Compiling for Heterogeneous Systems: A Survey and an Approach

Kathryn S. McKinley, J. Eliot B. Moss, Sharad K. Singhai, Glen E. Weaver, Charles C. Weems

CMPSCI Techincal Report 95-82

October 1995

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
{mckinley, moss, singhai, weaver, weems}@cs.umass.edu
(413) 545-1249 (fax)

Abstract. Large applications tend to contain several models of parallelism, but only a few of these map efficiently to the single model of parallelism embodied in a homogeneous parallel system. Heterogeneous parallel systems incorporate diverse models of parallelism within a single machine or across machines. These systems are already pervasive in industrial and academic settings and offer a wealth of underutilized resources for achieving high performance. Unfortunately, heterogeneity complicates software development. We believe that compilers can and should assist in managing this complexity. We identify four goals for extending compilers to assist with managing heterogeneity: exploiting available resources, targeting changing resources, adjusting optimization to suit a target, and allowing programming models and languages to evolve. These goals do not require changes to the individual pieces of a compiler so much as a restructuring of a compiler's software architecture to increase its flexibility. We examine the features and flexibility of six important parallelizing compilers to find existing solutions for flexibility. Where no solutions exist, we propose architectural changes to compilers.

1 Introduction

This paper surveys existing optimizing compilers and compiler frameworks for parallel machines and examines their potential contributions to a compiler for heterogeneous systems. The variety and variability of available resources in a heterogeneous system demands a compiler that is much more flexible than current compilers. This paper identifies where existing technology can meet these demands and where new techniques must be developed. We begin by discussing the original goals and features of several important, existing compilers. We then distill the impact of heterogeneity into four goals that an ideal compiler must address: exploiting available resources, targeting changing resources, adjusting optimization to suit a target, and allowing programming models and languages to evolve. For each goal, we isolate applicable technology that can be borrowed from existing compilers. Finally, we describe a compiler architecture that meets the needs of heterogeneity by incorporating this borrowed technology along with our own proposals in a flexible framework.

Heterogeneous processing
Current parallel machines implement a single homogeneous model of parallelism. As long as this model matches the parallelism inherent in an application, the machines perform well. Unfortunately, large programs tend to use several models of parallelism. Therefore, heterogeneous systems are being developed to provide consistent high performance by incorporating multiple models of parallelism within one machine or across machines, creating a virtual machine. Tightly-coupled heterogeneous computers integrate different parallel

This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0742, monitored by the Office of Naval Research.

architectures together within a single machine (*e.g.*, Meiko CS-2, IBM SP-2, and IUA [W+89]). Virtual heterogeneous machines treat a network of machines as a single virtual computer. This processing model is sometimes called metacomputing or, when the machines are similar, cluster computing. A virtual machine is often connected via message passing interfaces such as PVM [SGDM94], p4 [BL94], and MPI [Mes94].

Heterogeneous processing [SC92, Tur93, Tur95, KPSW93, GY93] is the well-orchestrated use of heterogeneous hardware to execute a single application [KPSW93]. When an application encompasses subtasks that employ different models of parallelism, the application may benefit from using disparate hardware architectures that match the inherent parallelism of each subtask. For example, Klietz *et. al.* describe their experience executing a single application, a simulation of mixing by turbulent convection, across four machines (CM-5, Cray-2, CM-200, and an SGI)[KMCP93]. The four machines form a single virtual machine, and the authors leverage the strengths of each machine for different tasks. Unfortunately, each task had to be hand parallelized specifically for the appropriate target machine, and each machine has a unique language dialect. If more machines became available or if the availability of machines changed, the application would have to be rewritten.

This example illustrates that although heterogeneous processing offers improved performance, it increases the complexity of software development. The complexity arises from three important features of heterogeneity: (1) variety, (2) variability, and (3) high performance. First, heterogeneous systems consist of a *variety* of hardware. For an application to take advantage of heterogeneity, it must be partitioned into subtasks, and each subtask mapped to a processor with a matching model of parallelism. The selection of available processors affects not only the mapping but also the partitioning, because subtasks should be chosen so that each uses a model of parallelism found in the set of available processors. Moreover, variability opens up the opportunity to trade local performance for overall performance of an application. If an application taxes certain components more than others, the overall execution time may be reduced by moving some subtasks to suboptimal component machines. Second, virtual heterogeneous systems experience *variability* as their makeup changes from site to site or day to day or based on load (*e.g.*, when component machines are highly utilized, added, or removed.) This variability of hardware resources requires rapid adaptation of programs to new configurations at compile and run time. The ability to adjust to different configurations will enable an application to further improve its performance by adjusting its partitioning and mapping at run time, according to the current work load of the heterogeneous machine. Furthermore, variability deters developers from using machine specific code (or languages) to improve performance. Third, heterogeneous systems can achieve *high performance.* If the execution time of a program did not matter, then it could probably run on a homogeneous processor with less trouble. The emphasis on high performance suggests that heterogeneous systems will be constantly updated with the latest hardware which programs will need to use. The demand for high performance also precludes simple solutions such as adding layers of abstraction that obscure the heterogeneity.

Compilers for heterogeneous systems

Developing software for heterogeneous systems would be overwhelming if each application needed to handle variety in its targets and variability of targets between executions. However, the variety, variability, and performance concerns inherent in heterogeneity are similar to the concerns of portability, where developers wish to move programs from platform to platform with minimal effort and no loss of efficiency. On current systems, portability is largely the result of using compilers. Developers write programs in source languages that present an abstract machine significantly simpler than the real machine. Compilers not only convert source code into object code, but through optimization and transformation they also manage the details of tailoring an application to specific hardware. Compilers can also assist in managing the variety and variability of heterogeneous systems.

Extending compilers to manage heterogeneity must address four goals: exploiting available resources, targeting changing resources, adjusting optimizations to suit a target, and allowing programming models and languages to evolve.

- Variability in a heterogeneous system's configuration affects how a compiler ensures efficient use of resources. With a static configuration the compiler knows exactly what hardware the program can use, but a changing configuration requires the compiler to leave some decisions to run time (or to recompile

the application), much like existing systems resolve some addresses in the linker. For example, if a subtask can exploit multiple models of parallelism, the compiler may choose to create alternate compiled versions of the module and leave the decision of which version to use until link or run time.

- The variety and variability of target machines for which a compiler for heterogeneous systems generates object code suggests that the compiler should not have a fixed set of targets. Instead, the compiler should be structured so that a new target can be added to the compiler's repertoire without significantly disturbing the rest of the compiler. Because heterogeneity implies variety in the models of parallelism as well as instruction sets, the compiler would ideally make provision for adapting to new models of parallelism.
- The variety of component processors in heterogeneous systems impacts the strategies and technology used in optimization. Compilers use optimization technology (analyses, optimizations, and transformations) to specialize code to a specific machine or component processor. On a homogeneous machine, compilers use a static ordering (strategy) of analyses, optimizations, and transformations, which are themselves drawn from a fixed set. However, heterogeneity requires that compilers adjust their strategy to suit the target. Moreover, different component processors may need new optimizations and different orderings of optimizations to exploit their unique hardware.
- As new processors with new hardware features are developed, source languages typically evolve to give programmers access to these features. Some hardware features (*e.g.*, instruction pipelining) can be handled within the compiler, but other features (*e.g.*, a new interconnection network) may need language support to get the maximum benefit. It is possible to simply build a new compiler or modify an existing compiler, but current monolithic compilers make these changes a costly endeavor for a small language extension.

These goals do not require changes to the individual pieces of a compiler so much as a restructuring of the compiler's software architecture to increase the compiler's *flexibility*. It may not be feasible to build a completely flexible compiler, but the burden of developing software for heterogeneous systems is diminished to the extent that a compiler can meet these goals. Researchers already have made some steps towards increasing compiler flexibility.

The extensions to compilers necessary for heterogeneity are not totally new. Retargetable compilers generate code for several machines and often accept multiple source languages. However, they are limited in their range of target hardware and source languages, and their internal structure is rigid. Current compilers, including retargetable compilers, tightly couple the compiler with both the source language and the target machine. Source-to-source translators couple the translator with a language and a model of parallelism, and rely on other compilers to couple the translator's output with a physical machine. If either the target or the source language changes, the compiler must either be substantially altered or simply rewritten. This coupling is natural for homogeneous machines, where a single compilation can only target a single machine. However, this coupling limits the compiler's flexibility in dealing with diversity in targets, optimization strategies, and source languages. Heterogeneity is the first compiler application that requires and therefore justifies this level of flexibility.

The purpose of this paper is to find existing compiler technology that can be used in a compiler for heterogeneous systems and identify where new techniques must be developed. Therefore, we begin in Section 2 with a review of six important parallelizing compilers. Section 3 considers which of their features can be borrowed. Because adding flexibility largely impacts the compiler's structure, Section 4 proposes a new compiler architecture.

2 High Performance Parallelizing Compiler Survey

This section compares six existing compilers and frameworks: Parafrase-2, ParaScope, Polaris, Sage++, SUIF, and VFCS. We selected these systems because of their significant contributions to compiler architectures and because they are well documented. ParaScope and SUIF are mature compilers with a wide assortment of features; Parafrase-2, Polaris, and VFCS are more recent systems. Sage++ is the only compiler develoment

framework. Since it is not feasible to change each system to handle heterogeneity and compare the efforts, we instead describe each system's architecture, and discuss their potential benefits and drawbacks for compiling for heterogeneous systems (see Section 3). This section describes the general approach, facilities for interacting with users, programming model, organization, intermediate representation, and optimizations and transformations of the six systems. Table 1 summarizes this information and Table 2 lists specific optimizations and transformations.

