Compiler Representations for Heterogeneous Processing

Glen Weaver

CMPSCI Techincal Report 95-102

November 1995

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

Abstract

The emergence of heterogeneous parallel systems opens the possibility of higher performance for complex, heterogeneous applications. Unfortunately, heterogeneous parallel systems are even more complex to program than homogeneous parallel systems. Programmers should not have to handle all the added complexity of these systems. Instead, compilers should be extended to automatically handle as much of this complexity as possible. In previous work [51, 49], we argue that a compiler for heterogeneous systems requires a substantially more flexible software architecture than found in current compilers. Achieving this flexibility requires changes in the compiler's intermediate representation (IR).

This paper proposes an intermediate representation, Score, designed to support heterogeneity. We identify four capabilities that an IR for heterogeneous systems should support (*i.e.*, reordering of transformations, representing of multiple models of parallelism, extending to new models of parallelism, and reusing transformations for different models of parallelism). We survey existing intermediate representations with respect to their support for these capabilities. We then present our IR design which adapts ideas found in our survey section to suit the needs of heterogeneity. Score's adaptations for heterogeneity support several existing models of parallelism, facilitate extension to new models of parallelism, and allow low and high-level transformations to be interleaved. Finally, we describe our approach for evaluating Score.

1 Introduction

This paper proposes an intermediate representation for use in a compiler for heterogeneous systems. Heterogeneous systems are a form of parallel system that combines disparate architectural models, so that the benefits of each model can be leveraged for a single application. The variety and variability of available resources in a heterogeneous system demands a compiler that is much more flexible than current compilers. Achieving this flexibility requires modifications to the software architecture of the compiler, which in turn implies changes to the compiler's intermediate representation (IR). The IR must support the reordering of transformations, represent multiple models of parallelism, be extensible to cover new models, and enable the reuse of transformations.

We propose a new intermediate representation which is an extension of recent work in program representation with features appropriate for heterogeneity. Our IR, named Score, addresses each of the four concerns listed above. Score separates a node's attributes from its operation, represents important language constructs at high- and low-levels simultaneously, and supplies the IR nodes to represent diverse models of parallelism. We have planned a series of six experiments to evaluate Score's effectiveness as a representation for a heterogeneous system's compiler.

The remainder of this section describes the importance and complexity of heterogeneous processing, and how a compiler can help reduce that complexity for programmers. Section 2 reviews current intermediate representations with respect to their appropriateness for heterogeneity. Section 3 presents the novel features of Score, and Section 4 details our plans for evaluating Score. Finally, we summarize in Section 5.

1.1 Heterogeneous processing

Recent product introductions (*e.g.*, Meiko CS-2, IBM SP-2) and research (*e.g.*, IUA [67], PVM [62], p4 [17], and MPI [34]) demonstrate growing interest in heterogeneous parallelism. Previous efforts in parallelism focused on homogeneous parallel machines, which implement a single model of parallelism throughout the machine. The performance of large applications on homogeneous parallel machines is often limited by the time required to execute sections of code that do not match the machine's model of parallelism. These sections must either be parallelized inefficiently or executed sequentially. Heterogeneous systems can deliver consistent high performance by incorporating multiple models of parallelism within one machine or across machines, creating a virtual machine.

Heterogeneous processing [60, 65, 66, 44, 36] is the well-orchestrated use of heterogeneous hardware to execute a single application [44]. When an application encompasses subtasks that employ different models of parallelism, the application benefits from using disparate hardware architectures that match the inherent parallelism of each subtask. For example, Klietz *et al.* describe handcoding and executing a single application, a simulation of mixing by turbulent convection, across four machines (CM-5, Cray-2, CM-200, and an SGI)[45]. The four machines form a single virtual machine, and the authors leverage the strengths of each machine for different tasks.

1.2 Compilers for Heterogeneous Systems

Heterogeneous processing is complex because of the variety of hardware available, the variability of hardware availability (in virtual machines), and the consistent demand for high performance. Developing software for heterogeneous systems would be overwhelming if each application needed to handle this complexity. Klietz *et al.*hand parallelized each task specifically for the appropriate target machine in each machine's unique language dialect. If the configuration of hardware changes, the authors must rewrite parts of the program.

Instead of always having to modify programs, some of the complexity of heterogeneous systems should be automatically handled by a compiler. With certain modifications to a compiler's software architecture, it could use transformations to adjust a program to execute efficiently on a heterogeneous machine [51]. A compiler for heterogeneous systems is unique in that it must simultaneously compile a single program to diverse targets[1]. When compiling for homogeneous machines, a fixed strategy for applying analyses and transformations is effective, but for diverse models of parallelism the compiler must be more flexible in applying its transformations. Our approach to achieving this flexibility is for the compiler to customize its strategy by *planning* the compilation. For heterogeneous virtual machines, the compiler must also be capable of adapting to changing hardware configurations and be extensible in order to handle changing models of parallelism.

[1]In our terminology, a *heterogeneous system* consists of diverse *component processors*, each of which is a separate *target*.

1.3 Intermediate Representations for Heterogeneous Systems

Extending a compiler to meet the needs of heterogeneity requires changes to its intermediate representation. We have identified four areas in which current intermediate representations must be altered to support compilation for heterogeneous systems (see Table 1).

- First, an intermediate representation must support the reordering of transformations. The features of the target (and the source code) determine the appropriate sequence of translation steps (*i.e.*, analyses and transformations). Thus, a compiler with multiple targets will need to reorder transformations flexibly. Existing systems reach a point at which they lower the IR represenations so that high-level transformations are no longer applicable. However, for maximum flexibility high-level transformations should be applicable at any time. For example, consider compiling for a superscaler processor with a vector co-processor, similar to processing elements in the CM-5. An operation on a short vector may not be worth the overhead of initiating a vector operation, but if the other function units are already busy with other instruction level parallelism, the operation may as well be scheduled on the vector co-processor. Trading off parallelism (vectorization) found at the high-level for low-level parallelism (scheduling functional units) is facilitated by interleaving of high- and low-level transformations.

- Second, an intermediate representation must express the features of different machines and models of parallelism. For a compiler to generate high-quality code, its representation must match the features of the machine (and its inherent model of parallelism). A compiler for heterogeneous systems is unique, because its IR must have a spectrum of nodes to represent the various machines and models of parallelism. For example, to compile for message-passing machines, a compiler needs IR nodes that represent message operations, but the compiler also needs IR nodes for matrix operations to compile for mesh connected SIMD machines.

- Third, an intermediate representation must be extensible so it can cover new machines and models of parallelism and corresponding language extensions. Virtual heterogeneous machines are not necessarily the product of a single vendor and so no one organization is responsible for producing a compiler. Instead, system programmers should be able to extend the compiler for new machines, which implies that the IR is extensible. For example, if a manufacturer wishes to extend the compiler to cover its new message-passing MIMD architecture, these extensions should be relatively easy to make and not impact existing IR nodes.

- Fourth, an intermediate representation must support reuse of transformations. Once a transformation is written it should not have to be rewritten for every new machine or model of parallelism, nor should it have to be rewritten to function on programs in every different source language. For example, message operations in message-passing MIMD machines and SIMD meshes have different characteristics. Whereas nodes in a MIMD machine generally can arbitrarily send messages to other nodes, SIMD meshes must use coordinated communication patterns and generally can directly communicate with only a few other nodes (*i.e.*, its neighbors). Nevertheless, some transformations, such as message coallescing, can be used for both targets.

2 Literature Survey

This section reviews intermediate representations from the literature with respect to their appropriateness for heterogeneous processing. Some of these IRs are implemented in a compiler and others are simply conceptual without a specific implementation. For each IR, we discuss the goals of the system in which the

- Supports reordering of transformations
- Represents multiple models of parallelism
- Permits extension to cover new models of parallelism
- Enables reuse of transformations for different models of parallelism

Table 1: Requirements of an intermediate representation for heterogeneity.

IR is used (if applicable), the design goals of the IR itself, and its support for the IR requirements listed in Table 1.

Several well-known representation techniques occur repeatedly in our survey. Abstract syntax trees (ASTs) [1] represent a program by treating source constructs as operators and operands, and organizing these nodes into trees where the root is generally a procedure or an entire program. ASTs are popular in source-to-source translators. Scalar compilers, on the other hand, often use a control flow graph (CFG) [1] as their primary representation. CFGs represent programs as a graph of basic blocks. Edges in a CFG represent control flow (*e.g.*, decisions, loops, jumps), and basic blocks contain sequentially executed code. Program dependence graphs (PDG) [32] connect nodes together with control dependence edges and data dependence edges. The control dependence edges in PDGs reflect less of a source program's syntactic structure than the control flow edges in CFGs or the syntax based edges in ASTs. Any of these three graph representations can be put into static single assignment (SSA) form [25]. In SSA form, no variable is assigned a value in more than one location.

We divide the IRs in our survey into five groups by focusing on the extent to which each IR constrains the ordering of its nodes. Sections 2.1–2.4 trace a progression of intermediate representations from heavily constrained ASTs to loosely constrained Optimistic IRs. Optimistist IRs are those representations that ignore most control or control flow dependences, and instead rely heavily on data dependences to define the relationships between nodes. Section 2.5 includes miscellaneous IRs which do not fit neatly in this progression.

