AUTOMATIC CONSTRUCTION
OF
OPTIMIZING, PARALLELIZING COMPILERS
FROM SPECIFICATIONS

A Thesis
Submitted to the Faculty

of

Purdue University

by

William Eden Cohen

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

December 1994

ACKNOWLEDGMENTS

There have been many people at Purdue that have made it possible for me to complete my Masters and PhD work. First, I would like to thank my advisor, Dr. Hank Dietz, for the many insightful comments and the fequent discussions we had. I would also like to thank my officemates: Tai, Gayathri, and Terence. Their support, entertaining diversions from academic studies, and feedback about my research made my time at Purdue much more productive and pleasant.

I would like to thank the National Science Foundation for support during my Masters and PhD work at Purdue University through NSF grants 8624385A3-CDR and CDA-9312649. I would also like to thank the National Science Foundation for the infrastructure grant, 9015696-CDA, that provided the MasPar MP-1 for the School of Electrical Engineering's Parallel Processing Laboratory.

Finally, I would like to thank the Office of Naval Research which provided the grant, N00014-91-J-4013, to research the area of compiler and architecture interactions.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

ABSTRACT

Cohen, William Eden. Ph.D., Purdue University, August 1994. Automatic Construction of Optimizing, Parallelizing Compilers from Specifications. Major Professor: Dr. Hank Dietz.

Most people write their programs in high-level languages because they want to develop their algorithm and not concern themselves with the low-level details of the computer on which the program will be executed. However, execution speed of the program is still a concern. Translating a program written in a high-level language into efficient code targeting a particular parallel machine requires a powerful compiler to perform transformations appropriate for that particular machine. Developing such a compiler requires a great deal of effort, and much of this effort must be repeated for each new target machine.

A number of tools to assist in the generation of lexical analyzers, parsers, optimizers, parallelizers, and code generators have been developed. Each of these tools generates a part of the compiler from a specification, but the compiler writer must combine and augment these components to synthesize a complete compiler. In contrast, this dissertation suggests that complete compilers can be mechanically derived by processing a set of specifications defining the source language; available analysis, optimization, and parallelization phases; and the relevant attributes of the target computer. Further, an integrated framework for this process is presented; this includes new methods for description and management of parallelism, selection and ordering of analysis and transformation phases, and computation and use of estimates of execution time. Most aspects of this approach have been prototyped in a

proof-of-concept tool, the Parallel Instruction Generator Generator (PIGG), which is also described in this dissertation.

1. INTRODUCTION

A tool as defined by Webster's Ninth New Collegiate Dictionary is "an instrument or apparatus used in performing an operation or necessary in the practice of a vocation or profession" [Web83]. The design of current computer systems would be difficult without software tools, which significantly reduce the time and effort required to develop computer systems while managing details that would otherwise have been the designer's burden.

The hardware designer has many programs to aid in the development of computer hardware. An example of a relatively complete software tool that aids in the design of hardware is the Mentor Graphic CAD environment [Men93], which performs a wide range of tasks from schematic capture and circuit simulation to circuit board layout and thermal analysis. Tools such as this make it possible to design extremely complex hardware in a relatively short amount of time.

However, a computer system does not consist solely of hardware; the software is also an important component of the computer system. Few people program the "raw" machine; instead, they resort to using a compiler to map their relatively machine-independent representation of the computation into efficient code for the target machine. Thus, for most people programming the computer, the execution model and the performance observed is not that of the hardware, but rather the performance of the entire system, both hardware and software. Classic benchmark sets such as NAS [BBB+91], Perfect [CFN90], Linpack [DBMS79] [Don92], and SPEC [Dix92] all measure the integrated performance of both the hardware and software.

As a variety of software tools simplify hardware development, other tools reduce the amount of time and effort required to produce a good compiler for a given machine.

The use of software tools for certain aspects of compiler construction has become commonplace, e.g., most front-ends for compilers are now constructed with the aid of lexical-analyzer generators such as LEX [Les75] and parser generators such as YACC [Joh75].

The typical compiler takes a high-level language program as input and translates it into a sequence of machine-executable instructions. Many early compilers accomplished this translation by directly replacing each construct in the high-level language with a group of machine instructions that would perform the same function. This syntax-directed translation leads to relatively inefficient code when the execution models for the input language and output language differ significantly. The generated code incurs extra overhead because it emulates the execution model of the input language. Thus, compilers for high-performance computers perform extensive analysis and transformation to efficiently map each program to the execution model most appropriate for that target machine.

With an appropriate set of specifications, it is possible to mechanically select, build, and integrate all the components needed to implement an effective compiler for a wide range of machines, including those supporting various types of parallel execution. Towards this goal, we propose a framework, and a set of tools, that can be used to automatically generate complete compilers for a wide range of source languages and target machines. Although all aspects of automating compiler construction are addressed, the primary contributions of this dissertation lie in three specific areas. What information must the intermediate representation (IR) encode? How can the salient features of the target machine be accurately and concisely specified? Given a library of analysis and transformation phases, how can a tool select the phases appropriate for the specified target machine and synthesize a compiler from them?

The intermediate representation defines the limits of what can and cannot be translated. Every instruction for the target machine is described as a mapping between the intermediate representation and output instruction. Thus, due to the limitations in the intermediate representation, it may be impossible to describe certain useful machine instructions. Either the performance of the code generated by the compiler will suffer or the compiler writer must add hand-written code to the compiler that will produce these instructions. In addition to translating the intermediate representation to machine instructions, most of the transformations take the intermediate representation as input and produce a modified intermediate representation as output. Characteristics of a flexible intermediate representation that can be used for a wide variety of machines are discussed in this dissertation.

Part of the machine description specifies the mapping of the intermediate representation to the instructions of the target machine. Each of the instructions described covers a portion of the intermediate representation of a program. Thus, to cover the entire intermediate representation of a program, multiple instructions will be required. There may be many possible coverings of the intermediate representation by the available instructions. Each of these coverings of the intermediate representation may have a different cost. The goal of the compiler is to select a covering that has a lower cost, indicating that the generated code is of better quality than other translations. Quality is typically measured by execution speed. So that the compiler can estimate execution speed, the machine specification associates timing properties with each unit of hardware, and each instruction's description summarizes how hardware function units are used. Thus, this dissertation links the hardware implementation of the target machine with the instructions to obtain a cost for each mapping.

High-performance compilers do not directly translate the program's initial intermediate representation to the instructions of the target machine. Transformations are performed on the intermediate representation to make the intermediate representation better match the target machine's execution model. These modifications to the intermediate representation of the program improve the performance of the code generated to implement the program. Because these transformations work on the intermediate representation, they often can be used within compilers for several different architectures.

Even though transformations operate on the intermediate representation and hence could be applied to almost any architecture, some transformations do not improve the quality of the code generated for particular execution models. For some execution models, certain transformations reduce the quality of the code produced. Thus, it is important to select the transformations appropriate for the target machine. With the information linking the instructions to the hardware implementing the instructions, many of the characteristics of the target machine can derived from the machine description. This information about the machine characteristics, combined with information about the compiler components, can be used to select the compiler components appropriate for producing a compiler for each target machine.

We claim that the complete compiler construction process can be automated, and our research suggests how, but such a radical claim should be backed by some experimental evidence. As both an illustration and a proof-of-concept, a prototype tool was constructed: the Parallel Instruction Generator Generator (PIGG), which was designed to work with the Purdue Compiler Construction Tool Set, PCCTS [PDC91].

Figure 1.1 illustrates a compiler constructed with the assistance of the prototype tools. Each of the boxes in the figure represents a component that could be constructed by a tool. The figure shows the general flow of information from one component to another in the compiler. Multiple passes may be made through "phase selection" and analysis and transformation phases. Once all the appropriate analysis and transformations phases have been executed, the intermediate representation is passed to the phase that selects the instructions to implement the intermediate representation and schedules the instructions.

The order that information is presented in this dissertation is the order that it required to construct the various components of the compiler. Figure 1.2 shows the flow of information required to create the components in the compiler. The small boxes along the top of the figure are supplied by the compiler writer, the tools are indicated by dashed boxes, and the final compiler components generated by the tools

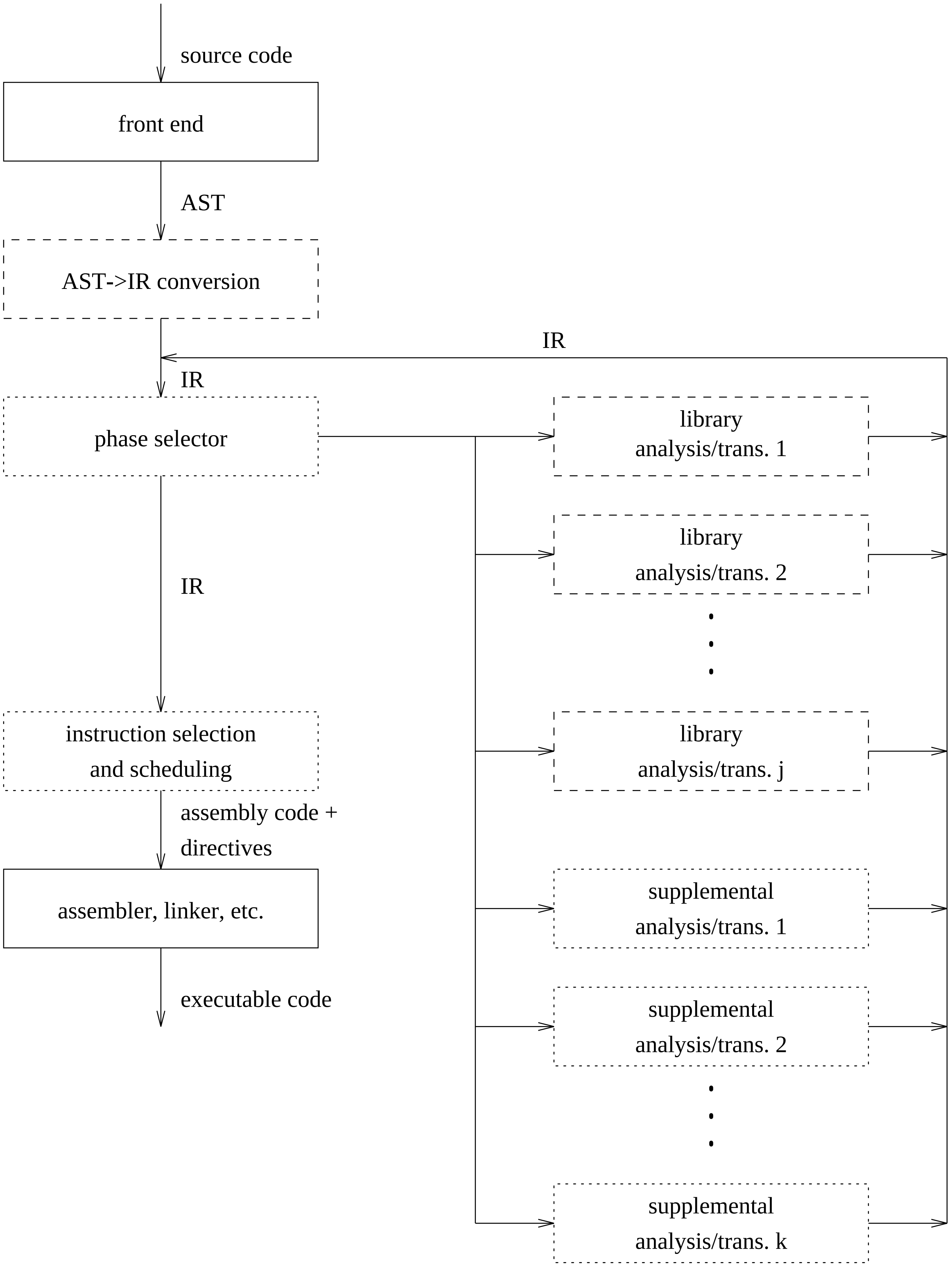

Figure 1.1 Overview of compiler of generated with thesis tools

are indicated by the small dotted boxes at the bottom of the figure. The arrows show where each tools obtains required information and which components are generated by each tool. Each smaller box is contained within a larger dotted box that indicates the chapter that discusses that item. A more detailed description of each chapter's content follows.

Chapter 2 discusses tools designed to automate the construction of the components that convert the source code into an intermediate representation of the program's syntactic structure. The front end's intermediate representation may be an Abstract Syntax Tree, and an additional conversion step may be required to convert the AST into the desired intermediate representation (IR) as shown in figure 1.1.

The intermediate representation is used to carry information between phases of the compiler. Chapter 3 describes various proposed intermediate representations and the characteristics required of the intermediate representation to allow it to be flexible for a wide range of transformations and target machines.

The phase selection module of the compiler applies the analysis and transformation phases of the constructed compiler in the appropriate order. Some of these phases may be applied multiple times. The method used to construction this portion of the compiler is detailed in chapter 6. The construction of the analysis and transformation phases is outside of the scope of this dissertation; a library of pre-existing phases is assumed to be available. However, to automate the selection and ordering of the phases, information about the characteristics of each of the phases is required. To select phases appropriate for the target machine's execution model, information about the target machine also must be gathered.

To provide the information about the target machine required to construct machine-dependent components of the compiler located in the instruction selection and scheduling component of the diagram, chapter 4 outlines the machine description language. In addition to describing the machine instructions and their mapping to the intermediate representation, this machine description language also maps the instructions to the hardware of the target machine.

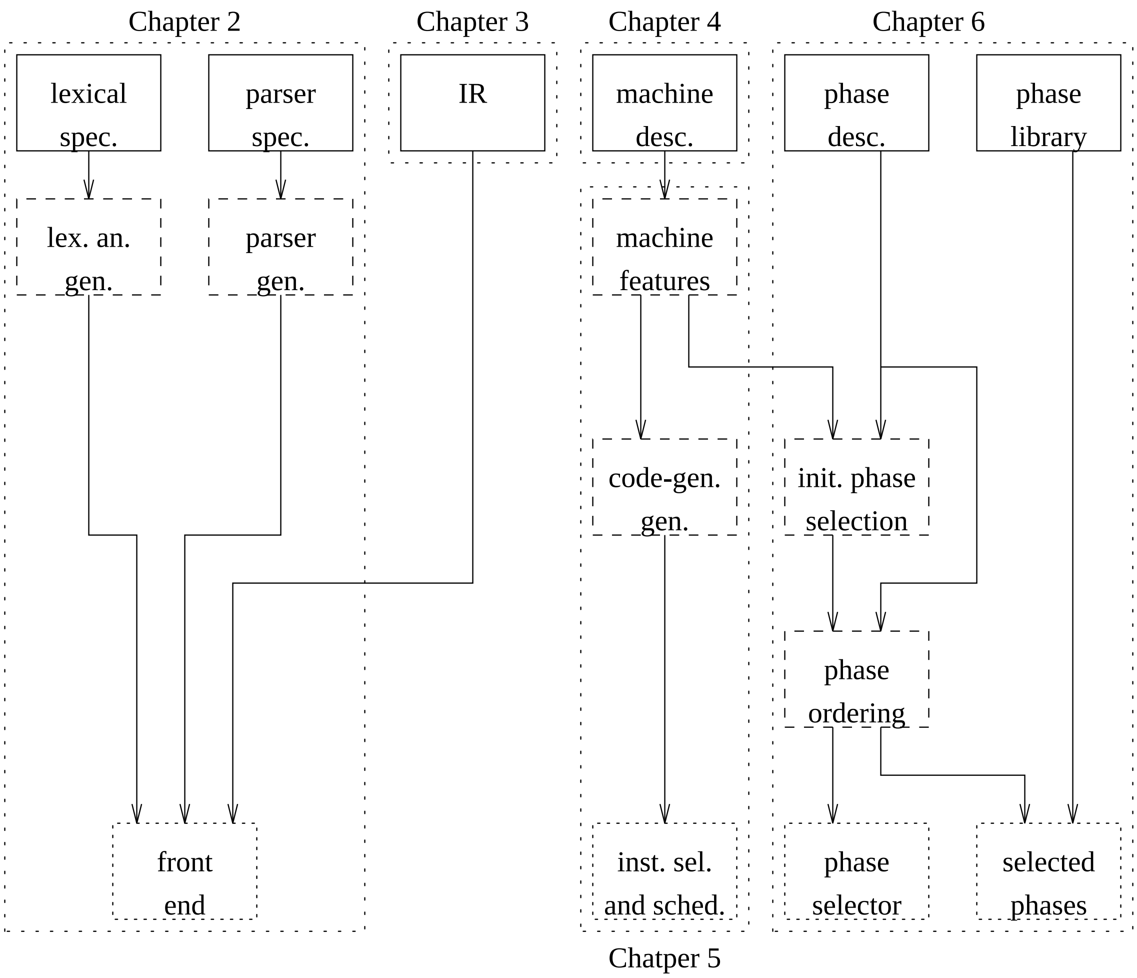

Figure 1.2 Data flow in creation of compiler

The mapping of the machine instructions to the hardware of the target machine allows a great deal of information about the target machine to be derived from the machine description. The methods used to derive the machine features from the machine description are given in chapter 5. The phase selection and ordering described in chapter 6 also uses this information.

To test some of the theory presented in this dissertation, a prototype tool, PIGG, was built. Chapter 7 describes the prototype tool and how the tool works with the other tools in PCCTS. Preliminary experiments using PIGG to generate compilers for parallel machines are described in chapter 8.

Finally, chapter 9 summarizes the dissertation and describes future avenues of research in this area.

2. FRONT END

No matter how trivial the compiler may be, every compiler must incorporate some method for recognizing the structure of input phrases. The first compilers intertwined hand-crafted syntax recognition and code generation. Later, more sophisticated compilers separated the language recognizer from the code generator; the language recognizer became part of the machine-independent front end. Thus, it is not surprising that the automatic production of recognizers from specifications was one of the first problems in compiler generation to be solved for most practical cases.

Traditionally, the language recognition problem has been divided into the lexical analyzer and parser. The lexical analyzer groups the individual characters into lexemes. The parser groups sequences of lexemes into phrases.

When early computer languages, such as Fortran [Bac81] and Cobol [Sam81], were designed, the problems of specifying language structure and implementing recognizers were not well understood. As a result, these languages had many syntactic quirks and inconsistencies. In some cases, a particular compiler's implementation was the specification of the language. For example, as the foreward of the ANSI C specification points out [ANS90], the "de facto C programming language standard" was the C book by Kernighan and Ritchie [KR78], which was sufficiently open to interpretation that a "variety of differing translator implementations" had come into common use.

With experience, language design became more rigorous, and languages were specified before they were implemented. In later languages, such as Algol 60 [Nau63] and Ada [DoD83], a specification of the grammar for the languages existed before the language was implemented. The grammar in the language specification, rather than an early implementation of a compiler, defined the syntax of the language. To concisely and unambiguously indicate the syntax of computer languages, specialized grammar notations were developed. Backus-Naur Form (BNF), which was used in the Algol 60 [Nau63] language description, is an example of a grammar specification language.

However, given a grammar, it is not sufficient to merely construct a program that can determine if the input conformed to the grammar, i.e., to construct a recognizer. There must be some means to effect a translation of the input either into the output language or some intermediate representation. This translation requires augmentation of the recognizer.

In addition to the mechanics of building the front end, the design of the language must be considered. The language design encompasses both the syntax and semantics of the language. The design of the language is influenced by many factors, such as the type of applications users will write, the recognizer technology that can be used, and the computers targeted. In particular, parallel computing significantly complicates language design.

These aspects of producing a compiler front end are covered in this chapter. Section 2.1 discusses the specification and generation of lexical analyzers, which groups characters into lexemes used by the parser. The specification and generation of parsers, which recognize the input language, is discussed in section 2.2. The translation of the parsed input into the initial intermediate representation is described in section 2.3. Finally, parallel language design issues are discussed in section 2.4.

2.1 Lexical Analyzers

Although the lexical analyzer groups characters together in the same manner as a parser, there are two reasons for having a software module separate from the parser accomplish this task. First, it simplifies the parser. The parser can sidestep issues such as handling of white space or special lexical conventions. A good example is the column-sensitive meaning of characters in Fortran 77 [Ett83]. The second reason is efficiency. The specialized recognizers for lexical analyzers are often simpler and faster than the code used to implement the parsers. Thus, to improve performance of the compiler, a specialized recognizer is used for lexical analysis.

Some languages, such as Lisp [SG93], require only simple lexical analyzers, which easily can be written by hand. However, many languages require more complex lexical analyzers. Extremely fast lexical analyzers for these more complex lexical problems can be implemented either directly in code or by a small interpreter loop, which only executes a few instructions to recognize each character of input [FH91]. For example, Deterministic Finite Automata (DFA) [HU69] or Non-deterministic Finite Automata (NFA) [HU69] yield simple state-machine implementations. It is possible to write these types of lexical analyzers by hand; however, for most computer languages, this becomes tedious and error prone because of the size and complexity of the resulting state transition matrix.

To reduce the amount of effort required to build these fast lexical analyzers, a number of tools have been implemented. Each of these tools takes a specification and builds a lexical analyzer that groups characters into lexemes. Examples of widely available lexical analyzer generators include LEX [Les75], FLEX [Pax90], and DLG [PDC92]. When the lexical analyzer successfully groups characters into a lexeme, it executes an action associated with the pattern matched. The action often only indicates to the parser that this token has been found. In other cases, the action may be more complicated, e.g., converting a character pattern that is a floating point number into the compiler's internal representation of that number. In most cases, the information is passed to the parser. However, lexemes not significant to parsing the language, such as white space and program comments, may simply be recognized and ignored.

Because existing tools are strong enough to directly handle the creation of lexical analyzers for most computer languages, lexical analysis is not a limiting factor in automating construction of complete compilers.

2.2 Parsers

Even when grammar specifications exist, parsers are often implemented by hand due to the lack or inappropriatness of tools; however, parser generation tools are also becoming strong enough to be used in a complete compiler construction system. Parsing theory has developed a number of parser algorithms, such as LR(k) [Knu65], LALR(k) [DeR71] [DeR71], and LL(k) [RS70] [Par93]. With the development of grammar analysis, it is possible to determine which grammar specifications could be implemented using a particular parsing method. This is an important capability for compiler tools. If a parser generator blindly generates an incorrect implementation of the parser without warning, it may make it very difficult for the programmer to determine what is causing an error. Worse yet, an error may surface much later in the development of the compiler, perhaps even after it is released.

In addition to determining whether a grammar can be implemented using a particular parsing method, parsing theory provided algorithms to translate from the grammar specification into parser implementation. Numerous "compiler-compilers" or parser generator tools have been developed to implement parsers from grammar specifications. Examples of widely available parser generators which generate LALR(1) table driven parsers include YACC [Joh75] and Bison [Fre92]. There are also tools that generate LL parsers such as ANTLR [PDC92], which produces LL(k) parsers coded in C, and Coco-2 [DP90], which generates LL(1) parsers coded in Modula-2.

The available parser generators have influenced the design of computer languages in that the syntactic structures of many computer languages have been designed to meet the restrictions of particular parser generators. Thus, for many computer languages, it is relatively straightforward to implement a parser for the language, and the task of building parsers has been well automated.

However, by default, most parser generators produce only recognizers, i.e., a program that merely determines whether the input is valid in the language. For a compiler, a translation from the input language to the output language must be effected.

Thus, the parser must be augmented to do more than simply recognize phrases in the input language.

2.3 Intermediate Representation Construction

Most parser generation tools need actions to be inserted into the grammar to translate the parsed input into an appropriate form. For a simple compiler, the translation can be done completely by the actions inserted into the grammar specification. However, for more advanced compilers, the translation is not completed in the language recognizer; rather, an intermediate representation of the input is produced by the parser and this intermediate representation is manipulated by other phases of the compiler and translated to the output language.

While input to the parser is a sequence of lexemes, the typical intermediate representation used by other sections of the compiler may not be linear. This difference derives in part from the fact that meaning is associated with nested phrase structures, hence, an intermediate representation that allows nesting is often more useful than a linear form. Commonly used intermediate representation structures include trees, Directed Acyclic Graphs (DAGs), and tuples. The parser must generate the appropriate intermediate representation to pass information about the input being recognized to the other sections of the compiler. The particular characteristics of the intermediate representation are discussed in more detail in chapter 3.

