Compiler Architectures for Heterogeneous Systems

Kathryn S. McKinley, Sharad K. Singhai, Glen E. Weaver, Charles C. Weems

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

Abstract. Heterogeneous parallel systems incorporate diverse models of parallelism within a single machine or across machines and are better suited for diverse applications [25, 43, 30]. 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 handling this complexity. We identify four goals for extending compilers to manage 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 existing compilers so much as a restructuring of a compiler's software architecture to increase its flexibility. We examine six important parallelizing compilers to identify both existing solutions and where new technology is needed.

1 Introduction

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. By incorporating multiple models of parallelism within one machine (e.g., Meiko CS-2, IBM SP-2, and IUA [42]) or across machines, creating a virtual machine (e.g., PVM [38], p4 [9], and MPI [28]), heterogeneous systems provide consistent high performance.

Heterogeneous processing [36, 40, 41, 24, 18] is the well-orchestrated use of heterogeneous hardware to execute a single application [24]. When an application encompasses subtasks that employ different models of parallelism, the

We are designing a new compiler architecture to meet the needs of heterogeneity. Another paper [27] has a preliminary description of our design along with an expanded survey section.

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.

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)[25]. The four machines form a single virtual machine, and the authors leverage the strengths of each machine for different tasks to achieve high performance.

Their experience illustrates that although heterogeneous processing offers improved performance, it increases the complexity of software development. The complexity arises from three important features of heterogeneity: variety, variability, and 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. Variety also opens up opportunities to trade local performance for overall performance. Second, virtual heterogeneous systems experience ***variability*** as their makeup changes from site to site or day to day or based on load. This variability of hardware resources requires rapid adaptation of programs to new configurations at compile and run time. Furthermore, variability deters programmers from using machine specific code (or languages) to improve performance. Third, heterogeneous systems can achieve ***high performance***. If the execution time of a program does not matter, it could run on a homogeneous processor with less trouble. The demand for high performance precludes simple solutions such as adding layers of abstraction that obscure heterogeneity.

Compilers for heterogeneous systems

Developing software for heterogeneous systems would be overwhelming if each application needed to handle the complexity caused by variety, variability, and high performance. In Kleitz et al., they hand-parallelized each task specifically for its target machine in that machine's unique language dialect. If the hardware configuration changes, they must rewrite parts of the program. Instead of manually modifying programs, the variability of heterogeneous systems should be automatically handled at least in part by a compiler. With certain modifications to their software architecture, compilers can use transformations[1] to adjust a program to execute efficiently on a heterogeneous system.

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***. (Sect. 3 explains why these goals are important.) Meeting these goals does not require changes to the individual pieces of a compiler so much as a restructuring of the compiler's software architecture to increase its ***flexibility***. Current compilers, including retargetable compilers, tightly couple the compiler with both the source language and the target machine. 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.

[1] For brevity, "transformations" refers to both optimizations and transformations.

Overview

This paper surveys optimizing compilers and compiler frameworks for parallel machines and examines their potential contributions to a compiler for heterogeneous systems. In this section, we have motivated compiler support for heterogeneous systems, and distilled the impact of heterogeneity into four goals for an ideal compiler: exploiting available resources, targeting changing resources, adjusting optimization to suit a target, and allowing programming models and languages to evolve. Sect. 2 reviews six well-known parallelizing compilers. Sect. 3 discusses the applicability of techniques found in our survey to heterogeneity and the impact of heterogeneity on the overall architecture of the compiler. For each goal and compiler, we identify where existing technology can be adapted to meet the needs of heterogeneity.

We find that heterogeneity's variety and variability of available resources requires a compiler architecture that is much more flexible than current ones.

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. 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, programming model, organization, intermediate representation, optimizations and transformations of the six systems and summarizes them in Table 1.[2]

System Overviews and Goals

Parafrase-2 is a source-to-source translator from the University of Illinois [19, 33]. Its design goal is to investigate compilation support for multiple languages and target architectures. It easily adapts to new language extensions because its IR emphasizes data and control dependences, rather than language syntax.

Rice University's ParaScope is an interactive parallel programming environment built around a source-to-source translator [12, 23]. It provides sophisticated global program analyses and a rich set of program transformations. Here we concentrate on the D System which is a specialized version of ParaScope for Fortran-D [17]. The output of the D System is an efficient message-passing distributed memory program [21, 22, 39].

Polaris is an optimizing source-to-source translator from the University of Illinois [4, 31, 15]. The authors have two major goals for Polaris: to automatically parallelize sequential programs and to be a near-production quality compiler. The authors focus on parallelization for shared memory machines. Polaris is written in C++ and compiles Fortran 77. Programmers may convey extra information, such as parallelism, to the compiler by embedding assertions in source code.