2.1 Abstract Syntax Trees

Many parallelizing compilers [23, 12, 13, 70, 55] use an abstract syntax tree (AST) because it is simple to build, retains the original structure of the program, and readily supports high-level transformations. Each AST node represents an operation, and the node's children represent the operation's operands [1, 48]. The type of an AST node corresponds to a construct from the source language such as a minus operation, **while** loop, or variable declaration. ASTs are also used in other types of systems such as compiler toolkits, software engineering environments, and distribution formats.

ParaScope

Rice University's ParaScope [23, 43] is a source-to-source translator for Fortran. ParaScope was built to explore automatic parallelization and has been used for a variety of research projects. It provides sophisticated global program analysis and a rich set of program transformations. ParaScope's core intermediate representation is an AST with other analysis data (*e.g.*, data dependences, control dependences, and SSA form) woven into it. ParaScope has been used to compile to several models of parallelism (*e.g.*, shared-memory, control-parallel MIMD [50] and distributed memory, data-parallel MIMD [35]), and has IR nodes to represent parallel constructs. Modifying ParaScope's IR is difficult because the implementation does not use data hiding, so all the components in ParaScope directly access the AST. Transformations in ParaScope may be reordered and even applied to sections of a program during interactive editing. Some transformations incrementally update analysis data, but implementing incremental update is solely the responsibility of the individual transformations.

Polaris

Polaris from the University of Illinois, is an optimizing source-to-source translator [12, 53, 30] for Fortran derivatives. The authors have two major goals for Polaris: to automatically parallelize sequential programs for distributed shared memory machines and to be a production quality compiler in terms of reliability and speed. Polaris seeks to achieve this level of quality by implementing its IR as a C++ class hierarchy and using class methods to control access. The methods ensure that updates reflect legal Fortran syntax, maintain the integrity of the IR, and correctly free memory. The underlying representation is an AST with most nodes defined as children of one of five major classes: the *Program* class represents the entire program, the *ProgramUnit* class corresponds to top level units (subroutines or data declarations), 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's IR is easy to extend through C++'s data abstraction and inheritance mechanisms. Transformations and analyses in Polaris are organized as separate passes over the program that may be statically reordered. If a transformation needs analysis data that is not yet computed, the analysis routine is automatically invoked. Once analysis data has been computed, Polaris does not ensure that the data is still accurate when subsequent transformations access it.

Sage++

Sage++ from Indiana University is a toolkit for building source-to-source translators [13]. Sage++ seeks to facilitate the construction of source-to-source translators for Fortran and C++ derivatives by packaging common routines such as parsers and data dependence testing. Sage++ implements an AST representation in a C++ class hierarchy with five major classes that represent whole programs and files, statements, expressions, symbols, and types. Sage++ does not have node types to represent parallelism, but it does have node types for both C++ and Fortran. The authors expect developers to extend the IR through data abstraction and inheritance. Sage++ uses methods to hide the IR's implementation but not to constrain operations on the IR. As a toolkit, Sage++ does not provide applicability criteria for transformations and therefore does not consider reordering.

Reprise, IRIS-Ada, IRIS-C++

The Arcadia project investigates software engineering tools and techniques. Though not a compiler, Arcadia software represents programs in an AST and can derive other compiler representations (*e.g.*, CFGs) from the AST. Arcadia uses IRIS-Ada [33] to represent Ada programs and IRIS-C++ [63] to represent C++ programs. IRIS-C++ descends from both IRIS-Ada and Reprise [57], a representation for C++ from AT&T. These IRs use a separate node to represent each semantic concept (*e.g.*, variable attributes) in the source language. All three representations represent only the parallelism inherent in their associated source language, which means IRIS-Ada has constructs for task level parallelism while IRIS-C++ and Reprise include only sequential constructs. The separation of each semantic concept into distinct nodes means that nodes can be easily reused for new languages. However, these representations do not use inheritance, so transformations must operate on specific node types.

SUIF

Stanford University Intermediate Format (SUIF) is a compiler framework that can be used as either a source-to-C translator or a native code compiler [70, 6, 38] for Fortran 77 and C. SUIF serves as a research tool for investigating automatic parallelization of sequential programs.

SUIF's IR (which is also called SUIF) includes both high and low-level nodes to enable code generation. High-level nodes reflect source language constructs (*i.e.*, for loops, general loops, if statements, and array references), whereas low-level nodes are modeled after a generic RISC machine instruction set (*e.g.*, load, store, and add). The SUIF compiler initially represents programs as a forest of abstract syntax trees with a mixture of high and low-level nodes called high-SUIF. Later, the high-level nodes are translated into low-level nodes, and routines are represented as a linear list of low-level nodes. The IR is then called low-SUIF. The nodes in low-SUIF are a subset of those in high-SUIF.

SUIF is an AST implemented as a C++ class hierarchy with 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 AST nodes. Each 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 (*i.e.*, 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.

Like other C++ implementations, SUIF facilitates extension through data abstraction and inheritance. SUIF is used for multiple languages, so it does not include source specific checks in its methods. By including high and low-level representations within a single IR, SUIF supports intermixing of high and low-level transformations. However, when the representation is converted to low-SUIF, high-level transformations may no longer be used. The SUIF compiler does not reorder transformations during compilation. SUIF exchanges data between passes through an annotation mechanism, which is very flexible but provides no support for ensuring the annotation's accuracy.

ANDF

Architecture-Neutral Distribution Format (ANDF) [55, 47] is a distribution format for portable software[46]. ANDF tries for a clean break of the compiler into a language-dependent, machine-independent front end called a *producer*, and a language-independent, machine-dependent back end called an *installer*. If this division is clean, then ideally a single producer can be used for many machines and a single installer can be used for many languages. Software manufacturers can ship an ANDF version of their program to their customers who run an installer to custom optimize the program.

ANDF is tree based like many other intermediate representations, but its form is strongly affected by its goals of being a distribution format and supporting portable software. As a distribution format, ANDF must protect against reverse engineering. Therefore, ANDF removes all identifiers and replaces them with *tags*. Tags are integers, and though they do not obey the source language's scoping, tags are only valid within their declaration[47]. Therefore, a declaration must carry the code for which the declaration is valid. For example, the declaration of a local variable will have as one of its arguments, the routine's body.

ANDF attempts to straddle the line between languages and machines. Besides linkage concerns, the representation consists largely of types, conditions, expressions (including simple control structures), and exceptions[27]. This level of abstraction is barely above that of an idealized uniprocessor. ANDF is only intended to support uniprocessors, so it does not consider multiple models of parallelism. ANDF also has no mechanism to protect the rest of the compiler from changes to the IR.

Discussion

ASTs have proven to be useful, especially for source-to-source translators, and they are even used for other applications besides compilers. Yet, they have limitations as an intermediate representation:

- Each node type in an AST corresponds to a source language construct, and therefore bears the semantics of that construct. Adding a new construct to the source language requires a new node type for the AST which is likely to impact other parts of the compiler that use the IR (*e.g.*, parsers and transformations). Polaris, Sage++, and SUIF partially accommodate extensibility by using a C++ class hierarchy which allows new constructs that are specializations of existing constructs to be added with little impact to transformations.

- The structure of an AST follows from the source language's syntax, which is not particularly important for most transformations. Instead, transformations need to know the data and control dependences between each IR node. For ASTs, this dependence information has traditionally been obtained through analysis and either woven into the AST or stored in a separate structure.

- Though some research has investigated generating code from trees [2, 19, 4, 39], ASTs are not adequate for low-level optimizations. ASTs do not capture the features of the target hardware (*e.g.*, the instruction pipeline and register set). SUIF and ANDF attack this problem in two separate ways. SUIF includes low-level nodes as well as high-level nodes and eventually linearizes the tree. ANDF tries to find a balance between high and low-level representations.

2.2 Program Dependence Graphs

Ferrante *et al.* introduce the program dependence graph (PDG) which overcomes some of the limitations of an AST [32]. A PDG eliminates the infiltration of the source language's syntax into the IR and supports a broader range of transformations. As in an AST, PDG nodes represent program actions (usually operators and operands), but in a PDG nodes are connected via data and control dependences. Although these dependences are uniformly represented, a PDG can be thought of as consisting of two subgraphs: the control dependence subgraph (CDG) and the data dependence subgraph (DDG). The control dependence subgraph has *region* nodes inserted to summarize the control conditions for a node and to organize the children of a node with distinct control outputs (*e.g.*, conditional statements).

A PDG has several inherent advantages as a compiler IR. The structure of a PDG naturally exposes some parallelism, because a PDG does not impose any execution order on the nodes other than that specified by dependences. In contrast, a CFG or an AST captures constraints that are artifacts of the original program's linear order. A PDG simplifies the coding of transformations by connecting computationally relevant parts of the program and capturing the relationships among nodes that are most important to transformations.

PDG and Vectorization
Baxter *et al.* [10] demonstrate the effectiveness of a PDG for high-level optimizations by using loop innermosting (the shifting/interchanging of loops with no associated data dependences to the deepest nesting level) and vectorization to vectorize loop nests automatically. Conditional statements such as **if** statements complicate vectorization, but can be handled by *if-conversion* which transforms **if** statements into guarded statements. If-conversion can be viewed as converting a control dependence to a data dependence. Previous vectorization research first performed if-conversion, and then found opportunities for vectorization. The PDG's unification of control and data dependence allows a compiler to delay the decision of whether or not to apply if-conversion. Control dependences are also more natural than control flow for transforming conditionals because each control dependence explicitly links statements with their controlling condition.

