a representation using a single task.

The second ITG transformation inserts scheduling points into a non-linear ITG, making corresponding modifications to the RTL graph, to make the number of its execution episodes data independent. This is a necessary condition for the correct implementation of synchronous communication using task boundaries and precedence constraints. However, we will first discuss its application to our critical section example. Figure 8 shows how the transformation inserts simple scheduling points along the conditional path which does not contain the critical section, producing ITG'.

This enables us to reduce the conditional block to a linear subgraph, producing TG'. Note that the **P** scheduling point in TG' records the WCET of the *original* critical section using the notation $P(X, D)$. A further reduction to combine the A and B blocks is not shown. Since there is only a single path through a linear ITG, construction of the task group is simple. One subtlety is that the resource use notation, $Res(X, D)$, for the second task records the duration of the original critical section D. Compare this task representation to that in Figure 6. The execution time of the first task has been reduced from $A+B+C+E$, but the execution time of the task using resource X is now the maximum of C and D. If $D > C$ simply noting that resource X is used would be correct, since D was the original duration of the critical section. However, if $D < C$ we can avoid holding X for longer than necessary by informing the scheduler that X is actually used for a maximum duration of D. This is the reason for the addition to the resource notation. We can now consider how this transformation plays a role in implementing synchronous communication.

There are two significantly different ways to define the semantics of synchronous communication. The first assumes that the synchronization is *simple*; meaning that no application level processing on the receiving end is required to produce a reply. The second assumes a send-accept-reply semantics. We will limit ourselves to the simple semantics for the time being.

A reasonable definition of these semantics would be that the sender and receiver may enter their communication calls in any order, but that the sender may not return until the receiver has entered its call, and the message is available on the receiving side. Under this definition the communicating processes will have task boundaries representing the entry and exit from the **send** and **receive** calls. Each task terminated by the entry to the synchronous call will have a precedence relation with the tasks beginning at the exit from the calls in *both* processes. In addition, the precedence constraints on the sending side have the communication delay associated with them. This is the time that must elapse to ensure the message has arrived at the receiving side. The value is specified by the real-time network service when the scheduler sets up the communication. When the scheduler constructs an execution plan for the process group, it will leave a gap at least this large between the task ending with the entry to the send, and the two tasks beginning with the exit from the communication calls in both processes.

Figure 9 illustrates several important properties of synchronous communication. First, processes which engage in synchronous communication form a *group* which is defined in the program source. The communication relation must hold between specific *calls* in

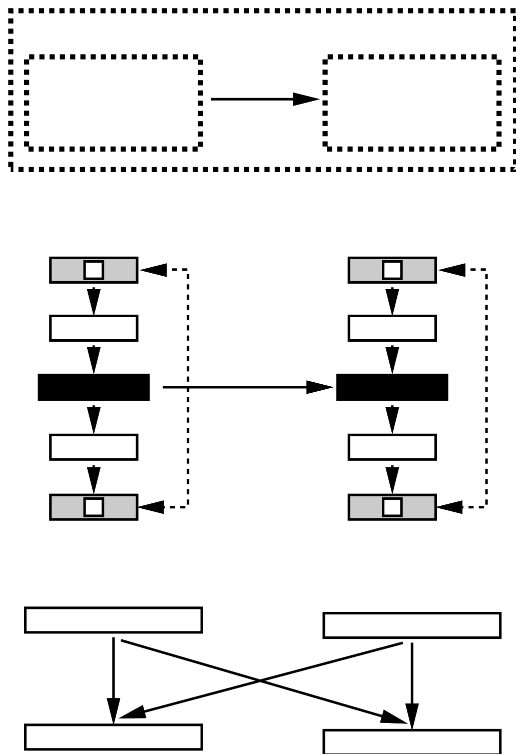

Figure 9: Simple Synchronous Communication

the sender and receiver, to permit proper translation into task representations. The process name and label arguments to the synchronous calls uniquely identify the sending and receiving calls. The communication relation at the process level translates into an equivalent relation between the **send** and **recv** nodes in the TGs. When the communication nodes are used to create the task representation, the precedence constraints across process boundaries are created as illustrated in Figure 9.

The requirement that we be able to pair the synchronous send and receive calls syntactically is a strong one, but we believe it is necessary condition for the construction of a reasonable task group representation which implements the synchronous communication semantics correctly for *all* input data. When the task representation is constructed, precedence constraints across process boundaries enforce the communication semantics. We create these constraints at the task boundary where the communication call occurs in each process. We want to do this at only one place in the task group for each communication act to keep the task representation reasonably simple. However, for this to be correct, the ordinality of the communication scheduling point must be path invariant. That is, if the communication call is the boundary between execution episodes i and $i+1$ along one execution path in a process's ITG, it must be so for every execution path.