The parser recognizes the syntax of the input; creation of the intermediate representation requires semantic information. Most parser tools manage this semantic information by associating attributes with each grammar symbol (i.e., with each terminal or non-terminal). These attributes contain information about the symbol (e.g. variable type, constant value, symbol table location) and the information can be referenced within the actions to build the intermediate representation. In parser generation tools, such as YACC [Joh75] and ANTLR [PDC92], the actions in the grammar specification can be arbitrary C code. When a production in the grammar specification is recognized, the associated action is executed to accomplish the translation between the input language and the intermediate representation.

Some tools, such as ANTLR, have added the capability to build Abstract Syntax Trees (ASTs) that record the syntactic nesting of phrases by constructing a tree-structured representation of the input recognized. The AST may form one of the intermediate representations used within the compiler. Thus, the effort required by the programmer to produce the intermediate representation may be reduced. The programmer can specify which symbols to include, where to place them in the AST, and which symbols to ignore with just a couple of annotations in the grammar specification. However, the programmer may still need to add actions to the parser specification to produce the appropriate intermediate representation. The complexity of generating the intermediate representation is closely related to the design of the input language.

2.4 Language Design

Although existing languages imply rigid guidelines for how each construct is to be recognized and executed, defining a new language or dialect allows the language designer some degree of freedom in adjusting the syntax and semantics. There are two areas that should be addressed when designing a language. The first is the ability to build a parser for the language. It must be possible to build a parser for the language using the available technology. The second is the ability to generate efficient code for each of the target machines. Some language constructs are difficult and inefficient to implement on particular computer architectures, e.g. independent threads of control on a SIMD machine.

Being able to implement a parser has not been a major problem. Most languages, e.g. Fortran [Bac81], Cobol [Doc73], C [KR78], HyperC [CP94], Pascal [Wir93], and Sisal [MSA$^+$85], have been designed to match the abilities of the available parser technology. Thus, the syntax of the languages was restricted in a manner to make it possible to construct the lexical analyzer and parser generators with available parser technology.

The second concern is also relatively minor: in most cases, porting between different target machines is feasible. However, portability issues for languages targeting parallel machines are far more troublesome. Care must be taken not to bind the execution model for the language too tightly to one particular computer architecture, so that programs become very inefficient to execute on other computer architectures. One of the advantages of using a high-level language is allowing a program to be ported from one target machine to another by simply recompiling the program, and this would be lost if the program must be extensively modified for it to execute with reasonable efficiency on other target machines.

This section covers some of the work done by other researchers to provide high-level languages that run on a variety of parallel machines. This is not an exhaustive literature review, but rather a discussion of issues that need to be addressed when designing a language that must execute with reasonable efficiency on a number of different parallel architectures. Section 2.4.1 describes the work done in XPC [Phi89] to keep the language relatively machine independent. The design of a parallel Fortran standard, HPF, is discussed in section 2.4.2.

2.4.1 XPC

Early parallel programming languages were closely tied to the machine that the language was originally intended to execute on. Thus, there were languages that had synchronous models of execution that followed SIMD semantics, and other languages that had asynchronous models of execution that followed MIMD semantics. XPC (eXplicitly Parallel C) attempted to unify and resolve the differences between asynchronous and synchronous execution models for parallel languages. XPC only defined the semantics of the input language and did not specify other aspects of the compiler, such as the intermediate representation. The syntax of XPC is based upon Kernighan and Ritchie C [KR78]. There have been a number of extensions to the base language to accommodate arrays as first class objects, to allow the declaration of parallel data structures, and to express parallel programming constructs.

2.4.1.1 Control Flow

In a MIMD model, it would be possible to have each one of the processing elements executing a different code image. However, in a SIMD model, there is only one thread of control and only one code image. Thus, to keep the two models consistent, the MIMD processors are also limited to one program code image. This is not as limiting as it initially appears. Given that each processor is uniquely numbered, a conditional test can be done to allow each processor or group of processors to select a different group of statements in the code image. In the logical extreme, separate programs for each processor can be unified into a single code image by concatenation, and each processor would execute only the portion of the unified image that corresponds to its program. Thus, restricting the program to a single code image does not impose any restrictions on the program, but may yield larger code images

Synchronous languages have more restrictive semantics for `if-then-else` statements. There is an ordering of execution across the processing elements in the execution model; the statements are executed in the `then` clause before the statements in the `else` clause. Asynchronous programming languages make no implicit ordering of the `then` and `else` clauses across the execution model. Therefore, if the ordering of the clauses makes a difference, the synchronous programming model's semantics are the ones that should be followed.

A synchronous language model characteristic that is not mentioned in XPC, but is apparent in MPL [Mas92c] is that if one processor is active and a function call is in the code, the control unit will execute that code. If there is an `all` statement executed within the function call, all processing elements will become active. This differs greatly from the asynchronous programming model, since it implies that processing elements must be able to execute code that does not appear on that processing element's MIMD control flow path.

2.4.1.2 Synchronization

Synchronous languages tend to package the synchronization within the language constructs. This models the SIMD mode of execution available on the machine originally targeted by the language. If a direct translation of a SIMD construct to code for a MIMD machine is performed, performance may be degraded significantly. However, some traditional synchronous languages have be successfully implemented on MIMD machines [QHS91] [Sab92]. This is made possible by two things. The first is compiler technology that can remove unnecessary synchronization. The second is use of hardware to implement a barrier synchronization across processors in a MIMD machine, rather than using a large number of direct synchronizations between pairs of processors, because this can significantly reduce the overhead of implementing the synchronization required.

2.4.2 High Performance Fortran

A number of different modifications to Fortran have been proposed [HKK+91] [BCZ92] [OPE+93] to enable programmers to write programs that make better use of parallel machines. The most visible standard is High Performance Fortran [Zos93], HPF, which is designed to execute on a variety of parallel architectures. This standard has been influenced by earlier parallel extended versions of Fortran such as CM Fortran [Thi89a] and MasPar Fortran [Mas92b].

HPF is described as an extension to the Fortran 90 standard, but early versions of HPF are an incompletely specified subset of Fortran 90 [Zos93]. The areas that the HPF committee has focused on to make the language portable and efficient on various parallel architectures are the control flow (discussed in section 2.4.2.1) and the distribution of data (discussed in section 2.4.2.2).

2.4.2.1 Control Flow

Because an HPF program may be executed on either a SIMD or MIMD architecture, a data parallel model is used for the language. This model was selected because

SIMD and vector machines cannot have more than one stream of control flow, and it would be inefficient to emulate multiple streams of control flow on these machines. Even on multiple processor machines, all the processors conceptually execute the same statement at the same time. However, asynchrony between the processors is allowed if it does not affect the results of the computations.

2.4.2.2 Data Distribution

The method that HPF uses to enhance the performance of Fortran is by using data parallel constructs. Computations that operate on arrays expose the parallelism of the program to the HPF compiler. Thus, rather than performing only one computation within a single statement, the number of elements in the array are the number of operations performed.

One of the impediments to performance that the HPF committee identified was the decomposition and layout of the data in arrays. Particular data layouts may allow operations to be performed concurrently and keep the communication overhead low. Thus, a number of directives have been included in the language to allow the programmer to control the layout of the arrays. There are two basic categories of directives: alignment and distribution. Both types of directives specify where to locate data across the processor in the machine. However, the alignment directives locate data relative to other data layouts.

There are two problems with the current set of layout directives in HPF. The performance directives are machine specific, but there is no means available to indicate that. The second problem is that the HPF assumes that machines can be modeled with single-level parallelism across processors in the machine, but many machines have nested hardware parallelism and other multi-level structures.

Performance tuning for a particular machine must be done with the data layout and alignment directives. However, there is no means available to indicate that these directives apply only for a particular type of architecture. Thus, when compiling the program on another, machine the directives will still be visible, but may be inappropriate for the other machine. Because the directives are always visible, optimizing the directives for a new machine requires that the old directives be replaced by new ones. Thus, it is unlikely the same source program can be used on dissimilar architectures without incurring significant performance penalties on at least one of the architectures.

The second problem stems from the HPF's single-level model of parallelism being spread across processors. Because many parallel machines are now using high-performance microprocessors as the processing elements, there are other architecture features within the machines that affect the performance of the machine. Most machines have hierarchies for both memory and parallelism.

An example of the performance difference caused by the memory hierarchy within a single processor is the DEC Alpha 21064 [Dig92], the processor used in the Cray T3D [Cra93]. This implementation of the Alpha architecture includes an 8 KByte data cache and an 8 KByte instruction cache. Both of these on-chip caches are direct mapped [KS93]. In a direct-mapped cache a memory address always maps to the same cache line and there are a number of other memory addresses that map to the same cache line. These caches are the only caches within a processing node on the Cray T3D. Therefore, certain memory access patterns will lead to very poor cache performance and low processor utilization caused by the constant swapping of entries between cache and memory. However, HPF has no directives to control the layout of data with a processor.

Similarly, several of the current parallel architectures have multiple levels of parallelism, each with a different granularity. The Convex Exemplar architecture [Con93] is an example of such a machine. Within a node there are eight processors. Communication between processors within the node is much less expensive than with processors outside the node. The HPF parallel arrays can have several dimensions distributed across the machine, but these are all assumed to be at the same granularity, and there is no way to specify how to lay out the data between levels in the hierarchy to make best use of the machine.

2.5 Conclusion

This chapter has discussed the issues relative to building a compiler front end for a compiler targeting a parallel machine. Although research is continues in parsing theory, the techniques and tools have been well developed. Building a parser for a well designed language does not pose a significant problem.

However, preserving the semantics of the language and representing the meaning of a program is a more difficult problem. Chapter 3 will discuss intermediate representations used within the generated compiler.

3. INTERMEDIATE REPRESENTATION

A key to producing high performance code is to translate the code into the execution model available on the machine rather than having the target machine emulate the input language's execution model. Syntax-directed translation makes it difficult to significantly change the execution model of the code, because the translation is based totally on the input language. To allow significant modifications to the code, an intermediate representation of the input program is made. The intermediate representation must be able to represent phrases of both the input language and output language of the compiler. A number of transformations are performed on the program's intermediate representation to modify it to match the execution model of the target machine.

The intermediate representation provides a media for expressing the program. Thus, often it is the interemediate representation that limits what can be described within it. Hence, the intermediate representation is a extremely important component of the compiler. If the intermediate representation is poorly designed, then the ability of the compiler to convert an input language to output language will be severely limited.

Early compilers targeting parallel machines avoided many of the issues involved with developing an intermediate representation by either designing an input language that was very closely tied to the target machine as Glypnir [LLBR75] or designing the compiler be a source-to-source translator as Parafrase-2 [PGH+90]. Glypnir's conversion from input to output language could be accomplished with syntax directed translation. Although Paraphrase-2 performs a number of different transformations, the execution models of the input and output language are very similar. Paraphrase-2 avoids many of the details of machine restrictions by generating FORTRAN code with directives as the output language, and the final translation into the native code of the target machine must be completed by another compiler.

Solutions such as those taken by Glypnir and Paraphrase are not suitable for implementing a portable parallelizing compiler. Either the input language will be hopelessly tied to a particular target machine as in Glypnir's case or the intermediate representation will be restricted to a high-level language as in Paraphrase-2's case.

Intermediate representations that have been used for compilers targeting parallel machines will be discussed in section 3.1. Section 3.2 describes the intermediate representation used for this thesis work.

3.1 Previous Work

The following sections will discuss three intermediate representations that have been used in compilers targeting parallel machines. The intermediate representations use several different execution models; RTL [Sta93] [Bur93] and VCODE [BC90] have been used for data-parallel languages with serial control flow, while GR [Sar89] has a concurrent task execution model with single assignment semantics. McCat's intermediate representation [HDE$^+$93] is a test bed intermediate representation that builds upon the RTL intermediate representation.

3.1.1 RTL

To separate the front end and back end of the GNU C compiler [Sta93], RTL (Register Transfer Language) is used to communicate information between the front-end, the optimizations, and the back end. This isolation of the back end is essential because of the large number of architectures the compiler generates code for; the 2.5.0 release of GCC include code generators for 27 different processor architectures.

RTL has been heavily influenced by the C language's execution model. The available data types, execution model, and memory model of RTL are tailored to C's execution model and their impact on code generation for parallel machines will be covered in the following sections. Even though RTL was originally designed for C, parts of the compiler have been used for other languages such as Ada [CGS93]. The GNU C compiler has also been used as a basis for compilers for SIMD architectures [Mas92c] [Bur93]; the modifications for these compilers will also be discussed in the following sections.

3.1.1.1 Control Flow

In the original execution model of C there is only one flow of control for the program. This characteristic also exists on all the processor architectures that the code generators in the 2.5.0 release produce code for. Because both the input and output languages have single flow of control, it is only natural that RTL can only express a single flow of control. All the parallelism due to multiple flows of control, e.g. execution on a MIMD machine, must be managed by the programmer. Even if the operating system would starts a number of concurrently executing copies of the compiled program, no communication or synchronization would occur between the executing task unless additional communication code was inserted into the source code by the programmer.

3.1.1.2 Memory Model

The RTL model has two basic types of locations to store data values: memory (`mem`) and registers (`reg`). Code is generally considered to be stored in a separate and distinct read only memory, because C does not allow the program to be modified while executing.

Any C variable that may have a pointer referring to its location must be located in memory. The C model of memory becomes apparent when examining how C handles larges arrays; large arrays must be contiguous in memory. When pointers are used to access items of a certain type, all the items must be accessed in the same manner (at least from the source code); regardless if the object is stored on the stack or a static location. This model of memory being a single uniform block of locations is also in RTL.

Many parallel machines have a segmented memory model, with several classes of storage available. Also in many machines large arrays must be distributed for two reasons. First, local memory in many cases is not large enough to hold the entire array and each processing element can only hold a section of it. Second, to achieve good performance, the array should be distributed to minimize access time for the processing elements.

An additional type of memory in RTL are registers (`reg`), which can store compiler generated temporary values and user variables that are scalars and do not have their addresses taken. These registers do not necessarily relate to the physical registers in the target machine. The GCC C compiler maps the abstract RTL registers into registers physically present in the machine as a later step.

In the standard RTL, each register is typed and can store a single scalar value of that type. This is a problem on machines that have multi-element registers such as vector registers. These items can not be represented in the standard RTL notation. The following section on typing will discuss how compilers based on the GNU C compiler circumvented this problem.

3.1.1.3 Types

All values in the registers are typed, as quarter integer, half integer, integer, double integer, float, and double float. This typing mechanism has been used as a mechanism to use RTL as an intermediate representation to generate code for SIMD machines in the MPL compiler [Mas92c]. Types indicate that the variable is stored on the processor array of the MasPar MP-1 are added to the compiler. Since operands of operations are typed, the correct instructions are generated for values that are stored on the parallel section of the machine. The data movement to and from the parallel part of the machine is modeled as a type casting.

This particular example shows a weakness in RTL for parallel machines. The compiler still treats the parallel variable as a single value, which causes two problems. First, it is impossible to perform any transformations that require treating the entire array variable as anything other than one monolithic element. An example of this is either all the values in the array are live or all the values in the array are dead; the attributes about the array must refer to the characteristics of the entire array. Second, the intermediate representation can only represent a data type that is directly available on the machine. Arbitrary sized array are hard to produce using the typing available in RTL.

This problem with arbitrary sized arrays is particularly visible in MPL [Mas92c], a SIMD dialect of C which is based on GCC. The plural keyword indicates a variable that is stored on the data parallel section of the machine. The number of elements a plural variable contains is always the natural size of the machine. A MasPar MP-1 [Bla90] can have 1,024; 2,048; 4,096; 8,192; or 16,384 processing elements. The size of the plural variable is equal to the size of the machine the code is executed on. Thus, to run the same executable across the MP-1 family of machines, the programmer must added virtualization code to the program to account for the variations in the parallelism available.

With the typing information and type conversions available in GCC that it would be possible to add some types and conversion rules to handle the virtualization. However, this is not practical, because the typing in GCC is not parameterized. Each possible type would have to be enumerated. Thus, there would be a different type for each combination of data type and array dimensions, leading to a immense number of types and type conversion rules.

Even worse is the manner to extract elements from an parallel data type and converted to another type or parallelism. Each possible conversion from one parallel data type to different type must be described, results in approximately n^2 rules to describing conversions for n data types.

3.1.2 McCat

The McCat compiler [HDE+92] is a test bed for compiler technology. The purpose of this compiler is to be able to provide a tool set that allows compilers for fine-grained parallelism to be developed. To make this possible the McCat compiler uses a family of intermediate representations for manipulating programs. These intermediate representations are broken into two basic categories: high-level intermediate representation (FIRST and SIMPLE) and low-level intermediate representation (LAST). Transformations and analysis for coarse-grained parallelism improvement are generally performed in the high-level intermediate represenatations, while low-level optimizations and scheduling, which are machine dependent on the target machine, are performed in the low-level intermediate representation.

Parts of the McCat compiler are based on other available tools. The front end is based on the GCC [Sta93] C Compiler, which produces RTL. Thus, FIRST, the initial intermediate representation generated is based on RTL. However, there are extentions to allow the entire program to stored in FIRST. RTL only allows a single function of a program to be stored in it, because GCC generated code on a function-by-function basis. Most of the transformations and analysis are done in SIMPLE and the program is translated into the SIMPLE intermediate representation as soon as possible.

SIMPLE is designed to minimize the complexity of the program representation. All assignment operations in C [KR78] are converted into simple assignment operations. Also complex expressions are broken down into either unary or binary operations that produce a single result. Unstructured `gotos` are not allow SIMPLE intermediate representation because they interfere with the analysis. This also requires constructs such as the `switch-case` statement in C, which can produce unstructed code because of the fall through nature of the alternatives, to be modified into structured code.

Once the analysis and transformations in the SIMPLE, the program is converted to the LAST intermediate representation, which allows additional information for register allocation and scheduling to be included in the intermediate representation for code generation. Pointers back to the SIMPLE represention are included in the LAST representation of the program, so that the aliasing information is available for the scheduling and code generation. The LAST intermediate representation also makes the address calculations explicit, the notion of variables is replaced with the concept that values are either stored in memory or registers. With many parallel machines this may present a problem, since there is often more than one flavor of memory, e.g. local and shared memory.

Unless things have changed significantly on the McCat compiler it provides a test bed for compiler construction, but doesn't currently have the framework in place for generating parallel code. They do not have a library of constructs for expressing parallelism or synchronization in the intermediate represention.

3.1.3 VCODE

In an attempt to produce a portable intermediate representation, VCODE [BC90] was developed. This intermediate form is modeled after Pascal PCODE [NAJ+81]. The execution model of the code is SIMD in nature, but it is rather independent to the target machine model. This intermediate representation uses a relatively small number of construct in the language and uses vectors as the basic data type to express parallelism.

3.1.3.1 Execution Model of VCODE

The execution model in VCODE is SIMD in nature, because there is only one flow of control running through the program. All decisions for branches are made on scalar values. The model also assumes implicit synchronization on every instruction in the code. However, on implementations for MIMD machines synchronization would only be required for operations that communicate between processors.

Parallelism is obtained by using vectors in the intermediate representation. The elements in the vector can be operated on concurrently. The vectors are single dimension arrays of arbitrary length. The vectors in VCODE can be dynamically sized at run time for the number of elements required. Since the vectors can be an arbitrary size, they may be larger or smaller than the size of vectors normally processed by the target machine.

Vectors required for computations must first be placed on a stack, and each vector takes up exactly one slot in the stack regardless of the number of elements the vector contains. All computations remove the arguments from the top of the stack and replace them with the results of the computation.

Computations on vectors fall into several classes. The simplest class is elementwise operations, where each element of the vector is operated on independent of any other element in the vector. Permutation operations in the language change the order of elements in the vector. Scan operations operations do parallel-prefix operations on the vectors.

Operations that only affect certain elements in vectors can be accomplished the vector merges and scatter/gather operations. Vector merges take two vectors of the same size and produces a new vector of the same size by using a selector array of the same size to indicate which vector each element is from. The second method of excluding elements from operations is to do a gather operation to select elements to operate on. This produces a dense vector that all elements in are operated on. Once the computations are completed a scatter operation is done to return the elements to the appropriate locations in the vectors.

The only looping construct allowed in the intermediate representation is tail recursion. It is expected that the later stages of the compiler would optimize out the calls and replace the tail recursion with a while loop.

3.1.3.2 Disadvantages with VCODE

VCODE is an extremely clean and concise intermediate representation. However, it is still several levels above the actual target machines' true representation. There are a number of operations in VCODE that are not particularly efficient on many parallel machines.

One problem is the stack based model of computation. It is very hard to express the register-to-register operations of a target machine with this notation. Thus, common subexpression elimination and caching of values in registers cannot be accomplished in this intermediate representation. These types of optimizations can give significant speed improvements on most machines, since accessing registers is significantly faster than accessing memory.

Another feature most machines lack is arbitrary parallelism. With the exception of the CM-2 paris model [Thi89b], vector up to a certain size can be computed in parallel on the physical machine, but larger vectors must be strip mined into section that fit the parallelism of the target machine. Reducing the vectors to the natural width of the machine may produce other benefits, such as reducing the amount of memory required for temporary values and allowing high speed registers in the machine to be used to store temporaries. Later programming models on the CM-2 [Sab92] [BHMS91] had much higher performance due to storing values in machine registers and to strip-mined loops for the appropriate parallelism width.

The method of including and excluding certain elements from calculations may be inefficient on some target machines. Either the calculation can be done and then unwanted calculations are nullified by a merge or scatter/gather operations can be done so only desired elements are operated on. Both methods have their disadvantages. By using vector merges, which combine elements from two arrays into one array, the communication costs of the gather/scatter operations can be avoided, but to nullify a calculation with a merge can be particularly expensive for a save, since an extra load must be done. Scatter/gather operations avoid the unneeded calculations, but the scatter/gather operations to produce a dense vector of elements to operate on implies communication operations to move the elements between processing elements. Worse yet, is that a gather operation is required for each operand fetch and a scatter, load, and merge are required for each store.

Finally, recursion is not a typical construct in most parallel machine languages. This construction will need to be converted into the appropriate test and branches for the target machine. However, it is impossible to represent the result of the transformation in the intermediate representation. Therefore, the transformation will have to be done outside VCODE.

3.1.4 GR

GR [Sar91] is the intermediate representation used in the SISAL [Can92] compiler, IF1 [SG85] produced by Lawrence Livermore National Laboratories. SISAL is a single assignment functional language. Hence, GR is also a single assignment language. Much of the work done by Sarkar [Sar89] focused on partitioning and scheduling the program represented in GR to multiprocessors machines. The following sections will discuss the execution model GR and limitations of this intermediate representation.

3.1.4.1 GR Execution Model

As stated previously, GR is the intermediate representation used in the Lawrence Livermore National Laboratory SISAL compiler. Since it is desirable to have a relative simple translation from the input language to the intermediate representation, GR is a single assignment functional language like SISAL. However, GR is a graph based language.

There are two basic objects in the GR intermediate representation. The nodes in the graph show places that possible computations occur (it is possible for a node to contain no computations at all). The edges in the graph represent transfer of data from one computation to another. An edge that transfers no data between nodes is an edge that preserves synchronization of the nodes.

Nodes cannot execute until all the input edges have supplied tokens (data) to them. The one exception to this are nodes that have *non-strict* transfer functions. These nodes many not consume tokens from all input edges or produce tokens for all output edges. Nodes that perform selection for case statements or conditions are examples of *non-strict* nodes.

There are several additional types of nodes to make the intermediate representations complete. The basic nodes that represent a series of sequential operations is a *simple* node. A *function call* node represents a call to a function. A *compound* node represents a subgraph. The only parallelism expressed in GR is the *parallel* node, which is equivalent to the `FORALL` expression in SISAL, or the DOALL loop in parallel versions of FORTRAN. A parallel node indicates iterations that can be concurrently executed.

Functions in GR have many of the characteristics of *compound* nodes. All the tokens for the input to the function must be present before the function is started. The function then is started and all computations in the function are completed before any of the output are return. To enforce these conditions, each function has a top node which all tokens must funnel into and a bottom node that all generated token must pass through.

The intermediate form carries informations about the expect number of times a node is expected to execute, the expected amount of time a node will take to execute, and the amount of data an edge carries. This information is used to partition the intermediate representation graphs into tasks and to schedule these tasks. Separate tasks can be scheduled to run in parallel.

3.1.4.2 Disadvantages with GR

The basic model of execution requires that all inputs to a node are present before the node executes. A node may actually be a compound node that does a large amount of calculations. However, several of the inputs may not be required until considerably later in the calculation, and much of the calculation could be started before these inputs are required and could save much of the time of processing.