Kennedy and McKinley show how to use the control dependence graph (CDG) to avoid if-conversion altogether when performing loop distribution [42]. Given a desired partitioning of a single loop into multiple loops, they determine where *execution variables* are needed by examining control dependence edges that cross partition boundaries. An execution variable transforms control dependences into data dependences.

GURRR
Berson *et al.* [11] use a PDG as the basis of an IR, Global Unified Resource Requirements Representation

(GURRR), that supports reordering of (very) low-level optimizations as well as high level transformations. GURRR grows out of earlier work in integrating register allocation with instruction scheduling but seeks to include other optimizations such as loop transforms.

GURRR is a collection of resource usage information (*e.g*., registers and functional units) superimposed on an instruction-level PDG. Using an instruction-level PDG makes GURRR machine specific, but this specialization may be necessary for such low-level optimizations. Region nodes in GURRR summarize resource usage information and execution count estimates as well as control dependences. GURRR also defines resource hole nodes (both free and slack holes) which represent unused resources. GURRR includes three new types of dependence edges: the transitive data dependence edge to simplify node ordering by representing indirect dependences, temporal dependence edges to represent sequential ordering constraints imposed by resources, and reuse edges to connect nodes that can temporally share a resource under a legal ordering of instructions.

Despite discussing loop transformations, the authors consider only instruction-level parallelism (ILP) for uniprocessors. GURRR may be limited to this domain because the overhead of accounting for the machine details on each transformation may prove too costly for general transforms. Though a particular implementation of GURRR is restricted to a single instruction set, a different instruction set could be used for a different implementation. GURRR is designed to support reordering of optimizations, including the lowest level optimizations. But once again, managing the smallest of details about the target for each transformation may be too costly.

Discussion

A PDG's structure better captures those aspects of a program that transformations need. Hence, PDGs have several advantages over ASTs.

- A PDG better supports transformation reordering. High-level transformations are simple on an AST, but low-level transformations are more difficult. Hence, AST-based systems generally run all high-level transformations first and use a different representation when applying low-level transformations. A PDG, on the other hand, is appropriate for both high- and low-level transformations, and therefore can interleave transformations from the different levels.

- Transformations may traverse a PDG uniformly which eases the development of transformations and improves their reusability. The CDG has only two types of nodes: normal operations which have a single control input and decision/branch nodes which have a single control input and multiple control outputs. A transformation may walk the DDG without considering the function of each node.

- A PDG exposes more low-level parallelism than an AST or CFG because it requires fewer links (*i.e*., constraints) between nodes. A CFG requires a linear ordering of nodes, and an AST captures syntactic constraints as well as implicit dependences.

A PDG cannot be used as a compiler's sole IR. Dependences are extracted through analysis and not directly by a parser. Hence, a compiler must represent programs in some other form for dependence analysis, quite possibly an AST. A PDG can be thought of as an intermediate level form between an AST and a CFG. GURRR includes sufficient information to forgo a low-level representation but may be expensive to maintain.

2.3 Incorporating Static Single Assignment

A PDG increases opportunities for optimization by using control dependences rather than control flow dependences (as in a CFG). Similarly, Static Single Assignment (SSA) form increases opportunities for optimization by modifying data dependences. In SSA form, a variable use has exactly one definition [25]. To meet this requirement, every place a variable is defined or two control paths merge, a new variable name is created. Conceptually, pseudo-assignments, called ϕ-nodes, are inserted when two control paths merge to combine the values from each of the incoming control paths. Implementations do not actually create new variable names, but rather modify def-use (or use-def) chains so that each use has exactly one definition. SSA can be thought of as approximating value flow, and it eliminates output dependences and some anti-dependences.

Though semantically equivalent, SSA form is a substantial change from the original program and cannot be executed directly. ϕ-nodes are not executable by hardware because their operation depends on the execution path taken to reach the node. In addition, converting to SSA form creates an extraordinary number of new variable names; many of which can and should be removed (Cytron *et al.* [25] suggest using a graph coloring algorithm for removal).

Aliases play an important role in the construction of SSA form. When several memory accesses are potentially aliased, they must be treated as identical. Hence, an update to any one of them is the same as an update to them all. Treating them as identical ensures proper program behavior when variables are aliased but diminishes the precision of the SSA graph when the alias relationship is unwarranted. Alias analysis tries to disprove that variables are aliases and thereby improve the precision of the representation [3].

Specializing Phi Nodes

Alpern *et al.* propose a modest change to SSA to support detecting equality of variables [5]. Their algorithm for detecting variables with equal values is similar to value numbering but works across control structures such as conditionals and loops.

The authors use congruence to find variables with equal values. Two nodes are *congruent* if they have the same phi function label (which means they are in the same basic block) and their arguments are the same. The authors find congruent variables by computing the maximal fixed point of the congruence relation. Congruence is a subset of equivalence and is independent of position in the program, whereas equivalence is defined at a point in the program. Hence, variables must *always* be equivalent in order to be congruent, which is reasonable in SSA form because each variable has only one assignment.

The authors extend congruence beyond basic blocks by using specialized ϕ-nodes that correspond to specific control structures (*i.e.*, conditionals and loops). High-level ϕ-nodes make SSA form interpretable by including as a parameter the condition that determines which value to assign. High-level ϕ-nodes are also called gated ϕ-nodes because they accept additional arguments that determine the node's function (*i.e.*, gate the value assigned). They define ϕ_{if}-nodes that combine values at the bottom of a conditional, ϕ_{enter}-nodes that combine the incoming and loop-carried values of a loop index, and ϕ_{exit}-nodes that combine a variable's value from the loop with its value if the loop is never executed. The ϕ_{if}-node, for example, has three parameters: the value coming through the `then` clause, the value coming through the `else` clause, and the `if` predicate. With high-level ϕ-nodes, two nodes are congruent if they have the same value under the same control condition. High-level ϕ-nodes combine high and low-level information in a single representation which in turn enables a broader range of transformations to be used on the representation.

Extended SSA

Stoltz *et al.* describe the intermediate form they use in their parallelizing Fortran 90 compiler, Nascent [61]. Rather than incur the cost of actually creating many new variable names, most systems use factored def-use chains. A def-use arc originates at a variable's definition and points to a use of the variable; a use-def arc points in the other direction. Combining SSA form and use-def chains is advantageous because each use has only one definition, which greatly reduces the number of links. The authors recommend using use-def chains instead of def-use chains. Use-def chains require only constant space per node and facilitate demand-driven analysis. The authors also add a new type of ϕ-node, a ψ-node, which combines the values of a variable from the execution of *parallel* sections. Unlike a ϕ-node, both control flow paths leading to a ψ-node are executed. Hence, ψ-nodes allow the explicit representation of control parallelism.

Dataflow and SSA

Even with *use-def* chains as Stoltz *et al.* [61] recommend, SSA form does not support solving backward data flow problems. Johnson and Pingali [41] devised the dependence flow graph (DFG) to support forward and backward data flow. The algorithm to construct DFGs finds single-entry single-exit (SESE) regions in the CFG. The single entry edge and the single exit edge of a SESE regions have the same control dependences. *Switch* nodes are inserted at the top of the SESE region to collect incoming definitions, and *merge* nodes are inserted at the bottom to collect outgoing definitions. Switch and merge nodes are inserted symmetrically so that the graph can be traversed in either direction. Merge nodes are equivalent to ϕ-nodes in SSA form.

Program Dependence Web

Ballance *et al.* combine many of the preceding features into a single representation, the Program Dependence Web (PDW) [9]. The PDW consists of SSA form on top of a PDG with modifications to make it interpretable and provide support for data flow parallelism. The PDW is built in three steps. The first step uses the standard algorithm [25] to build SSA form on a PDG. The second step translates an SSA-form PDG into Gated Single Assignment (GSA) form, which is essentially equivalent to Alpern *et al.*'s gated SSA form. In SSA form, a single ϕ-node may have as many arguments as necessary, but GSA form's specialized (or high-level) nodes have a fixed number of arguments. Hence, a single ϕ-node in SSA form becomes a tree of specialized ϕ-nodes in GSA form. The third step converts from GSA form to a PDW by inserting switch nodes (similar to those in Johnson and Pingali [41]). Switch nodes are inserted symmetrically to γ-nodes.

The authors do not expect transforms to operate over the entire PDW. Instead, they define an interface that provides subgraphs called Interpretable Program Graphs (IPGs). They define three IPGs: the control IPG which includes only control dependences, the data flow IPG which includes switches and μ- and η-nodes, and the demand IPG which includes all three specialized ϕ-nodes. The authors use the data flow IPG in their efforts to compile to data flow architectures. The other IPGs are appropriate for compiling to other models of parallelism.