2.1 Parafrase-2

Parafrase-2 is a source-to-source translator from the University of Illinois [P+89, GP94]. The goal of Parafrase-2 is to be a research tool for investigating compiler support for multiple languages and target architectures. Rather than try to build everything into Parafrase-2 from the beginning, the authors strive to make it flexible enough for later additions. Parafrase-2 currently accepts C, Fortran 77, and Cedar Fortran.

System architecture
The overall architecture of Parafrase-2 is typical for source-to-source translators. Language specific parsers convert source programs into a common intermediate representation. Transformations operate over the IR and then a postprocessor generates source code in the original language using a language specific pretty printer. Parafrase-2 has an optional graphical user interface (GUI) for selecting and applying transformations.

Programming model
Parafrase-2 is unique among these systems in that it focuses on control parallelism as well as data parallelism. Parafrase-2 first partitions the program into separate tasks, such that the dependencies between tasks are minimized. A task may be executed as soon as the dependencies, for which it is a sink, are satisfied. Then, Parafrase-2 parallelizes the execution of loops. The optimizations used for this are the same as those used by compilers to generate loop parallelism and data parallel code (***e.g.***, loop interchange).

Intermediate representation
The authors of Parafrase-2 have designed their own intermediate representation data structure: Hierarchical Task Graphs (HTG). The HTG hierarchy is built by repeated substitution of a single node for each strongly connected region in the CFG. The leaves in an HTG represent simple statements and subroutine calls. Interior nodes represent loops and basic blocks in the flow graph. In addition, the control flow and data dependence graphs are included as subgraphs of the HTG. The HTG representation obscures the syntax of a program, but it highlights dependencies between sections of the program. The emphasis on dependencies aids program decomposition, because a natural way to select tasks is to find highly interdependent clumps of nodes. The hierarchical organization of HTGs facilitates choosing tasks at a granularity appropriate for the target architecture. The HTG representation is largely immune to extensions of the source language; calls and loops are the only language constructs with distinct nodes in an HTG. Much of the semantics of loops and calls are captured by the dependence and control information, further separating the programming language from the representation and thereby making the HTG language independent.

Optimizations
Parafrase-2 has a large set of analyses, optimizations, and transformations (see Table 2). Parafrase-2 has the base analyses (data dependence and control dependence) necessary for handling parallelism, as well as, symbolic analysis and interprocedural analysis. For each routine, Parafrase-2's interprocedural analysis records summary information of aliases, modified variables (MOD), and referenced variables (REF) and propagates interprocedural constants. In addition to an assortment of traditional optimizations, Parafrase-2 has loop blocking, distribution, interchange, and task and loop parallelization for high-level transformations, and inlining for interprocedural optimization. Parafrase-2 also has an optional graphical user interface that allows programmers to manually alter dependence information and guide optimization. In batch mode, Parafrase-2 uses a fixed order of optimizations for a given target architecture.

Parafrase-2 structures its analyses and optimizations into separate passes that can be reordered. This modularity largely insulates other passes from changes made to just one (or a few) passes. Parafrase-2's modularity also means that adding new analyses and optimizations to the system is easy. The major difficulty

will be that as the number of potential passes increases, so does the burden on the user to ensure that analyses are rerun when an optimization pass invalidates analysis data. Unfortunately, Parafrase-2 neither supports incremental updates to analysis data nor supplies a mechanism for ensuring that analysis passes are rerun when necessary. Parafrase-2 does, however, automatically invoke an analysis pass, when it is needed but not yet executed.

Summary
Parafrase-2 has several powerful features that make it an interesting system. First, Parafrase-2 easily adapts to new language extensions. Accommodating language extensions is relatively easy because the IR emphasizes data and control dependences rather than source language syntax. Hence, the impact from extending an existing language should be largely localized in the front end parser and the back end pretty printer. Second, Parafrase-2 supports interprocedural analysis and transformation (*i.e.*, inlining) via summary based techniques. Third, Parafrase-2 tackles the automatic identification of control parallelism which, when combined with techniques for finding loop parallelism, gives the compiler more choices in transforming a program to match a particular target architecture.

2.2 ParaScope

Rice University's ParaScope is an interactive parallel programming environment built around a source-to-source translator [C+93, KMT93]. It provides sophisticated global program analysis and a rich set of program transformations. ParaScope is a research tool and has been specialized for several Fortran derivatives. This section concentrates on the Fortran-D version of the ParaScope called the D system. Fortran-D enhances Fortran 77 and 90 with data decomposition specifications [F+90]. The output of the D System is an efficient message-passing distributed memory program [HKT91, HKT92, Tse93].

System architecture
The D System accepts programs written in either a subset of Fortran 77D or Fortran 90D and converts them into an abstract syntax tree (AST) representation. The Fortran-D back end uses loop transformations and communication optimizations to build efficient message-passing, SPMD node programs. These node programs can be compiled and further optimized by scalar compilers.

Programming model
The D System implements the data-parallel SPMD programming model. The programmer specifies data decompositions using language constructs of Fortran-77D and Fortran-90D (and hence in a sense also detects parallelism.) The D System uses these data decompositions to partition computation into node programs and assign these programs to processors. The assignment of computation to processors follows the *owner-computes* rule. The compiler analyzes communication to find non-local data accesses that would require message passing, and then optimizes the communication.

Intermediate representation
ParaScope uses an abstract syntax tree (AST) as its intermediate representation. Various modules of ParaScope use auxiliary data structures associated with the AST to represent and manipulate data and control dependence information. Unfortunately, ParaScope's implementation of the AST does not use data abstraction; therefore, changes to the AST potentially impact the entire ParaScope system.

Optimizations
ParaScope offers an unusually large selection of analyses, optimizations, and transformations. The ParaScope developers have built a whole suite of analyses over the years including the following: scalar analyses, symbolic analyses (using static single assignment form), interprocedural analyses and synchronization analyses. Users can interactively change the results of dependence analysis. ParaScope also has an extensive palette of interactive transformations (loop distribution, fusion, strip mining, interchange, reversal, skewing, array privatization, scalar replacement, scalar expansion, unroll-and-jam, and loop peeling). Descriptions of the applicability and safety of these transformations are built into ParaScope, and it interactively advises about the profitability of transformations. The Fortran-D compiler adds communication transformations such

as message vectorization, communication selection, message coalescing, message aggregation, and message pipelining. In addition, recent work addresses automatic data partitioning [BKK94].

Summary
ParaScope is a mature system for automatic and interactive parallelization which provides a wide selection of analyses, optimizations, and transformations. As a source-to-source translator, it is not tightly coupled to any particular machine. Unfortunately, it is difficult to modify or extend because the internal structure of the AST is exposed to the entire system. Changing the IR to support new languages or models of parallelism requires modifications throughout ParaScope. The Fortran-D compiler has a monolithic structure which makes it difficult to extend.

2.3 Polaris

Polaris from the University of Illinois, is an optimizing source-to-source translator [B+95, P+93, F+94]. The authors goal for Polaris is automatic parallelization of sequential Fortran programs for a variety of architectures. The authors have thus far focused on parallelization for shared-memory machines. Polaris is written in C++, and it compiles Fortran 77.

System architecture
Polaris organizes its analyses and optimizations as separate passes that operate over the whole program, but a pass may call the body of another pass to operate over a subset of the program. Programmers can affect the operation of passes through command line parameters. Programmers may also convey extra information, such as parallelism, to the compiler by embedding assertions in source code. Polaris outputs Fortran 77 code augmented with annotations indicating parallelism for several dialects of parallel Fortran. In most cases, Polaris leaves the actual transforming of code to the platform-specific, backend compiler. The annotations tell the backend compiler where to apply specific transformations, but the backend compiler is free to perform additional transformations of its own (***e.g.***, at least one of their backend compilers performs loop coallescing [Gro95b]). For its internal IR, Polaris exploits the data abstraction of a C++'s class hierarchy to improve the quality of Polaris code. The IR may only be updated via method calls which ensure the consistency of the IR with respect to the syntax of the source language, the structure of the internal representation, and the accuracy of analysis information.

Intermediate representation
Polaris uses an AST as its internal intermediate representation. Polaris has five classes that correspond to the major syntactic elements of a Fortran program: the ***Program*** class represents the entire program, the ***ProgramUnit*** class corresponds to top-level units (programs, subprograms or block data), the ***Statement*** class represents a single statement, the ***StmtList*** class holds a series of statements (***e.g.***, a loop body), and the ***Expression*** class corresponds to expressions. Polaris also provides classes for symbols, symbol tables, and various generally useful data structures. All of these classes may be specialized, via inheritance, for a specific purpose. For example, headers for Fortran DO loops are represented by a subclass of ***Statement*** that has additional fields for items such as the end of the loop and the index variable.