One possible solution of waiting for all the input before starting the calculation is to split the calculations into smaller tasks, so that the calculations are only waiting for the inputs that it requires. However, this results in high scheduling overhead, because there are more tasks. Costs for communication can also be higher, because more communication edges will cross between tasks and possibly processors.

The one form of parallelism expressed in the GR intermediate representation is the parallel node, which allows concurrent execution of iterations of loops. This allows a very compact form of indicating parallelism, but this is not the only form of parallelism available on machines. However, on some machines, such as PASM [SSKD87], significant performance increases can be achieved by overlapping the control unit's operation with the processing element's operations. With the semantics of GR this overlap would be impossible to obtain.

The GR intermediate form has no method of controlling the layout of which processors get particular tasks to do, or which processors obtain data. All processors are assumed to be equivalent, and the things that are controllable are number of tasks and whether a communication arc crosses between tasks. The CM-5 [Thi92] network groups of 4 processors have a high communication bandwidth between themselves and other groups of processors. Another reduction of communication occurs between groups of 16 processors. It is impossible to group communicating tasks in GR and improve communication bandwidth between the tasks. Edges either stay within a processor or cross between processors. There is no distinction made in GR between edges to processors that are close and ones that are far.

3.2 Our Method

This section of the dissertation will describe the intermediate representation designed with flexibility to describe both high-level constructs in the input language and individual machine language instructions effectively. The front end of the complier will not initially generate the individual machine language instructions, but rather machine independent intermediate representation. This intermediate representation will be modified by transformations to fit the intermediate representation constructs available in the machine language. This scheme has much in common with the approach taken with the McCat compiler.

While the previous sections have described the work by other researchers in the area of designing and building intermediate representations for compilers targeting parallel machine, this section will cover the design of the intermediate representation used for this dissertation.

3.2.1 Typing System

As essential component of a compiler for high-performance computer languages is the typing system. Much of the performance for supercomputers comes from the ability to perform calculations on aggregate sets of data, e.g. vector instructions. Therefore, high performance compilers must be able to handle such data types in the intermediate representation to allow the analysis of the program to be performed on the appropriate level, to make it possible to generate instructions for the target machine, and to avoid tying the compiler to a particular machine because of restrictions in the typing system.

There are three aspects of the typing system that will be covered in the following subsections. Section 3.2.1.1 will discuss how quatities are represented in the intermediate representation. The next section will describe how aggregate data structures such as arrays are represented in the intermediate representation. Finally, in section 3.2.1.3 the rules for data conversion and movement in the intermediate representation

3.2.1.1 Data Formats

One of the most basic attributes of a data is the format the information is stored in. All computers must encode the data into some format. In a simple case the data may simply be a two's complement representation of a number. A more complex case would be to represent a floating point number as a two's complement number with a scaling factor. There are other data formats, such as encoding character sets. However, for this work only integer and floating point number formats are supported, `TypeInt` and `TypeFloat` respectively.

This simplification of the possible data formats was done to insure that machine instructions using different data formats can be handled by the system, but avoid spending a large amount of time with many different data, since the thrust of the work is on generating code for parallel systems.

The restriction that both types are the same size again is relatively minor. This may make the compiler have some artificial alignment constraints. Because aggregate types, such as arrays, are supported in the intermediate representation, the system does support different sized objects for the aggregate types.

Conversions from one data format to another in the intermediate representation must be explicit. The results of not including a data format conversion in places where it is required is undefined.

Element operators in the intermediate representation such as element-wise addition and subtraction are assumed to produce the same data format for a result as the data format of the operands. Thus, all the operands must have the same data format.

3.2.1.2 Data Geometry

Virtually all languages have some form of aggregate data type, the most commonly known are array, and many languages also have support for non-homogenous data types like C's structures. However, in this intermediate representation only homogenous data types, such as arrays, are supported. There are two aspects of these data types that will be discussed in this section of the dissertation: dimensions and layout.

Many high perfomance machines improve performance by using operations that perform computations on aggregate data types. One of the best known examples of this type of instruction are vector instructions as on the Cray-1 [Rus78] [BK77] and the CDC Cyber 205 [Lin82]. Thus, rather than performing an operation on scalar operands and returning a scalar result, these vector instructions take one-dimensional arrays, vectors and return a result of the same type.

This presents the first concern for the geometry. What shape are the aggregate data types. Almost all machines have constraints on the geometry of the aggregate data types. For the Cray-1, the vector registers can contain 64 elements and require vector operands to be grouped into one dimensional arrays with 64 elements. Larger array operations must be converted to using operands of this size. Some parallel machines have the processing elements arranged in either a two dimensional or a three dimensional array. These machine configurations could be described as an array of arrays and an array of arrays of arrays, respectively.

This brings up the second concern. How are the arrays distributed across the machine? For the initial intermediate representation of the program the layout of the program on the target machine is unspecified due to the machine independent nature of the front end that generates the intermediate representation. Arrays with the same dimensions are of the same type in the initial intermediate representation. However, after the layout of the array is performed, which gives the array a location on the machine, data types with the same dimensions may not be the same type because of the differences in location.

In the initial intermediate representation the arrays are whatever size specified in the input program. However, the target machine may not be able to directly operate on arrays of all dimensions. The arrays that the target machine can directly operate on are *natural* data types. Data types that may not be directly available on the target machine are *synthesized* data types. The dimensions of natural data types are based on machine constants, such as the number of processing elements in the machine or the number of elements in a vector. There is also the concept of location of the data in the machine. Thus, there would be separate natural data types for variables stored in registers and variables stored in memory. The method used to obtain the information about the natural types is discussed further in chapter 5.

There may be no difference in the dimensions between the sythesized and natural data types. Even if there are differences, transformations such as strip mining (section 5) can be used to eliminate these differences. The rules for such transformations will be covered in more detail in section 3.2.1.3.

The intermediate representation does not assume a layout pattern other than one-to-one for a particular dimension of a data type. Cyclic and block distribution become the same for this case. An additional dimension can be added if more storage locations are required. Transformations can modify how elements in one array are mapped to another. However, this mapping must be explicitly given in the intermediate representation through the equations that calculate the address. This choice was made beacuse of the machine level instructions that could be represented in the intermediate representation. The machine instructions do not describe the distribution of the data. Thus, this intermediate representation does not either.

3.2.1.3 Data Typing Rules

Because the intermediate representation supports aggregate objects as first class objects, there are additional considerations in the design of intermediate representation to fully utilize these agregate objects. There are a number of rules for representing aggregate data types and performing transformations on them in the intermediate representation. This section of the dissertation will describe the rules governing these aspects of the data typing.

Changes in the data typing is explicitly expressed in the intermediate representation. Thus, if an addition is performed and one of the operands is of the integer data format and the other is of the float data format, the integer data format operand must be explicitly converted into the float data format. This follows the C convention that if one of the operands is a float, then the operation is performed as a float.

Similar to the explicit conversion of data formats in the intermediate representation changes in the data type must be explicit in the intermediate representation. Thus, data type conversions for converting a scalar value to an an array for a broadcasts must be in the intermediate representation.

Many of the data movement operations are modeled as specialized type conversions in the intermediate representation. A broadcast operation is a type conversion from a scalar data type to an array data type. Loads and stores that are of the same dimensions can be modeled as conversions from one natural data type to another. In some sense this is rather natural, since type conversions are not useful in themselves, they only allow other operations to be generated.

To maintain the machine independence of the front end of the compiler system, the initial intermediate representation generated by the front end is abstract and does not attempt to map the program to the natural data types available on the target machine. Thus, the data types used in the intermediate representation may not exist on the target machine. Later transformations in the compiler will have to modify the intermediate representation to use only the natural data types available on the target machine.

When a transformation changes the data type used to store information in the intermediate representation, the transformation must ensure that a sufficient number of locations are available in new data type. Thus, it may be necessary to pad arrays and have some locations that are unfilled. An example of this would be a a vector that has only 20 elements. When the vector is mapped to a vector machine that manipulates 32 element vectors, the vector will be padded out so that it has 32 elements.

3.2.2 Execution Model

For the target machines there is a wide range of execution models. This section of the dissertation will discuss how various execution models for different target machines are encoded in the intermediate representation. In particular, there are two aspects that will be covered: expression of parallelism and expression of synchronization in the intermediate representation. Chapter 5 will discuss how the execution model for the target machine is selected, and this section will be restricted to how a particular parallel construct would be represented in the intermediate representation.

3.2.3 Transformations to Fit Constraints

During the translation process from the input language to the output language transformations may be required to ensure that the intermediate representation meets the constraints imposed by the target machine. In particular, the data structures in the input language must be converted into the data structures that the machine can manipulate and the intermediate representation should only use constructs available in the output language.

3.2.4 Code Generation

The goal of the code generation stage of the compiler is to have a relatively straightforward conversion from the intermediate representation into the target machine's assembly language. There may be many possible translations of the intermediate representation to the assembly language, but there should be a simple almost word-for-word conversion process.

There are two observations about this conversion process. First, it appears that this straightforward conversion violates the basic premise, that this direct translation will generate inefficient code. Second, no translation between the intermediate representation and the output language may exist, since some constructs in the intermediate representation do not have direct translations in the assembly language. Neither of these observations are problems, since the compiler has a list of constraints that the intermediate representation must be met before code generation can be implemented. Thus, the intermediate representation is very closely matched to the output language of the compiler at the time of code generation because the transformations in the compiler enforces these constraints.

3.2.5 Unresolved Issues

There are several issues that need further investigation in the intermediate representation.

3.2.5.1 Different Sized Arrays

The layout form has a number of good characteristics, it reduces local communications for linear nearest neighbor, and fits well with giving a section of the problem to every processing element. However, the distribution for different sized arrays on the machine will have locations map to different processors.

Elements in arrays will not line up for different sized arrays. Therefore, they may be cases where some conforming of the array is required before operations occur. Equivalent to the scatter gather problems in VCODE.

This may be solved by allowing nested parallelism, vectors of vectors. Thus scatter and gathers can be done on the appropriate level of the model to improve performance of the code while avoiding communication operations.

3.2.5.2 Families of Machines

With many programmers wanting to minimize the number of different versions of object code it is desirable to have the same code run across a family of machines. The machines basically have are the same design, but differ in a basic parameter, such as the number of processing elements the machine has.

Changes in parameters such as the number of processors significantly change the distribution of arrays on the machine. It is possible to have the code generated be parameterized for such run-time constants as the number of processors, but this significantly increases the complexity intermediate representation for a program. If the size of the machine is known at compile time, many of the parameters can be folded turned into constants. If there are several possible sizes of machine, then the calculations must be deferred until run time and optimizations such as dead code elimination and loop unraveling cannot be done on the intermediate representation.

An generating code for a family of machines make the memory allocation more complex, since The amount of space required for arrays will depend on the number of processing elements in the system.

One solution to this problem is to have a modified loader that uses information about the machine that the code is being loaded into to allocate the required space for a given machine and to patch the code for the appropriate sized machine. The file generated by assembler could also track the information about the amount of space needed in segment for the program and use this information to calculate the amount of space required by a program.

3.3 Conclusion

The intermediate representation is clearly an essential component of the compiler because it provides a bridge that connects the input language to the output language of the compiler. The intermediate representation semantics also specify the range of execution models that can be efficiently represented by it.

With the examination of available intermediate representations, there are several observations that have been made in this chapter: "the intermediate representation" of the compiler may actually be several intermediate representations, the intermediate representation needs to be parameterized to tailor the intermediate representation to the target machine, and geometry and layout issues are a concern.

Several intermediate representations that may be required within a compiler. Some may have very similar semantics, but exclude certain features. Other may have larger differences between them. However, when the actual code is generated for the target machine, the intermediate representation used should be a closer match to the target machine than the other intermediate representations used in the compiler. The intermediate representation used for final code generation may vary from target machine to target machine.

Differences in target machine implie that the intermediate repesentation be parameterized. These parameters would be influenced by the machine characteristics, e.g. what constructs does the target machine directly support, the types of parallelism available in the target machine, and whether the same instruction could be used to access both local and global memory.

With the wide variations in machine architecture the data layout has become a much greater concern. Few machine have a uniform memory access (UMA) model, where all memory access cost the same and data layout has no effect on the resulting cost of the assesses. Thus, models used in serial intermediate representations like RTL and McCat that do not allow any control over the layout of data and are clearly inadequate for parallel machine. However, even intermediate representations such as VCODE and GR have problems with the methods the methods they use for data layout.

This chapter has reviewed other intermediate representations and identifies some areas that require futher consideration for parallel intermediate representations.

4. MACHINE DESCRIPTIONS

The machine description language has one purpose: to provide enough detail about the target machine so that a good compiler can be generated. Although a variety of hardware description languages exist, they are not a good match to this task. Languages such as VHDL [Arm89] [Int89] and Verilog [Arn90] [Tho90] provide a model of the the machine that is too detailed. For our purpose, many of the details of circuitry and physical implementation of the machine can be ignored, while other features of the machine are simplified. Thus, this chapter focuses on the theoretical aspects that should be considered when designing the machine description language; the particular language used in the prototype tool is presented in chapter 7.

The design of a machine description language can be broken down into four topics. Section 4.1 explains the methods used in the machine description language to express the available parallelism of a target machine. The fundamental hardware resources available in the machine description language are defined in section 4.2 and the methods used to group hardware resources together are given in section 4.3. Section 4.4 covers the mapping of machine instructions to the available hardware resources.

4.1 Parallelism

The machine description should make the parallelism available within the target machine visible to the compiler tools. By making the parallelism visible to the compiler tools, the compiler tools can make decisions on how to map the parallelism available in the program to the target machine.

There are many types of parallelism available in a target machine, and the methods used to determine the constraints on the available parallelism of the target are discussed further in chapter 5. This section restricts itself to giving the basic means by which parallelism can be expressed in the machine description language.

On the most basic level, parallelism implies that there is concurrency, and that more than one operation is occurring simultaneously. There are two dimensions to the parallelism available. Multiple function units can be calculating values in parallel, and each function unit may have parallelism available within it (i.e., pipelining).

An approximate upper bound on the maximum parallelism available on the target machine can be formulated as follows. Let f, p, m be the number of a particular type of function unit in the machine, the parallelism available within the function unit, and the number of different types of function units in the machine, respectively. Thus, the total parallelism of the machine, P, can be estimated as shown in equation 4.1.

$$P = \sum_{i=1}^{m}(f_i p_i). \tag{4.1}$$

This upper bound can be overly generous for several reasons. First, the instructions may not be able to use apparently unrelated function units independently. An example of such a machine is the CDC STAR-100 [HT72], which pairs function units together to compute double precision floating point numbers. Second, due to resource usage conflicts between instructions, it may not be possible to have all function units perform useful calculations at all times. Some function units may remain idle, reducing the effective parallelism of the machine. A more accurate method for estimating hardware parallelism will be discussed in chapter 5.

An estimate of the maximum throughput of the machine also can be made. However, to compute this estimate the timing of the components of the machine must be known. There are several relative time bases that can be used to describe the temporal operation of the function units. The most basic is the measured number of nanoseconds required for each function unit to accomplish its task. For a synchronously-clocked machine that has a single clock from which all other clocks are derived, it might be simplest to use ticks of that clock as the basic unit of time. It does not matter which method is used, so long as it is used consistently throughout the description.

Equation 4.2 describe the maximum throughput available on the machine. For each type of function unit there is a minimum enqueue time e required before the next calculation can start on that function unit. This equation assumes no startup overhead and assumes the optimistic parallelism given in equation 4.1.

$$w = \sum_{i=1}^{m} (\frac{f_i p_i}{e_i}) \qquad [4.2]$$

4.2 Fundamental Hardware Resources

Much of the information about the hardware implementation is unimportant to the compiler. For example, the design of compilers generally is not affected by the implementation circuitry, packaging of the circuitry, gates used, power dissipation, and many other features of the target machine clearly have no direct impact on the compiler; thus, these aspects need not be described.

The hardware modeling described in this thesis requires timing information about the hardware resource. Often the timing of hardware resources is deterministic and can be counted in machine clock cycles. However, some computer hardware has been designed to operate asynchronously and is not clocked, e.g. some HP PA-RISC processors [GFK+91] and experimental ST9 implementation [CDF+93]. Despite the asynchronous nature of the hardware, estimations of the time required for each hardware resource to perform its task can be made and this estimate is sufficient for the modeling of the machine.

The second concern is how to express the relationships between hardware resources. Some function units act together as a single hardware resource, while other act independently. Obviously, hardware gates that are always used together, such as the ones that compose an adder or a register, can be treated as a single resource. A more complex example of this grouping is a pipeline in which stages are always used in a fixed sequence can be modeled as a single entity with timing characteristics identical to the pipeline's aggregate behavior. Hardware components that may be used separately should be separate resources.

To identify the resources in the machine, each resource must be given a name. These names will be used both as a tag to assemble collections of resources together into larger resources (as described in section 4.3) and to link each instruction to the machine resources it uses (as described in section 4.4).

There are at least three different kinds of resources needed to describe a typical parallel computer: pipeline, register, and memory. Register and memory resources can hold values until those values are explicitly overwritten. The distinction between register and memory resources is that at compile time the register accessed are always known. Thus, the compiler can always determine whether two register accesses use the same register or different registers. For memory accesses it may not be possible to determine which accesses to memory are to the same location due to the run-time calculation of the memory address. In contrast to the register and memory resources, a pipeline can also hold data, but it holds data only for a specified time interval. The following three sections will describe these basic hardware resources.

4.2.1 Pipelines

A *pipeline* resource is a basic hardware resource which takes input and generates output a specified time later. As data "flows" through the pipeline the calculations are performed at each stage in the pipeline. While one set of values are being calculated, other sets may enter the pipeline before previous sets have exited the pipeline.

The number of inputs and outputs to the pipeline are not important, but time required for a calculation to be completed after the inputs are presented, which is latency, and the time between consecutive set of inputs can be presented to the pipeline, which is enqueue time, are important. The relationship between these two timing characteristics of the pipeline indicates what the resource is capable of performing.

The traditional pipeline as described in [HB84] assumes that the latency is longer than the enqueue time. Thus, there are several sets of calculations being performed simultaneously by the pipeline. An example of such a function unit is the Cray-1 floating point add pipeline [BK77].

If the latency and enqueuing times are equal, then the resource can be calculating only one value at any time. This serial resource is not a traditional pipeline. However, the pipeline model in this thesis includes it because it is a natural result of the definition of pipeline used in this thesis.

Finally, there are situations where the enqueue time is greater than the latency of the unit. An example of this might be a function unit that is designed with dynamic circuitry. The latency may be relatively low and the result produced quickly. However, a precharge or set up time may be required before another operation can be performed by the unit. An example of this effect can be seen in the access time for dynamic rams being less than the time required between accesses [Mos80].

The generalized model of pipelines in this thesis allows other components in the computer, such as busses, to be accurately represented as pipeline resources. The timing information provided about these components represented as pipeline resources will be used by the compiler's code scheduler to reorder instructions, hopefully reducing the execution time of the resulting code.

4.2.2 Registers

The *register* resources in a machine description differ from the pipeline resources in that they can hold values for extended or indefinite amounts of time, while the *pipeline* resources always flush the values from the pipeline within a specified amount of time. Thus, the management of *register* resources is somewhat different than *pipeline* resources.

Most computer architectures have groups of equivalent registers. Each register in the group of equivalent registers can be accessed like any other register in that group, with the same machine instructions. The only time that they are distinguished

```
register DATA_REG 1 1 "d" "D%d" 8
register ADDR_REG 1 1 "a" "A%d" 8
register PC 1 1 "p" "PC"
register CCR 1 1 "f" "CCR"
```

Figure 4.1 Compiler visible register of the Motorola 68020

between each other is when they hold a particular value. Any register in the equivalent group that does not have a live value in it is suitable for holding a new value.

Because there could be several register resources that could be used to hold a value, additional information about the register resources is required. In particular, register resources that are equivalent are indicated with the same register class. Second, the compiler will need to have the register names (e.g. R0, R1, etc.), so instructions with the appropriate register names can be generated.

Figure 4.1 illustrates how this information is encoded in the prototype PIGG tool. The first field on the line indicates the that the resources described are register resources. The second field on the line indicates the resource name that will be used when grouping the resources into a complete processor. The third and fourth fields are the number of cycles for latency and enqueuing an access to the register, respectively. The fifth field is the register class name used by the compiler when generating instructions, and indicates what other register resources the resources are equivalent to. The fields following the register class field describe the register names. These name are printed by the compiler when generating the instructions.

The register resource `DATA_REG` describes the general purpose registers `D0` through `D7` as register class "d". Similarly, `ADDR_REG` describes the address registers `A0` through `A7`. The program counter is given its own class of "p" and is named `PC`. Finally, the condition code register is specified as the `CCR` register resource on the last line. All of the registers can be accessed in a single cycle on the Motorola 68000. Thus, both the enqueue times and latencies are 1.

On some machines registers are paired together to provide registers for larger data types. An example of this is on the Gould NP-1 [Gou87], which there are 8 32-bit general purpose registers. When working with 64-bit numbers, an even numbered general purpose register holds the most significant 32-bit word and the odd numbered register following it holds the least significant 32-bit word of the 64-bit quantity. This pairing of registers and overlapping another register class is not handled in the model presented in this thesis. However, with the current trend in computers, this should not be a problem. Overlapping register classes were developed to conserve register memory bits. Current machines commonly have registers that the largest data type will fit into. Smaller data are padded to fill the entire register. Thus, overlap between register types does not exist on newer architectures, such as the DEC Alpha [Dig92].

4.2.3 Memories

The *memory* resource is the third type of resource available for describing machines. Like *register* resources, *memory* resources can have values that are held for indefinite amounts of time. However, *memory* resources have an additional quality that does not exist in the *register* resources; the name or address of the memory cell can be generated at run-time, unlike the register names, that must be known at compile-time.

The example memory description given in figure 4.2 give the information about the memory. First, the description indicates that this is a memory resource with the `memory` keyword followed by the name of the resource, `RAM`. Like the pipeline and register resources, memory resources also have an latency and enqueue time. For the Gould NP-1 it takes 8 cycles to access the memory, and the memory accesses are not pipelined. Thus, the enqueue time is the same value as the latency. Finally, the size of the memory is listed in the smallest addressable unit. For this example the machine has 64 million words of memory. This was a simplification in the prototype description, because the NP-1's memory is byte addressable rather than word addressable.

```
memory RAM 8 8 67108864
```

Figure 4.2 Gould NP-1 memory

4.3 Hierarchy

An important property of the machine description language is the ability to describe the machine in a hierarchical fashion. This allows one to decompose the machine into smaller, more manageable components. It also simplifies the description of machines made of multiple smaller machines, such as a workstation cluster or MIMD supercomputer using commodity microprocessors.

This hierarchical approach is commonly used in VLSI design so that a component can be specified once and used repeatedly throughout the machine description, factoring out much of the redundant information. Consider the MasPar MP-1 [Bla90], a SIMD machine with up to 16,384 processing elements. The basic processing element can be specified once in the machine description and then this component can be used within the processing element clusters that each contain 16 processing elements. In turn, the machine has 1,024 cluster in the machine to give the machine a total of 16,384 processing elements.

This points out a second advantage in using a hierarchical description language. The designer of a parallel machine naturally groups components of the machine together. These groupings can be mirrored in the machine description. Typically, these groupings share resources. An example of this is the processing element clusters in the MasPar MP-1 [Bla90]. The 16 processing elements in the cluster share both a byte-wide memory interface and a network interface. Another example of the grouping of machine resources is the processing node in the T3D [Cra93]. Each processing node contains two DEC Alpha processors, which share an interface to the network. By including this grouping information in the machine description, estimates of the cost of sending information from one section of the machine to another can be made more accurate. This information is extremely important for deciding what granularity of parallel program the machine can efficiently execute at each level in the machine hierarchy.

Finally, the hierarchical nature of the machine description allows machine descriptions for new machines to be built from earlier machine descriptions. Most of the parallel machines currently available are built using conventional microprocessors, e.g. Cray T3D [Cra93], Thinking Machine CM-5 [Thi92], Convex SPP1 [Con93], IBM SP1 [SJ93], and the Intel Touchstone Delta and iPSC/860 [van91]. With the hierarchical method of building the machine description, a machine description for the parallel machine can be built upon the description for the processor used for a single processor version of the compiler, and would reduce the amount of work to build a compiler for the complete machine. Similarly, a compiler for a cluster of workstations could also be built from a single-processor version of the machine description.