Thinned Gated Single Assignment

Havlak describes a minor variation of Ballance, *et al.*'s GSA form, called Thinned Gated Single Assignment (TGSA) [40]. TGSA extends high-level SSA form (*i.e.*, GSA) to unstructured code and removes information from GSA that is unnecessary for symbolic analysis. Supporting unstructured code requires constructing a directed, acyclic graph (DAG) of γ-nodes rather than a tree (as Ballance *et al.* [9] does). *Thinning* refers to the removal of information the authors do not need. For example, ϕ-nodes for which only one argument will ever be selected are eliminated.

Discussion

SSA form complements rather than competes with other intermediate representations such as ASTs and PDGs. The primary benefit of SSA form is that it creates new opportunities for transformation by reducing data dependences and by reassigning variable names. SSA form also aides in the reordering of transformations by unifying low and high-level information through specialized ϕ-nodes. A PDW merges the benefits of SSA form and a PDG by combining them, and even improves SSA form's usefulness for data flow machines by including switch nodes.

The combination of SSA form and a PDG is a powerful intermediate representation, but as we will see in the next section, it can be improved by removing unnecessary control dependences.

2.4 Optimistic IRs

Recent representations improve on the PDG representation by removing as many control dependences as possible. The idea is that only instructions that affect the program's output need to be executed under their control dependences. The remaining instructions are scheduled according to their data dependences. For example, code in a conditional may be executed *speculatively* outside of an *else* clause, as long as memory is not updated if either the condition turns out to be true or an exception occurs. The existence of speculatively executed code increases possibilities to combine code, improves flexibility in assigning registers, and creates opportunities for moving the definition and uses of a value closer together.

In the CFG
Click describes a simple SSA form intermediate representation [22, 20]. Click's IR is based on the CFG, but he represents basic blocks by a single node, called a *region* node. The other nodes in the IR have pointers to their region node which allows the hierarchy of the CFG to be flattened into a single layer. Not only does this flattening allow transformations to treat control and data dependences identically, but it also facilitates the reduction of control dependences. The dependence between a node and its region node can be broken by simply clearing the pointer. Once the control flow dependence is broken, the execution order of these nodes is constrained only by data dependences; they are free to float around with respect to basic blocks. Click calls this a "sea" of nodes.

In Click's IR control is represented by a Petri net with a single token, and none of its nodes represent parallel actions. Hence, it is limited to representing sequential programs. The IR is implemented by a C++ class hierarchy, which facilitates extension but does not have any other provisions for extending the IR. Currently, only low-level optimizations (*i.e.*, conditional constant propagation, global value numbering, and global code motion) have been implemented on top of this IR. Click takes a different approach to optimization ordering by combining optimistic optimizations within a single data flow framework.

In a PDG
Weise *et al.* present the Value Dependence Graph (VDG) for representing imperative programs [69, 58]. The VDG grew out of Fuse, a representation for functional programs, and VDG retains a functional flavor. The VDG is much like a PDG coupled with SSA form. Like GSA, the VDG has γ-nodes that represent the merge of a value from the two halves of a conditional, but the VDG uses function calls to represent loops. The VDG also uses *demand dependence* rather than control dependence. A demand dependence graph (DDG) is similar to a control dependence graph (CDG) but the predicates in a DDG are those that lead to computations that contribute to the output. The predicates in a CDG are those that lead to a computation being performed (perhaps needlessly).

To build a VDG, the authors start with a CFG and produce a Store Dependence Graph (SDG) which makes changes to memory (such as adjusting the stack size) explicit and represents loops as recursive function calls. The authors then perform several transforms and finally symbolic execution on the SDG to produce the VDG. The VDG only represents value flow, so control and data dependences are folded into value flow. To get out of VDG, the authors build a demand PDG (dPDG). A dPDG is very much like a PDG except that control dependences are replaced by demand dependences. Finally, dPDGs are sequentialized using one of the standard algorithms for sequentializing PDGs.

Though the VDG should be easy to extend for other models of parallelism, it currently supports only uniprocessor targets. Ruf presents transformations that produce a sparse form of the VDG tailored for specific analyses [58]. He reuses the VDG structure, so the compiler does not need an entirely different representation for this specialized IR. The use of specialized representations may aid compilation to different models. Though the authors are not interested in extensibility and do not provide explicit support for it, new nodes can be handled fairly easily because many transformations key off of the γ-nodes and function calls. The authors also suggest that a different group of transformations should be applied to the IR at each stage in the VDG's construction. Though this simplifies the construction of each transformation, it partially orders transformations.

Discussion

These representations are a natural progression from the PDG representation. A PDG reduces dependences by using control dependences instead of control flow dependences, and these representations remove control dependences from those nodes which are sufficiently ordered by data dependence. Having only data dependences gives compilers greater flexibility in scheduling nodes, which implies greater flexibility in molding a program to fit the target model of parallelism. However, neither of these intermediate representations is used for compiling parallel programs. Hence, they do not include nodes to represent parallelism, support extensions to cover new models of parallelism, or reuse transformations for different models of parallelism.

2.5 Other Representations

This section contains several important IRs which do not fit in the preceding sections. The designs of these IRs represent different objectives. Although intended for source-to-source translation, the Hierarchical Task Graph uses a hierarchy of control flow graphs rather than abstract syntax trees because it is intended for extracting source parallelism. XIL/YIL's unique combination of low and high level representations is, in part, intended to preserve existing compiler code that uses the low level representation. Explicit Data Placement focuses solely on representing communication.

Hierarchical Task Graphs

The University of Illinois' Parafrase-2 is a source-to-source translator designed for investigating compiler support for multiple languages and target architectures [56]. Parafrase-2 is unique in that it focuses on control parallelism as well as data parallelism. For control parallelism, Parafrase-2 partitions a program into parallel tasks while minimizing dependences between tasks, and for data parallelism, it parallelizes the execution of loops.

Girkar and Polychronopoulos designed an intermediate representation to meet the needs of Parafrase-2: Hierarchical Task Graphs (HTG) [37]. HTGs are hierarchical graphs much like the output of interval analysis. 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 structure of the HTG obscures the syntax of a program, but it highlights dependences between sections of the program.

The HTG is well suited for finding control parallelism because its emphasis on dependences aids program decomposition; a compiler can identify tasks by finding highly interdependent clumps of nodes. Furthermore, the hierarchical organization of HTGs facilitates choosing tasks at a granularity appropriate for the target architecture. The HTG is also largely immune to extensions of the source language because 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.

XIL/YIL

The XIL and YIL intermediate representations [52] jointly comprise the intermediate representation for IBM's TOBEY compiler. Initially, TOBEY's design included only traditional, low-level optimizations performed on XIL. However, when high-level transformations were added to TOBEY, they developed YIL to work with XIL. XIL nodes represent instructions typically found in RISC processors. Each node participates in two graphs. One graph is a linear list of nodes, so that a total order is always maintained. The other graph maintains nodes as expression trees with any redundant trees removed. The compiler extracts a YIL representation from an XIL representation. The nodes of a YIL graph correspond to statements with the body of each statement expressed in XIL. Each source procedure has a separate YIL graph. YIL represents important constructs such as loops, conditionals, and array references at a high-level.

The combination of XIL and YIL has limitations as an IR for heterogeneity. Though XIL/YIL includes the same basic information as representations in other parallelizing compilers, they have only been used for sequential and vector machines, and the authors do not discuss any nodes for representing parallelism. The information necessary for reordering transformations is present in an XIL/YIL graph, but the dichotomy between the two implies that a single transformation may impact both graphs. The authors expect that at some point during compilation the YIL subgraph will be ignored and subsequent optimization will use only the XIL subgraph. Hence, the separation of graphs hinders reordering.

Explicit Data Placement

Bala *et al*. describe Explicit Data Placement (XDP), which is an IR extension for representing communication [8]. The authors designed XDP to give compilers the power of manipulating both data and ownership transfer operations, with the loosest possible semantics. To achieve this goal, XDP incorporates five features: separation of data transfer from local computations, language and machine independent representation, unified and explicit treatment of data and ownership, flexible compute rules, and delayed binding of communication primitives to transfer operations. XDP supports only the SPMD model of parallelism and data distribution capabilities of High Performance Fortran. Since XDP is independent of both the source language and target machine, it encourages reuse and reordering of transformations by the compiler. Moreover, most existing compilers do not represent communication explicitly, so XDP exposes new opportunities for optimization.

2.6 Summary

Throughout this survey, we have examined each IR with respect to the features listed in Table 1. None of them meets all the requirements needed to accommodate heterogeneity. This section summarizes our findings.

Of the four requirements listed in Table 1, the reordering of transformations has received the most attention. High level transformations are primarily concerned with conditionals, loops, call sites, and array references; whereas, low-level optimizations prefer to operate over machine instructions. ASTs are convenient for expressing high-level transformations because they explicitly represent conditionals, loops, call sites, and array references as the programmer wrote them. But ASTs do not handle low level transformations well, which prevents the interleaving of high- and low-level transformations. SUIF attempts to overcome this difficulty by sharing nodes between high-SUIF and low-SUIF, but the compiler still cannot apply high-level transforms to low-SUIF. PDG based representations are much better for reordering optimizations because conditionals, loops, and call sites can easily be identified in a PDG. Nevertheless, the PDG based representations do not represent array references conveniently for both high- and low-level transformations.