Optimizations
The authors of Polaris performed a study in which they parallelized the Perfect Club benchmark by hand [EHLP91]. They found that a few new analyses and optimizations, in addition to ones already prevalent in commercially available parallelizing translators, would significantly improve a translator's ability to parallelize code. For analysis, they added constant propagation, inlining (also used for optimization), and their own version of symbolic data dependence analysis. Polaris performs ***auto-inlining***, which is the automatic inlining of subroutines that are fifty lines or less. In their experience, auto-inlining is sufficient for Polaris to carry out flow-sensitive analysis [Gro95a, Gro95b]. Instead of using traditional dependence analysis, Polaris builds symbolic lower and upper bounds for variable references and propagates these ranges as needed throughout the program using a demand driven algorithm [BE95b, BE95a]. Polaris's range test then uses these ranges to disprove dependences[BE94]. For optimization, Polaris includes scalar and array privatization, induction variable substitution, and reduction recognition and replacement. When Polaris recognizes

a reduction, it replaces accesses to the reduction variable to one of ***blocked*** (variable access is in a critical section), ***privatized*** (each processor has its own copy of the variable), and ***expanded*** (processors build partial sums in disjoint sections of a global array)[PE95].

Summary
Because Polaris's authors focus on automatic parallelization rather than multiple source languages or diverse architectures, it parses only Fortran 77 extended with assertions. The inclusion of consistency checks in the IR access methods supports correct and quick optimization development. The automatic, incremental update of analysis data needed ensures the accuracy of this information but requires that knowledge about the analysis routines be included in the IR. Modifying the IR to support additional languages may require changes to the consistency checks and analysis routines, but adding new nodes for an existing language should be easy because of Polaris's object-oriented design.

2.4 Sage++

Sage++ from Indiana University is a toolkit for building source-to-source translators [B+94]. The authors foresee optimizing translators, simulation of language extensions, language preprocessors, and code instrumentation as possible applications of Sage++. Sage++ is written in C++ and provides parses for C, C++, pC++, Fortran 77, and Fortran 90. Because Sage++ is a compiler toolkit instead of a compiler, it is not limited to any hardware architectures.

System architecture
Sage++ assumes that a source-to-source transformation will proceed in three phases. Sage++ provides the first and last phases, parsing and unparsing (***i.e.***, pretty printing) respectively. Between phases, the program is stored on disk in a Sage++ supplied intermediate representation. The middle phase that actually analyzes and transforms the program is the responsibility of the Sage++ user, but Sage++ supplies several basic analysis and transformation routines as building blocks. In Sage++ parlance, a specific middle phase is called a restructuring tool.

Intermediate representation (IR)
Like many other source-to-source translators, Sage++'s intermediate representation resembles an abstract syntax tree. Sage++ uses a C++ class hierarchy for defining the nodes in its IR, where each class corresponds to a syntax component of the source language. Sage++ has five major classes that represent whole programs and files (***SgProject*** and ***SgFile***), statements (***SgStatement***), and expressions (***SgExpression***), symbols (***SgSymbol***), and types (***SgType***).

Sage++ emphasizes extensibility by providing a single, simple intermediate representation that users can extend to represent new languages. Sage++ IR currently supports C, C++, pC++, Fortran 77, and Fortran 90.

Optimizations
Sage++ provides a general framework for data dependence and flow analysis for Fortran 77. A separate class, ***depGraph*** stores results of dependence analysis using the Omega test. A basic framework for data flow analysis is provided, but the compiler writer is responsible for writing flow routines. Sophisticated interprocedural or alias analyses are not performed.

Sage++ provides a basic set of loop transformations like loop interchange, fusion, tiling and distribution. However, these transformations do not check the legality of the transformation. The rationale being that Sage++ is a toolkit, not a complete compiler. Using Sage++, a compiler writer can implement tests for legality and a richer set of transformations and optimizations. The compiler writer is responsible for writing the code to update the dependence graph after program changes.

Applications of Sage++
Sage++ has been used to support new language extensions [BPMG94, Y+94, M+94, B+93]. Sage++ is useful not only for building source-to-source translators but also other tools that depend on programs as input. Some traditional uses of Sage++ include optimizing expressions in scientific library code, instrumenting user code, and preprocessing user annotations in Fortran-S. But, Sage++ has also been used to construct a suit

of programming environment tools: tuning and analysis utility (TAU), file and class display (Fancy), call graph extended display (Cagey), class hierarchy browser (Classy), routine and data access profile display (Racy) and event and state display (Easy) [MBM94, BHMM94].

Summary
Sage++ is a convenient tool for building source-to-source translators. It is not tied to any particular hardware architecture which makes it portable, and it provides basic routines needed by compilers. Though Sage++ has been extended to accept new source languages several times, the process is, unfortunately, somewhat tedious. It also lacks many of the sophisticated analyses, optimizations, and transformations because of its compiler framework nature.

2.5 SUIF

Stanford University Intermediate Format (SUIF) is a compiler framework designed for research in compilation techniques, especially automatic parallelization [W+94, AALL93, Sta94]. SUIF functions as either a source-to-C translator or a native code compiler. SUIF has been used to study parallelization for both shared-memory and distributed shared-memory machines [W+94]. SUIF accepts source code written in either Fortran 77 or C, but a modified version of `f2c` [FGMS93] is used to convert Fortran code to C code. The modifications to `f2c` retain some Fortran specific information that further aids in analysis.

System architecture
SUIF has a flexible organization, with each analysis and optimization coded as a separate, independent pass. Because passes are independent, they can execute in any order, but pragmatically, analysis passes must run prior to passes that use the analysis data. SUIF does not check that the ordering of passes is valid, but a transformation pass can easily determine that prerequisite analysis passes have not yet executed. The sole means of communication between passes is attaching annotations to the intermediate representation. An annotation consists of a name field which identifies the type of the annotation and a body whose structure depends on the annotation's type. Hence, an analysis pass will attach annotations of a certain type, and optimization passes search for these annotations. The annotation mechanism does not ensure consistency or accuracy of information; when a transformation pass corrupts analysis information, the analysis pass must be rerun.

Intermediate representation
The design of SUIF's intermediate representation reflects the needs of code generation. Source-to-source translators generally use a single, high-level representation that is appropriate for loop transformations. However, because high-level representations (such as ASTs) do not totally order instructions, they are inappropriate for certain low-level optimizations (*e.g.*, instruction scheduling). SUIF's IR (which is also called SUIF) includes both high- and low-level representations. SUIF initially represents programs as a forest of ASTs using high-SUIF nodes (a combination of high- and low-level nodes). SUIF's AST is unique in that it represents only a few language constructs (*i.e.*, for loops, general loops, if statements, and array references) with high-level nodes; the remainding constructs are represented by low-level nodes that resemble RISC instructions (*e.g.*, load, store and add). These low-level nodes are known as low-SUIF. When the AST is lowered to a linear form, SUIF translates high-level nodes into low-level nodes, *i.e.*, low-SUIF. Hence, SUIF shares low-SUIF nodes between the AST and linear representations.

SUIF is implemented in C++ and uses a class hierarchy for its IR. SUIF has three major classes for representing programs: the *file_set* class collects all the files composing a single program, the *file_set_entry* class represents a single source file, the *tree_node* class defines nodes in the AST. In SUIF, an AST represents a single procedure and has three parts that divide the tree horizontally. The top part is simply the root of the tree, and the second part consists of high-level nodes representing source language constructs. The bottom part is made up of low-level nodes (*e.g.*, the nodes common to low-SUIF and high-SUIF). Thus, converting from high-SUIF to low-SUIF is essentially compiling away the middle part of an AST. In addition to the three major classes for representing programs, SUIF also has classes for symbols, symbol tables, types, annotations, and generic data structures.

Optimizations
SUIF does not rely on another compiler to perform traditional optimizations, and therefore has a full complement of compiler analyses and optimizations. For analysis, SUIF can compute data dependence, control dependence, symbolic constants, and interprocedural information. The data dependence analyzer uses a suite of tests of varying complexity to obtain accurate results quickly. The interprocedural information is computed by an unusually extensive collection of analyses [HMA95]. SUIF structures its interprocedural analyses as a bottom-up pass that summarizes the behavior of each subroutine, followed by a top-down pass that applies calling contexts to each subroutine's summary description to compute its final analysis result. For optimization, SUIF has a wide assortment of traditional and loop optimizations (see Table 2), including unimodular loop transformations (***i.e.***, interchange, reversal, and skewing) [WL91]. SUIF also has an extensive collection of interprocedural transformations: parallelization, data privatization, inlining, cloning, and reduction recognition. Lastly, SUIF includes a code generator which allows it to not only transform a sequential program to a parallel program, but also immediately generate assembly code for the parallel program.

Summary
SUIF is most unique for the two distinct levels of abstraction in its intermediate representation. With two levels, SUIF can generate optimized assembly code as well as perform source-to-source transformations. The definition of high-SUIF as a combination of low- and high-level representations eases the development of passes that work on both levels of abstraction, and allows the interleaving of loop-level and low-level optimizations. Because the authors focus on loop-level parallelism for MIMD machines, SUIF's IR has a high-level representation for only a few source language constructs, which may limit SUIF's ability to target other architectures. Fortunately, SUIF's IR is relatively easy to extend because it uses object-oriented techniques; new IR nodes will have little effect on existing ones. Adding new analyses and optimizations is also easy because they are independent, and new annotations (with new names) do not impact existing annotations. SUIF's other major strength is its flexible ordering of compiler passes. Though this flexibility facilitates exploration of optimization ordering for different architectures, SUIF neither assists in dynamically selecting passes nor provides guards against inappropriate orderings.