To facilitate this hierarchical approach for the machine description, several concepts have been developed. Section 4.3.1 describes the issues with grouping resources together and section 4.3.2 describes the concept of reserving resources for special uses.

4.3.1 Grouping of Resources

To accomplish this hierarchical structure in the machine description, resources can be grouped together into larger groups. There are two issues that must be resolved for grouping machine resources: how are the groupings formed and which group forms the entire machine.

Composite resources are the method used to implement the hierarchy grouping. Each composite resource is composed of other composite and fundamental resources. A composite resource can be used elsewhere in the machine description to represent all the components it contains.

One reason for creating a composite resource is that the resources it contains will be replicated throughout the machine. A simple MIMD architecture with 8 processing element is shown in figure 4.3, and the machine description for this architecture is given in figure 4.4.

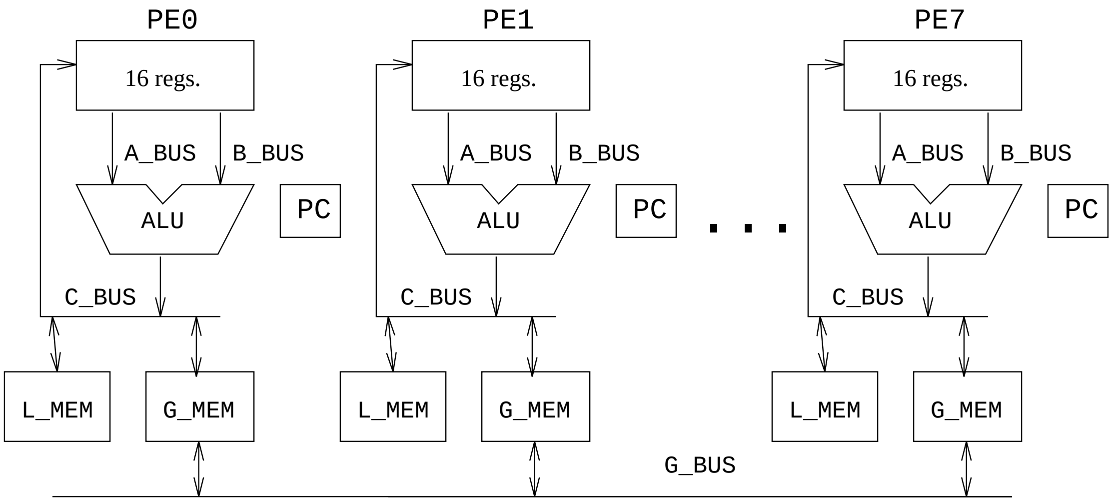

Figure 4.3 Example MIMD architecture

At the highest level there must be a single resource that describes the entire machine and this resource must be indicated. This resource is treated in the same way a parser generator treats the start symbol of a grammar.

```
pipeline ALU 1 1
pipeline A_BUS 1 1
pipeline B_BUS 1 1
pipeline C_BUS 1 1
register REGS 1 1 "r" "r%d" 16
register PC  1 1 "p" "pc"
memory L_MEM 2 2   65536
memory G_MEM 10 10 65536
resource PE (ALU REGS PC L_MEM G_MEM A_BUS B_BUS C_BUS)
pipeline G_BUS 1 1
machine mimd (8(PE) G_BUS)
```

Figure 4.4 Hardware description of the example MIMD architecture

4.3.2 Reservation of Resources

When a machine description is made it is desirable to describe the entire machine, even the parts of the machine that are reserved or "off limits" to the compiler. The approach of describing the complete machine and then marking some of the resources as reserved provides a number of advantages. The advantages are explained further in this section.

A *reserved resource* is a resource in the machine that has a dedicated purpose. The code generated by the compiler may still use this resource. However, the resource will not be free to be allocated to some other use. Thus, the resource will only be used if it is explicitly referenced in an instruction template in the machine description.

If the entire machine is described, the machine description can take advantage of the regular nature of most machine designs. Most machines have a large number of replicated components. If there are no exclusions or holes in the listed resources, the components can be numbered consecutively. Thus, it is easier to specify all components in the machine and later remove components from the specification.

With the advent of microprocessors, many machines use the same core for the processing element, e.g. a number of parallel machines have used the Intel i860 as a processing element, such as the Intel Paragon [Int92], the Sky SKYbolt Shamrock [Sky93], and the IBM Vulcan prototype [SSA+94]. Although these machines differ enough to require vastly different compilers, the description of the i860 can be used for each. For example, the registers that are reserved for the operating system and runtime support can be specified separately. In addition, if the registers reserved by the operating system change, the i860 core description does not require modification.

Even if a resource is reserved, there may be code generated by the compiler that uses it. Thus, the timing information available from the machine description is still required, making it impossible to just exclude that resource from the machine description. An example of this would be the stack pointer in the Motorola 68020 register

```
reserve ADDR_REG["A7"]
```

Figure 4.5 Stack pointer reservation for the Motorola 68020

description given in figure 4.1. The operation to reserve the stack pointer, address register A7, for the prototype PIGG is given in figure 4.5.

It is important to point out that registers are not the only resources that may be reserved for special use. Parallel machines may have certain processors assigned to particular tasks in the machine and hence not available for execution of user code. In general, any resource may be excluded from compiler management.

4.4 Relating Resource use to Instructions

The machine description must link the three aspects of the machine description together. The three sections are the resources available on the machine, the intermediate representation each instruction covers, and the information to output when the instruction template is selected. By linking the resource usage to the instruction templates, the relative cost of each instruction template can be determined. In addition, the register usage information can also be stated concisely in the resource usage. Previous work on associating the hardware use for instruction templates is discussed in section 4.4.1. Then, the method taken in this thesis to group the intermediate representation, resource usage, and the translated output is explained in section 4.4.2.

4.4.1 Previous Work

Other machine descriptions, such as GCC version 2 [Sta93] and Marion [BHE91], can include information about the resource usage of the machine instructions. Both of these systems provide information for building the instruction schedulers. The approaches differ in the respect that GCC takes a higher level view that resources are complete pipelines, while Marion takes a cycle-by-cycle view of how the instructions use the hardware. Sections 4.4.1.1 and 4.4.1.2 describe in more detail the method that resource usage is specified in GCC version 2 and Marion respectively.

4.4.1.1 GCC Version 2

The resource usage information made available in GCC version 2 [Sta93] is a relatively recent addition to GCC; GCC version 1 [Sta89] did not include any of this information in the machine description. This capability was added to the machine description language to allow the GCC compiler to schedule instructions and improve the performance of the code for pipelined processors.

The resources in GCC version 2 are described as pipelined function units that have latency and enqueue time included in the description. Multiple copies of the function unit can exist in the machine description. The latency, spacing between entries to the pipeline, and the maximum number of entries in the pipeline can all be specified in the description.

An instruction template indicates the use of the function unit by including the attribute name of the function unit. Thus, instruction templates that have the same attributes use the same type of function unit. Specifying a cycle-by-cycle use of the function units is not possible in GCC version 2. Also, specifying a particular resource out of a group of resources of the same type is not possible; it is assumed that the hardware will allocate the resource out of the pool of available resources of that type.

4.4.1.2 Marion

The Marion system [BHE91] provides a set of tools to build code generators and schedulers from machine descriptions of RISC processors. Target processors include the Motorola 88000, the MIPS R2000, and the Intel i860. However, special scheduling algorithms had to developed for the i860's explicitly advanced floating point pipelines.

In the resource description, with the exception of registers and memory, each resource is unique and only one copy of it exists. The machine resources in Marion each take a single cycle to accomplish their tasks and cannot be grouped together, causing the resource description to be a single level. The description of large parallel machines is difficult using this notation because resources of the parallel machine would be replicated, which cannot be described in Marion.

Each instruction resource usage is an explicit cycle-by-cycle description of the resources used. If the instruction uses a resource for more than one cycle, the resource will be listed in the usage description multiple times. The instruction resource usage in Marion does not indicate the use or definition of values. The level of detail required for coding of the instruction usage of the resources is much greater. This information cycle-by-cycle is not easily available, making it more difficult to code the resource usage.

4.4.2 Our Approach

This section concentrates on how the resource usage of the machine instructions are encoded, opposed to chapter 5 which describes how information required for building the compiler is extracted from the machine description. To calculate the cost and characteristics of the instructions, each instruction template in the machine description indicates the resources required to execute that instruction on the target machine.

There are several advantages to the system described in this chapter over other code generator methods of describing target machines. First, data dependencies between function units can be accurately described in the instruction template resource usage, discussed in section 4.4.2.1. Section 4.4.2.2 explains how hardware resources in the resource usage can be uniquely identified, while section 4.4.2.3 shows how this information is linked to the intermediate representation and the output for the instruction templates.

4.4.2.1 Resource Dependencies

Unlike GCC version 2 and Marion, the data-flow dependencies from one resource to another need to be marked. GCC makes the assumption that if multiple resources are used by an instruction that there are no serializing dependencies between the units. Thus, for GCC, the function units are monolithic entities that cannot be decomposed into a chain of smaller units. Marion's uniform single cycle units require multiple stage pipes to be broken down into smaller units. The dependencies are implicitly encoded by starting a dependent stage in the time unit following the completion of the stage that it was dependent on. Neither of these solutions works with the different time resources and chaining required to describe the possible resource usage patterns in the machine description. In addition, neither of these resource usage descriptions takes advantage of the hardware interlocks that are included in the hardware. The hardware interlocks can be represented in our method as a simple dependency. Different version of a processor may be very similar with the exception of a couple pipelines latencies and enqueue times. With our method only the definition of the resource may need to be changed, while the resource usages for the instructions remain the same.

Serialization of resource use caused by data flow dependencies from resource to resource is allowed in the specification. Resource usage for a hypothetical multiply/accumulate operation is shown in figure 4.6. The data-flow graph of this operation is shown in figure 4.7. Both of the registers for the multiply can be fetched in parallel, and the multiply depends on the completion of those fetches before it can start. Likewise, the add pipeline must wait until the multiply and the third register fetch are complete before it can start. The final result is written into the last register.

Each vertical bar in the resource usage indicates a dependency exists. The resource group may depend on several different groups, as is the case for `ADD_PIPE` in figure 4.6. The numbers immediately preceding the vertical bar indicate which resource groups must complete their operations before use of the current resource may start. In the example, we also use the notational convenience of excluding the resource group number when the resource group to satisfy the dependency immediately precedes the current group.

Both GCC version 2 pipelines and Marion's individual function units can be expressed easily using this notation. GCC-style pipelines can be encoded as pipeline

```
<[ REG[i], REG[j];
   REG[k];
   1 | MUL_PIPE;
   2 3 | ADD_PIPE;
       | >REG[m]
]>
```

Figure 4.6 A hypothetical multiply/accumulate operation

function units and then single resources can specify instruction resource usage. Marion-style pipelines can be represented by each function unit being defined as a single-cycle resource. Resources starting in the same cycle would just be separated by commas, while resources starting in separate cycles would use the vertical bar to serialize their use.

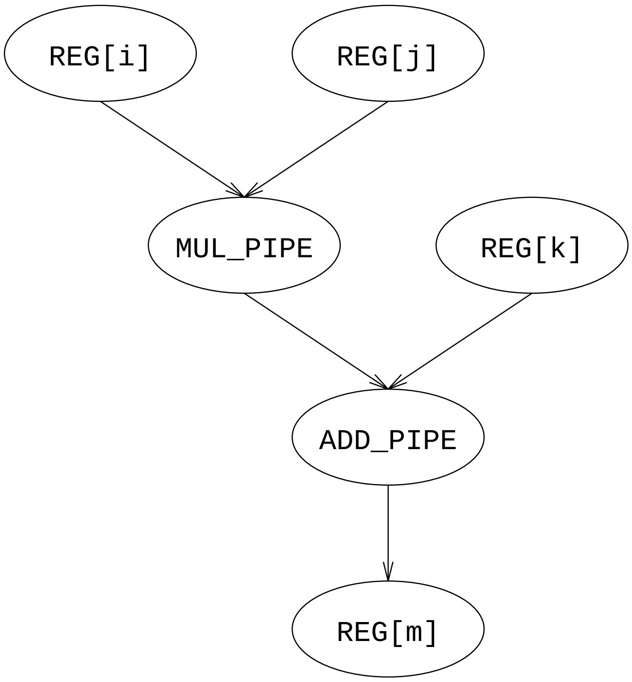

Figure 4.7 Dependence graph of multiply/accumulate operation

```
<[ REG[i], REG[j];
   | ADD_PIPE;
   | >REG[i]
]>
```

Figure 4.8 An addition instruction

4.4.2.2 Identifying Resources

Because multiple copies of resources may exist in the machine, it is necessary to indicate which copy of a resource is being used. This has not been addressed in GCC version 2 [Sta93], Marion [BHE91], or Wang [Wan94]. All of these approaches have either assumed that only one copy of the resource exists or resource allocation does not need to be addressed by the compiler if multiple copies of the resource exist. The exceptions to this assumption are registers and memory allocation.

Using the resource description language presented in this thesis, it is possible to have multiple resources of the same type in the description. There must be some means of identifying the specific resources in the resource usage, so that multiple references to the same resource can be identified. Likewise, references to different resources of the same type can also be expressed.

An example in which a resource is used multiple times is an addition instruction that takes two operands from registers and replaces the operand in one of the registers with the result of the calculation. The fact that one of the operand registers is also used to store the result must be encoded in the resource usage. The resource usages for such an operation is shown in figure 4.8. In this example, `i` and `j` are tags that allow us to specify that the resource `REG[i]` is used both as a source and a destination, while `REG[j]` is used only as a source. Notice that `REG[i]` and `REG[j]` could be the same register because there are no restrictions on the tags.

Another example to show the importance of being able to distinguish between specific resources of the same type would be local and non-local memory references

```
<[ PE[i].G_MEM[j];
   | PE[i].C_BUS;
   | >PE[i].REG[k]
]>
```

Figure 4.9 Load from local G_MEM

for the MIMD architecture shown in figure 4.3. The description of this architecture is given in figure 4.4. For a load from the G_MEM local to the processor, no access to the shared bus needs to be made. However, for a load from the shared memory of another processor, the shared bus must be accessed. The resource usage for the local G_MEM is shown in figure 4.9 and the access for G_MEM from a different processor is shown in figure 4.10. The remote access requires the use of the G_BUS, while the local access does not.

An additional concern is minimizing the changes to the resource usage when a smaller description is used to create a larger description. The resource usages from the smaller description do not include the new, higher levels in the hierarchy. Thus, to prevent ambiguity as to which specific resources are used it is assumed that they are all from the same branch of the resource tree unless otherwise specified. For

```
<[ (i!=m)
   G_BUS;
   | PE[i].G_MEM[j];
   | G_BUS;
   | PE[m].C_BUS;
   | >PE[m].REG[k]
]>
```

Figure 4.10 Load from remote G_MEM

```
<[ PE[*].PE_PREG[i], PE[*].PE_PREG[j];
   | PE[*].PE_ALU;
   | >PE[*].PE_PREG[i]
]>
```

Figure 4.11 Resource usage from MP-1 integer addition instruction

the example MIMD machine, the resource usage given for an operation for only one processor would still refer only to resources within a single processor, even through there are other resources of the same type in other processors.

Multiple resources of the same type may be required by an instruction. A common example of this would be SIMD instruction on a data parallel machine such as the MasPar MP-1. When data parallel instructions are executed on the machine, each of the active processing elements in the machine will participate in the instruction.

An example resource usage for the MasPar MP-1 integer addition instruction is given in figure 4.11. The instruction requires every PE to participate in the instruction. Thus, the `PE[*]` precedes each of the resources being used to indicate all PEs are used. If only one `PE` was participating in the instruction, then the `[*]` would not be included in any of the resources in figure 4.11.

4.4.2.3 Resources Allocation in Instruction Template

An instruction template outputs the machine instruction when the template is matched. These instructions consist of the instruction itself and its operands. The specification of the output has place holders for the operands, which will be filled in with the actual operands, such as labels for jump destinations, constants, and register names, when the instruction is generated.

However, there are many possible choices for the operands; the compiler must select resources out of the pool of available resources for that operand. This requires an allocator that manages the use of resources. In most compilers only the registers are allocated, but allocation could be done for other resources, such as processors in the machine. The compiler tools should be able to construct allocators for each type of resource that the compiler must manage.

The resource allocation information available in the template must be used to ensure that when the instructions are re-ordered that the correction of the program is preserved. The resource conflicts between different instructions can be determined by the resource usage information available in the instruction template.

Because the information about the resource allocation is included in the template, the compiler tools have flexibility in the method used to decide the allocation and scheduling. There is no explicit separation or ordering of these operations in the instruction templates. The problems of resource allocation and scheduling are intertwined. With this method of specification a combined resource allocator and scheduler could be generated. This may improve the quality of the code generated by the compiler [Bra91].

To better explain the interaction of the different parts of the instruction template required for resource allocation, and example instruction template that performs an addition of two operands and stores the results is shown in figure 4.12. Each instruction template is composed of five elements: the template name, where the computation stores the results, the resource usage of the template, the intermediate representation matched by this template, and the code to output if the template is matched.

The first element is the name of the template; in this example the name is `reg`. There could be a number of instruction templates that all have the same name. Basically, this indicates that all the templates produce either the same type of results or do the same operation. Instruction template names are also used in the intermediate representation section of the template to indicate the other instruction templates required for this template.

The second element of the instruction template, `<[REG[i]]>`, indicates where the results of the instruction template are stored. This information must be passed to other instruction templates that use the results of this template.

The resource usage, the third element in the instruction template, describes the resources required when the operation is performed on the target machine. The resource allocation information about the number of a particular type of resource during the execution of the operation is given in this section.

As discussed in section 4.4.2.2, it is often necessary to differentiate resources of the same type in the machine description. Identification of the resource is needed for another reason, tracking the allocation of resources. This information is required for both schedulers and register allocators. In this particular example between one and three registers will be used because there are no restrictions on whether the registers must be distinct and separate.

The intermediate representation that the instruction template matches is a `Add` node with two subtrees. Each of the subtrees must match some `reg` instruction template for this template to match.

The final element in the instruction template is the specification of what to output if the template is matched. For the most part the output specification is simply text of what to output when the template matches. However, the operands cannot be hardcoded into the output specification. Place holders are used instead of the operands, and when code is generated, the place holders are replaced with the actual operands. The resource placeholders in the output specification link the output specification with the resource usage and the intermediate representation patterns.

The place holders in the output specification gather information from the appropriate elements in the instruction template. The simplest information gathered is where the actual resource name should be inserted into the output, which is simply indicated by the position of the place holder in the output text. The first placeholder in the example, `%(1).k`, links that place holder with the actual resource associated

```
reg
    : <[ REG[i], REG[j]; | ADD_PIPE; | >REG[k] ]>
      <[REG[k]]>
      (Add reg reg)
      <"add %(1).k, %(1.2).i, %(1.1).j\n">
    ;
```

Figure 4.12 An addition instruction template

with the root of the intermediate representation. The `k` at the end of `%(1).k` connects the actual resource at the root with the resources tagged with `k`, `REG[k]`, in the resource usage. Since `REG[k]` is also in the results element of the instruction template, it is also linked to the root of the intermediate representation. Similarly, the `%(1.2).i` links the second child, `reg`, of intermediate representation pattern, `(Add reg reg)`, to `REG[i]` in the resource usage. Finally, `%(1.1).j` placeholder links the first `(Add reg reg)` to `REG[j]`.

5. MACHINE FEATURES

There are many different types of computer architectures available. These different computer architectures require the use of different types of coding techniques to make efficient use of each machine. Thus, the the compiler tools need to determine the computer architecture features available on the target machine and generate a compiler that produces code suitable for that target machine.

This chapter of the dissertation discusses how the needed information about the target machine is gather from the machine description. Compilers for serial machines are rather restricted in the information they collect about the target machine. In the most basic case, virtually no information may be gathered about the target machine and the compiler writer produces a translation for each of the constructs of the input language. For many serial computers this was acceptable, because all of target machines followed a basic "von Neumann" model with one central processing unit and a separate memory that stored all instructions and data, which was similar to many serial language execution models. In addition, the central processing units of early machines executed instructions in a totally serial fashion, so the execution order of the instructions did not affect performance. However, with parallel architectures there is a large variety of designs, and coding strategies that yield good performance on one type of architecture but yield poor performance on another, e.g. MIMD code on a SIMD machine [DC92]. The compiler tools must be able to identify the target machine to select the appropriate coding strategy to make good use of the target machine.

The compiler generation tools can control the code generation strategy used by the compiler. In most cases, this is accomplished by using certain transformations from a library in the compiler, and these transformation are not modified by the compiler generation tools. However, there are some phases of the compiler that will need to be generated for each target machine, such as the code selection phase and the allocation routine to manage registers and other resources.

Because the coding strategies vary from machine to machine, to ensure portability the coding strategies are not included in the machine-independent section of the compiler. The machine-independent front end generates a naive initial intermediate representation. The transformations implement the coding strategies by modify the intermediate representation to better match the target machine.

To match the intermediate representation to the target machine, determination of the target machine's execution model is crucial. However, it is not possible to simply describe the machine as a particular type of machine, such as a SIMD machine. This method of description will not work for two reasons. First, describing the execution model of the target machine with a single word is not detailed enough; there are many other aspects of the execution model, such as the parallelism width of the machine, that must also be known by the compiler. Second, most machines contain a combinations of different execution models, e.g. the CM-5 [Thi92] contains SIMD in the vector units, pipelining in the control processors, and MIMD across all processing nodes. The relationship between these components in the machine cannot be described with a single word.

A better method of gathering information about the target machine features is required. This method uses the resource based machine descriptions detailed in chapter 4 to gather information about the target machine. Both the resources available in the target machine and how the resources are used in the instruction templates must be examined. This chapter of the dissertation describes how this information is obtained from the machine descriptions.

This chapter first describes the internal data structured in section 5.1, the resource tree, used to hold information about the available hardware of the machine. The information stored in the resource tree is used both when analyzing individual instruction templates in the machine description as in section 5.2 and when analyzing the entire machine as in section 5.3.

5.1 Resource Tree

The resource tree is a crucial component to determine the machine features and this section describes the characteristics of the resource tree. The resource tree is a data structure that is built from the hardware resource description. The grouping and count of each resource component of the target machine can be extracted from the resource tree. With the resource usage associated with each instruction, the parallelism of the target machine can be determined. Other sections in this chapter describe how machine features are determined with the use of the resource tree.

Each resource in the machine description that is a part of the machine is included resource tree. Resources in the machine description that are not a part of some other resource included in the machine are not in the resource tree. These defined, but unused, resources may be in the machine description because a standard file of resources is shared between several machine descriptions. This exclusion of unused resources from the resource tree allows more accurate calculation of the available parallelism of the machine. It also allows some error checking to be performed by the software. It is clearly an error when a resource is in the resource usage of an instruction template, and that resource is not in the resource tree.

The resource tree structure mirrors the grouping in the hardware description; the composite resources are in the resource tree. The resources that make up a composite resource are the children of the composite resource. The number of copies of the resource contained within that composite resource is included in the tree.

The smallest and most basic resources are at the leaves of the tree. The leaves of the resource tree are fundamental resources, e.g. pipelines, memories, or registers. Because all the resources are fully described in the machine description, no composite resource can be a leaf in the resource tree, because the resources the composite resource contains are its children.