Most of the work on parallelizing compilers has been implemented in source-to-source translators, so AST based representations have been used to target several models of parallelism. ParaScope, for example, has a variety of nodes for representing parallel actions, and SUIF can use its annotation mechanism to indicate parallelism. However, few of the other representations have been used for representing parallel programs, and those that have been used for parallelism have been used only for a small (usually one) number of models. In particular, the IRs which minimize control dependences (*i.e.*, Click's IR and the VDG) are used only for sequential machines. XDP is an interesting departure from the typical IR design. It captures one particular model of parallelism and is designed to work with other IRs to represent other models of parallelism.

What little work has been done in designing an IR to be extensible falls into two categories. Intermediate representations, such as PDGs, incorporate into the structure of the graph much of the information that transformations need. Hence, transformations are much less dependent on the types, *i.e.*, operation, of IR nodes. The other approach is to use a class hierarchy to define IR nodes. If a transformation operates on a node class that is high in the class hierarchy then lower classes may be modified or added without impacting the IR. This approach is sensitive to the structure of the class hierarchy.

None of the intermediate representations in our survey consider the reusability of a transformation for different models of parallelism. For a transformation to be reusable its function must somehow be separated from the nodes over which it operates. The techniques used for making an IR extensible somewhat address reusability. If a transformation is totally unaware of the node types on which it operates, then it will work for any model of parallelism. However, most interesting transformations need more knowledge of the underlying program.

3 Score

Our long term goal is to build a compiler for heterogeneous systems [51]. This section describes our intermediate representation, Score, which is the first step in building this compiler. In Section 1 we observed that the variety, variability, and high performance of heterogeneous systems demand a compiler that generates efficient code for a variety of hardware models. An intermediate representation can assist a compiler in this task by meeting the requirements from Table 1, but in Section 2 we found that none of the existing intermediate representations meet these requirements. Hence, we have designed Score to address the needs of heterogeneity by supporting several existing models of parallelism, facilitating extension to new models of parallelism, and allowing low- and high-level transformations to be interleaved.

The remainder of this section gives an overview of Score. Section 3.1 begins by describing Click's intermediate representation, which is the basis for Score. Section 3.2 presents our extensions to Click's representation that enable Score to support compilation for heterogeneous systems. Finally, Section 3.3 lists other enhancements to Click's IR for handling parallelism.

3.1 Click's Representation

Click describes an IR with simple nodes appropriate for representing instructions for a uniprocessor RISC machine (*e.g.*, load, store, and conditional branch) [22, 20]. His representation uses SSA form on top of a CFG, but each CFG block is explicitly represented by an IR node, called a *region* node. Operation nodes in a basic block point to the region node representing that basic block. Region nodes are connected via control flow edges, and operation nodes within a basic block are ordered only by data dependences. This representation for basic blocks allows the two-level CFG hierarchy to be flattened into a single layer. Moreover, for nodes representing most data computations,[2] Click removes the edges between them and their associated region nodes. Thus, the execution order of these nodes is constrained only by data dependences; they are free to float around with respect to basic blocks.

Click's sea of nodes provides new opportunities for reorganizing instructions, but currently the IR only supports sequential programs. All the operations represented by nodes are sequential, and the program representation is too low level for most parallel transformations. Hence, we plan to start with Click's IR and extend it to include support for parallelism.

[2]The exceptions are nodes which cause exceptions such as loads, stores, and division.

3.2 Score's Enhancements to Click's IR

This section describes the novel extensions to Click's IR that support heterogeneity. We divide the extensions into three categories: separation of nodes' operation from attributes, nodes to capture high-level knowledge, and nodes for parallel actions. Together these extensions meet the requirements listed in Table 1.

3.2.1 Supporting Reuse and New Models

To enable transformations to be reused with different models of parallelism and extended to new models of parallelism, Score separates a node's attributes from its operation. Transformations are generally written to operate on certain nodes and ignore others. In existing compilers, transformations examine the node's operation (*e.g.*, addition or **for** loop). Whenever a new node type is defined, any transformation that can operate over the new node must be updated so that it recognizes the new node. This updating of transformations complicates the extension of a compiler and is unnecessary. Transformations usually operate on several different types of nodes because these nodes share certain properties, which we call attributes. By separating a node's attributes from its operation, transformations can exploit a new IR node without being modified. Thus, this feature supports both extension of the IR to cover new models of parallelism and the reuse of transformations for different models of parallelism.

As a simple example of the potential benefits of separating a node's attributes from its operation, consider the Fortran code in Figure 1. Figure 1(a) shows a simple **for** loop to which any parallelizing compiler is able to apply many different transformations. In languages such as Fortran 90 and HPF, this same loop can be written using vector notation as in Figure 1(b). If a programmer uses vector notation, the IR may use a separate node to represent the vector operation. Nevertheless, we wish to be able to apply relevant loop transformations to this vector operation node. For example, a typical transformation is strip mining which tailors a vectorizable loop for a vector register of a specific size. Figure 1 shows a strip mined version of the loop (this example is taken from Bacon *et al.* [7]). The strip mining transformation could be explicitly coded to handle both types of loops and recoded when another type is introduced, but a more extensible approach is for the transformation to be written such that it operates over attributes of a node rather than the node's operation. A vector operation loop node would have the following attributes: it is a loop, it is iterated, the number of iterations is fixed before the loop begins, and data dependences do not form a recurrence. These attributes are sufficient to determine the strip mining transformation's applicability.

In addition to meeting the needs of heterogeneity, the separation of attributes from operations has the side benefit of enabling transformations to operate on user defined constructs. For example, algebraic simplification relies on algebraic properties such as associativity, commutativity, and identity. These same algebraic simplifications can apply to user defined data types, if the compiler is able to tag nodes corresponding to user defined constructs with the appropriate attributes. In separate research, we are investigating an annotation tool for conveying such information to the compiler [28].

3.2.2 Supporting Reordering

A compiler's compilation strategy determines the order in which transformations are performed. Different targets often require different compilation strategies, which means different transformations and orderings. We want our compiler to be as flexible as possible in its potential orderings of transformations, so that it can generate efficient code for any architecture, even ones that do not yet exist. Intermediate representations can restrict the ordering of transformations. For example, traditional parallelizing compilers (*i.e.*, AST based) impose a partial order on their transformations by employing a series of IRs with each one at a successively lower level. To avoid this restriction, scalar compilers immediately translate to a low-level representation and extract any necessary high-level information. Such a low-level representation simplifies traditional optimization but complicates high-level transformations.

```
do $i = 0, N$
    $A(i) = A(i) + c$
end do
```

(a) Original Loop

```
A(1:N)=A(1:N) + c
```

(b) Vector Form

```
$TN = (N/64) * 64$
do $i = 0, TN$
    $A(i : i + 63) = A(i : i + 63) + c$
end do
do $i = TN + 1, N$
    $A(i) = A(i) + c$
end do
```

(c) Strip Mined Loop

Figure 1: Example of separating a node's operations and attributes. 1(b) is equivalent to the simple loop in 1(a). To improve efficiency, 1(b) should be strip mined to account for the hardware's vector length, as in 1(c).

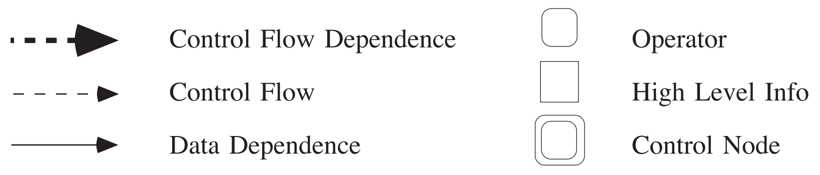


Figure 2: Key for figures.

To allow the interleaving of high- and low-level transformations, Score simultaneously maintains a high- and low-level representation. The program is completely represented in a low-level representation, similar to Click's IR, but Score also redundantly represents certain constructs (*i.e.*, loops, array references, procedure calls, and record accesses) in a high-level form. This high-level information is associated with a low-level node and can be thought of as part of the low-level node or as a separate node as long as the association is maintained. The high-level information participates independently in the IR graph through use/def chains. The independence of high- and low-level information allows the compiler to optimize the two levels of representation independently. The association between high- and low-level information permits the compiler to maintain consistency between the two levels of abstraction.

The SUIF IR is similar to Score in that it is a mixed level representation. However, high-SUIF does not have any redundancies between its high- and low-level information. For low-level optimizations to be applied to SUIF, high-SUIF must first be translated to low-SUIF which prevents further high-level transformations and related analyses.