2.6 Vienna Fortran Compilation System (VFCS)

The Vienna Fortran Compilation System (VFCS) from the University of Vienna is an interactive, source-to-source translator for Vienna Fortran [CMZ92a, CMZ92b, ZC93, ZCMM93]. VFCS is based upon the data parallel model of computation with the Single-Program-Multiple-Data (SPMD) paradigm. The Vienna Fortran language extends Fortran with constructs for distributing data across the processors of a distributed-memory multiprocessing system (DMMP). The language features focus mainly on the issue of distributing data across a virtual processor structure based on the default rule of ***owner computes***. VFCS is written in C and generates explicitly parallel, SPMD programs.

System architecture
Input languages accepted by VFCS are Vienna Fortran, Fortran 77, HPF and a subset of Fortran 90. The compiler outputs message passing programs in Intel features, PARMACS, or MPI. VFCS allows users to write programs for distributed-memory systems using global addresses. It allows the data distributions of arrays to change dynamically, based upon runtime conditionals. Sophisticated memory management and buffering schemes are used to optimize communication.

VFCS operates in both interactive and batch modes. In interactive mode, users have access to a set of analyses and a catalog of transformations via graphical user interface. In batch mode, a batch command language enables the creation of batch files which may be automatically applied to a Fortran program. Alternatively, the sequence of transformations executed during an interactive session can be ***protocolled***. The protocol can then be automatically executed in batch mode.

Intermediate representation (IR)
VFCS uses abstract syntax trees for intermediate representation, and a ***program database*** for communication among components of the compiler. The program database contains syntax trees, call graphs, interprocedural

information, dependence graphs, and data partitioning information. Optimizations work directly on the program database, and transformations are guided by an interactive kernel.

Optimizations
VFCS performs traditional data flow and data dependence analyses. It also has interprocedural communication and dynamic distribution analysis. It optimizes irregular access pattern using PARTI routines [SCMB90].

VFCS was specifically designed for distributed-memory machines and has an extensive set of communication optimizations besides loop optimizations. It performs overlap analysis to determine which non-local elements of a partitioned array are used in a processor and inserts communication primitives based on the analysis. To reduce communication overhead VFCS *aggregates* communication primitives within a loop whenever appropriate. It performs further optimizations to simplify or eliminate communication *masks* in SPMD programs.

Vienna Fortran has many optimizations in common with Fortran D. However, Vienna Fortran has a richer set of data distributions. In particular, it has static distributions, dynamic distributions, user-defined distribution functions, and run-time selectable conditional distributions. Similar to HPF, it allows conditional code execution based on run-time data distribution. However, dynamic data distribution is quite expensive and Vienna Fortran does not provide mechanisms to measure the trade-off between the cost of redistribution and the additional communication cost. Some recent work uses static performance measurements to guide optimization process [Fah94, FZ93].

Summary
The design goal of Vienna Fortran is to generate optimized code for distributed-memory machines. Vienna Fortran borrows upon SUPERB, a predecessor of VFCS [ZBG88]. In turn, many features of Vienna Fortran have been incorporated into HPF. VFCS is an interesting system because many other supporting tools have been developed besides the compiler. Some recent work addresses task and data parallelism in Vienna Fortran and HPF [CMRZ94, HHM+94]. It is not readily adaptable to other languages because the optimizations are too tightly coupled with the rest of the system.

2.7 System Comparison

This section compares and contrasts the systems we have just reviewed and summarized in Table 1 on page 13 and Table 2 on page 15. We compare their general approach, programming model, organization, intermediate representation, analyses, optimizations and transformations, and interaction with users. The goal of this section is to better understand the contributions of these systems, especially with respect to their software architecture.

ParaScope has several different compilers built on top of its core routines. Where noted, we refer to a specific compiler, *e.g.*, the D System; otherwise, we refer to the ParaScope system as a whole. Similarly, Sage++ is a framework for assembling compilers, so where noted, we refer specifically to the pC++ compiler.

General
Reflecting their common mission of compiling for parallel machines, the systems are similar in their general approach. All the systems (except Sage++) are designed for automatic parallelization, and Sage++ supports features necessary for building parallelizing compilers (*e.g.*, data dependence). Nevertheless, they have slightly different emphases, *e.g.*, Polaris targets shared-memory machines, and VFCS emphasizes targeting distributed memory machines. All the compilers can operate as source-to-source translators and Sage++ facilitates the construction of source-to-source translators. SUIF also compiles directly into assembly code for the MIPS family of microprocessors. All the systems parse some variation of Fortran, and half of them also handle C. These are the traditional languages for high performance computing.

Programming model
The compilers in our survey expect to receive and generate a variety of programming models. SUIF accepts only sequential C and Fortran 77 programs, and therefore must extract all the parallelism automatically. Polaris parses only Fortran 77 but interprets assertions in the source code that identify parallelism. ParaScope allows programmers to use data parallel languages as well as sequential languages, and attempts to

find additional loop and data parallelism. Though VFCS accepts data parallel languages, it requires that programmers supply the data distribution. VFCS then finds opportunities to apply data parallel operations. Parafrase-2 accepts Cedar Fortran which includes vector, loop, and task parallelism and generates programs with control and data parallelism.

Most of the compilers in our survey generate data parallel programs, but Parafrase-2 extracts control parallelism as well. SUIF and Polaris take a classic approach to parallelization by identifying loop iterations that operate on independent sections of arrays and executing these iterations in parallel. For scientific applications loop level parallelism has largely been equated to data parallelism. The D System and VFCS, on the other hand, output programs that follow the SPMD model; the program consists of interacting node programs. Parafrase-2 identifies non-loop sections of a program that can be executed in parallel and transforms these sections into tasks. Parafrase-2 also finds loop level parallelism.

Organization

The organization of analyses and optimizations varies slightly between the systems. All of the systems support batch compilation and optimization. However, Parafrase-2 and ParaScope also provide a graphical interface that enables users to interact with the compiler and affect its optimization. Moreover, ParaScope and Polaris support incremental updates of analysis data. Though incremental analysis is not more powerful than batch analysis, it dramatically increases the speed of compilation and therefore encourage more extensive optimization.

Most compilers have additional tools to assist users in writing and understanding their parallel programs. ParaScope strives to provide a complete parallel programming environment, including an editor, debugger and an automatic data partitioner. Sage++ provides a rich set of tools for pC++ named ***Tuning and Analysis Utilities*** (TAU). TAU includes utilities for file and class display, call graph display, class hierarchy browsing, routine and data access profile display, and event and state display. Almost all of these systems are research tools that encourage user experimentation. Experimentation is further facilitated by having graphical user interfaces in Parafrase-2, ParaScope, Sage++, and VFCS which display various aspects of the compilation process in windows that the user can interact with to get more details or provide inputs to the compiler.

Intermediate representation

Most of the systems use an abstract syntax tree (AST) as an intermediate representation. ASTs retain the source level syntax of the program which makes them convenient for source-to-source translation. SUIF uses a mixed-level AST which allows both a broader range of transformations and the sharing of nodes between the AST and linear representations.

Parafrase-2 uses the hierarchical task graph (HTG) representation instead of an AST. HTGs elucidate the control and data dependencies between sections of a program and are convenient for extracting control parallelism.

Analyses

All the systems in our survey provide the base analysis necessary for parallelism, but beyond that their capabilities diverge. Data dependence analysis is central to most loop transformations and is therefore built into all the systems. However, Parafrase-2, ParaScope, and SUIF perform control dependence analysis, albeit in a flow-insensitive manner. All the systems (except Sage++) perform intraprocedural symbolic analysis to support traditional optimizations, but ParaScope and Parafrase-2 have extensive interprocedural symbolic analysis such as forward propagation of symbolics. VFCS provides intraprocedural irregular access pattern analysis based on PARTI routines [SCMB90]. Parafrase-2, ParaScope, Polaris, SUIF, and VFCS provide interprocedural analysis. Polaris recently incorporated interprocedural symbolic constant propagation [Pau95]. Parafrase-2, ParaScope, and VFCS [Zim95] perform flow-insensitive interprocedural analysis by summarizing where variables are referenced or modified. SUIF's FIAT tool provides a powerful framework for both flow-insensitive and flow-sensitive analysis [HMA95].

Optimizations

A wide range of optimizations is supported by these compilers. Optimizations performed by uniprocessor compilers are termed as *traditional*. Except Sage++, all compilers provide traditional optimizations as listed in Table 2. Notice that SUIF generates native code with two levels of IR and has additional low level

optimizations such as register allocation. Again all six provide loop transformations, but ParaScope has an unusually large set of independent loop transformations.

Communication and synchronization optimizations, though not always distinct from loop optimizations, refer to the optimizations specifically performed for distributed memory machines, like bunching messages together. The VFCS system, which is designed exclusively for distributed memory machines, has a richer set of communication optimizations than the others. SUIF is able to derive automatic data decompositions for a given program. ParaScope and VFCS do this to some degree, however, the primary computation partitioning mechanism for them is the ***owner computes*** rule and the data partitioning is programmer-specified.

Parafrase-2 is unique in that it exploits ***control*** parallelism. It includes the partitioning of a program into parallel tasks as a transformation.