If the ordinality of the synchronous communication scheduling point is not constant across all execution paths, then we would have to impose the communication related precedence constraints, and the associated communication delays, *everywhere* they might occur. If the communication scheduling point was the ith scheduling point along one path in the send process's ITG, but the jth along another, then the communication delay would have to be imposed between tasks i and $i + 1$, *and* between tasks j and $j + 1$ in the sending process's task group. Precedence constraints would also be required between tasks i and j in the sending process, and the task beginning with the exit from the receive in the receiving process. This greatly increases the number of constraints that must be considered by the scheduler. If the ordinality of the scheduling point for the receive is also data dependent, the situation becomes extremely complicated.

Since task group construction is done at compile time, unique labeling of the calls in the source is required for us to know which send corresponds to which receive, enabling us to check, in turn, the invariance of the communication scheduling point ordinality. The techniques discussed earlier for inserting process suspension points along paths are used to make the ordinality of the communication points invariant. Note that we can permit more than one send or receive in each communicating process to correspond, as long as all of the corresponding calls in each process have the same ordinality.

These are not the only transformations that are possible, and we continue to investigate the application of transformations for several purposes. However, the techniques explained here are sufficient to implement a working system. Optimizations and extensions of many kinds are possible, and provide ample opportunity for long-term development of this approach to program translation and representation.

4 Summary and Future Work

This paper has described our basic approach to the problem of program translation for systems using explicit plan scheduling. The need for translation between the programming and run-time models arises because this type of scheduling assumes nonblocking tasks as the schedulable entities, yet we wish to preserve the process model as the programming model. The time graph representation of a process's temporal control flow structure was introduced. The three phases of the translation method were described and illustrated: time graph construction, subgraph reduction, and task group construction.

The design of the basic translation method is complete and we are integrating it into our compiler and related tools. We are also implementing, in parallel, the next version of the Spring operating system which provides predictable low level process management and general system support. The scheduling code from the previous version of the system is integrated into the predictable version, and we are now

conducting simple tests of the process code and task representations produced by the compiler.

Extensions of the translation method will follow several directions. We would like to develop more ITG transformations that will enable us to simplify the generated task group. Consideration of caching effects will require modifications to the subgraph reductions, and improvement of the processor models used to generate the WCETs for the RTL graph's basic blocks should increase the accuracy of our behavioral predictions. Extension of the reductions to construct symbolic expressions for execution time will enable us to do scheduling-time evaluation of task execution times in terms of specific input data or system configuration characteristics, which should decrease excess allocation of resources.

Acknowledgements

I would like to thank my advisors, Jack Stankovic and Krithi Ramamritham for their advice and encouragement. I would also particularly like to thank Chung-Huei Kuan and Decao Mao for their invaluable assistance with the kernel implementation work so necessary to support the work described here.

References

[1] P. Amerasinghe. An Interactive Timing Analysis Tool for the SARTOR Environment. Master's thesis, University of Texas at Austin, 1985.

[2] W. A. Halang and A. D. Stoyenko. Comparative Evaluation of High-Level Real-Time Programming Languages. *Real-Time Systems Journal*, 2(3), 1990.

[3] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proceedings of OOPSLA/ECOOP*. ACM, October 1990.

[4] K.Kenney and K. Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5):70–78, May 1991.

[5] E. Kligerman and A. D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, September 1986.

[6] P. S. Lavoie. Tool to Analyze Timing on 68020 Processor. Master's Project, University of Massachusetts-Amherst, 1991.

[7] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *JACM*, pages 46–61, February 1973.

[8] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Calculations. *IEEE Computer*, 24(5):58–68, May 1991.

[9] D. Niehaus. Program Representation and Execution in Real-Time Multiprocessor Systems. Phd. Thesis Proposal, University of Massachusetts-Amherst, 1991.

[10] D. Niehaus and C. Kuan. Spring Software Generation System. Technical report, Spring Project Documentation, 1990.

[11] D. Niehaus, E. Nahum, and J. A. Stankovic. 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.

[12] V. M. Nirkhe, S. K. Tripathi, and A. K. Agrawala. Language Support for the Maruti Real-Time System. In *Proceddings of the IEEE Real-Time Systems Symposium*. IEEE, December 1990.

[13] C. Park and A. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[14] R. Rajkumar, L. Sha, and L. Lehockzy. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE, 1988.

[15] K. Ramamritham, J. A. Stankovic, and Perng-Fei Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[16] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[17] R. Stallman. Using and Porting GNU CC. Technical Report 88-85, Free Software Foundation, October 1989.

[18] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, January 1991.

[19] A. Stoyenko and T. Marlowe. Schedulability, Program Transformations and Real-Time Programming. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 33–41. IEEE, May 1991.

[20] G. Zlokapa. Hard Real-Time Multiprocessor Scheduling with Precedence Constraints. Phd. Thesis Proposal, University of Massachusetts-Amherst, 1991.