Loops

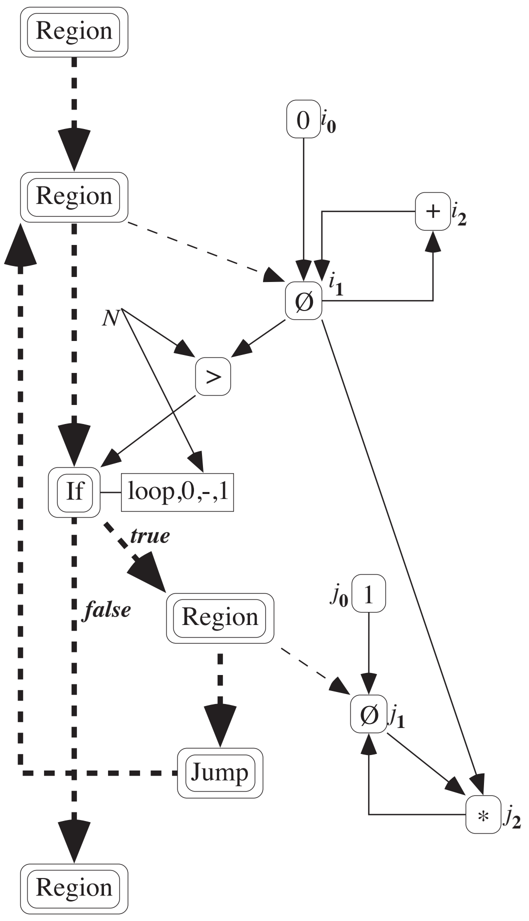
Loops are an important construct for compiler manipulation as evidenced by the plethora of loop transformations [7]. Due to the importance of loops, researchers have developed techniques for extracting loops and related information from low-level representations [71]. Given these techniques, our compiler will find loops through analysis rather than through interacting with the compiler's front end. Using analysis allows us to identify loops that programmers have obscured and thus overcome semantic deficiencies (*e.g.*, C/C++ does not have a true indexed loop, but these can be identified via analysis). Hence, the compiler

$j = 1;$
for $(i = 0; i < N; i++)$ {
    $j = j * i;$
}

(a) Original Source Code

S_1		$j_0 = 1;$
S_2		$i_0 = 0;$
S_3	loop:	$i_2 = \phi(i_0, i_1);$
S_4		if $(i_2 < N)$ {
S_5		$j_2 = \phi(j_0, j_1);$
S_6		$j_1 = j_2 * i_2;$
S_7		$i_1 = i_2 + 1;$
S_8		goto loop;
S_9		}

(b) Unstructured code in SSA form



(c) CFG-based Representation for code in 3(a)

Figure 3: Loop Example

generates descriptions of loops and associates the descriptions with a low-level node. Figure 3(b) shows the effect of translating the loop in Figure 3(a) to lower level language features (and converting to SSA form). Figure 3(c) contains the representation of the loop (using a CFG base for simplicity), and Figure 2 explains the graph symbols. The compiler associates the results of analysis with a low-level node (for **for** loops the compiler uses the **if** node that represents the end of the loop predicate). In our example, the representation has constants for the lower bound and step but has a use/def pointer for the unknown upper bound since later analysis may fix its value.

Array References

Array accesses and updates are an important source of data parallelism. Hence, researchers have developed a variety of tests to determine data dependences between array references. A high-level representation for array subscripts simplifies data dependence testing and many loop transformations. Array references are also an important source of low-level optimization because they abstract a substantial amount of arithmetic which is easy to optimize but which the programmer cannot optimize. Existing IRs initially use a high-level representation and eventually lower it in order to optimize address arithmetic. Score, on the other hand, retains both a high-level and low-level view of each array reference.

Figures 4 and 5 show a contrived example with two array references, one before optimization and one after. Figure 4(a) lists the original source code, and Figure 4(b) contains the same code but with SSA indices and handpicked ϕ-nodes. Figure 4(c) shows the Score representation for this code (see Figure 2 for symbol definitions). For brevity the control subgraph is not shown; hence, control dependence edges have no source node. Note that though the example is written in C, the address calculation shown assumes true multidimensional arrays which C does not provide. Hence, the example assumes some language and/or machine specific knowledge of array layout. The array references themselves are represented by the array *access* operator. Score uses a low-level representation for address arithmetic, so that the calculation of memory offsets can be optimized. Figure 5 shows the result of applying several optimizations (*e.g.*, algebraic transforms, node splitting, and common subexpression elimination) to the subscript expressions in Figure 4. Figure 5(b) uses only four operators instead of eight in computing the memory offset and only one multiplication instead of two. Both Figures 4(c) and 5(b) show high-level subscript information attached to the access operator. The high-level subscript representation retains a pointer to each variable referenced by a subscript operation, which is important because the representation is in SSA form. The *store* node represents a portion of potentially aliased memory. If the array A is unaliased, then the memory represented by the store node is equivalent to the array.

Subscript expressions are now effectively represented at both a high- and low-level. The compiler may apply low-level optimizations, but a high-level representation is always available for analysis. The compiler has two choices if the subscript expression changes. Transformations, such as loop skewing, may affect subscript expressions by only requiring a quantity to be added to the subscript expression. Addition's algebraic properties allow an addition node to be inserted without disturbing the low-level optimizations already performed on subscript expressions. On the other hand, the compiler may simply disconnect from the low-level subscript computation, modify the high-level representation, construct a new low-level representation from the high-level representation, and redo optimization on the new low-level representation. The old low-level representation can be removed by dead code elimination (DCE). In Figure 5(b), the computation of array offsets has been intertwined (in the computation of the compiler generated temporary t). If one array reference were removed, DCE would only remove the computation unique to that array reference.

Call Sites

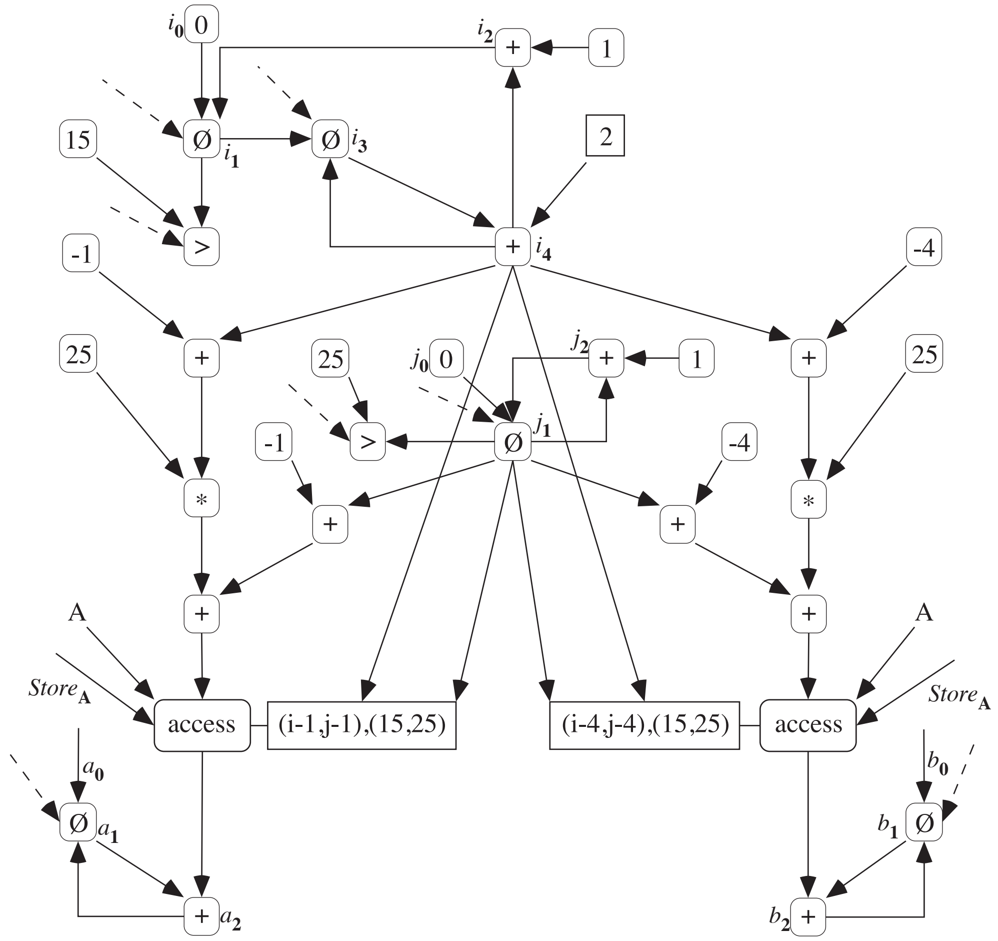
A high-level representation for procedure calls benefits control parallelism and interprocedural optimiza-

```
$S_1$:  char $A[15][25]$;

$S_2$:  for $(i = 0; i < 15; i = i + 1)$ {
$S_3$:      for $(j = 0; j < 25; j = j + 1)$ {
$S_4$:          $i = i + 2$;
$S_5$:          $a += A[i - 1][j - 1]$;
$S_6$:          $b += A[i - 4][j - 4]$;
$S_7$:      }
$S_8$:  }
```

(a) Original source code

```
char $A[15][25]$;

for $(i_0 = 0; i_1 < 15; i_2 = i_4 + 1)$ {
    for $(j_0 = 0; j_1 < 25; j_2 = j_1 + 1)$ {
        $i_3 = \phi(i_1, i_4)$;
        $i_4 = i_3 + 2$;
        $a += A[i_4 - 1][j_1 - 1]$;
        $b += A[i_4 - 4][j_1 - 4]$;
    }
}
```