[2] We give more detailed descriptions in [27].

Properties	**Parafrase-2**	**ParaScope/ D System**	**Polaris**	**Sage++/ pC++**	**SUIF**	**VFCS**
General						
Goals	Multiple Languages and Target Architectures, Extensibility	Automatic and Interactive Parallelization	Production Quality Translator for Automatic Parallelization	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[3], Fortran D	Fortran 77	C, C++, Fortran 77, Fortran 90, pC++	C, Fortran 77[4]	Fortran 77, Fortran 90[3], HPF, Vienna Fortran
Code Generation	Tuples	Tuples	—	—	MIPS Assembly	—
Programming Model						
Input	Sequential or Control Parallel	Sequential or Data Parallel	Sequential	*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
Intermediate Representations	HTG, Linear Tuples	AST	AST[5]	AST	Hybrid of AST and Linear Tuples[6]	AST

SM	Shared Memory	**DSM**	Distributed Shared Memory	√	Yes	—	No or None
DM	Distributed Memory	**HTG**	Hierarchical Task Graph	√+	Exceptional Implementation	NA	Not Applicable

[3] Language subset. [4] Preprocesses Fortran with f2c. [5] Has pattern matching language for manipulating IR.
[6] Single IR has two levels of abstraction.

Table 1. Comparison table for surveyed systems.

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

Properties	**Parafrase-2**	**ParaScope/ D System**	**Polaris**	**Sage++/ pC++**	**SUIF**	**VFCS**
Analyses						
Data Dependence	√	√	√	√	√	√
Control Depend.	√+	√	√	—	√	—
Symbolic[7]	√+	√+	√	—	√	√
Interprocedural	Alias, MOD, REF, Constant Propagation	Alias, MOD, REF, Constant Propagation	Inlining (for analysis), Constant Propagation	—	MOD, REF, GEN, KILL, Constant Propagation, Array Summary, Array Reshape, Reductions[8], Induction Variables, Cloning[8]	USE, DEF, Alias, Overlap, Constant Propagation, Communication, Dynamic Distribution
Incremental	—	√	√	—	—	—
Optimizations and Transformations						
Traditional	√	√	√	—	√+	√
Loop	√	√	√	√	√	√
Comm/Sync	—	√[9]	√	—	√	√+
Interprocedural	Inlining	Inlining, Cloning	Inlining, Cloning	—	Parallelization, Data Privatization, Inlining,Cloning, Reductions	Inlining, Cloning
Data Partitioning	—	√	*In Progress*	—	√	√
Task Partitioning	√	—	—	—	—	—
Applicability Criteria	√	√	√	—	√	√
Profit. Criteria	Queries User	Queries User	Fixed for Arch.	—	Fixed for Arch.	Static Measure

[7] Intraprocedural. [8] Used for both analysis and optimization. [9] Only for D System compiler.

Sage++ is a toolkit for building source-to-source translators from Indiana University [6]. The authors foresee optimizing translators, simulation of language extensions, language preprocessors, and code instrumentation as possible applications of Sage++ [7, 46, 26, 5]. Sage++ is written in C++ and provides parses for C, C++, pC++, Fortran 77, and Fortran 90. Because Sage++ is a toolkit instead of a compiler, it is not limited to particular hardware architectures.

SUIF from Stanford University is a compiler framework that can be used as either a source-to-C translator or a native code compiler [44, 1, 20, 37]. SUIF is designed to study parallelization for both shared memory and distributed shared memory machines as well as uniprocessor optimizations. SUIF accepts source code written in either Fortran 77 or C, however a modified version of `f2c` [16] is used to convert Fortran code to C code.

The Vienna Fortran Compilation System (VFCS) from the University of Vienna is an interactive, source-to-source translator for Vienna Fortran [10, 11, 48, 49]. VFCS is based upon the data parallel model of computation with the Single-Program-Multiple-Data (SPMD) paradigm. VFCS outputs explicitly parallel, SPMD programs in message passing languages, Intel features, PARMACS, and MPI.

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 a parallelizing compiler (e.g., data dependence). All the systems parse some variation of Fortran and half of them also handle C, the traditional languages for high performance computing. Except for Sage++, the systems in our survey can operate as source-to-source translators, and Sage++ facilitates the construction of source-to-source translators. In addition, SUIF compiles directly into assembly code for the MIPS family of microprocessors, and Parafrase-2 and ParaScope generate tuples.