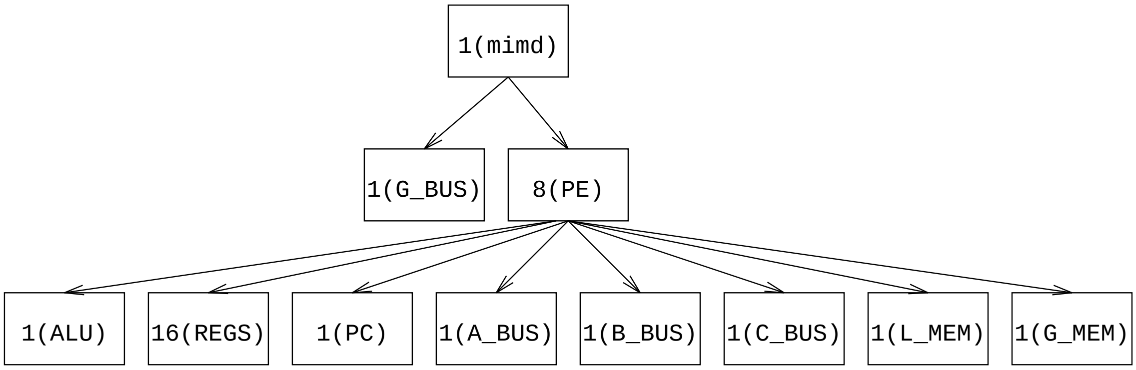

Figure 5.1 Resource tree of example MIMD machine

The root of the resource tree is the composite resource that describes the entire machine. There is only one copy of the root resource, which is natural considering that the compiler is generating code for a single machine. However, the machine could be composed of smaller less complex machines and there could be multiple copies of these smaller resources, e.g. a cluster of workstations.

Each resource in the tree has a count to indicate the number of times that resource is replicated at that level in the tree. Thus, to calculate the total number of that resource in the machine, the count of the resource at that level would be taken as the base value, then multiply by the replication of each composite resource that contains it.

An example resource tree for the machine description given in figure 4.4 is shown in figure 5.1. To find the number of `REGS` the machine has, start at the resource `REGS`, then select a path to the root of the tree. Multiply by the count of each resource on that path. The machine has 16 `REGS` in each processing element, `PE`. Therefore, the machine has a total of 128 `REGS`.

Although it is not shown in figure 5.1, the resource tree also provides a place to store timing information about each of the machine resources available. In general, the resource tree is the structure that provides the information about the hardware resources in the target machine. The information within the resource tree is used to calculate the cost of the instruction templates, to determine the type and width of parallelism available in the machine, to build schedulers, to build resource managers, and to determine the memory hierarchy in the target machine.

5.2 Instruction Template

The instruction templates in the machine description contain a large amount of information about the target machine. This section describes the information that can be extracted for the individual templates in the machine description. Section 5.3 describes aggregate characteristics that hold for groups of instruction templates.

Each of the instruction templates in the machine description is expected to represent one scheduleable unit on the target machine. Typically, this will be a single instruction on the target machine. However, in some cases, it may be most practical to specify a series of instructions that collectively implement the semantics of an operation in the intermediate representation; for example, a single template might encompass the series of instructions that implement a function prologue (entry point). Although useful, such templates can adversely affect the compiler's analysis and code scheduling, because each such group of instructions is treated as an atomic unit. For example, function prologue sequences often contain many store instructions that could benefit from interleaving with the register operations that follow that sequence, but the definition of a function prologue template would prevent such reordering.

It is possible to separate a group of instructions in a single instruction template, by converting the single template into a chain of singleton productions. Thus, each instruction template in the chain of singleton productions contains a single instruction. This allows greater flexibility in the scheduling of the instructions.

The relation of the instruction template to the intermediate representation is discussed in section 5.2.1. Because the instruction template does not fully specify the code generate when the template is matched, section 5.2.2 discusses where this additional information to completely specify the output is obtained. To discriminate between different possible translations of the intermediate representation, section 5.2.3 explains how to provide cost information for each template.

5.2.1 Intermediate Representation Covered by Instruction

In simplest case, a instruction template may not cover any of the intermediate representation. Examples of instruction templates that does not cover any of the intermediate representation are "glue" templates or singleton productions that convert from one instruction template to another, e.g. data movement to relocate operands to the appropriate registers for a particular instruction.

However, there must be some instructions templates that cover portions of the intermediate representation. The patterns that the templates match are partially specified. An example of this would be a template to perform an addition operation; the addition operator would be specified in the pattern as would the general form of the intermediate representation, but the operands would be specifies as any of a number of possible other instruction templates. Any of the possible instruction templates for the operands would produce results in the appropriate place for the addition operation and match the intermediate representation for the operands of the addition.

When translating the intermediate representation to the individual instructions, every portion of the intermediate representation must be covered by instruction templates. If the intermediate representation cannot be completely covered by the available templates, the uncovered portion of the intermediate representation cannot be translated into the available instruction templates in the machine description.

With the information about the portion of intermediate representation that the instruction template matches, the code that matches the intermediate representation to templates can be implemented. Traditionally, this translation of the intermediate representation to the available instructions has been referred to as code generation. Many researchers have developed theory on how to implement automata to perform the matching of the templates [AG85] [HC86] [Cha87] [HD87] [Wen90]. Several code-generator generators have be developed using these theories [Gla77] [Bir82] [Hen84] [Tji86]. However, simply matching the intermediate representation is not sufficient to generate the instruction, because the instruction template does not fully specify the output of the template. Section 5.2.2 describes what additional information must be obtained to generate the output for an instruction template match.

5.2.2 Instruction Operands

Each instruction template describes the form of the instruction to output. However, the instruction template does not fully specify the output. There are positions in the output specification that must be filled in appropriately before the instruction generated.

The information to complete the specification must be obtained from places other than the output specification. However, the output specification includes enough to locate the information to completely fill in the operands for the instruction. An operand could be any of a number of valid operands, e.g. a constant, a label, or a register name.

The operands are indicated in the output of the template by place holders, which are similar to the `printf()` function formatting specification for variables in the C language [KR78]. Each of the place holders in the output specification is replaced with an operand when the output is generated.

The compiler must have the ability to select and manage these operands. The information about the operands must be available to other instruction templates that need that information.

5.2.3 Template Cost

When instructions are selected to cover the intermediate representation, there are many coverings of the intermediate representation possible with the instruction templates. However, some method must be used which of the many possible parses to select. This selection is typically accomplished by the use of a cost function, which is minimized, e.g. space required for the program or the instruction execution time. A lower cost would indicate a better translation of the intermediate representation.

Even for serial machine, it can be computationally expensive to find code that is optimal with the respect to the cost function used [KA82] [AJU77], NP-complete. With parallel machines, the problem becomes more complex. Even though it may not be feasible to find an optimal solution, providing cost information about the instructions available on the target machine can improve the selection of instructions.

Because there are many possible parses (and translations) of the intermediate representation, some cost evaluation method must be included in the code selection of the compiler to decide which parse of the intermediate representation to use. Several methods have been used to select the "best" parse of the intermediate representation: template ordering, "maximal munch," dynamic programming, and exhaustive search.

The simplest method used to select a code implementation is template ordering, as used in PCC [Joh78] and GCC [Sta89]. The list of templates that describe the machine are ordered so the lower cost templates are earlier in the machine description than the more expensive templates. When compiling code, the first valid template match is the one applied, since it is assumed to have the lowest cost.

Template ordering puts the burden of determining the cost of templates on the programmer. Every time the machine description file is updated or changed the programmer must re-evaluate the cost of the templates. Further, selecting a locally inexpensive template can cause the total cost to be higher by forcing more expensive templates to be used in a later portion of the coding. Because there is no backtracking, there may be templates selected that will later lead to *blocking*, where some part of the intermediate representation cannot be matched because of the earlier choices made in the code selection.

Rather than taking the first (locally cheapest) match, "maximal munch" [Gla77] prefers the template which matches the largest portion of the intermediate code. This makes the assumption that larger templates are more efficient than smaller ones. Glanville treats this conversion of intermediate representation to final code as a LR parsing problem. If the grammar produced from the templates has a shift/reduce conflict, it is resolved in favor of the shift to allow matching the largest template possible.

Some dynamic programming techniques have been used to improve the quality of code generation [BS76] [AJU77] by finding an optimal order of instructions to minimize the number of registers used by the code. Dynamic programming breaks the problem of finding the translation of the best translation of the intermediate representation into smaller sub-problems of finding the best translation for each sub-tree and combining the solutions for subtrees into the complete solution. This translation can be performed in linear time [ASU86].

However, there could not be common sub-expression in the intermediate representation. As shown in both [BS76] and [AJU77] common sub-expressions make the problem NP-complete. In addition, the program evaluating the intermediate representation had to be *contiguous*. All the values to be calculated in memory are completed before evaluating the rest of the tree. Then for each node in the tree, the sub-trees are evaluated in any order, so long as each sub-tree is completely evaluated before another sub-tree at the same level is evaluated.

A method that has been tried with some success is an exhaustive search [KA82]. In the worst case, the search space grows exponentially, but branches of the search are pruned once they exceed the current lowest cost parse. For this reason, the number of matches considered is fairly small. The code generated was optimized for a linear combination of time and space costs that could be changed without the need to rebuild the compiler.

There are many possible measures of cost. Typically, the one that most programmers are interested in is the execution time. However, accurately estimating the execution time of a program is equivalent to the halting problem, because in addition to determining when the program completes, the time for all instruction must be totalled. Despite the inability to estimate the run time of a complete program, it is still useful to determine which set of instructions implement a particular construct in the intermediate representation with the least cost. Code-generator generators such as

[KA82], BEG [ESL89], Marion [BHE91], and BURG [FHP92], have the programmer associate a cost, usually execution time for the instruction, with each template.

However, manually determining the execution time of each instruction template can be tedious. In this dissertation we propose a method of automatically obtaining the cost information from the resource usage included in the instruction template. With the machine description language given in chapter 4, there are two measures of cost, latency and enqueue time, that can be determined from the resource usage information included with each instruction template.

Similar to GCC version 2 [Sta93], the cost information is not directly stored in the instruction template, but it is stored with the resource. GCC version 2 includes latency and enqueuing time for the pipeline units and templates can indicate the pipelines they use. However, this information is not used for instruction selection in GCC.

The information about the latency and enqueue time for the instruction template is obtained from the resource usage for the template. Thus, these costs are derived from the hardware resource information in the machine description. Each resource has a latency and enqueue time associated with it. The default latency and enqueue time for a composite resource are the maximum latency and enqueue time for the resources within the composite resource This default is reasonable, because it uses the worst case values unless more exact specification of how the resources within the composite resource are used is given.

The latency and enqueue times for each instruction template is calculated from the resource usage using the following rules:

- The maximum enqueue time of the resources in the resource usage is the enqueue time for the resource usage.
- As discussed in section 4.4.2.1, a flow graph of the resource usage can be constructed. The longest latency path in that flow graph is selected as the latency for the resource usage.

These estimates of the latency and enqueue time are not exact. The combined latency of multiple instructions may be less than the sum of the instructions' latencies due to the ability of the processor to overlap the execution of the instructions. A common example of this overlap would be the fetch and decode stages of most pipelined processors, e.g. AMD 29000 [Adv90]. The execution of one instruction could occur while another instruction is being decoded and a third instruction is being fetched. Thus, the portion of the latency cause by the fetch and decode is hidden.

With the latency of the instruction known and the number of result the instruction produces, another measure of the instruction template, throughput, can be calculated. To determine the results produced by the instruction, the intermediate representation that the instruction covers must be analyzed, which is given in more detail in section 5.2.1 and section 5.2.2.

To calculate the throughput, w, for the instruction template i, the number of operations done by the instruction, p, is divided by the latency of the instruction l.

$$w_i = \frac{p_i}{l_i} \qquad [5.1]$$

This equation provides an approximate estimate of the rate that the instruction calculates values. The throughput is useful for determining the point at which it it would be profitable to use the parallelism of the machine. This is an approximate value since the there may be other overhead to using a particular instruction, such as moving the data to particular registers.

An example would be two addition instructions, one instruction is operates on scalar data, the other operates on vector data up to a particular size, such as on the MasPar MP-1. The scalar instruction requires 1 cycle to produce a single result, while the vector instruction requires 24 cycles to produce up to 16,384 results [Mas90]. Thus, using the throughput calculation, 24 elements must be in a vector before it is profitable to use the vector version of the instruction.

5.3 Machine Aggregate Characteristics

A more complete picture of the machine than the one provided by the individual instruction templates can be derived from analyzing portions of the machine description larger than individual resources or instruction templates. This section describes the machine features information that can be gathered from the machine description language give in chapter 4.

The information obtained from the machine description described in the following sections is not exhaustive, but is designed to illustrate the utility of the this machine description language and to show information useful for the phase selection and ordering discussed in chapter 6.

5.3.1 Independent Processors

One type of feature required for many of the transformation is information on the number of independent processors on the target machine, which is used to determine the parallelism available on the machine. A related type of information is finding the group of resources that compose the processor. With the knowledge of the resource that compose a processor, data transfers between processors can be determined. Thus, this section describe how the number of processors available at each level of parallelism in the machine is determined and how the group of resources that make up a processor is found.

An *independent processor* is a resource in the machine that can perform computations asynchronously to other resources in the machine and has its own thread of control. Thus, a processing element, PE, on SIMD machines, which perform computations synchronously with other PEs in the machine, would not be an independent processor. However, an entire SIMD machine with the control unit would be a single independent processor because it has one thread of control.

Each independent processor has a single `PC` register resource, which is the program counter that indicates what instruction in the program the processor is executing. The

```
For each PC in the resource tree
Let processor=PC
While resource count of PC = 1 in processor
processor=parent(processor)
processor is resource that describes a type of processor in the machine
```

Figure 5.2 Processor resource extraction algorithm

method of finding the number of independent processors in the machine at each level in the machine is similar to the method given in section 5.1 to find the number of a particular resource that exists in the machine. By locating the PC resources in the resource tree and multiplying by the replication factor of the composite resources at each level in the resource tree the number of independent processors at each level can be determined.

The other aspect of the processors in the machine that must be determined is the boundaries of each independent processor. This information is required to determine when synchronizations between processors is required during scheduling (section 5.3.7. It is also used when analyzing the memory structure of the machine in section 5.3.2.

The algorithm to determine the reesources included in the independent processors is shown in figure 5.2.

Note there are a couple of interesting characteristics to this method of finding the processors in the macine. First, is that there are definite groupings of the processors in the machine. The size of each of these groups is identified by the algorithm used to count number of processors in the machine. Second, on machine such as the CM-5 [Thi92], two different types of processors would be found: one type that composes the processing nodes with the vector units and the host processor that controls the machine. The counts for these two types of processors would be the size of the machine and one respectively.

5.3.2 Data Movement

Because the machine description describes individual instructions, it takes a radically different approach for data movement than compilers such as GCC [Sta93]. Compilers, such as GCC, assume that there is a generalized move instruction that can move data from any part of the machine to any other part of the machine. This makes the move templates very complicated for GCC because the instruction template must be able to generate many different types instructions to perform all possible moves.

In this dissertation, each instruction template is assumed to represent a single instruction. Thus, a single template may not be sufficient to implement some moves from one storage location to another storage location. The compiler tools must be able to synthesize moves between different memory and register resources. In addition, some data movement operations may require operations on separate processing element, e.g. a message passed between processing element.

A table of the cost for the different move operations can be constructed to determine the memory hierarchy. Moves between closely located elements highest in the hierarchy would take the least amount of time, while moves between distance elements at the bottom of the hierarchy would take the longest amount of time.

Examining the data movement operation can be used to indicate the cost of a spilling a value from one level to another in the memory hierarchy.

5.3.3 Synchronization

Because high-performance machine often have separate hardware components, it is possible that the hardware components may operate asynchronously with respect to one another. However, to ensure the proper calculations are performed, the hardware components may need to be synchronized. The methods used to determine whether asynchrony can exist between hardware components and how the components are synchronized is discussed in this section.

Within an instruction template, it is assumed that all synchronization between the hardware resources is taken care of and no asynchrony exists within this operation. The instruction is assumed to be deterministic in the respect that the result it produces does not change from execution to execution due to the asynchonous exectution of operations within the instruction. With this assumption it is possible to determine whether the machine has SIMD or vector operations. These operations are synchronous because each operation is described as a single instruction template.

The next step up in the heirarchy for machines is multiple instructions within a single processor. It is assumed that the processor preserves the semantic of a sequence of instructions. Thus, no synchronization is required between instructions on the same processor. However, this does not accurately model some processors that have separate caches for data and instructions. It is possible to have the information between the caches to be conflicting. This may occur for programs that dynamically create code to execute. The newly created code may only exist in the data cache. The memory and the instruction cache may not have the information. Thus, references to this new code may obtain incorrect instructions. A flush of the data cache must occur to make the memory and the instruction cache consistent. This thesis only deals with statically compiled code, and this problem is not a concern.

Asynchrony may exist between processors in a machine. There are two causes of these variations in timing between processors. Some instructions have data dependent timing characteristics. There may also be variations in the sequence of instructions executed by each processor.

The timing for individual instructions may not be a fixed unit of time. Some processors may check for special case of data for instructions to descrease the amount of time required to execute the instruction, e.g. multipy by zero being faster than a non-zero multiply as the `MUL` on the Intel i386 [Int86]. Thus, even if the same sequence of instructions start execution at the same time on different processors, the different values being computed may cause the processors to be at different locations in the sequence at a later time.

Processors may also take different control flow paths throught the code they are executing. This can also introduce variations in timing. One processor my only need to iterate a loop a couple of times, while another processor may need to iterate a loop many more times. In the general case, it is not possible to predict this information before the program executes. Thus, if the processors communicate after the execution of the loops, some synchronization must occur to indicate that it is possible to complete the communication. It may be possible to include the synchronization with the data being sent as with some message passing systems.

5.3.4 Data Types

For code generation the compiler must be able to break operations into operations that perform operations on the data types that are available on the target machine. The compiler tools make the assumption that the at the time that code selection is performed by the compiler, the intermediate representation only uses data types that exist on the target machine. Thus, the machine feature extraction must determine the data types available on the target machine, so the transformations make the appropriate modifications in the intermediate representation.

This section of the dissertation will discuss how the information about the available data types for the target machine is gathered from the machine. In chapter 3 the assumption that the different data formats have the same size is also assumed for this work. This restriction is not a major restriction, because smaller data formats can be padded to the size of the largest data format. There are two steps to determining the data types: first determining the available data types handled by individual instructions on a single processor and determining aggregate data types produced by the available parallelism in the target machine.

The most basic data types available on the machine are the data types manipulated by individual instructions. This infomation is obtained by examining the linkage between the intermediate representation the template matches and the resource usage for the template, provided by the output specification. The intermediate representation indicates the semantics of the operation, which operands provide values to the operation and which operands store the results of the operation. The output specification described in section 4.4.2.2 links these operands of the intermediate representation to the resources in the resource usage. The dimension of the operands can be determined from the resource usage.

This analysis is performed for each instruction template in the description. The machine feature analysis must track which processors each data type is associated with. This becomes needed heterogenous machines, e.g. CM-5 which has a separate type of processor for controlling the processing nodes in the machine. Elements of data types need to be distributed across the same type of resources in the machine. Thus, the control processor in the CM-5 could not hold part of an aggregate data type because it is different from other processors in the machine.

5.3.5 Register and Memory Allocation

As discussed in section 4.4.2.3 many machines have a number of equivalent resources and any one of them could be used interchangeably. However, once that resource is used in the generated code, it must be separated from the other resources that it was equivalent to until it is returned to the pool of equivalent resources. The allocation routines perform the tasks of selecting resources to be used for particular instructions, ensuring that there are conflicting uses of the resource, and returning it to the set of equivalent resources when it becomes available again.

Routines that manage the allocation of resources such as registers and memory must be created from the information given in the machine description for each target machine. The information required for these allocation routines can be extracted easily from the machine description because the machine description clearly indicates these types of resources as `register` and `memory` resources. The routines can also exclude certain resources from the automatic allocation by examining the resources that are reserved in the machine description.

The other aspect of the allocation that must be determined is how the allocation routines are invoked in the compiler. With the machine description language given in chapter 4, it is possible to determine how and when a resource is used in an instruction. The resource usage specification in a template indicates the type of resources that must be allocated for the instruction to execute.

5.3.6 Available Parallelism

A crucial step in generating a compiler for a particular machine is to determine the parallelism available. The parallelism of interest is the parallelism that is directly supported by the hardware. Hence, software to allow the machine to support larger amount of parallelism, such as Paris on the Thinking machine CM-2 [Thi90] or the virtual SIMD execution model of the CM Fortran compiler for the CM-5 [Thi92] do not provide the information desired. If the compiler has detailed information about the parallelism supported by the machine, then the compiler can modify the code with such transformations as strip mining to better match the parallelism of the target machine and improve the performance the program on the target machine.

5.3.7 Scheduling

Because most machines allow concurrent execution of instructions, the same instruction ordered in different ways can produce different execution times. Thus, a goal of most compilers is to reduce the time for executing the instructions by selecting an order to maximize the concurrent execution of the instructions [Tho67].

Naive code generation schemes for a single processor that simply walk the intermediate representation to generate the instructions often produce code that has poor execution time characteristics. This is due to the related instructions that have dependencies between themselves occuring very close together in the instruction sequence. Thus, instructions must often wait to execute due to dependencies on prior instructions that have not completed execution. By scheduling the instructions, the processor can accomplish more useful work in the same amount of time by executing instructions that are not dependent on other uncompleted instructions. Section 5.3.7.1 discusses how this information is extracted from the machine description and the instruction scheduler built.

Similarly, scheduling can be performed on a larger scale than on the instruction level. Often this has been broken into two topics, placement and actual scheduling of the computations. This interprocessor scheduling is discussed in more detail in section 5.3.7.2.

5.3.7.1 Instruction Scheduling

On many processors, some measure of concurrency is allowed in the execution of instructions. This overlap of the execution of instructions is allowed as long as a *hazard* does not exist. In [HP90], "There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle." Further, [HP90] classifies hazards into three categories:

1. *Structural hazards* occur when a simultaneous use of a machine resource caused by overlapping execution of instructions cannot be supported.

2. *Data hazards* occur from data dependencies (or anti-dependencies) of possibly overlapping instructions.

3. *Control hazards* are caused by instructions that modify the program counter (`PC`), and changing the control flow path.

This hazard information can be determined from the machine description of the computer. The structural hazards can be determined by intersecting of the pipeline resources required by the two instructions being analyzed. The data hazards are visible with the register resources the instruction uses. Finally, the control hazards are visible as instructions that write into the `PC` resource.

When constructing the scheduler for the processor, some assumptions are made about the processor. The processor has some interlock circuitry to prevent certain types of hazards from producing incorrect results. Some schedules may produce instruction sequences that have hazards on them. In particular, instruction sequences that have structural hazards are assumed to stall instructions until the resource is available. The scheduler doe not need to insert addition “filler” instructions to have correct execution. Similarly, data hazards caused by a result of an instruction not being produce before a later instruction in the sequence uses the result are also assumed to stall the instructions until the result is available.

The modelling of the instructions’ behavior is simplified. All the register and memory resource uses assumed to occur at the beginning of the instruction and all the register writes are assumed to happen at the end of the instruction template. All of the pipeline resource usages are assumed to occur in the same places as before.

With this simplified instruction behavior, the interactions between instructions can be analyzed, and a table of the number of cycles of stall caused by structural hazards can be built. This table is a two dimensional table indicating pairs of instructions, the current instrucion and a following instruction. This table contains the number of cycles that must pass before that next instruction can start. It is assumed that once this amount of time has passed, the stalled instruction can execute and there are no further restrictions on when the instruction can start.

5.3.7.2 Interprocessor Scheduling

In addition to scheduling instructions on individual processors, the compiler is required to distribute work among the processors in the machine to decrease the execution time of the program. To accomplish this task, the compiler must schedule work between the processors in the machine. There are some differences between this scheduling and the scheduling accomplished within a single processor. This section discusses how intraprocessor scheduling might be accomplished for a target machine given the information in the machine description.

In section 5.3.7.1 an approximate estimate of the time that passes is made by the instruction schedulers. The inaccuracy of this estimate does not cause a problem for the scheduling within a single processor because the processor is assumed to handle data and structural hazards correctly if they do occur. However, very few machines have hardware to detect data hazards across processors, and the data hazards must be resolved by the software. Thus, the purpose of interprocessor scheduling is to minimize the cost of resolving these data hazards to allow the machine to perform the computations as quickly as possible.

5.4 Conclusion

With the range of target machines now available the machine description language must be able to encode a wider range of machines than previous machince description languages. This chapter has shown a number of examples of how information about the target machine can be extracted from the machine description given in chapter 4. This machine feature information can then be to select transformations appropriate for the target machine and be used to calculate parameters needed for some of the transformations, which is discussed in more detail in chapter 6.

6. COMPILER PHASE SELECTION AND ORDERING

Few high-performance compilers use a one-step direct translation strategy for generating output from the input language, because the direct translations generated are often inefficient. Also constructing a complex compiler as one, large, monolithic step makes it difficult to modify and maintain the software. Thus, most compilers are composed of many steps; where each step or phase accomplishes a specific task.