All compilers include ***applicability*** criteria for the transformations since a transformation may not be legal, ***e.g.***, loop interchange is illegal when any dependence is of the form $(\cdots, <, >, \cdots)$. Sage++ is unique in this respect; though it has a few loop transformations, it does not have any applicability criteria built in. The Sage++ developers argue that in a preprocessor toolkit like Sage++ applicability should be defined by the compiler writer for individual applications.

Though an optimization may be applicable, it may not be ***profitable***. The six compilers surveyed in this article take varying approaches to this issue. Parafrase-2 and ParaScope rely on user input. ParaScope also offers a small amount of feedback to the user based on its analysis. SUIF and Polaris use a fixed ordering of transformations for each target, and therefore perform valid optimizations according to a predefined strategy. VFCS performs static performance measurement and dynamic performance measurements based upon external tools P^3T [Fah94] and MEDEA [CPZ95] respectively, to determine profitability.

Closely related to the profitability issue is ordering criteria. Optimizations applied in different orders may produce dramatically different results. In interactive mode, ParaScope, Parafrase-2, Polaris and VFCSallow the user to select any ordering of optimizations. All support fixed optimization ordering via their command lines as well.

3 Criteria for a Compilation System in a Heterogeneous Environment

In Section 1 we introduced four goals that a compiler for heterogeneous systems must meet: exploiting available resources, targeting changing resources, adjusting optimization to suit a target, and allowing programming models and languages to evolve. This section expounds upon these goals by determining their implications on the compiler and examining the existing systems from Section 2 for applicable technology.

3.1 Exploiting Available Resources

As with any computer system, compilers for heterogeneous systems should generate programs that take maximum advantage of the available hardware. However, the variability in resources complicates this task. To account for variability, programs could simply be recompiled whenever the hardware configuration changes. Recompilation works well when the configuration is stable but is inefficient if the configuration changes frequently. Recompilation at runtime to adjust to the current workload of the heterogeneous machine defeats the purpose of high performance.

Multiple object modules for different targets
Instead of recompiling when the configuration changes, the compiler could precompile for several machines. Hence, the compiler produces the building blocks for a program partitioning, but the linker assembles the final partitioning. The compiler generates alternate versions of subtasks (or routines), and passes along enough information for the linker to select a final partitioning. If the configuration changes slightly, the linker may simply recompute the partitioning and mapping without the program experiencing a complete recompile. It may also be possible to account for the system's workload by relinking at run time. None of the existing compilers provide this level of flexibility. Excluding Sage++ which is a framework, all of the compilers perform partitioning and mapping within the compiler.

Table 1. Comparison table for surveyed systems.

Properties	**Parafrase-2**	**ParaScope/ D System**	**Polaris**	**Sage++/ pC++**	**SUIF**	**VFCS**
General						
Goals	Multiple Languages and Target Architectures, Extensibility	Automatic and Interactive Parallelization	Automatic Parallelization of Fortran Programs	Framework for Building Source-To-Source Translators	Tool for Research in Compilation Techniques, especially Automatic Parallelization	Compiling for Distributed Memory Systems
Source-to-Source	√	√	√	√	√	√
Source Languages	C, Fortran 77, Cedar Fortran	Fortran 77, Fortran 90[1], Fortran D	Fortran 77	C, C++, Fortran 77, Fortran 90, pC++	C, Fortran 77[2]	Fortran 77, Fortran 90[1], HPF, Vienna Fortran
Code Generation	Tuples	Tuples	—	—	MIPS Assembly	—
Programming Model						
Input	Sequential or Control Parallel	Sequential or Data Parallel	Sequential or Data Parallel	*NA*	Sequential	Sequential or Data Parallel
Output	Task/Loop Parallel	SPMD, Loop Parallel	Loop Parallel	*NA*	Loop Parallel	SPMD
Target Architectures	Multithreaded, SM, DSM	Uniprocessor, SM and DM	SM, DSM	*Not Specified*	Uniprocessor, SM, DSM	DM
Organization						
Implementation Languages	C	C, C++	C++	C++	C, C++	C
Incremental Analysis	—	√	√	—	—	—
Batch Optimization	√	√	√	√	√	√
Interactive Optimization	√+	√+	—	*NA*	—	√
User Interface	CLI, GUI	GUI	CLI, Assertions	CLI, GUI	CLI	CLI, GUI
Supporting Tools	—	PED, D Editor	Delta	TAU[3]	Sharlit, Fiat	P[3]T, MEDEA
Intermediate Representation						
Forms	HTG, Linear Tuples	AST	AST[4]	AST	Hybrid of AST and Linear Tuples[5]	AST

Properties	**Parafrase-2**	**ParaScope/ D System**	**Polaris**	**Sage++/ pC++**	**SUIF**	**VFCS**
Analyses						
Data Dependence	√	√	√	√	√	√
Control Dependence	√+	√	√	—	√	—
Symbolic[6]	√+	√+	√	—	√	√
Interprocedural	Alias, MOD, REF, Constant Propagation	Alias, MOD, REF, Constant Propagation, Symbolic	Inlining (for analysis), Constant Propagation	—	MOD, REF, GEN, KILL, Constant Propagation, Array Summary, Array Reshapes, Reductions[7], Induction Variables, Cloning[7]	Alias, USE, DEF, Const. Propagation, Overlap, Communication, Dynamic Distribution
Optimizations and Transformations						
Traditional	√	√	√	—	√+	√
Loop	√	√	√	√	√	√
Communication/ Synchronization	—	√[8]	√	—	√	√+
Data Partitioning	—	√	*In Progress*	—	√	√
Task Partitioning	√	—	—	—	—	—
Interprocedural	Inlining	Inlining, Cloning	Inlining, Cloning	—	Parallelization, Data Privatization, Inlining, Cloning, Reductions	Inlining, Cloning
Applicability Criteria	√	√	√	—	√	√
Profitability Criteria	Queries User	Queries User, CLI	Fixed for Arch.	—	Fixed for Arch.	Static/Dynamic Performance Measure
Ordering Criteria	Fixed for Arch., Interactively Selected by User	Fixed for Arch., Interactively Selected by User	Fixed for Arch.	—	Fixed for Arch.	Fixed for Arch.

Table 1. Comparison table for surveyed systems, continued...

Table 2. Summary of optimizations supplied by surveyed compilers

Optimizations	Parafrase-2	ParaScope/ D System	Polaris	Sage++/ pC++	SUIF	VFCS
Traditional	Scalar Expansion, Induction Var. Subst., (Symbolic) Const. Propagation, Dead Code, Forward Subst., Variable Localization	Scalar Expansion, Loop Invariant Code Recog., Scalar Privatization, Induction Var. Recog., Const. Propagation, Expr. Folding	(Symbolic) Constant Propagation, Dead Code, Induction Var. Subst., Expression Folding	—	CSE, Constant Propagation, Dead Code, Forward Propagation, Induction Var. Subst., Register Allocation	Scalar Replacement, Scalar Expansion, Strength Reduction, Induction Var. Subst., Const. Propagation
Loop, Memory Hierarchy, and Parallelization	Blocking, Distribution, Interchange, Task Parallelization, Loop Parallelization	Fusion, Interchange, Distribution, Strip Mininng, Skewing, Peeling, Unrolling, Unroll-and-Jam, Reversal, Scalar Replacement, Loop Parallelization	Data Privatization, Reduction Recog. and Replacement[9], Loop Parallelization	Fusion, Interchange, Tiling, Distribution, Loop Parallelization	Interchange, Reversal, Skewing, Tiling, Reduction Recog. and Replacement, Loop Parallelization	Reductions, Unroll-and-jam, Interchange, Distribution, Fusion, Strip Mining, Unrolling, Loop Parallelization
Communication	—	Communication Vectorization, Overlap of Communication and Computation	—	—	Latency Hiding, Message Aggregation, Redundant Message Elimination, Data Privatization	Elimination of Redundant Communication, Aggregate Communication
Interprocedural	Inlining	Inlining, Cloning	Inlining, Cloning, Symbolic Constant Propagation	—	Parallelization, Data Privatization, Inlining, Cloning, Reductions	Inlining, Cloning

Table 3. Abbreviations for Table 1.

CLI:	Command Line Interface	**DM:**	Distributed Memory
DSM:	Distributed Shared Memory	**GUI:**	Graphical User Interface
HPF:	High Performance Fortran	**HTG:**	Hierarchical Task Graph
P^{3}T:	Parameter-based Performance Prediction Tool	**PED:**	ParaScope Editor
SM:	Shared Memory	**TAU:**	Tuning and Analysis Utilities
$\surd$:	Yes	$\surd$+:	Exceptional Implementation
—	No or None	NA	Not Applicable

Compiler communicates with run-time environment
Another approach to exploiting varying resources is for the compiler to embed code that examines its environment at run time and dynamically decides how to execute. For example, VFCS has optimizations that dynamically decide their applicability at run-time. These optimizations along with delaying the binding of subtasks to specific processors increase communication between the compiler and the run time system. This increased cooperation allows compiler-level information to be used in adjusting to variations in hardware resources without the cost of a recompilation.

3.2 Targeting Changing Resources

The variety and variability of hardware complicates code generation for individual components. Unlike existing systems, a compiler for heterogeneous systems must generate code for diverse processors during a single compilation, which not only requires flexible instruction selection but also flexible optimization selection. The compiler must choose the optimizations that suit the target processor. When the programming model does not match the model of parallelism implemented by the hardware, the compiler must adapt the code to the hardware's model. One of heterogeneity's main benefit is that it should eliminate most of the need for this adaptation, but some subtasks may still benefit when no appropriate hardware is available or the appropriate hardware is busy.