(b) Code with SSA indices and select ϕ-nodes

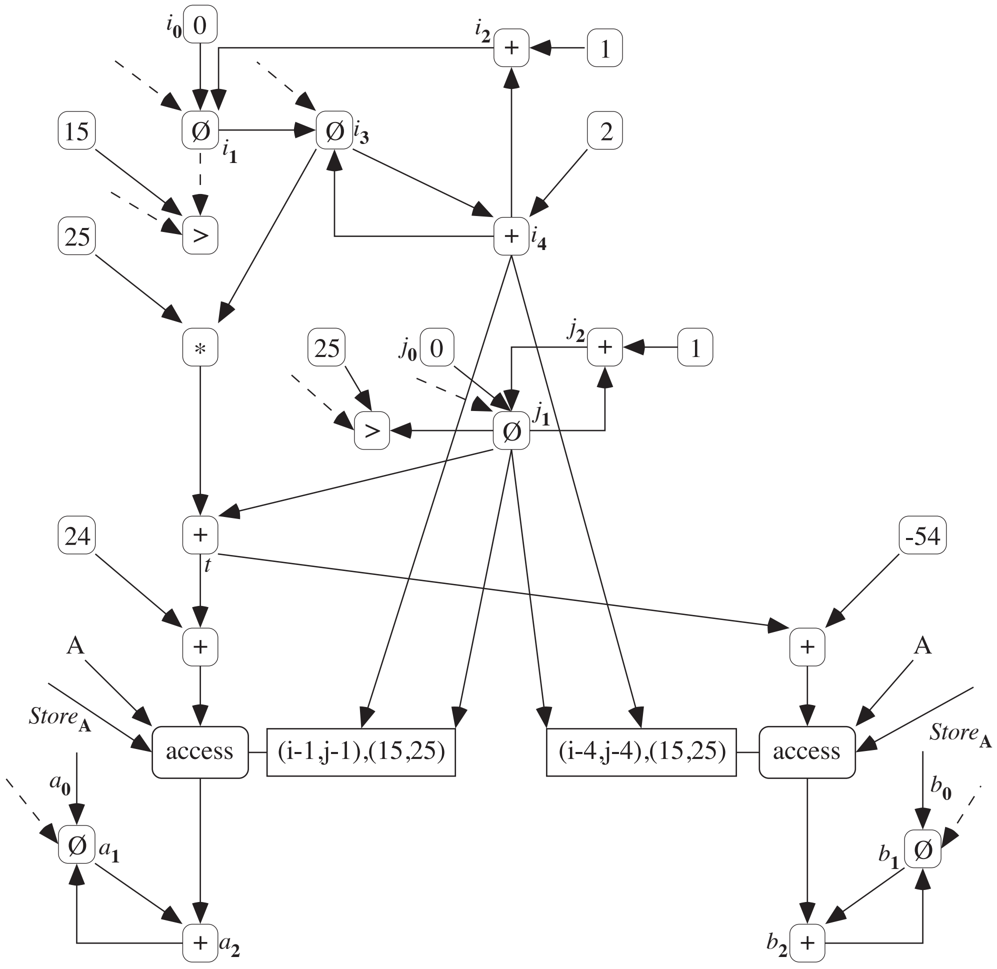


(c) Representation for code in 4(a)

Figure 4: Array Reference Example

```
S_1:  char A[15][25];

S_2:  for (i_0 = 0; i_1 < 15; i_2 = i_4 + 1) {
S_3:      for (j_0 = 0; j_1 < 25; j_2 = j_1 + 1) {
S_4:          i_3 = φ(i_1, i_4);
S_5:          i_4 = i_3 + 2;
S_6:          t = 25 * i_3 + j_1;
S_7:          a += A[t + 24];
S_8:          b += A[t - 54];
          }
      }
```

(a) Code from Figure 4(b) after optimizing array references



(b) Representation for code in 5(a)

Figure 5: Optimized version of Figure 4.

S_1: $f(x)$;
S_2: $fp(x)$;
S_3: $o.m(x)$;

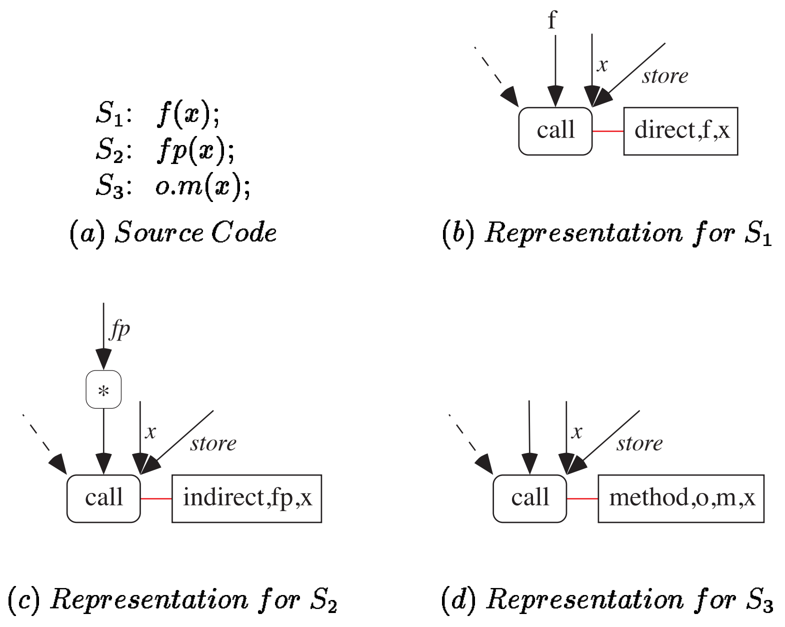


Figure 6: Procedure Call Examples: Part (a) shows three different types of calls found in C++. Parts (b)-(d) show the Score representation for each of these. Type information is not shown.

tion for object oriented languages. Low-level representations have a call node for representing procedure calls, but these nodes generally do not carry enough information about parameters. The problem is particularly acute for object oriented software where method invocations expand into a series of table lookups and pointer indirections. The indirect call inherent in a method invocation may be optimized by deducing constraints on the actual method called [29, 18, 26, 31, 54]. This deduction requires high-level knowledge about the method invocation.

Score distinguishes between four different types of calls: direct calls, indirect calls for languages with function pointers, closure calls for languages with nested procedures, and method calls for object oriented languages. At a low level these calls are identical. They input a control dependence, the address of the called routine, the arguments, and any necessary store values; they output a return value and any necessary store values. As shown in Figure 3.2.2, Score does distinguish between the different types of calls at the high level. For a direct call, Score records the name of the called routine, but this information is not (immediately) available for indirect calls. For method calls, Score represents both the object and the method which is being called. High-level analysis will also need type information that is not shown in Figure 3.2.2. The closure call, which does not apply to C, is also not shown but would be similar to a direct call.

Field References

Analysis of object oriented code benefits from a high-level representation for references to record fields [29]. Therefore, Score includes an annotation which holds the necessary high-level information. This annotation will be similar to the one for procedure calls.

3.2.3 Supporting Multiple Models

Our third contribution is to assemble the IR nodes necessary for compiling to multiple models of parallelism within a single IR. Most existing systems target only a few models of parallelism and so the IR represents only operations for a few target model(s) but not others. To compile for heterogeneous machines with the degree of diversity we envision, the IR must include nodes to represent the operations appropriate for a wide variety of models of parallelism. Though this is the first attempt at building an IR that supports so many diverse models of parallelism, the nodes can largely be borrowed from other systems such as XDP [8], the D System [64], and ParaScope [24]. It is unclear if we need to investigate new nodes for some models of parallelism or simply assemble existing ones. This feature meets our requirement of supporting compilation to multiple models of parallelism.

3.3 Additional Compiler Support

Click built his system on top of utilities provided by the Massively Scalar Compiler Project (MSCP) [15, 14]. MSCP includes Fortran and C compilers that output ILOC, a low-level intermediate representation. Click's system reads in ILOC, builds his representation internally, performs various optimizations, and outputs ILOC. MSCP also has a machine simulator that executes ILOC and collects statistics.

Enhancing Click's system to support parallelism and the PDG requires several changes. First, we must build a routine to convert his CFG based representation to a PDG based representation. Then we can still use the MSCP compiler to generate ILOC, and Click's parser to convert ILOC to his representation. Similarly, we will write a PDG to CFG conversion routine, and use Click's ILOC generation capabilities. Our new nodes require changes to the ILOC generator. Third, to support parallelism analysis, we will have to add dependence testing for subscript expressions. Several systems already do dependence testing (*e.g.*, SUIF and Sage++), so we plan to borrow and *adapt* their code. To support complete reordering of optimizations and analyses, we must be able to perform analyses which require a total ordering of operations at any time, which conflicts with our representation's goal of eliminating as many ordering constraints as possible. Our strategy will be to convert the PDG to a CFG [59], generate a legal ordering [21], run analyses, and then convert back to a PDG.

4 Experiments

4.1 Infrastructure

Our tests will include four models of parallelism: uniprocessor, SIMD, message passing MIMD, and distributed shared-memory MIMD. We have a simulator [16] for the IUA CAAPP [68], which is a SIMD machine. We expect to obtain access to a distributed shared-memory MIMD at one of the national supercomputing centers.