Typically, in most compilers, such as GCC [Sta93], the phases and their order in the compiler is selected by hand. This works reasonably well if all versions of the compilers require the same phases to generate code for target machines, which is the case if the target machines are similar. However, this may not be so for even sequential machines and the variations in architectures for parallel machines are much greater. Similar architectures would have more phases in common, and less similar architectures would have fewer phases in common.

The various phases of the compiler represent a large portion of the invested effort. It is extremely desirable to be able to re-use this code for various compilers. Thus, these reusable transformations could be collected into a library. The phases could be selected from the library and quickly assembled into a compiler for a target computer architecture.

It is extremely desirable to have a tool that automates the process of selecting the phases to be included in the compiler and ordering them, since there could be literally scores or hundreds of phases contained in the library. This chapter of the thesis will discuss the methods of encoding the necessary information about the phases for the tool and how the tool could use that information to automate the tasks.

First, this chapter will describe in more detail what is a phase in a compiler, in section 6.1. Then, the method used to describe the information about the phase to the phase selection and phase ordering stages is described in section 6.2. The methods used to select and order the phases are described in sections 6.3 and 6.4 respectively.

6.1 Phases

A *phase* is a software module in the compiler, which accomplishes a specific task. A phase takes the intermediate representation and symbol table, which specify a program, and performs some manipulation on them. The modified intermediate representation and symbol table can then be used by other phases. There are only two phases which do not follow this convention, the front-end phase, which produces the initial intermediate representation of a program, and the final code generation phase which translates the intermediate representation into assembly language instructions (or some other output language).

The method used to produce the phases is outside the scope of this thesis. For the purposes of this thesis, the phases could be "black boxes," written in any of a number of languages tailored to describing optimizations and transformations, such as GOSpeL [WS91], Sharlit [TH92], or SORCERER [Par94]. The information that must be known about the transformations is how they interface with other transformations.

6.2 Information about Phase

The method used to implement a phase is not important. However, how the phase interfaces to other phases, what properties must exist in the intermediate representation before the phase can be successfully executed, and what properties exist after the phase executes must all be specified to allow the ordering of the phases to be automated. This information would be produced by the programmer who wrote the phase.

The abstraction used to encode this information about the phase are attributes. Attributes describe the properties of the intermediate representation. They may indicate that all the expressions in the intermediate representation are constant folded or that certain fields of the intermediate representation currently hold valid information. Attributes can also indicate the phases that have been used to produce the current intermediate representation. Each phase can be summarized by the effects it has on the attributes of the intermediate representation. The effects can be categorized as:

`uses` An attribute in the `uses` category must exist as a precondition for an application of the phase.

`disabling` No attribute in the `disabling` category can exist when the phase is applied.

`kills` An attribute in the `kills` category may not exist after the phase is applied.

`defines` An attribute in the `defines` category is guaranteed to exist after the phase is applied, regardless of whether or not the attribute existed before the application of the phase.

Attributes that are not listed in the `uses`, `disabling`, `kills`, or `defines` categories for a phase are unchanged by that phase.

The attributes are independent of the target machine; the phase will have the same effects on the attributes no matter what the target machine is. However, some information about the phase is dependent on the target machine architecture. This information can be summarized by three fields:

`benefit` An expression that is evaluated to determine the usefulness of the phase given a particular target machine description. If the expression evaluates to greater than zero, the phase should be used on the target machine. If the expression evaluates to less than zero, it should not be used on the target machine.

`cost` — An expression that gives a cost of applying the phase to the intermediate representation. The summation of the phase costs can be used to determine which of two possible phase orderings is better (lower cost).

parameters — A list of the target machine dependent parameters that should be passed to the phase. Such parameters are needed for instance in loop parallelization which requires the number of processors contained in the target machine.

Finally, the description includes the name of the transformation. This allows the software to generate code that executes the phases in the appropriate order.

The attributes information and `cost` are used by the phase ordering stage to determine a valid sequence of phases to apply after the phase selection is complete. The phase selection only uses the `benefit` expression to decide whether to include or exclude the phase from the phase order. The phase selection process is discussed further in the following section.

6.3 Phase Selection

The first step in automating the assembly of the phases into a compiler is put the phases into three categories: phases that must be in the completed compiler, ones that must be excluded from the compiler, and those that may or may not be in the compiler. The method used to categorize the phases is to associate an expression with each phase as outlined in the previous section. The expression which computes the `benefit` determines which category the phase belongs to. The three categories of `benefit` are:

`benefit` > 0 — A valid phase ordering solution must include this phase.

`benefit` $= 0$ — A valid phase ordering solution may include this phase. It will be in the phase ordering solution if some other phase requires it.

`benefit` < 0 — A valid phase ordering solution must exclude this phase.

Two additional phases which will always be included in the compiler are `first` and `last`, which are the first and last phases of the compiler respectively. The `first` phase generates the initial intermediate representation and `last` generates the final output of the compiler. The selected phases are passed to the phase ordering algorithm, which uses the information to find a solution to the phase ordering.

6.4 Phase Ordering

Once the phases have been partitioned into the three sets: "must include" (`benefit` > 0), "may include" (`benefit` $= 0$), and "must exclude" (`benefit` < 0), a valid ordering of phases must be created. Initially, the only information about the ordering known is that no phase precedes the `first` phase and no phase follows the `last` phase. How can the order of the other phases in the compiler be found?

At first glance, a topological sort of the "must include" phases appears to be sufficient to find an ordering of the phases, but that is not the case. Phases in the "may include" set may be required for a valid phase ordering. Even if a valid solution can be generated with only the "must include" list, a topological sort may not be able to generate a valid ordering of the phases because the phases may destroy some attributes. These cases may be solved by letting certain phases be executed multiple times.

An example description of phases that cannot be solved with a topological sort is listed below:

`First`	Does nothing
`Last`	Only `uses folded`
`Fold`	Constant folding (with value propagation), which `defines folded`; `benefit` > 0
`Unroll`	Loop unrolling, which both `uses` and `kills folded`; `benefit` > 0

A phase ordering is generated where the constant folding is executed twice. The first execution is required to set the folded attribute for the Unroll and the second is required because Unroll destroys the folded attribute and Last requires it. A valid ordering of phases is the following:

First

Constant folding

Unroll

Constant folding

Last

For the previous example it was found that the attribute folded was required by Last, but Unrolling destroys folded. In the prototype implementation of the phase ordering software such information can determined by examining a bit vector which has one bit allocated for each attribute. It also has one bit allocated for each phase name. The goal state can simply mark the phase names of the "must include" list in the same manner that the `uses` are marked for the last phase; items the "must exclude" list are marked the in the same manner as the `disabling` attributes for the goal state. A more detailed explanation of the algorithm follows.

The algorithm developed to determine a valid phase ordering is based on the GPS (General Problem Solver) [EN69]. The attributes produced by the `first` phase create the initial starting state, and the attributes in the `use` category of the `last` phase represent the goal state. Additional requirements on the goal state are the made. Attributes that represent that execution of the "must execute" phases are added to the goal condition. And the attributes representing the the `disabling` category and "must exclude" phases are excluded from goal state.

Figure 6.1 describes the algorithm. Step 1 sets the initial conditions, while steps 2 through 5 form the recursive section of the algorithm. Step 2 finds the difference between the state that the algorithm is currently examining and the goal state. If there is no difference between the current state and the goal state, then the algorithm has successfully found an ordering of the phases.

However, more than likely there is a difference between the current state and the goal state. Some attempt must be made to reduce the difference between the current state and the goal state, which is accomplished by step 4 of the algorithm. The various phases in the "must include" and "may include" are tentatively tried. There is no reason to attempt using the phases with `benefit` < 0, because these cannot be in a valid solution. The functions that can be validly applied can be ranked, and the "best" candidate tried. If no solution is found using it, the lower ranked candidates can be tried in order.

The state generated by the tentative application of a rule is checked against a table of previously generated states. Further exploration from that state is attempted only if the cost to reach that state was less than the previous lowest cost. If the cost to reach that state was not less than previous lowest cost, there is no reason to explore further from that location, because a lower cost list of phases can be applied to achieve the same state. Checking a state table prevent the algorithm from getting stuck in a "infinite loop."

There are some differences between the GPS algorithm [EN69] and this algorithm. The GPS algorithm selects a difference between the current state and the goal state and then attempts to remove the difference by finding a solution to the subproblem. For the phase ordering algorithm it finds a list of currently applicable phases at the current state and tries to solve the subproblem from the tentatively applied phase to the final goal.

To demonstrate how this algorithm works, descriptions for some of the analysis and transformation phases from Wolfe's PhD thesis [Wol82] shown in figure 6.2. The purpose of each of the phases is listed below.

`data_dependence` This is an analysis phase which determines the data dependencies that exist in the code.

`loop_vectorization` The `loop_vectorization` phase transforms the statements in inner loops into vector operations if it will not change result of the calculations.

1 Let curstate=state at `first`

2 Find the difference between curstate and state at `last` $\cup$ set of phases with `benefit` > 0

3 If difference is covered, done (success)

4 For each phase that could be applied

  a Set curstate=state after phase

  b If this application gives the lowest cost path to this state add this phase and recursively call 2

  c Remove this phase

5 Return (failure)

Figure 6.1 Phase ordering algorithm

`loop_parallelization` The `loop_parallelization` transforms outer loops into `DOALL` [Wol82] loops where separate iterations can be independently executed.

`loop_scalarization` The `loop_scalarization` transformation converts vectorized loops into serial loops.

`statement_reorder` In some cases the loop carried dependencies can be minimized by reordering the statements in the loop. This can allow the statements in the loop to be vectorized.

`loop_fission` Within a loop some statements may be impossible to vectorize. These unvectorizable instructions are put in separate loops from statements that do not have dependencies. This transformation allows part of the loop to be make use of the parallelism available on the machine.

`loop_fusion` On most machines there is an overhead in executing loops. Statements contained in separate loops may be combined if the loop bounds are identical and combining the loops does not change the results of the program. This information is determined from the dependence analysis.

Each of the phases has a benefit associated with it. Notice that `benefit` $\equiv 0$ for `data_dependence`. This will be the case for any phase that only provides analysis information. The phase should not be included in the compiler unless some other phase requires that information.

Other phases are either excluded or included in the compiler. Phases such as vectorization are detrimental to performance on scalar machines, because of the loop overhead incurred to emulate vector constructs on scalar machines and the additional storage overhead required for temporaries. For this example `loop_scalarization` and `loop_fusion` will reduce the overhead for scalar machines.

```
phase: data_dependence; cost: 1; benefit: 0;
uses: IR; defines: depend_graph;

phase: loop_vectorization;
cost: 1; benefit: (vector_inst_exist() ? 1 : -1);
uses: IR, depend_graph, reorder, fission;
defines: IR, vectorize;

phase: loop_parallelization;
cost: 1; benefit: (res_count(PC)>1? 1 : -1);
uses: IR, depend_graph; defines: IR, parallel;

phase: loop_scalarization;
cost: 1; benefit: (!vector_inst_exist() ? 1 : -1);
uses: IR, depend_graph; defines: IR, scalar;

phase: statement_reorder; cost: 1; benefit: 0;
uses: IR, depend_graph; defines: IR, reorder;

phase: loop_fission; cost: 1; benefit: 0;
uses: IR, depend_graph; defines: IR, fission;

phase: loop_fusion;
cost: 1; benefit: (vector_inst_exist() ? 1 :  -1);
uses: IR, depend_graph, scalar;
defines: IR, fusion;

phase: first; benefit: 1; cost: 1;
defines: IR;

phase: last; benefit: 1; cost: 1;
uses: IR;
```

Figure 6.2 Phase ordering example

For each of the phases in the example the cost to execute that phase is assumed to be the same; this may not be true in all cases. An example is that there are several different methods of doing dependence analysis on a program and the cost for these different methods varies, but all the methods of dependence analysis determine the same information.

Table 6.1 shows the order that the phases are applied. Initially, the algorithm ordered only `first` at the beginning, `last` at the end of the list, and required `loop_vectorization` to be in the phase ordering. The other phases were included and ordered as requirement to get a valid phase ordering.

For a scalar machine, the resulting phase ordering produced is very different. The only phase besides `first` and `last` that the two have in common is the `data_dependence` phase. While it is quite likely that the `first` phase is the same for the two compilers, `last` is the final code generation, and in reality is not a common element between the two compilers.

Table 6.1 Phase ordering result for vector machine

Attribute	front_end	data_dependence	statement_reorder	loop_fission	loop_vectorization	code_selection
front_end	X	X	X	X	X	X
IR	X	X	X	X	X	X
symbol_table	X	X	X	X	X	X
code_selection						X
data_dependence		X	X	X	X	X
depend_graph		X	X	X	X	X
loop_vectorization					X	X
reorder			X	X	X	X
fission				X	X	X
vectorize					X	X
loop_parallelization						
parallel						
loop_scalarization						
scalar						
statement_reorder			X	X	X	X
loop_fission				X	X	X
loop_fusion						
fusion						

Table 6.2 Phase ordering result for scalar machine

Attribute	front_end	fold	select_inst	schedule_inst	gen_inst
front_end	X	X	X	X	X
valid	X	X	X	X	X
IR	X	X	X	X	X
symbol_table	X	X	X	X	X
select_inst			X	X	X
folded		X	X	X	X
inst			X	X	X
schedule_inst				X	X
schedule				X	X
gen_inst					X
fold		X	X	X	X

7. IMPLEMENTATION OF TOOLS

To test the concepts of the that described in the previous chapters, a prototype tool, the Parallel Instruction Generator Generator (PIGG), was developed. The tools was built primarily to study and refine the ideas of extracting information about the target machine and applying this information to select and order phases in the compiler appropriate for the target machine.

PIGG was developed with the aid of the Purdue Compiler Construction Tool Set, PCCTS [PDC92], a public domain software package containing lexical analyzers generators, parser generators, and a software library that includes symbol table, set, and abstract syntax tree manipulation routines. However, PCCTS does not contain a code generator. Thus, work on PIGG would complement PCCTS.

In addition of PIGG being constructed with PCCTS, it is also designed to generate code that is compatible with compilers generated with PCCTS. Eventually, the goal is to integrate PIGG into the PCCTS and release it with PCCTS. Thus, PIGG would function as both a code generator generator and a tool to select phases appropriate for the target machine to PCCTS.

This chapter of the dissertation describes the construction of PIGG, how the tools operates with PCCTS, the features the tool provides, and the restrictions on the current implementation of the tool.

7.1 Compiler Construction

PIGG has been incorporated into generating compilers. To better understand how PIGG was incorporated into PCCTS, this section of the chapter describes the files used and created by each of the tools to produce a complete compiler. Figure 7.1

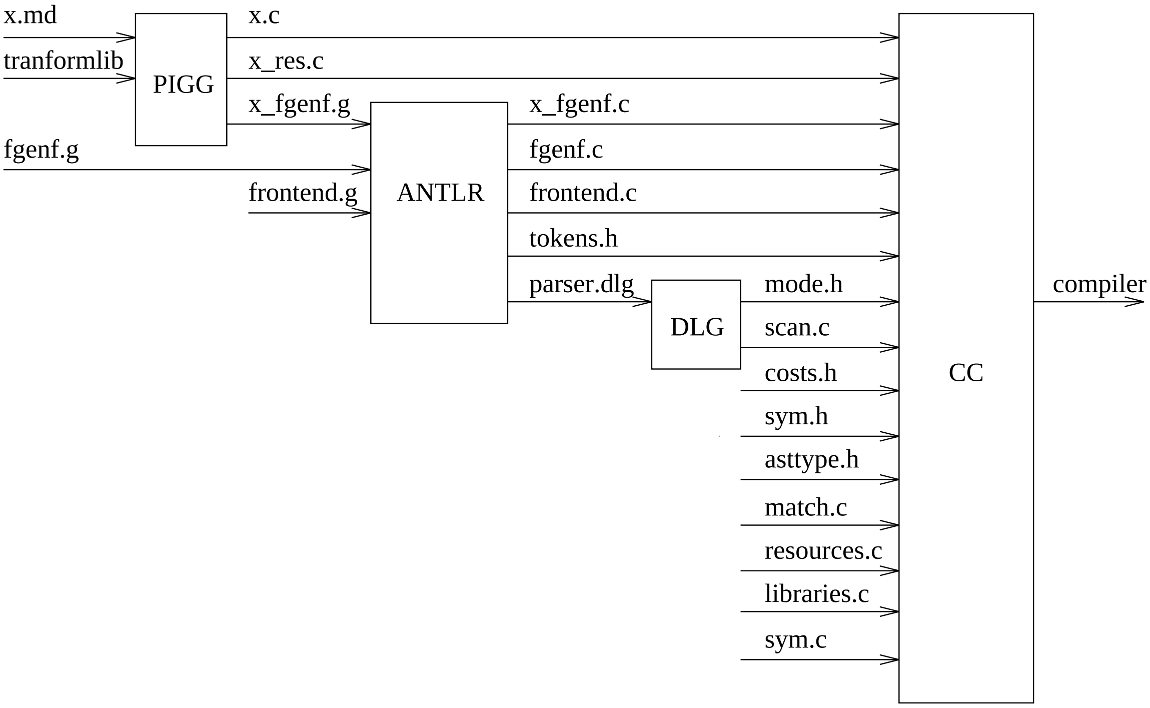

Figure 7.1 File flow in the creation of a compiler with PIGG

indicates the which tools creates certain files and which files are required as input to certain tools.

For our example a hypothetical machine, x, is the target machine for the compiler. For this machine the compiler writer must create a machine description `x.md`, which will be analyzed by PIGG. The other information required by PIGG is the description of the transformations available, `transformlib`. The transformation library indicates the architectures that the transformation is useful for, the attributes exist on the intermediate representation when the transformation is performed, and the attributes that must exist before the transformation is applied to the intermediate representation. PIGG can read multiple transformation descriptions. However, for this example it is assumed that thre is only one file describing all the transformations.

From the machine description and the transformation description, PIGG constructs a number of files. The code selection routines produced by PIGG are located in `x.c`. In addition to the code selection routines, the code for scheduling the instructions and the driver to call the transformations in the appropriate order are included in x.c. Two other files are produced by PIGG, x_res.c and x_fgenf.g. The information about the resources available on the target machine required for resource allocation is stored in x_res.c, which is used by resources.c, the resource allocation routines. To interpret the output specification of the instructions at runtime, the x_fgenf.g is created by PIGG. This file is a grammar specification of how to parse the output specification of the templates.

Because PIGG generates a parser specification, x_fgenf.g must be passed to ANTLR. ANTLR constructs parsers from specifications. For a compiler constructed with PIGG, ANTLR must construct at least two parsers, one for parsing the output specifications and one to parse the source language for the compiler. The output specification parser is created from x_fgenf.g and fgenf., the part of the output specification common to all output specifications. The source language is described in frontend.g.

ANTLR generates C code for each of the grammar specification files. Thus, for x_fgenf.g, fgenf.g, and frontend.g ANTLR generates x_fgenf.c, fgenf.c, and frontend.c respectively. ANTLR generates a header file, tokens.h, which contains the definitions for the token names and declarations of the functions in x_fgenf.c, fgenf.c, and frontend.c. Because the lexical analyzer is integrated with the parser generator, ANTLR extracts the patterns the lexical analyzer should find.

DLG translates the specification of the lexical analyzer into C code. Similar to ANTLR, DLG has the ability to have multiple lexical analyzers in the same specification. This feature is used to separate the lexical analyzers for the output specification and the front end. The lexical analyzer code is implemented in scan.c. Information to allow switching between the lexical analyzers is in mode.h.

Some of the transformations in libraries.c may have been created by transformation tools such as GOSpeL [WS91], Sharlit [TH92], or SORCERER [Par94]. However, it is assumed that the C code for them is contained within libraries.c.

There are a number of support files that must be included in the compiler to produce a working compiler. The header file costs.h has the structure used to determine the cost of a particular template during code selection and the header file `asttype.h` contains the structure used as the intermediate representation of the compiler. The symbol table manager is contained within `sym.h` and sym.c. The file `match.c` has code to implement the matching of the instruction templates with the intermediate representation and the resource allocation routines are contained within `resources.c`.

As indicated by figure 7.1, all of these files are compiled by the C compiler to produce the complete compiler.

7.2 PIGG

The PIGG prototype was constructed with the aid of the Purdue Compiler Construction Tool Set, which considerably simplified the generation of the sections of PIGG which parsed the machine description and the phase descriptions.

7.3 Machine Description

The machine description format closely follows the machine description language given in chapter 4. The fundemental resources of registers, pipelines, and memory are included in the specification language. However, a minor syntactic modification to make it easier for PIGG to recognize the resource that describes the entire machine was made. The machine is indicated with the `machine` keyword.

Another simplification in the specification language is that all of the timing specifications are simple integer values. This is not a major restriction because most processors have some basic unit of time counted by some clock in the machine. This count can be used to specify the timing characteristics of each of the hardware units in the machine.

7.4 Code Selection

The code selection phase for the target machine is generated from the instruction templates given in the machine description. This phase decides which instructions to use to implement the intermediate representation. The basic algorithm performs pattern matching and attempts to find the lowest cost covering of the intermediate representation with the available instruction templates. The algorithm is modeled after the bounded tree search discussed in [KA82].

Originally, PIGG generated a a set of C tables that indicated described the patterns and the costs associated with each machine and used an automaton to interpret the tables. However, the tables were the same structure used for the abstract syntax trees. The tables were large and the code in PIGG to generate them had to be changed when the structure was changed. Later verison of PIGG generated C code to accomplish the matching.

The tables generated by the early versions of PIGG were dynamically read by the PIGG generated compiler. Later versions of PIGG statically generated the tables. This avoided reading in the tables, but generated a significantly larger program because AST structure was fairly large. An additional problem was that everytime a change was made in the AST structure, a modification in the table generation code of PIGG. To avoid these problems a different implementation of the pattern matching was tried.

The newer version of PIGG generates C code for the code selection phase of the compiler. This had the advantage that the code produced by PIGG had to be recompiled when the AST structured was modified rather than requiring modifcations to PIGG, generating new code selection phase, and recompiling.

The code selection routines would build a new tree described the instruction templates applied and contained pointers back to the matched intermediate representation.

7.5 Resource Management

A set of table driven routines were developed to handle the management of resources. These routines implement on-the-fly register allocation. Tables used by the

```
reg
   : <[ REG, REG; | ADD_PIPE; | >REG ]>
     (Add reg reg)
     <"add %>(1).r, %<(1.2).r, %<(1.1).r\n">
   ;
```

Figure 7.2 An prototype PIGG addition instruction template

routines were automatically constructed by PIGG. These table were constructed from the `register` and the `reserved` resources.

The prototype has a different version of the output specification, which required some redunant information about the class of the resource being used to be in the specification. Figures 7.2 and 7.3 show equivalent add instruction templates. When code was being generated in the prototype PIGG, the place holders (indicated by a leading "management routines.

Each of the place holders in the output specification include enough information select the appropriate resource. In particular each place hold included information about which part of the intermediate representation that the resource is associated, the type of operation that is being done on that resource (read, write, or modify), and the class of resource that was required for that operation.

```
reg
   : <[ REG[i], REG[j]; | ADD_PIPE; | >REG[k] ]>
     <[REG[k]]>
     (Add reg reg)
     <"add %(1).k, %(1.2).i, %(1.1).j\n">
   ;
```

Figure 7.3 An thesis addition instruction template

With the prototype specification of how resources are manage three problems were noted: redundant information existed in the resource usage and output specifications, it was difficult to determine which resource in the resource usage was related to the output specification if there was more than one resource of the same type, and it was difficult to determine which resource held the result of interest when there were side effects. Thus, the language for the machine description was revised. However, the revised syntax has not been incorporated into the prototype tool.

7.6 Phase Selection

PIGG generates a driver that calls the appropriate phases of the compiler in the correct order. The phase section and ordering algorithm closely follows the algorithm described in chapter 6. However, the algorithm implemented in PIGG also uses the cost information associated with each transformation. Thus, the lowest cost sequences of transformations is the one selected if there are two sequences that includes the required phases.

The parameters of the phase descriptions are directly transcribed into the phase selection routine. Thus, for two calls to the same transformation, different arguments can be passed.