IR supports code generation for diverse hardware
A compiler transforms a program through a series of representations from source code to object code. At each step the intermediate representation resembles the source language less and the target machine more. The final step, selection of object code instructions, is facilitated by an intermediate representation that resembles the target instruction set. The more accurately the IR reflects the hardware, the greater the opportunity for optimization. On the other hand, including more hardware specific detail in the intermediate representation decreases its generality. All of the surveyed systems, except SUIF, do a source-to-source translation and leave the final code generation to the native compilers thus avoiding code generation issues. These compilers are losing the benefit of intertwining their high-level optimizations with machine level optimizations. While low- and high-level optimizations may be interleaved in SUIF, it targets only RISC-like processors.

Compiler accepts an extensible description of the target
Another concern for generating efficient code is knowing the details of the target hardware. Hardware features such as the number of registers, number of functional units, memory access latencies, and cache line size are important concerns during optimization. Even high-level optimizations can benefit from exploiting these

[1] Language subset.
[2] Preprocesses Fortran with f2c.
[3] Includes Fancy, Classy, Cagey, Racy, Easy.
[4] Has Pattern Matching Language for Manipulating IR.
[5] Single IR has two levels of abstraction.
[6] Intraprocedural.
[7] Used for both analysis and optimization.
[8] Only for D System compiler.
[9] Relies on KAP for many optimizations.

features. The variety of hardware in a heterogeneous system precludes embedding hardware knowledge in the compiler. Instead, we expect to provide target descriptions to the compiler. The Memory Compiler, which is part of ParaScope and is only for uniprocessors, uses hardware parameters such as latency, but none of the compilers for parallel machine accept hardware parameters as input.

Compiler detects/accepts programming model
In order to assign code to a processor with the appropriate model of parallelism, the compiler must know the model of parallelism used by the programmer. Programmers could annotate programs with this information, and analysis techniques might be able to detect the model of parallelism. For example, a simple analysis routine might use the source language constructs that appear in a section of code to estimate its model of parallelism. None of the surveyed systems bother to determine the model of parallelism used by the source program because they assume the input program is one of a small set of models. For example, Polaris and SUIF assume a sequential model, and ParaScope assumes the source program is either sequential or data parallel.

Compiler converts programming models with user assistance
Because of the variability of resources in a heterogeneous system, a compiler must have the ability to target code that uses one model of parallelism for a machine that implements a different model of parallelism. Thus, the compiler must convert, to some extent, the model of parallelism that a subtask uses. Extensive effort has gone into developing methods for converting sequential programs into parallel programs, and some forms of parallel code can be readily converted to other forms such as running data parallel programs on control parallel hardware. All the compilers in our survey transform programs to execute on a different model of parallelism. For example, SUIF exclusively convert sequential programs to a parallel model, and though VFCS inputs and outputs data parallel programs, its input model (SPMD) is more tightly coupled than its output model (Message-Passing MIMD). Collectively, the compilers in our survey represent the state of the art in automatic parallelization (*i.e.*, model conversion), and their techniques should be included in a compiler for heterogeneous systems. Yet, automatic techniques have had limited success because compilers must make conservative assumptions. Parafrase-2 and ParaScope address this issue with their interactive interface that allows programmers to guide code transformation. Unfortunately, for heterogeneous systems, this approach would require programmers to edit their programs each time the system's configuration changes. With interactive editors, the programmer's deep understanding of a program remains implicit. Instead programmers should convey their insights about the program to the compiler and allow the compiler to determine how to use these insights to improve optimization. In other words, the programmer should annotate the program with knowledge that current analyses cannot discover, but that enables optimizations and transformations. This approach allows the same facts to be used in different ways, depending on the target.

Annotating source programs with additional semantic knowledge is appropriate only when the algorithm changes very slightly for a new target. However, sometimes a change in the target requires a radical change in the algorithm to obtain good performance. If an algorithm extensively exploits the features of a processor, it is unlikely to be efficient on a substantially different processor. The two simplest solutions are, either only write generic algorithms that have mediocre performance on all processors, or rewrite the algorithm when the target changes, which is what currently happens. Neither of these solutions is acceptable for heterogeneous systems. Instead, the compiler should manage multiple implementations of a routine[10]. Programmers would not have to write alternative versions but probably will for routines that are critical to performance. None of the compilers in our survey support multiple versions of a routine.

3.3 Adjusting optimization to suit a target

Compilers have a set of tools that they use to tailor code to exploit unique features of a specific processor and to adjust the code's model of parallelism. These tools (analyses, optimizations, and transformations) give a

[10] This approach is related to our earlier discussion (see Section 3.1) about compilers generating alternative object code modules but differs in that here the programmer specifies alternatives. Each of these capabilities is useful independent of the other.

compiler its power to understand and alter a program. The strategy for using these tools (*i.e.*, ordering and parameterization) depends upon the target. Because current compilers have a limited number of targets, their analyses, optimizations, and transformations are applied in a fixed order, or else controlled by the user. A compiler for heterogeneous systems must target a variety of hardware within a single compilation, and therefore must be able to alter its strategy according to the particular hardware. Moreover, because heterogeneous systems have variable configurations, new analyses, optimizations, and transformations may need to be added. Hence, a compiler for heterogeneous systems should encode analyses, optimizations, and transformations in a form that facilitates reordering and addition.

Modular analyses, optimizations, and transformations
One implication of needing to reorder analyses, optimizations, and transformations as well as include new ones is that they should be modular. Parafrase-2, Sage++, and SUIF break optimizations into individual passes which communicate through the IR. This approach to modularity works well if the entire program needs the same ordering and is destined for the same model of parallelism. Because optimization strategies for subtasks vary depending on the target processor and a heterogeneous system has a variety of targets, the compiler must also be able to apply an analysis, optimization, or transformation to individual sections of code. Polaris supports this capability directly; passes may call the bodies of other passes as needed. ParaScope, Parafrase-2 and to some extent VFCS provide this capability through their interactive interface, but none of the compilers automatically select optimizations based on a varying model of parallelism and the hardware features of the target.

Compiler maintains consistency of analysis data
Though the compilers in our survey have modular implementations of their analyses, optimizations, and transformations, most of them still have severe ordering constraints. Despite their modularity, the passes have implicit ordering concerns that compiler developers must manage. SUIF, for example, divides its analyses, optimizations, and transformations into reorderable passes where each pass may modify the IR and annotate it with information for future passes. No mechanism exists to ensure that annotations are up-to-date when used, or even available when needed. Compiler developers must intimately know the annotations produced and consumed by each pass before they can assign an order. A fixed ordering of optimizations works when a compilation has a single target but has drawbacks even for a single target. Compilers for heterogeneous systems must vary the order of optimizations and transformations to suit a variety of targets. Polaris supports incremental update of flow information (and data dependence information is in progress). ParaScope can identify when analysis data is not current, but incremental update is the responsibility of individual transformations. Ideally, optimizations and transformations would be written in such a way that the compiler infrastructure would handle ensuring that the necessary analysis data is accurate. Not only would this prevent errors, but it would greatly simplify the addition of new optimizations and transformations.

3.4 Allowing Programming Models and Languages to Evolve

Languages typically evolve to support new hardware features, and the variability in heterogeneous systems dictates that their compilers should support changes in source languages. For example, HPF extends Fortran with structured comments and a few constructs to allow programs to specify SPMD parallelism. A compiler needs two capabilities to support evolving languages. The first capability is already common: a clean break between the front and back ends of the compiler. The second capability is much harder: despite the separation, the front end must still pass a semantic description of new language features to the back end.

IR hides source language from back end
The separation of front and back ends of a compiler protects the analyses, optimizations, and transformations in the back end from details of the source language's syntax. This separation of the front and back ends is realized by limiting their interaction to an intermediate representation. To the extent that the IR is unaffected by changes in the source language, the back end is insulated. Unfortunately, ParaScope, Polaris, Sage++, SUIF, and VFCS use an AST intermediate representation, which inherently captures the syntax of the source language. However, SUIF attempts to overcome the limitations of ASTs by immediately compiling source

language constructs it considers unimportant to a RISC-like IR nodes. Parafrase-2 uses a HTG which does not necessarily represent the syntax of the source language, and therefore can hide the source language.

IR is extensible

To pass a semantic description of new source language features through the intermediate representation, the IR must be extended. Some simple changes to a language (***e.g.***, a new loop construct) may be expressible in terms of the existing IR, but others (***e.g.***, adding message passing to C) require new IR nodes. Sage++, SUIF, and Polaris allow extension of their respective IRs through object-oriented data structures. Their IRs can then be extended to include new features of an evolving language or to reuse parts of the IR for a completely different language. Note that when a node is added to the IR, most likely new or enhanced optimizations and transformations will also be needed to process that node.

3.5 Summary

This section has examined the impact of our four goals on a compiler for heterogeneous systems (see Table 4). Though the impact is significant, it is important to note that the impact is primarily on a compiler's software architecture, and much less so on the core compiler technology. For example, we have not discussed specific, new optimizations or transformations, but we have considered how existing ones must be coordinated for heterogeneity. The basic technology such as parsers, analyses, and transformations (that is in all of our surveyed compilers) remains important and largely unchanged for heterogeneity; it is the way they work together that must change. We have also seen that the existing compilers often offer partial solutions, but they lack the flexibility needed by heterogeneous systems because homogeneous systems did not require it.