Programming model

SUIF accepts only sequential C and Fortran 77 programs, and therefore must extract all 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. VFCS accepts data parallel languages and requires that programmers supply the data distribution and the assignment of computation to processors, following the ***owner-computes*** rule as in ParaScope. Parafrase-2 inputs Cedar Fortran which offers vector, loop, and task parallelism.

Most of the compilers in our survey generate data parallel programs, but Parafrase-2 produces control parallel code 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 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 is unique in that it exploits task parallelism as well as loop parallelism.

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's AST is unique because it represents only select language constructs at a high-level; the remaining constructs are represented by nodes that resemble RISC instructions. These low-level nodes are also used in SUIF's linear tuple representation. Parafrase-2 uses the hierarchical task graph (HTG) representation instead of an AST. HTGs elucidate 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 analyses 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. Instead of using traditional dependence analysis, Polaris builds symbolic lower and upper bounds for each variable reference and propagates these ranges throughout the program using symbolic execution. Polaris' range test then uses these ranges to disprove dependences[3].

Polaris, Parafrase-2, ParaScope, and SUIF perform control dependence analysis, albeit in a flow-insensitive manner. Parafrase-2 has additional analyses to eliminate redundant control dependences.

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 [35]. Parafrase-2, ParaScope, Polaris, SUIF, and VFCS provide interprocedural analysis. Polaris recently incorporated interprocedural symbolic constant propagation [32]. Parafrase-2, ParaScope, and VFCS [47] perform flow-insensitive interprocedural analysis by summarizing where variables are referenced or modified. SUIF's FIAT [20] tool provides a powerful framework for both flow-insensitive and flow-sensitive analysis [20].

Optimizations and Transformations

The organization of analyses and transformations varies among the systems. SUIF has a flexible organization, with each analysis and transformation coded as an independent pass and the sole means of communication between passes being the annotations attached to the IR. Polaris also organizes its analyses and transformations 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 in both systems through command line parameters. All of the systems support batch compilation and optimization. However, Parafrase-2, ParaScope, and VFCS also provide a graphical interface that enables users to affect their transformation ordering. 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 encourages more extensive optimization.

Transformations performed by uniprocessor compilers are termed ***traditional***. Except Sage++, all the systems provide traditional optimizations. In addition, Polaris and SUIF perform array privatization and reductions [34]. Because SUIF generates native code, it also includes low-level optimizations such as register allocation.

All six systems provide loop transformations. ParaScope has a large set of loop transformations. SUIF too has a wide assortment of traditional and loop transformations including unimodular loop transformations (i.e., interchange, reversal, and skewing) [45].

All systems except Sage++, include inlining as one of their interprocedural optimizations. ParaScope, Parafrase-2, SUIF and VFCS also perform cloning. SUIF exploits its strong interprocedural analyses to provide data privatization, reductions and parallelization.

Communication and synchronization transformations, though not always distinct from loop transformations, refer to the transformations specifically performed for distributed memory machines, like message vectorization, communication selection, message coalescing, message aggregation, and message pipelining. VFCS, designed exclusively for distributed memory machines, has a richer set of communication transformations 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 default computation partitioning mechanism for them is the ***owner computes*** rule and data partitioning is specified by programmers (recent work in ParaScope addresses automatic data partitioning [2]).

Parafrase-2 is unique in that it exploits ***control*** parallelism by partitioning programs into separate tasks. The other compilers use only data parallelism.

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 the respect that although it has a few loop transformations, it does not have any applicability criteria built in. Sage++ developers argue that in a preprocessor toolkit applicability should be defined by the compiler writer for individual implementations.

Though a transformation 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 transformations according to a predefined strategy. VFCS relies on static performance measurement by an external tool, P^3T, to determine profitability [13, 14].

Closely related to profitability is ordering criteria. Transformations applied in different orders can produce dramatically different results. In interactive mode, ParaScope and Parafrase-2 allow users to select any ordering of transformations. All support fixed transformation ordering via their command lines.

Interaction with users

Most of the surveyed compilers have additional tools to assist users in writing and understanding their parallel programs. ParaScope strives to provide a parallel programming environment, including an editor, debugger and an automatic data partitioner. Polaris allows programmers to provide instructions to the compiler through source code assertions. Sage++ provides a rich set of tools for pC++ named ***Tuning and Analysis Utilities***, TAU [8, 29]. TAU includes tools 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 a graphical user interface in Parafrase-2, ParaScope, Sage++, and VFCS which display various aspects of the compilation process in windows so the user can request more details or provide inputs to the compiler.

3 Criteria of a Compiler for Heterogeneous Systems

In Sect. 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 with respect to the compiler, and finding where technology from Sect. 2 is applicable or new technology is needed.