7.7 Conclusion

Although the PIGG prototype was not very usable as a regular tool, it did provide a great deal of feed back on what problems existed. Chapter 8 discusses what projects were attempted with the prototype tool to evaluate the strengths and weaknesses of the tool.

8. PROTOTYPING OF COMPILERS

To test the ideas presented in this thesis, two prototype front ends and two prototype back ends were developed. The front ends, C-lite and Fortran-P [OPE+93], are subset languages based on C and FORTRAN respectively. The front ends generate similar intermediate representations. However, the target machines are very different. One goal of this thesis is to develop tools that allow the generation of compilers for very different computer architectures. One target is a massively parallel SIMD machine, the MasPar MP-1 [Bla90]; the other target is a MIMD machine composed of vector processors, the Gould NP-1 [Gou88].

This chapter of the thesis will first describe the front end languages in section 8.1. The target machines will be described in more detail in section 8.2. Finally, in section 8.3 the results of these prototype compilers will be described.

8.1 Source Languages

The front ends are designed aid in the demonstration of the techniques used to retarget the compilers for different parallel architectures. They are not full-scale implementations of C or FORTRAN. Rather than designing new languages, existing language grammars for C and Fortran written for the Purdue Compiler Construction Tool Set (PCCTS) [PDC92] were subsetted and modified to produce the intermediate representation for the prototype compilers.

The same intermediate representation is generated by both front ends. This intermediate representation is an extremely naive intermediate representation and is not tailored for the target machine at all. A series of transformations will will make modifications on the intermediate representation to produce an intermediate representation of the original program that will work more efficiently on the target machine.

The following subsections will describe the features and restrictions of the source languages understood by the front ends.

8.1.1 C-lite

C-lite is a simple language, which is based on a subset of C [KR78]. The goal when developing this front end was to make it recognize standard C code if the restrictions on the language are followed.

One significant different between C and C-lite is the way that arrays handled. Arrays are not first class objects in C. This leads to pointers being extensively used to pass arrays to functions. However, there is no means of determining if the pointer is pointing to one object or many. Thus in C-lite a distinction is made between arrays and pointers. Thus, using a the name of an object declared as a array does not give a pointer to the first element in the object, but instead refers to the entire array.

8.1.2 Fortran-P

Fortran-P was orginally developed to faciliate the translations of computational fluid dynamics (CFD) code from vanilla FORTRAN 77 to specific parallel machines' versions of FORTRAN. Since Fortran-P is a subset language, many of the complex constructs in Fortran have been removed. There are a number restrictions on Fortran-P.

The CFD codes have a 2 dimensional or 3 dimensional array represent the area or volumen being simulated. Each point in the array has a number of values associated with that point to indicate the physical properties of the point. In addition to the physical region being simulated, there is a boundary around the array to store the boundary conditions. The initial code does not explicitly state any parallelism and is simply plain FORTRAN 77.

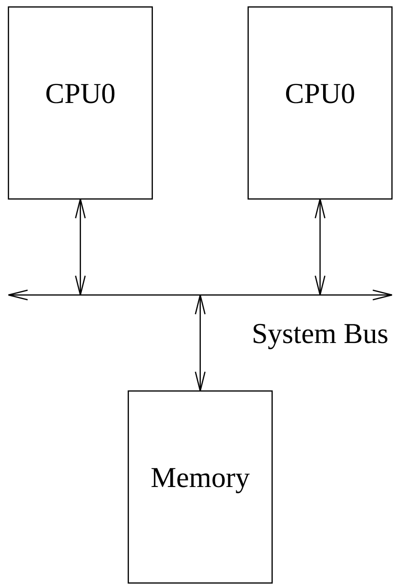

Figure 8.1 The Gould NP-1 system architecture

8.2 Target Machines

The computer architectures machines chosen to be targets for the compilers because they have different forms of parallelism.

8.2.1 Gould NP-1

The Gould NP-1 [Gou87] is a supermini computer. The machine is a symetric multiprocessor machine with all memory on a shared bus as shown in figure 8.1. The machine has two vector processors; each processor has a local cache to reduce the traffic on the system bus. A model of a single vector NP-1 processor is shown in figure 8.2. The vector registers in the machine each have 32 elements and some vector operations can be chained to improve the performance of the machine.

8.2.2 MasPar MP-1

In many respects the MP-1 [Bla90] is a fairly traditional massively parallel SIMD architecture. The largest version of the MP-1 has 16,384 processing elements, which can produce a peak performance of 1.3 GFLOPS. This section will cover in more detail some of the design feature of the MP-1 that will affect the compiler.

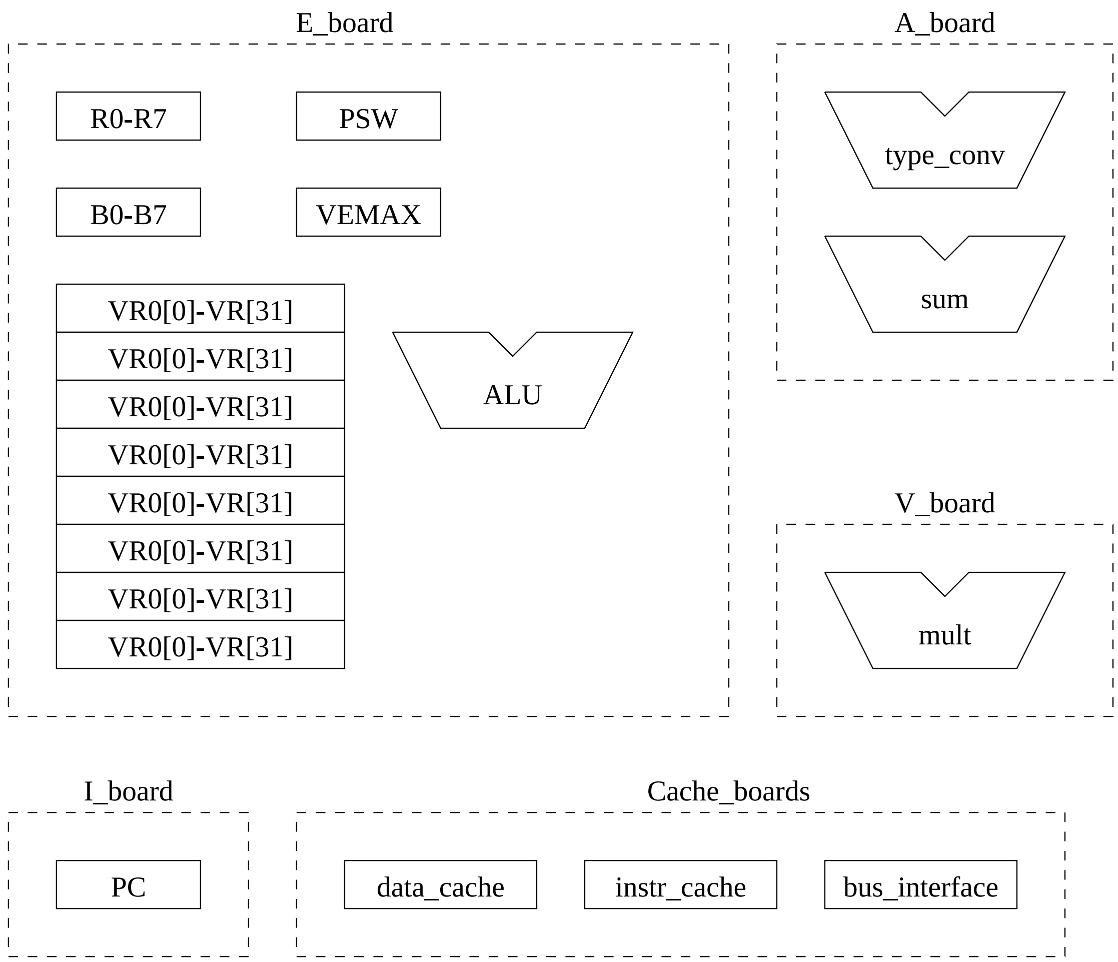

Figure 8.2 The Gould NP-1 CPU

The most obvious feature is that the machine is SIMD excution model, which is divided into the Array Control Unit (ACU) and the Data Parallel Unit (DPU). The ACU all serial and branching instructions. The ACU also stores the program and the serial data in memory located in the unit. All parallel instructions are broadcasted to the processing elements (PEs) in the DPU. The PEs can condtionally execute the broadcasted instructions based on several control bits that reside in each of the PEs. Each PE also has a local set of registers and memory associated with it.

The PEs are composed of three submachines, and a separate control bit controls each submachine in the PE. The three machines are the communication machine, computation machine, and memory machine. In many cases these machines can execute operations concurrently.

Depending on the MP-1, each PE in the machine has either 16 KBytes or 64 KBytes. Therefore, in most programs the data must be distributed across the PEs because none of the PEs have enough local memory. This leads naturally to how does the MP-1 move information from one processing element to another? There are two communication networks on the MP-1: the X-net and the router to move data from one PE to another PE.

The PEs in the array are laid out in 2 dimensional array either as a square or as a rectangle with one dimension twice the size of the other. Each "X" connects four PEs as shown in figure **??**. The X-net wraps around on the edges to form a toridal mesh. The X-net takes advantage of the SIMD nature of the machine; the PEs all communicate in the same direction (one bit every machine cycle), so there are no conflicts on any "X" in the network; each "X" has exactly one pair of PEs communicating.

The other network available in the MP-1 is the router network. This allows any PE to make a connection with any PE. Like the X-net, the network takes advantage of the SIMD nature of the machine. All the PE particpating in the network operation attempt to make circuit with the PE. The conflicts are abitrated and the PEs that fail obtaining a connection must wait until another router network cycle to attempt another connection. Once the PEs connected, the PEs can send data bit serially over the network.

Since the machine obtains its performance by using a large number of low performance PEs, it is imperetive to keep each PE doing useful work for as large a portion of the time as possible. An example of this is the directive of the MP-Fortran [Mas92b], `allbits`, to spread a 1-D array across the two dimensions in the machine. The normal distribution of a parallel array in the Fortran compiler would lay out one dimension horizonally and another dimesion vertically, which in some cases would leave many processor idle. Processors can also be idle when they are disabled due to conditional statments. To minimize the time that PEs are disabled a special optimization, Common Subexpression Induction (CSI) [Die92], was developed. This optimization attempts to move code common to different branches in coditional states outside the condition statement, and avoid excuting the instruction multiple times across the PE array.

To implement 1024 PEs on a single board, the PEs are much simpler than typical serial processors. A single chip contains 32 PEs. Each PE has a 4 bit ALU (a 32-bit ALU in the MP-2 [Mas92a]) and a register set. However, the package for the chip limits the number of external pins, so the 32 PE form two groups of 16 PEs. Each PE group shares a router network interface and an 8-bit memory interface (a 16-bit memory interface in the MP-2). Therefore, caching data in the PE registers to avoid repeated loads and router operations can significantly improve performance.

8.3 Lessons learned

Modeling history dependent resources such as caches is difficult in the current model. The resource modeling does not permit the behavior of the caching algorithm to be encoded in the machine description. In addition if the behaviour was accurately modelled in the description using that information for scheduling and data placement would be difficult.

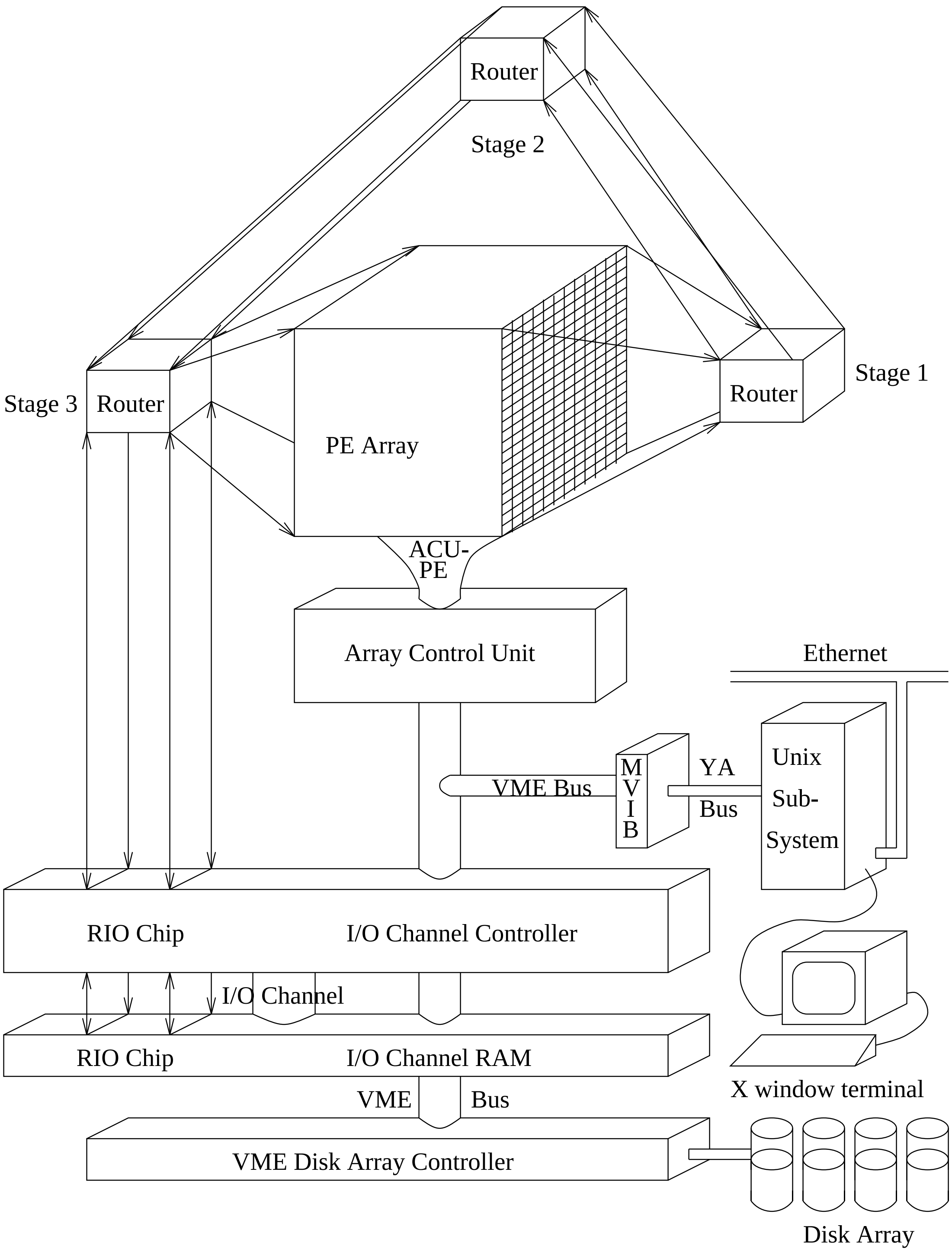

Figure 8.3 The MasPar MP-1 system architecture

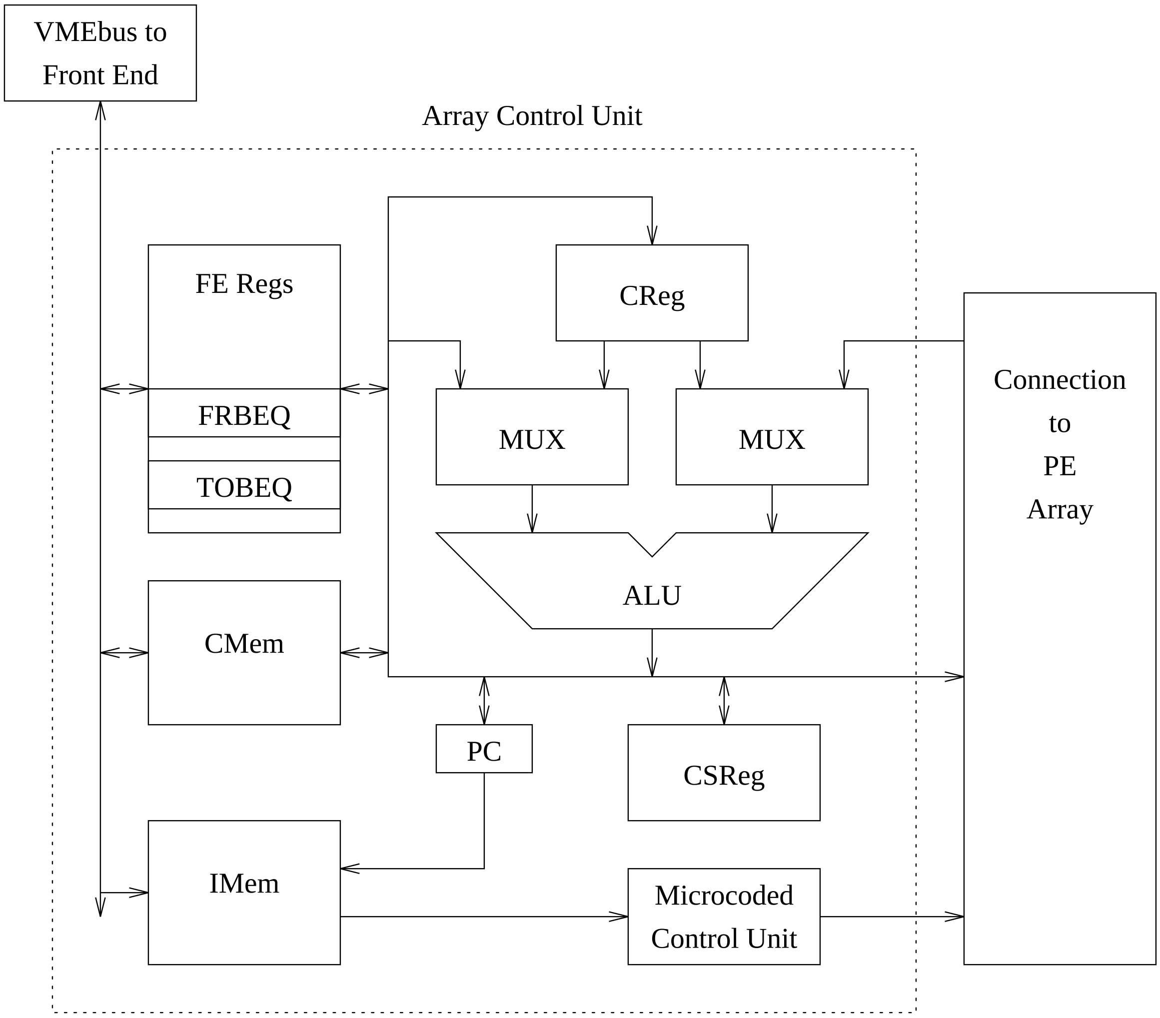

Figure 8.4 The MasPar MP-1 Array Control Unit

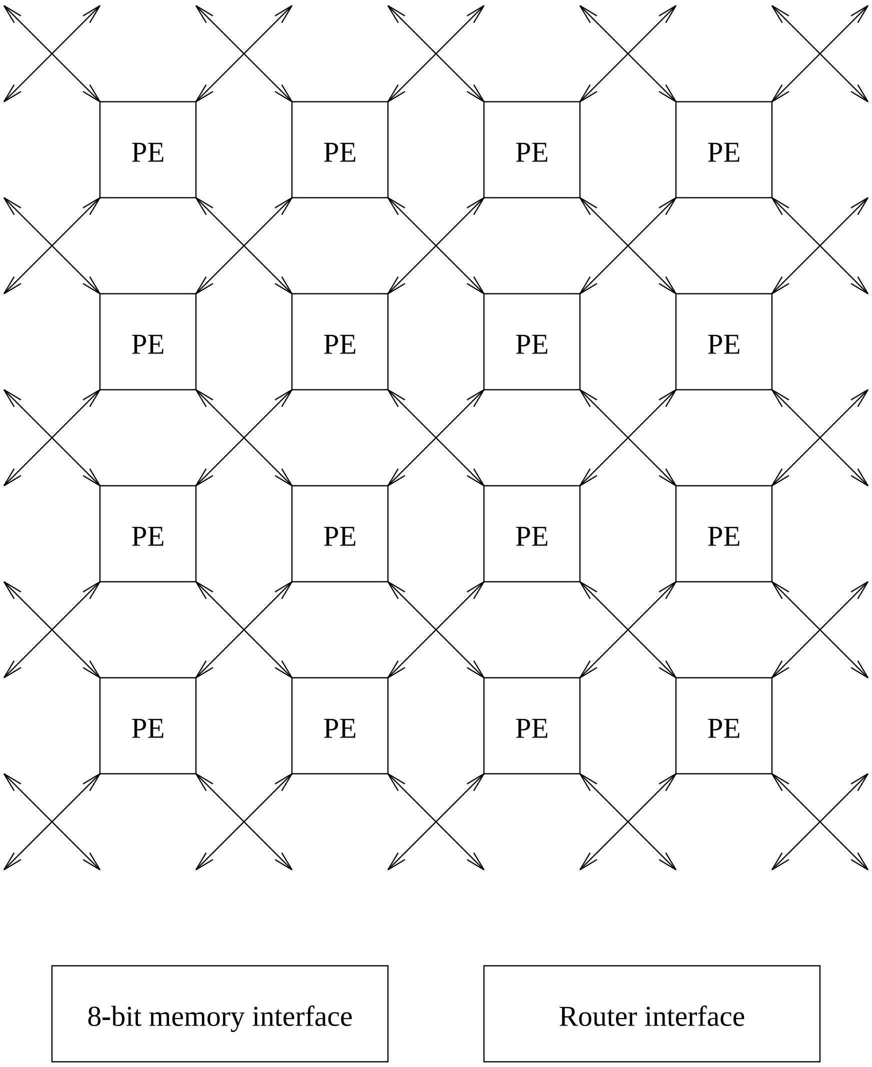

Figure 8.5 The MasPar MP-1 PE cluster

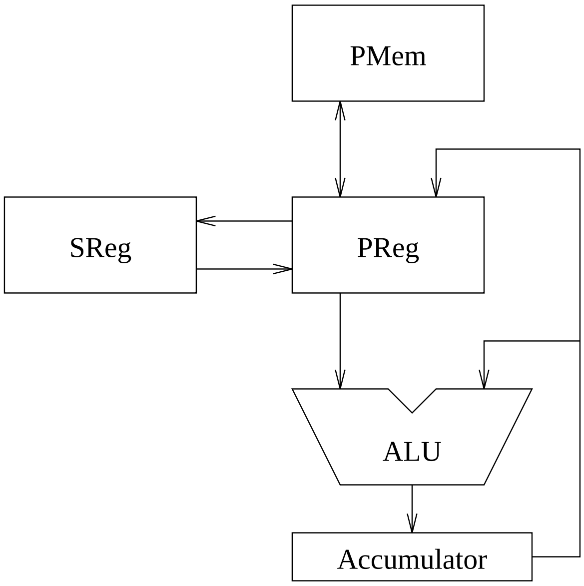

Figure 8.6 The MasPar MP-1 PE

In a similar vein communication operations may be difficult to model, because the communication times may dependent on the source and destination of the data. It is possible to make the distinction between local and non-local operations in the resource usage language.

9. CONCLUSION

Compilers have become an essential part of computer systems because few people write their programs in the native code of the target machine. Thus, the compiler must convert the user's program into reasonably efficient code for the target machine. Constructing powerful compilers that perform the appropriate translations for the target machine requires a great deal of effort, which, using conventional compiler construction methods, must be repeated for each new target machine.

To aid in the construction of compilers, a number of tools have been developed to generate specific components: lexical analyzers, parsers, optimizers, parallelizers, and code generators. Unfortunately, there are key components that existing tools cannot directly produce. Further, only recently have researchers focused on the problems of integrating some components [WS90] [WS91] [Far92]. This dissertation proposes a framework for the automatic construction of complete compilers, with special attention to the problems associated with construction of compilers targeting machine that employ various types of hardware parallelism.

Integrating many components is a relatively simple matter, for example, PCCTS [PDC92] already integrates lexical analyzer and parser construction. However, automating the selection of appropriate transformations to integrate into the compiler requires detailed knowledge of the machine features. Including this information about the target machine in the instruction set description as described in chapter 4 is a significant contribution of this dissertation. From this augmented description of the instruction set, timing estimates for the instructions and aggregate characteristics of the machine can be derived.

The information about the target machine is used to determine which transformations to include in the compiler. Effective compilers cannot simply use the same transformations in the same order for the wide range of target machines now in use; not all transformations are useful for all architectures. To automate the selection and ordering of the transformation, the characteristics of each transformation must be described, e.g. type of machine features the transformation is useful for, information produced by the transformation, information required by the transformation, and parameters to the transformation. With this information, the methods described in chapter 6 select appropriate transformations to include in the compiler and order their application so that good target machine code will be produced.