Table 4. Compiler Goals for Heterogeneous Systems.

1. Exploiting available resources:
   - Compiler generates multiple object code modules for different targets to support load balancing and maximize throughput.
   - Compiler communicates with Run-time environment.
2. Targeting changing resources:
   - IR supports code generation for diverse hardware.
   - Compiler accepts an extensible description of the target.
   - Compiler detects (or accepts from programmer) the source code's programming model.
   - Compiler accepts user assistance in converting code from one programming model to another.
3. Adjusting optimization to suit a target:
   - Modular analyses, optimizations, and transformations.
   - Compiler maintains consistency of analysis data.
4. Allowing programming models and languages to evolve:
   - IR hides source language from back end.
   - IR is extensible (via new constructs or annotations).

All of the goals of a compiler for a heterogeneous system still have significant unmet challenges. The nearest to being met is the fourth goal: allowing programming models and languages to evolve. Yet, meeting this goal requires the combination of IR features from Parafrase-2 with those from Polaris or SUIF, and even these could be improved upon to support a wider array of hardware or ease the addition of new analyses, optimizations, and transformations. Therefore, in Section 4 we propose a compiler architecture with sufficient flexibility to support heterogeneity.

4 Proposed Compiler Architecture

Section 3 shows that existing compilers lack the flexibility necessary for heterogeneous systems. This lack of flexibility is not the fault of the compilers, but rather a by product of their goals. These compilers were designed to target only a few models of parallelism, and then only for homogeneous targets. Because heterogeneous systems have different, more rigorous requirements, the simplifications used for homogeneous systems are no longer appropriate. This section presents a compiler architecture with the flexibility necessitated by heterogeneity.

4.1 Architecture Overview

Figure 1 shows a high-level diagram of our proposed compiler architecture. It consists of three major pieces: Translation Director, Compilation Library, and Persistent Store. The translation director provides the intelligence for deciding upon appropriate program partitioning and planning optimization strategies. The translation director uses the compiler tools in the compilation library to carry out its plans. The persistent store holds a variety of information for use by the compiler and other tools.

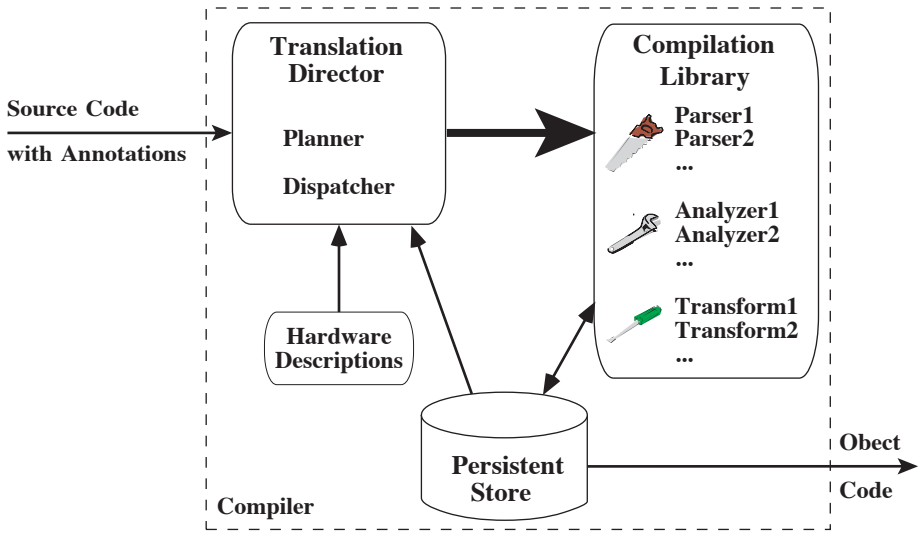

Fig. 1. Architecture of a flexible compiler for heterogeneous systems.

Translation director
An noted in Section 3, heterogeneity requires more flexibility in the compiler. Some of this flexibility needs to be in the form of additional intelligence within the compiler, so it can reason about how to exploit variety and adapt to variation. For example, given a subtask the compiler must decide for which target(s) to generate code (see Section 3.1) and must plan the optimization strategies which depend upon the specific target (see Section 3.2 and 3.3). These decisions are implicit for homogeneous processors but must be made dynamically for heterogeneous systems. The appropriate target and translation plan for a subtask depends not only on the subtask itself, but also on the target and plans for other subtasks. The intelligence needed to handle these complexities is collected into the translation director, which is responsible for making these decisions and capturing them in the form of a translation plan. The entire program has a translation plan, and when it is decomposed into subtasks, each subtask has its own translation plan. The translation director is also responsible for controlling the execution of plans by dispatching routines from the compilation library.

Compilation library
Section 3.3 points out that the variety inherent in heterogeneity demands more flexibility in the ordering of analyses, optimizations, and transformation. The compilation library serves as a tool box or repository for compiler components (*e.g.*, parsers, optimizations, and code generators). The compilation library ensures flexibility by explicitly maintaining the information the translation director needs to reason about a library member. This information can be thought of in general terms as preconditions and postconditions. A transformation may have, for example, applicability criteria and required analysis data as preconditions, and a description of its impact on the program and its affect on analysis data as its postconditions.

Annotations
Section 3.2 discusses how programmers could augment the compiler's analysis data with annotations. Annotations enable additional optimization. Given the limited success of automatic parallelization, we believe that programmer supplied information is essential for generating efficient code, especially when the model of parallelism used in the source code does not match the one the hardware implements. Annotations also provide an alternative to language extensions for accessing new hardware features, though for code with a long lifetime, language extensions may be preferable.

Persistent store
Though current compilers generally use the file system provided by the operating system, compilers for heterogeneous systems benefit from a more structured stable storage mechanism for several reasons:

- As in other planning systems, the translation plan produced by the translation director may need to be corrected during compilation. If the plan is updated, the compiler may need to roll back some of its optimizations and transformations. A persistent store facilitates the preservation of intermediate versions of the IR as well as recording of modifications to the IR (*i.e.*, reverse transforms).
- Having the intermediate representation on disk is also a first step towards incremental compilation. Compilers for heterogeneous systems are likely to use incremental compilation to offset the compile time cost of handling multiple versions of source code and object code as well as the dynamic planning of optimization.
- Managing source files for large software projects is already cumbersome for programmers, and heterogeneity exacerbates this problem by requiring additional information such as alternative object code modules and hardware target descriptions. With a persistent store, the compiler can manage these files and protect against version problems.
- The persistent store allows tools outside the compiler to communicate with the compiler and access its internal structures (*e.g.*, the intermediate representation) in a controlled manner. Examples of these tools are the linker, the run time system, debuggers, and performance tools. The linker is particularly important because it is the main user of the compiler's output and because it must be able to select from among several object code modules that have equivalent functionality.

The persistent store holds an assortment of items such as the translation plan, a description of each target's hardware features (*i.e.*, Hardware Descriptions), the intermediate representation with annotations, object code, and reverse transforms for debuggers.

Linker/run time system
To support variability of resources in heterogeneous systems, the compiler should maintain alternate versions of object code so that the linker can handle minor variations itself (see Section 3.1). This approach, however, implies a closer association between the compiler and the linker. Not only does the linker have to finish what is normally the compiler's job, but the linker will have to notify the compiler when the existing object code modules are no longer adequate.

4.2 Compilation Process

This section describes the compilation process, which is controlled by the translation director by invoking routines in the compilation library to carry out its plans. The compiler accepts three major inputs: the source

code, descriptions of targets, and (optionally) program annotations. The descriptions of a heterogeneous machine's component processors are likely to be maintained centrally and therefore are an implicit input to the compiler. The programmer may, however, indicate that only a subset of the available processors should be used. Programmers may convey information not captured by the source language through annotations or possibly using structured comments or pragmas. Annotations are created by an *Annotation Tool*, which is separate from the compiler.

1. When the compiler is invoked, the translation director immediately takes control because it is responsible for coordinating compilation. The translation director begins by determining the program's source language(s), examining the annotations, and finding the target descriptions.
2. After identifying source language(s), target descriptions, and annotations, the translation director builds an initial plan. This plan is not detailed but has enough information to evaluate the feasibility of compiling the given program for the indicated component processors. Because the IR is extensible, feasibility is not simply a matter of having the correct parsers and code generators. Some parsers may generate IR nodes that cannot (directly or indirectly) be translated into code for any of the specified targets. The compiler, therefore, checks the feasibility of transforming the IR produced by the parsers to a form that the code generators can consume.
3. For the remainder of the compilation, the translation director carries out and improves the translation plan. The first step in every translation plan is to invoke the appropriate parsers to build an intermediate representation and place it in the persistent store.
4. After the program is parsed, the compiler extracts models of parallelism used within the program. The translation director uses the annotations and may also invoke analyses, to isolate models of parallelism. Different sections of the program may use different models, so this process finds a combination of user-specified and natural boundaries for decomposing the program into subtasks.