3.1 Exploiting Available Resources

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

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. None of the existing compilers provide this level of flexibility. All the compilers perform partitioning and mapping within the compiler.

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 transformations that dynamically decide their applicability at run-time. These transformations, along with delayed binding of subtasks to specific processors, increase communication between the compiler and the run time system. This approach can be adapted to accommodate variations in hardware resources without the cost of recompilation.

3.2 Targeting Changing Resources

The variety and variability of hardware complicates code generation for individual components. Unlike existing compilers, 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 transformation selection. The compiler must choose the transformations that suit the target processor.

IR supports code generation for diverse hardware

A compiler transforms a program through a series of representations from source code to object code. 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 IR decreases its generality. All of the surveyed systems, except SUIF, do a source-to-source translation and leave code generation to native compilers, thus avoiding code generation issues. These compilers lose the benefits of intertwining their high-level transformations with machine-level optimizations. SUIF's unique representation allows it to capture RISC hardware specific details (for most source language constructs) and still perform high-level transformations on the program.

Compiler accepts an extensible description of the target

Another concern for generating efficient code is exploiting details of the target hardware. Even high-level transformations can benefit from exploiting features such as the number of registers, memory access latencies, and cache line size. The variety of hardware in a heterogeneous system precludes embedding hardware knowledge in the compiler. Instead, there must be some way to provide target descriptions to the compiler. The Memoria compiler, which is part of ParaScope and is only for uniprocessors, uses hardware parameters such as latency, but none of the compilers for parallel machines accept hardware parameters as input. Memoria accepts only a limited hardware description, but this approach can be extended to accept a richer description.

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, or analysis techniques might be able to detect the model of parallelism. None of the surveyed systems automatically determine the source program's model of parallelism because they assume it is one of a small set of models. For example, Polaris and SUIF assume a sequential model, and ParaScope assumes the program is either sequential or data parallel. Thus, new technology is needed to both accept and extract the programming model.

Compiler converts programming models with user assistance

Because of the variability of resources in a heterogeneous system, a compiler must be able 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 code module uses. Extensive effort has gone into developing methods for converting sequential programs into parallel programs (i.e., automatic parallelization), and some forms of parallel code can be readily converted to other forms. All the compilers in our survey transform programs to execute on a different model of parallelism, and they represent the state of the art in automatic parallelization. Their techniques should be included in a compiler for heterogeneous systems. Yet, automatic techniques have had limited success because they make conservative assumptions. Parafrase-2 and ParaScope address this issue with an interactive interface that allows programmers to guide code transformation. Unfortunately for heterogeneous systems, this approach requires programmers to edit their programs each time the system's configuration changes because the programmer's deep understanding of a program remains implicit in the code. Instead programmers should convey their insights about the program to the compiler and allow it to determine how to apply these insights.

Annotating source programs with additional semantic knowledge is appropriate only when the algorithm changes slightly for a new target. Sometimes a change in the target requires a radical change in the algorithm to obtain good performance. Programmers currently have two choices: write generic algorithms with mediocre performance on all processors or rewrite the algorithm when the target changes. A compiler for heterogeneous systems should provide a third choice by managing multiple implementations of a routine.

3.3 Adjusting optimization to suit a target

Current compilers have a limited number of targets and therefore apply their analyses and transformations in a fixed order (or under user control). Because of the variety of hardware in heterogeneous systems, a compiler must be able to reorder transformations to suit a particular target. Moreover, because heterogeneous systems have variable configurations, new analyses and transformations may need to be added. Thus, a compiler for heterogeneous systems should encode analyses and transformations in a form that facilitates reordering and addition.

Modular analyses, optimizations, and transformations

One implication of needing to reorder analyses and transformations as well as include new ones is that they should be modular. Parafrase-2, Sage++, and SUIF break transformations 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 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 also provide this capability through their interactive interface. New technology should use the flexibility these systems provide to automatically adapt the ordering of transformations based on varying models of parallelism and a target's hardware features.

Compiler maintains consistency of analysis data

Though the compilers in our survey have modular implementations of their analyses and transformations, most of them still have strong ordering constraints. Ideally, transformations would be written such that the compiler infrastructure would manage these constraints by ensuring that necessary analysis data is accurate and up-to-date. Not only would this capability prevent errors, but it would also simplify the addition of new transformations. Polaris already supports incremental update of flow information. ParaScope can identify when analysis data is not current, but incremental update is the responsibility of individual transformations. Extensions of these techniques can simplify the compiler developer's task in ordering transformations.