New computer architectures are still being developed. These new computer architectures may required new transformations to be written to generate efficient code for these architectures. These new transformations can be added easily to the library of transformations, and hence automatically incorporated in other compilers where appropriate using the methods described in chapter 6.

To further the goal of having a complete set of tools to construct compilers, much of the work discussed in this dissertation has been integrated into the public-domain Purdue Compiler Construction Tool Set (PCCTS) [PDC92] as a prototype tool, the Parallel Instruction Generator Generator (PIGG), as described in chapter 7. Future work on automating the process of compiler construction will focus on improving the analysis of the target machine performed by tools and expanding the library of analysis, optimization, and transformation phases.

LIST OF REFERENCES

LIST OF REFERENCES

[Adv90] Advanced Micro Devices. *Am29000 User's Manual*, 1990.

[AG85] A. V. Aho and M. Ganapthi. Efficient Tree Pattern Matching: an Aid to Code Generation. In *Conference Record Twelveth ACM Symposium Principles of Programming Languages*, January 1985.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Expressions with Common Subexpressions. *J. ACM*, 24(1):146–160, January 1977.

[ANS90] Programming Language C. Technical Report ANSI Document Number X3J11/90-013, American National Standard for Information Systems, February 1990.

[Arm89] J. R. Armstrong. *Chip-level Modeling with VHDL*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Arn90] L. F. Arnstein. Describing systems for high level sysnthesis in the VERILOG language. Master's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compiler Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Bac81] J. Backus. PAPER: THE HISTORY OF FORTRAN I, II, AND III. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, New York, 1981.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, et al. The NAS parallel benchmarks–Summary and preliminary results. In *Proceedings of Supercomputing 1991*, pages 158–165, Albuquerque, New Mexico, November 1991.

[BC90] G. E. Blelloch and S. Chatterjee. VCODE: A Data-Parallel Intermediate Language. In *The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, College Park, Maryland, October 1990.

[BCZ92] S. Benkner, B. M. Chapman, and H. P. Zima. Vienna Fortran 90. In *Proceedings of the Scalable High Performance Computing Conference - SHPCC-92*, pages 51–59, Williamsburg, Virgina, April 1992.

[BHE91] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion System for Retargetable Instruction Scheduling. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, Ontario, Canada, June 1991.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 145–156, Toronto, Ontario, Canada, June 1991.

[Bir82] P. L. Bird. An Implementation of a Code Generator Specification Language for Table Driven Code Generators. In *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, pages 44–55, Boston, Massachusetts, June 1982.

[BK77] F. Baskett and T. W. Keller. An Evaluation of the CRAY-1 Computer. In Kuck et al., editors, *High Speed Computer and Algorithm Organization*, pages 71–84. Academic Press, New York, New York, 1977.

[Bla90] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of COMPCON Spring 90 – The Thirty-fifth IEEE Computer Society International Conference*, pages 20–23, San Francisco, California, February 1990.

[Bra91] D. G. Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991.

[BS76] J. Bruno and R. Sethi. Code Generation for a One-Register Machine. *J. ACM*, 23(3):502–510, July 1976.

[Bur93] J. H. Burrill. Using the Gnu C++ Compiler to Generate Code for a Massively Parallel SIMD Processor, April 1993.

[Can92] D. C. Cann. The Optimizing SISAL Complier: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, California, April 1992.

[CDF+93] A. Costa, A. De Gloria, P. Faraboschi, G. Nateri, and M. Olivieri. Asynchronous approach to the RISC design of a micro-controller. *Microprocessing and Microprogramming*, 38(1-5):447–454, September 1993.

[CFN90] Z. Cvetanovic, E. G. Freedman, and C. Nofsinger. Perfect Benchmarks decomposition and performance on VAX multiprocessors. In *Proceedings of Supercomputing 1990*, pages 455–464, November 1990.

[CGS93] C. Comar, F. Gasperoni, and E. Schonberg. The GNAT Project: A GNU-Ada9X Compiler. Working Document, December 1993.

[Cha87] D. R. Chase. An Improvement to Bottom-up Tree Pattern Matching. In *Conference Record Fourteenth ACM Symposium Principles of Programming Languages*, pages 168–177, Munich, West Germany, January 1987.

[Con93] Convex Computer Corporation, Richardson, Texas. *Exemplar Architecture*, first edition, November 1993.

[CP94] P. Clermont and N. Paris. HyperC: Portable Paralle Programming in C. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 682–687, Cancùn, Mexico, April 1994.

[Cra93] Cray Research, Inc. *CRAY T3D System Architecture Overview*, 1993.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide*. Philadelphia, Pennsylvania, 1979.

[DC92] H. G. Dietz and W. E. Cohen. A Massively Parallel MIMD Implemented by SIMD Hardware. Technical Report TR-EE 92-4, Purdue University, West Lafayette, Indiana, January 1992.

[DeR71] F. DeRemer. Simple LR(k) grammars. *Computer Languages*, 1(2):105–119, 1971.

[Die92] H. G. Dietz. Common Subexpression Induction. In *Proceedings of the 1999 International Conference on Parallel Processing*, volume II, pages 174–182, St. Charles, Illinois, August 1992.

[Dig92] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[Dix92] K. M. Dixit. New CPU benchmark suite from SPEC. In *Proceedings of COMPCON Spring 92 – The Thirty-seven IEEE Computer Society International Conference*, pages 305–310, San Francisco, California, February 1992.

[Doc73] V. T. Dock. *COBOL: American National Standard.* Reston Pub. Co., Reston, Virgina, 1973.

[DoD83] American National Standard reference manual for the Ada programming language / sponsor: United States Department of Defense. Technical Report ANSI/MIL-STD-1815A-1983, American National Standards Institute, February 1983.

[Don92] J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. *Computer Architecture News*, 20(3):22–44, June 1992.

[DP90] H. Dobler and K. Pirklbauer. Coco-2 A New Compiler Compiler. *ACM SIGPLAN Notices*, 25(5):82–90, May 1990.

[EN69] G. W. Ernst and A. Newell. *GPS: A Case Study in Genrality and Problem Solving.* Academic Press, New York, 1969.

[ESL89] H. Emmelman, F. Schr̈oer, and R. Landwehr. BEG — A generator for effiecient back ends. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 227–237, Portland, Oregon, July 1989.

[Ett83] D. M. Etter. *Structured Fortran 77 for engineers and scientists.* Benjamin/Commings Publishing Company, Inc., Menlo Park, California, 1983.

[Far92] C. Farnum. Dora — An Environment for Experimenting with Compiler Optimizers. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 86–95, Oakland, California, April 1992.

[FH91] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. *ACM SIGPLAN Notices*, pages 29–43, October 1991.

[FHP92] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

[Fre92] Free Software Foundation, Cambridge, Massachusetts. *Bison, The YACC-compatible Parser Generator*, 1.20 edition, December 1992.

[GFK+91] C. Gleason, M. Forsyth, C. Kohlhardt, S. Mangelsdorf, B. Arnold, and R. Luebs. CMOS processor circuit design in Hewlett-Packard's series 700 workstations. In *Proceedings of the 1991 IEEE International Conference on Computer Design - VLSI in Computers and Processors - ICCD*, pages 288–292, Los Alamitos, California, USA, October 1991.

[Gla77] R. S. Glanville. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers.* PhD thesis, University of California, Berkeley, California, December 1977.

[Gou87] Gould, Fort Lauderdale, Florida. *Gould NP1 Central Processing Unit (CPU) Model 4020 Reference Manual*, September 1987.

[Gou88] Gould, Fort Lauderdale, Florida. *NPL Service Reference Guide*, February 1988.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

[HC86] P. J. Hatcher and T. W. Christopher. High-Quality Code Generation Via Bottom-Up Pattern Matching. In *Conference Record Thirteenth ACM Symposium Principles of Programming Languages*, 1986.

[HD87] R. R. Henry and P. C. Damron. Code Generation Using Tree Pattern Matchers. Technical Report Technical report 87-02-04, University of Washington, Seattle, Washington, February 1987.

[HDE+92] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Conference Record, 5th Workshop on Languages and Compilers for Parallel Computing*, pages 261–276, Stamford, Conneticut, August 1992.

[HDE+93] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, chapter 26, pages 406–420. Springer-Verlag, Berlin, 1993.

[Hen84] R. R. Henry. *Graham-Glanville Code Generators.* PhD thesis, Computer Science Division University of California, Berkeley, California, May 1984.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An Overview of the Fortran D Programming System. In *Preliminary Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, pages e1–e17, Santa Clara, California, August 1991. Intel.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[HT72] R. G. Hintz and D. P. Tate. Control Data STAR-100 Processor Design. In *COMPcon Proceedings*, pages 1–4, September 1972.

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[Int86] Intel, Santa Clara, California. *80386 Programmer's Reference Manual*, 1986.

[Int89] Intermetrics. VHDL Language Reference Card, August 1989.

[Int92] Paragon Supercomputers, 1992. Order No. 203/6/92/10K/GA.

[Joh75] S. C. Johnson. Yacc — Yet Another Compiler-Compiler. Technical Report Compuster Sciecne Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[Joh78] S. C. Johnson. A Portable Compiler: Theory and Practice. In *Conference Record Fifth ACM Symposium Principles of Programming Languages*, pages 97–104, Tucson, Arizona, January 1978.

[KA82] D. W. Krumme and D. H. Ackley. A Practical Method for Code Generation Based on Exhaustive Search. In *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, pages 185–196, Boston, Massachusetts, June 1982.

[Knu65] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[KS93] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Proceedings of COMPCON Spring 93 – The Thirty-eighth IEEE Computer Society International Conference*, pages 176–182, San Francisco, California, February 1993.

[Les75] M. E. Lesk. LEX — a lexical analyzer generator. Technical Report Computer Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[Lin82] N. R. Lincoln. Technology and Design Trade Offs in the Creation of a Modern Supercomputer. *IEEE Transactions on Computing*, C-31(5):363–376, May 1982.

[LLBR75] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir — A Programming Language for the Illiac IV. *Communications of the ACM*, 18(3):157–164, March 1975.

[Mas90] MasPar Computer Corporation. *MasPar Assembly Languaage (MPAS) Reference Manual*, April 1990.

[Mas92a] MasPar. The Design of the MasPar MP-2: A Cost Effective Massively Parallel Computer. Technical Report MP/P-11.92, MasPar Computer Corportation, Sunnyvale, California, March 1992.

[Mas92b] MasPar Computer Corporation. *MasPar Fortran Reference Manual, Software Version 3.0*, July 1992.

[Mas92c] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 3.0*, July 1992.

[Men93] Mentor Graphics, Wilsonville, Oregon. *Getting Started with Design Architect Training Workbook, Software Version 8.2*, 1993.

[Mos80] Mostek, Carrollton, Texas. *Memory Data Book and Designers Guid*, June 1980.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal, Streams and Iteration in a Single Assignment Language Reference Manual, Version 1.2. Technical Report M-146 Rev. 1, Lawrence Livermore National Laboratory, Livermore, California, March 1985.

[NAJ+81] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. The Pascal P-code compiler: Implementation notes. In D. W. Barron, editor, *Pascal — The language and its implementation*. Wiley, New York, 1981.

[Nau63] P. Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[OPE+93] M. O'Keefe, T. Parr, B. K. Edgar, S. Anderson, P. Woodward, and H. Dietz. The Fortran-P Translator: Automatic Translation of Fortran 77 Programs for Massively Parallel Processors. Technical Report Preprint No. 93-021, Army High Performance Computing Research Center, February 1993.

[Par93] T. J. Parr. *Obtaining Practical Variants of LL(k) and LR(k) for $k > 1$ by Splitting the Atomic k-Tuple*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1993.

[Par94] T. J. Parr. *The SORCERER 1.00B Reference Manual*, February 1994.

[Pax90] V. Paxson. *Using flex — A Fast Lexical Analyzer*. The Regents of the University of California, May 1990.

[PDC91] T. Parr, H. Dietz, and W. Cohen. PCCTS Reference Manual, Version 1.0. Technical Report TR-EE 90-14, Purdue University, West Lafayette, Indiana, August 1991.

[PDC92] T. Parr, H. Dietz, and W. Cohen. PCCTS Reference Manual, Version 1.0. *ACM SIGPLAN Notices*, 27(2):88–165, February 1992.

[PGH+90] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 423–453. The MIT Press, Cambridge, Massachusetts, 1990.

[Phi89] M. J. Phillip. Unification of Synchronous and Asynchronous Models for Parallel Programming Languages. Master's thesis, Purdue Univesity, West Lafayette, Indiana, May 1989.

[QHS91] M. Quinn, P. Hatcher, and B. Seevers. Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor. In A. Nicolau and D. Padua, editors, *Advances in languages and compilers for parallel processing*, pages 385–401. The MIT Press, Cambridge, Massachusetts, 1991.

[RS70] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, 1970.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, pages 63–72, January 1978.

[Sab92] G. Sabot. A Compiler for a Massively Parallel Distributed Memory MIMD Computer. In *Proceedings of Frontiers Massively Parallel Computation*, pages 12–20, McLean, Virginia, October 1992.

[Sam81] S. E. Sammet. PAPER: THE EARLY HISTORY OF COBOL. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 199–243. Academic Press, New York, 1981.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, Massachusetts, 1989.

[Sar91] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5/6):779–804, September/November 1991.

[SG85] S. Skedzielwski and J. Glauert. IF1 An Intermediate Form for Applicative Languages, Version 1.0. Technical Report M-170, Lawrence Livermore National Laboratory, Livermore, California, July 1985.

[SG93] G. L. Steele, Jr. and R. P. Gabriel. The Evolution of Lisp. In *History of Programming Languages Conference*, pages 231–270, Cambridge, Massachusetts, April 1993.

[SJ93] P. M. Siegel and J. Jordan. Scalably Parallel Computing at the Theory Center: an IBM Partnership. *Forefronts*, 8(3), Winter 1993.

[Sky93] SKYlight. SKY Computers, Inc., May-June 1993.

[SSA+94] C. B. Stunkel, D. G. Shea, B. Abali, et al. Architecture and Implementation of Vulcan. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 268–274, Cancùn, Mexico, April 1994.

[SSKD87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV. An Overview of the PASM Parallel Processing System. In D. D. Gajski, V. M. Milatinovic, H. J. Siegel, and B. P. Furht, editors, *Tutorial: Computer Architectures*, pages 387–407. IEEE Computer Society, Washington DC, 1987.

[Sta89] R. M. Stallman. *Using and Porting GNU CC for version 1.35.* Free Software Foundation, April 1989.

[Sta93] R. M. Stallman. *Using and Porting GNU CC for version 2.5.* Free Software Foundation, October 1993.

[TH92] S. Tjiang and J. Hennessy. Sharlit — A tool for building optimizers. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, California, June 1992.

[Thi89a] Thinking Machines Corporation. *CM Fortran Reference Manual, Version 5.2-0.6.* Thinking Machines Corporation, Cambridge, Massachusetts, September 1989.

[Thi89b] Thinking Machines Corporation. *Introduction to Programmin in C/Paris.* Thinking Machines Corporation, Cambridge, Massachusetts, June 1989.

[Thi90] Thinking Machines Corporation. Connection Machine Technical Summary, November 1990.

[Thi92] Thinking Machines Corporation. Connection Machine CM-5 Technical Summary, November 1992.

[Tho67] J. F. Thorlin. Code generation for PIE (parallel instruction execution) computers. In *AFIPS Conference Proceedings*, volume 30, pages 641–643, Atlantic City, New Jersey, April 1967.

[Tho90] D.E. Thomas. Verilog: A Tutorial Introduction, 1990.

[Tji86] S. Tjiang. Twig Reference Manual, January 1986.

[van91] R A. van de Geijn. MASSIVELY PARALLEL LINPACK BENCHMARK ON THE INTEL TOUCHSTONE DELTA AND iPSC/860 SYSTEMS. Technical Report CS tr91-28, University of Texas at Austin, Austin, Texas, 1991.

[Wan94] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 73–84, June 1994.

[Web83] Webster's ninth new collegiate dictionary, 1983.

[Wen90] A. L. Wendt. Fast Code Generation Using Automatically-Generated Decision Trees. In *Proceddings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 9–15, White Plains, New York, June 1990.

[Wir93] N. Wirth. Recollections about the Development of Pascal. In *History of Programming Languages Conference*, pages 333–342, Cambridge, Massachusetts, April 1993.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, October 1982.

[WS90] D. Whitfield and M. L. Soffa. An Approach to Ordering Optimizing Transformations. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, Seattle, Washington, March 1990.

[WS91] D. Whitfield and M. L. Soffa. Automatic Generation of Global Optimizers. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 120–129, Toronto, Ontario, Canada, June 1991.

[Zos93] M. E. Zosel. High Performance Fortran: An Overview. In *Proceedings of COMPCON Spring 93 – The Thirty-eighth IEEE Computer Society International Conference*, pages 132–136, San Francisco, California, February 1993.

APPENDICES

```
md : ( HEADER )* hardware instr "@"
   ;

hardware
    : ( hrules | mapping )*
    ;

hrules
    : resource
    | machine
    | pipeline
    | registers
    | memory
    | reserve
    ;

block
    : ( number | ) "\(" ( block )* "\)"
    | ident
    ;

pipeline : PIPELINE ident number number ;

registers
    : REGISTER ident number number STRING ( STRING ( number | ) )+
    ;

memory
    : MEMORY ident number number STRING ( STRING ( number | ) )+
    ;

resource
    : RESOURCE ident block { CACTION cycles ACTION_END }
    ;

reserve : RESERVE r_tree ;

r_tree
    : ident ( "\[" r_index ( "," r_index )* "\]" | ) { "." r_tree }
    ;
```

```
r_index
    : ident
    | STRING
    | number
    ;

machine
    : MACHINE ident block { CACTION cycles ACTION_END }
    ;

ident
    : NAME
    | TERMINAL
    ;

mapping
    : MAPPING ident "\(" { one_map ( "," one_map )* } "\)"
    ;

one_map : TERMINAL MAP STRING ;

instr : ( rule )* ;

rule : NAME ":" product ( "\|" product )* ";" ;

product : cost_action gen_order pattern gen_action ;

cost_action
    :  ACTION
    | CACTION cycles ACTION_END
    |
    ;

cycles : cycle ( ";" cycle )* ;

cycle
    : depends modifier r_tree ( "\*" | )
        ( "," modifier r_tree ( "\*" | ) )*
    ;

depends
    : ( number ( "," number )* | ) "\|"
    |
```

```
    ;

modifier
    : LESS
    | MORE
    | LESS_MORE
    |
    ;

gen_action
    : ACTION
    | SACTION
    |
    ;

gen_order
    : "!"
    |
    ;

pattern : tree ;

tree : "\(" ident { els } "\)" ;

els : ( ( ident | tree ) { "\*" } )+ ;

number : DEC_NUM ;

lib_des
    : ( lib_entry )* "@"
    ;

lib_entry
    : "phase:" ATT_NAME ";"
         ( p_cost
         | p_ben
         | p_parm
         | p_disabling
         | p_uses
         | p_kills
         | p_defines )*
    ;
```

```
p_ben : "benefit:" p_expr ";"
    ;

p_cost : "cost:" p_expr ";"
    ;

p_parm : "parameters:" PARMS
    ;

p_disabling : "disabling:" p_name_list ";"
    ;

p_uses : "uses:" p_name_list ";"
    ;

p_kills : "kills:" p_name_list ";"
    ;

p_defines : "defines:" p_name_list ";"
    ;

p_name_list
    : ATT_NAME ( "," ATT_NAME )*
    |
    ;

p_expr
    : ATT_NAME
    | DEC_NUM
    | "\-" DEC_NUM
    ;
```

Appendix B: C-lite Syntax

```
prog : decls funcs "@" ;

decls : ( decl )* ;

decl : stc WORD { "\[" NUM "\]" } ";" ;

stc    : Mono
       | Poly
       ;

funcs : ( func )* ;

func : word "\(" word "\)" block ;

word : WORD ;

slist : slist1 ;

slist1 : ( stat )* ;

block : "\{" decls slist "\}" ;

stat : array Assign cond ";"
       | If cond stat { Else stat }
       | While cond stat
       | Ret cond ";"
       | Print cond ";"
       | Wait ";"
       | block
       | ";"
       ;

cond : expr { ">" expr | LT expr } ;

expr : term { Add term | Subtract term } ;

term : fact { Mul fact | Div fact | Mod fact } ;

fact : num
       | array ( "\(" cond "\)" | )
       | Subtract fact
```

```
        | Not fact
        | "\(" cond "\)"
        ;

num : IConst ;

array : WORD { "\[" cond "\]" } ;
```

Appendix C: Fortran P Syntax

```
program : ( decl )* ( subroutine )* "@" ;

subroutine : subdef ;

subdef
    : ( Subr | Function | Program )
          WORD { "\(" arglist "\)" } ( decl )* ( slist ) End
    ;

arglist : ( WORD ( "," WORD )* )* ;

decl
    : Dimension WORD "\(" expr_list "\)"
         ( "," WORD "\(" expr_list "\)" )*
    | param
    | RealType varlist
    | IntegerType varlist
    | DoubleType varlist
    | ComplexType varlist
    | LogicalType varlist
    | CharacterType varlist
    | Common "/" WORD "/" comlist
    ;

expr_list : expr ( "," expr )* ;

varlist
    : WORD ( "\(" expr_list "\)" | )
         ( "," WORD ( "\(" expr_list "\)" | ) )*
    ;

comlist
    : WORD ( "\(" expr_list "\)" | )
         ( "," WORD ( "\(" expr_list "\)" | ) )*
    ;

statement : { IConst } stat0 ;

stat0
    : WORD { "\(" ( expr ( "," expr )* )* "\)" } Assign expr
    | Do IConst { "," } WORD Assign expr "," expr ( "," expr | )
```

```
    | If "\(" expr "\)" ( ifstat Endif | stat0 )
    | Goto IConst
    | Call WORD { "\(" { expr ( "," expr )* } "\)" }
    | Ret
    | Continue
    | Stop
    | Cdir WORD
    ;

param : Param "\(" WORD Assign expr ( "," WORD Assign expr )* "\)" ;

ifstat : Then slist ( Elseif "\(" expr "\)" ifstat | Else slist ) ;

slist : ( statement )* ;

expr : expr0 ( ( NEQV | EQV ) expr0 )* ;

expr0 : expr1 ( Or expr1 )* ;

expr1 : expr2 ( And expr2 )* ;

expr2
    : Not expr2
    | expr3
    ;

expr3
    : expr3a ( ( LT | LE | EQ | NE | GT | GE ) expr3a )*
    ;

expr3a: expr4 { RangeOp expr4 } ;

expr4 : expr5 ( ( Add | Sub ) expr5 )* ;

expr5 : expr6 ( ( Mul | Div ) expr6 )* ;

expr6
    : Minus expr6
    | expr7
    ;

expr7 : expr8 { Exp expr7 } ;
```

```
expr8
    : FConst
    | IConst
    | LConst
    | WORD { "\(" expr ( "," expr )* "\)" }
    | "\(" expr "\)"
    ;
```

VITA

VITA

William Cohen was born to Dr. and Mrs. Cohen, August 2, 1965 in Topeka, Kansas. In his spare time while a student of Topeka's Unified School District Number 501, he learned the joys of taking things apart. More importantly, he learned the importance of understanding how to put them back together so they still functioned. Much of this examination of things mechanical was encouraged by his parents by not punishing him for taking things apart (as long as he put them back together), making many trips to the library and museums, getting subscriptions to magazines such as Popular Mechanics and Popular Science, and buying books such as The Way Things Work.

Cohen pursued a bachelor's degree in electrical engineering at the University of Kansas and obtained the degree in May 1987. Deciding to learn more about electrical engineering, he started the master's program in electrical engineering at Purdue University in August 1987. During the masters program, he began studying compilers; by the completion of his masters, he decided to pursue a Ph.D. in electrical engineering, and graduated in December 1994.

While a graduate student at Purdue, he taught sophmore electrical engineers digital logic, retargetted the Free Software Foundation Gnu C Compiler to the Gould NP1 in the engineering computer network, and developed compiler tools software which is in use by over 1,000 people.

Some friends of his noted that he still took things apart to see how they work.