   Source languages themselves follow a model of parallelism, and when a program matches that model it compiles to efficient code. When a program uses a different model or a more specialized model (***e.g.***, wave front parallelism in a loop parallel language), compilers for heterogeneous machines are no more able to recognize this different model than are current compilers. So at least initially, annotations will be important for identifying alternative models of parallelism.
5. When it has enough information, the translation director divides the program into subtasks, where a subtask executes as a single unit on a single machine of the heterogeneous system. The translation director uses the description of the heterogeneous system's configuration from the target information store to guide its selection of subtasks. Note that though the mapping of subtasks is not performed until link or run time, each subtask may initially be associated with one or more component processors to enable specific optimizations.
6. For each subtask, the translation director builds a more detailed translation plan. The translation director will probably start with an initial plan and refine it to better suit the individual needs of the code. The advantage of actually formulating a plan is that the translation director can use the preconditions and postconditions (maintained by the compilation library) to group optimizations and transformations so that reanalysis is minimized.
7. During compilation, the translation director may decide that its attempts to translate a section of code is not yielding the expected performance benefits. The translation director may further decide to seek greater benefits by reversing an earlier decision. This reversal requires rolling back some number of optimizations and transformations. For example, when compiling a data parallel subtask, the translation director may discover or suspect that its original data decomposition can be improved. The translation director may choose to rollback the transformations performed on the subtask after the decomposition was chosen.
8. The last step of the translation director's plan is to invoke an appropriate object code generator for each subtask-machine pair. The resulting object code is stored in the persistent object store.

4.3 Extending the Intermediate Representation

An IR intended to support heterogeneity should avoid language and hardware dependencies, so that it is unaffected by minor changes in source languages or hardware. Nevertheless, changes to the IR are inevitable. This section roughly describes the impact of extending the IR, but it is necessarily vague because we have not discussed any one IR in sufficient detail.

Creating new intermediate representation nodes
Occasionally, the intermediate representation will be extended with new nodes that represent new models of parallelism, new language constructs, or new machine features. Besides defining a data structure for the new node itself, compiler developers must modify or create routines in the compilation library, because these routines must produce and consume an IR node in order for it to be useful. On the other hand, many library members will be uninterested in the presence of the node and so should not have to be modified. For example, dead code elimination is largely uninterested in the function represented by an IR node; it simply needs to know its parameters and result. Adding a new arithmetic operator should not impact dead code elimination.

As noted in Section 4.2, a compiler that expects its IR to evolve cannot be certain that it can compile all of its source languages to all of its targets. The translation director accounts for this uncertainty by verifying the feasibility of its plans. Uncertainty arises because it is possible that given a language, no series of parsers, optimizations, and transformations can generate a particular IR node, or conversely given a target, no series of optimizations, transformations, and code generators can translate the new node to object code for the target. For example, when compiling a program in a sequential language, a node representing a parallel action can only occur in the IR if a transformation (or series of transformations) exists to convert the program to use this parallel action.

Creating new annotations
An alternative to extending the IR is creating a new type of annotation. Annotations and IR nodes are closely related and can often represent the same concept. For example, the compiler could represent a parallel loop with a parallel loop node or with a sequential loop node that is tagged as parallel. Generally, IR nodes represent information expressed in the source language, and annotations represent information gleaned from other sources such as analyses and the annotation tool.

A compiler developer that extends a compiler to handle a new concept (*i.e.*, new language, model, or hardware feature) may choose which approach (*i.e.*, new IR node or new annotation type) seems best. Creating a new type of annotation is preferable for experimentation because the annotation tool can be easily extended to generate the annotation. Then only a consumer of the annotation needs to be written for the compilation library. When the new concept is refined, the developer may wish to replace the annotation type with a new IR node. The developer will also have to add routines to the compilation library that generate the new node.

5 Summary and Conclusions

Compiling for heterogeneous systems is a challenging task because of the complexity of efficiently managing multiple languages, targets and programming models in a dynamic environment.

In this paper, we reviewed six state-of-the-art high performance optimizing compilers with respect to their programming models, organizations, intermediate representations, analyses and optimizations. We identified the areas in which these compilers lack the necessary flexibility to be successful in a heterogeneous environment and the areas from which the existing technology can be borrowed.

We presented four important goals of an ideal compiler for heterogeneous environment, namely, exploiting available resources, targeting changing resources, adjusting optimization to suit a target, and allowing programming models and languages to evolve. In light of these requirements, we concluded that a more flexible, open compiler architecture is needed which promotes reuse of existing software, enables the addition of new programming paradigms and targets, and makes efficient use of resources. We proposed an architecture which resembles a reflective planning system composed of several interacting components rather than a

passive, monolithic piece of software. This architecture is extensible, *i.e.*, various supporting tools for editing, debugging, performance analysis, etc. can be added as needed.

This paper offers a first step in understanding what an ambitious compiler for achieving high performance on heterogeneous systems will entail.

Acknowledgements: We want to thank the development teams of each compiler in our survey for their comments and feedback, especially John Grout, Jay Hoeflinger, David Padua, Constantine Polychronopoulos, Nicholas Stavrakos, Chau-Wen Tseng, Robert Wilson, and Hans Zima.

References

[AALL93] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[B+93] F. Bodin et al. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[B+94] F. Bodin et al. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Second Object-Oriented Numerics Conference*, 1994.

[B+95] W. Blume et al. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[BE94] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. CSRD 1345, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1994.

[BE95a] Bill Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[BE95b] William Blume and Rudolf Eigenmann. Symbolic range propagation. *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[BHMM94] D. Brown, S. Hackstadt, A. Malony, and B. Mohr. Program analysis environments for parallel language systems: the TAU environment. In *Proceedings of the 2nd Workshop on Environments and Tools For Parallel Scientific Computing*, pages 162–171, Townsend, Tennessee, May 1994.

[BKK94] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Montreal, August 1994.

[BL94] R. Butler and E. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[BPMG94] F. Bodin, T. Priol, P. Mehrotra, and D. Gannon. Directions in parallel programming: HPF, shared virtual memory and object parallelism in pC++. Technical Report 94-54, ICASE, June 1994.

[C+93] K. Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[CMRZ94] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary appications: Integrating task and data parallelism. Technical Report 94-18, ICASE, March 1994.

[CMZ92a] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[CMZ92b] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, 1992.

[CPZ95] B. Chapman, M. Pantano, and H. Zima. Supercompilers for massively parallel architectures. In *Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '95)*, pages 315–322, Aizu-Wakamatsu, Fukushima, Japan, March 1995.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[F+90] G. Fox et al. Fortran D language specification. Technical Report TR90-141, Rice University, December 1990.

[F+94] K. Faigin et al. The polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[Fah94] T. Fahringer. Using the P^3T to guide the parallelization and optimization effort under the Vienna Fortran compilation system. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, May 1994.

[FGMS93] S. Feldman, D. Gay, M. Maimone, and N. Schryer. A Fortran-to-C converter. Computing Science 149, AT&T Bell Laboratories, March 1993.

[FZ93] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, July 1993.

[GP94] M. B. Girkar and C. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5), 1994.

[Gro95a] J. Grout. Inline expansion for the polaris research compiler. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[Gro95b] J. Grout. Personal communication, September 1995.

[GY93] A. Ghafoor and J. Yang. A distributed heterogeneous supercomputing management system. *Computer*, 26(6):78–86, June 1993.

[HHM+94] M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale, and H. Zima. Runtime support for data parallel tasks. Technical Report 94-26, ICASE, April 1994.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR91-149, Rice University, Jan. 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HMA95] M. Hall, B. Murphy, and S. Amarasinghe. Interprocedural analysis for parallelization. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[KMCP93] A. E. Klietz, A. V. Malevsky, and K. Chin-Purcell. A case study in metacomputing: Distributed simulations of mixing in turbulent convection. In *Workshop on Heterogeneous Processing*, pages 101–106, April 1993.

[KMT93] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience*, 5(7):575–602, October 1993.

[KPSW93] A. Khokhar, V. Prasanna, M. Shaaban, and C. Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, June 1993.

[M+94] A. Malony et al. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium*, 1994.

[MBM94] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *Proceedings of CONPAR 94 - VAPP VI*, University of Linz, Austria, September 1994. LNCS 854.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard, v1.0. Technical report, University of Tennessee, May 1994.

[P+89] C. Polychronopoulos et al. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1), 1989.

[P+93] D. A. Padua et al. Polaris: A new-generation parallelizing compiler for MPPs. Technical Report CSRD-1306, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1993.

[Pau95] D. A. Pauda. Personal communication, September 1995.

[PE95] W. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, July 1995.

[SC92] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, June 1992.

[SCMB90] J. Saltz, K. Crowely, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and P. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[Sta94] Stanford Compiler Group. The SUIF library. Technical report, Stanford University, 1994.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[Tur93] L. H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical Report MSSU-EIRS-ERC-93-2, NSF Engineering Research Center, Mississippi State University, February 1993.

[Tur95] L. H. Turcotte. Cluster computing. In Albert Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 26. McGraw-Hill, October 1995.

[W+89] C. Weems et al. The image understanding architecture. *International Journal of Computer Vision*, 2(3):251–282, 1989.

[W+94] R. Wilson et al. The SUIF compiler system: A parallelizing and optimizing research compiler. *SIGPLAN*, 29(12), December 1994.

[WL91] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Y+94] S. Yang et al. High performance fortran interface to the parallel C++. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC93] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.

[ZCMM93] H. Zima, B. Chapman, H. Moritsch, and P. Mehrotra. Dynamic data distributions in Vienna Fortran. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[Zim95] H. Zima. Personal communication, September 1995.