3.4 Allowing Programming Models and Languages to Evolve

The inherent complexity of a compiler for heterogeneous systems along with the variety of targets favors a single compiler with multiple source languages and targets. Because languages typically evolve in response to new hardware capabilities and virtual machines allow the introduction of new hardware, a compiler should include two capabilities to support changes in the source languages it accepts. 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 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 protects the analyses and transformations in the back end from details of the source language's syntax. Compilers separate their front and back ends by limiting their interaction to an intermediate representation. To the extent that the IR is unaffected by changes in the language, the back end is insulated. Unfortunately, ParaScope, Polaris, Sage++, SUIF, and VFCS use an AST representation, which inherently captures the syntax of the source language. SUIF attempts to overcome the limitations of ASTs by immediately compiling language constructs it considers unimportant to RISC-like IR nodes. Parafrase-2 uses an HTG which does not necessarily represent the syntax of the source language, and can therefore hide it. Extending this approach of reducing source language syntax dependences in the IR can improve the separation of front and back ends.

IR is extensible

To pass a semantic description of new language features through the intermediate representation, the IR must be extended. Some simple changes to a

Table 2. 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 user) the code's programming model.
   - Compiler accepts user assistance in converting programming models.
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).

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++, Parafrase-2, SUIF, and Polaris allow extension of their respective IRs through object-oriented data structures. Their IRs can be extended to include new features of an evolving language or to reuse parts of the IR for a completely different language. Note that a new IR node may require new or enhanced transformations to process that node. Object-oriented features improve extensibility but they may not be sufficiently flexible for unanticipated extensions.

4 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 survey six state-of-the-art high-performance optimizing compilers. We present four goals of an ideal compiler for heterogeneous systems and examined their impact on a compiler summarized in Table 2. Existing compilers satisfy some of these goals, but they lack the flexibility needed by heterogeneous systems because homogeneous systems do not require it. We identify areas from which existing technology can be borrowed and areas in which these compilers lack the necessary flexibility for heterogeneity. Achieving this flexibility does not require substantial changes to core compiler technology, e.g., parsers and transformations, but rather the way that they work together, i.e., the compiler's software architecture.

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

1. 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.
2. 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.
3. 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.
4. W. Blume et al. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.
5. F. Bodin et al. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.
6. 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.
7. 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.
8. 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.
9. R. Butler and E. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.
10. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.
11. 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.
12. K. Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.
13. 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.
14. 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.
15. K. Faigin et al. The polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.
16. S. Feldman, D. Gay, M. Maimone, and N. Schryer. A Fortran-to-C converter. Computing Science 149, AT&T Bell Laboratories, March 1993.
17. G. Fox et al. Fortran D language specification. Technical Report TR90-141, Rice University, December 1990.
18. A. Ghafoor and J. Yang. A distributed heterogeneous supercomputing management system. *Computer*, 26(6):78–86, June 1993.

19. M. B. Girkar and C. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5), 1994.
20. 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.
21. 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.
22. 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.
23. 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.
24. A. Khokhar, V. Prasanna, M. Shaaban, and C. Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, June 1993.
25. 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.
26. 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.
27. K. S. McKinley, S. Singhai, G. Weaver, and C. Weems. Compiling for heterogeneous systems: A survey and an approach. Technical Report TR-95-59, University of Massachusetts, July 1995. http://osl-www.cs.umass.edu/~oos/papers.html.
28. Message Passing Interface Forum. MPI: A message-passing interface standard, v1.0. Technical report, University of Tennessee, May 1994.
29. 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.
30. H. Nicholas et al. Distributing the comparison of DNA and protein sequences across heterogeneous supercomputers. In *Proceedings of Supercomputing '91*, pages 139–146, 1991.
31. 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.
32. D. A. Pauda. Private communication, September 1995.
33. 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.
34. 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.
35. 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.
36. L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, June 1992.

37. Stanford Compiler Group. The SUIF library. Technical report, Stanford University, 1994.
38. 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.
39. C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.
40. 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.
41. L. H. Turcotte. Cluster computing. In Albert Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 26. McGraw-Hill, October 1995.
42. C. Weems et al. The image understanding architecture. *International Journal of Computer Vision*, 2(3):251–282, 1989.
43. C. Weems et al. The DARPA image understanding benchmark for parallel processors. *Journal of Parallel and Distributed Computing*, 11:1–24, 1991.
44. R. Wilson et al. The SUIF compiler system: A parallelizing and optimizing research compiler. *SIGPLAN*, 29(12), December 1994.
45. 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.
46. 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.
47. H. Zima. Private communication, September 1995.
48. H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.
49. H. Zima, B. Chapman, H. Moritsch, and P. Mehrotra. Dynamic data distributions in Vienna Fortran. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.