We are currently designing a compiler testbed, of which Score will be part. Though the design is not finalized, some details appear to be settled. The compiler will initially accept C, C++, and Fortran, and we expect to include Modula-3 shortly thereafter. We plan to use a Fortran 77 front end and a combined C and C++ front end from Eddison Design Group (EDG). We will build a tool to translate EDG's intermediate language into Score. We will also revise our annotation tool [28] to work on either the EDG intermediate language or Score. In addition, we are constructing a native code generator for the Alpha family of microprocessors. Click's IR is written in C++, so Score will also be written in C++. The annotation tool is currently written in C. The source language for the other tools is undetermined.

4.2 Evaluation

Though the impact of an intermediate representation is not directly measurable, we will perform experiments to evaluate the utility of our IR in meeting the goals of the overall project. The experiments are organized around several questions:

- *How effective is our IR for compiling to different models of parallelism?*

  In this set of experiments, we will compile a series of programs to several machines with different models of parallelism. These experiments examine three interrelated features of Score: its support for reordering transformations, its inclusion of multiple models of parallelism, and its reuse of transformations for different models of parallelism. We will select programs from standard benchmark suites as our test programs, which we will compile and execute on machines representing our four different models of parallelism. We will use the IUA annotation tool to overcome any deficiencies in our compiler analyses and the ILOC simulator to collect run time statistics. Since we do not have a planner, we will iterate our transformations multiple times. We will use the native compilers with the best matching optimization levels for comparison. Some of the codes may need to be hand modified to give the native compilers the same advantage the annotation tool gives us.

- *How important are specialized IR nodes in compiling for different parallel machines?*

  This set of experiments overlaps the previous test, but focuses on the role of specialized nodes (for representing different models of parallelism). We expect specialized nodes will improve the quality of the resulting object code. Hence, we will compare the performance of object code when the compiler can use specialized nodes versus object code when the compiler cannot use those nodes.

- *How does a small change in Score's operator set impact its memory usage?*

  Memory consumption is an important attribute of intermediate representations because it partially determines the representation's practicality. So in addition to measuring Score's memory usage, we will also evaluate if the selection of operators available to Score for representing programs impacts its consumption.

- *How effective is our IR for optimizing user-defined constructs?*

  In these experiments, we evaluate the benefit of optimizing user defined data types and operations. We expect the benefit to be maximized for code written using data abstraction. Hence, we will find C applications that use data abstraction and annotate them as necessary with our annotation tool. We will compare the execution time of the original application compiled with GCC versus that of the annotated application compiled by our system. We will try to match the level of optimization in each system as much as possible.

- *How extensible is our IR with respect to new transformations or new models of parallelism?*

  These experiments measure the extensibility of Score by estimating the ease with which programmers may modify the IR. During the course of the project, we will add new transformations and new IR nodes to support different models of parallelism. For each of these changes, we will record the incremental number of lines of code and the time required to make the change.

- *How does the performance of our resulting object code compare to that produced by Click's original system?*

  The final set of experiments is a regression test that quantifies any impact on execution performance caused by Score's new features. We will compare the execution performance of code compiled by Click's original system with that compiled with our new system. We will use only sequential code, because Click's system is only for uniprocessors. The two systems will not have identical transformations, but we will constrain our selection of transformations to those shared by both systems. This restriction will give us a fair comparison and illuminate advantages and limitations in Score's capabilities as an IR.

5 Summary

We have proposed a new intermediate representation, Score, that extends recent work in IRs to encompass the features needed for heterogeneous processing. Score separates the attributes with which transformations determine their applicability from the operation which a node represents. The separation allows the IR to be modified easily, facilitates the reuse of optimizations, and allows user-defined constructs to be optimized. Score also completes the trend towards IRs that allow transformations to be reordered by representing array references so that analyses and low-level transformations can both operate over them. Finally, Score collects together the node types necessary for a single compiler to generate code for diverse models of parallelism.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, MAf, 1 edition, 1986.

[4] Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, and Eduardo Pelegri'-Llopart. Experience with a graham-glanville style code generator. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 13–24, Montreal, Canada, June 1984.

[5] B. Alpern, M. Wegman, and K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[6] 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.

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 28(4):345–420, December 1994.

[8] Vasanth Bala, Jeanne Ferrante, and Larry Carter. Explicit data placement (XDP): A methodology for explicit compile-time representation and optimization of data movement. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 139–148, MAY 1993.

[9] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a repreesentation supporting control-, data- and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

[10] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[11] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. GURRR: A global unified resource requirements representations. In *Workshop on Intermediate Representations*, San Francisco, CA, January 1995.

[12] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Peterson, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[13] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Second Object-Oriented Numerics Conference*, 1994.

[14] P. Briggs. The massively scalar compiler project. Technical report, Dept. of Computer Science, Rice University, In Progress.

[15] P. Briggs and T. Harvey. ILOC '93. Technical Report CRPC-TR93323, Dept. of Computer Science, Rice University, 1993.

[16] James H. Burrill. The class library for the iua: Tutorial. Technical report, Amerinex Artifical Intelligence, February 1992.

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

[18] Brad Calder and Dirk Grunwald. Reducing iindirect function call overhead in c++ programs. In *Proceedings of the Twenty-first Annual ACM Symposium on the Principles of Programming Languages*, pages 397–408, Portland, OR, January 1994.

[19] R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, April 1980.

[20] Cliff Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Dept. of Computer Science, Rice University, 1995.

[21] Cliff Click. Global code motion, global value numbering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 30, June 1995.

[22] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *ACM SIGPLAN Workshop on Intermediate Representations*, San Francisco, CA, January 1995.

[23] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[24] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[25] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[26] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report 94-12-01, University of Washington, December 1994.

[27] Defense Research Agency. TDF specification. Technical report, Defence Research Agency, June 1994.

[28] Arundhati Dhagat. ANN: A software tool for annotatin IRIS-Ada graphs. Master's Project, June 1995.

[29] Amer Diwan. Analyzing statically-typed object-oriented programs for modern processors. In Progress.

[30] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The polaris internal representation. Center for Supercomputing Research and Development CSRD-1317, University of Illinois at Urbana-Champaign, February 1994.

[31] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[32] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[33] Kari Forester. IRIS-Ada reference manual. Arcadia UM-90-07, University of Massachusetts, May 1991.

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

[35] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

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

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

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

[39] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, St. Petersburg Beach, FL, January 1986.

[40] P. Havlak. Construction of thinned gate single-assignment form. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[41] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[42] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

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

[44] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaaban, and Cho-Li Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, June 1993.

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

[46] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation Research Institute, June 1993.

[47] Stavros Macrakis. The structure of ANDF: Principles and examples. Technical report, Open Software Foundation Research Institute, June 1993.

[48] J. McCarthy. Towards a mathematical science of computation. In *Information Processing*, pages 21–28, Holland, 1962.

[49] K. McKinley, S. Singhai, G. Weaver, and C. Weems. Compiling for heterogeneous systems: A survey and an approach. Computer Science TR-95-59, University of Massachusetts, July 1995. http://osl-www.cs.umass.edu/-∼oos/papers.html.

[50] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[51] Kathryn S. McKinley, Sharad K. Singhai, Glen E. Weaver, and Charles C. Weems. Compiler architectures for heterogeneous systems. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[52] Kevin O'Brien, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ron Unrau. XIL and YIL: The intermediate languages of TOBEY. In *Workshop on Intermediate Representations*, San Francisco, CA, January 1995.

[53] David A. Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for MPPs. Center for Supercomputing Research and Development CSRD-1306, University of Illinois at Urbana-Champaign, June 1993.

[54] Jens Palsberg and Michael I. Schwartzbach. Object-oriented. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Pheonix, AZ, October 1991.

[55] N. E. Peeling. ANDF features and benefits. Technical report, Defense Research Agency, 1992.

[56] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1), 1989.

[57] David S. Rosenblum and Alexander L. Wolf. Representing semantically analyzed C++ code with reprise. In *Proceedings of the USENIX C++ Conference*, pages 119–134, DC, April 1991.

[58] Erik Ruf. Optimizing sparse representations for dataflow analysis. In *Workshop on Intermediate Representations*, San Francisco, CA, January 1995.

[59] B. Simons and J. Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR 03.465, IBM Corp., Santa Teresa Laboratory, February 1993.

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

[61] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism. In *Proc. Hawaii International Conference on Systems Sciences*, Maui, Hawaii, Jan 94.

[62] 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.

[63] Peri Tarr, Alexander Wise, and Glen Weaver. IRIS-C++ 1.0 vs. IRIS-Ada 2.0: A comparison of the internal representations. Arcadia Technical Report UM-94-02, University of Massachusetts, November 1994.

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

[65] 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.

[66] L. H. Turcotte. Cluster computing. In Albert Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 26. McGraw-Hill, October 1995. ISBN 0-07-073020-2.

[67] Charles Weems, Steven Levitan, Allen Hanson, Edward Riseman, J. Gregory Nash, and David Shu. The image understanding architecture. *International Journal of Computer Vision*, 2(3):251–282, 1989.

[68] Charles C. Weems, Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, David B. Shu, and J. Grefory Nash. The image understanding architecture. *Internation Journal of Computer Vision*, 2:251–282, 1989.

[69] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the Twenty-first Annual ACM Symposium on the Principles of Programming Languages*, pages 297–310, Portland, OR, January 1994.

[70] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. The SUIF compiler system: A parallelizing and optimizing research compiler. *SIGPLAN*, 29(12), December 1994.

[71] Michael Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, CA, 1992.