AN ABSTRACT OF THE THESIS OF

Karen Meyer-Arendt for the degree of Master of Science in

Computer Science presented on May 29, 1987.

Title: Parallel Code Generation via Scheduling

Abstract approved: Redacted for Privacy

A translator has been designed and implemented which generates parallel code for a long instruction word parallel computer with local memories. Its main methods are to translate the sequential source code into single assignment, two-operand form, and to then assign the operations to processors so as to minimize the number of resulting long instruction words. The scheduling of an operation can be myopic, or it can look ahead at the consequences of scheduling on the next several operations. The system can generate optimal code if all subsequent operations are included in the lookahead. For computational reasons, the operational system is limited to a fixed number of lookahead steps.

Parallel Code Generation via Scheduling

by

Karen Meyer-Arendt

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed May 29, 1987

Commencement June 1988

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented May 29, 1987

Typed by Karen Meyer-Arendt for Karen Meyer-Arendt

ACKNOWLEDGMENTS

Many thanks go, above all, to Dr. Don Chiarulli, for his encouragement, and countless comments and suggestions.

Thanks also go to Dr. Walter Rudd, Dr. David Sandberg, Dr. Ted Lewis, and Dr. Curtis Cook, for their ideas and advice.

Thanks to my friends who helped me with this thesis: Jim Holloway, Russell Ruby, Bradley Babin, and Ritchey Ruff.

And finally... Danke, Eltern!

TABLE OF CONTENTS

LIST OF FIGURES

PARALLEL CODE GENERATION VIA SCHEDULING

INTRODUCTION

This thesis is a study in the automatic detection of parallelism in sequential programs and the translation from sequential into parallel representations. The motivation for this study is the need to create a high-level language for the Dynamically Reconfigurable Architecture for Factoring Things (DRAFT), a local memory, single sequencer, parallel computer (Chiarulli, Rudd, and Buell 1984, Chiarulli 1986). This document is an examination of the issues involved in such a translation effort.

Why detect parallelism automatically? The alternative would be to utilize or design a high-level language which includes explicit parallel programming constructs. There are several disadvantages to using such parallel constructs. First of all, programming is a complicated and error-prone activity. Requiring the programmer to keep track of several concurrent flows of control adds significant complexity. Second, there is always an obstinacy by the computing public against new language constructs. Several such constructs for parallel programs have been tried yet none have gained wide acceptance. Thirdly, there are reams of programs written in conventional high-level languages, which it would be most useful and cost-efficient to machine-translate into parallel versions, rather than have to reprogram using such new constructs.

The problem is to analyze a sequential program to attempt to find instructions or groups of instructions which can be executed concurrently without loss of semantic equivalence. Now a sequential program is no more than a number of textually ordered statements which are embedded in control structures. The key to parallelization is identifying those groups of statements which must be executed in strict sequential order. Textually adjacent statements which have no common operands and are in a common control structure are likely candidates for being concurrently executable. Exactly how statements will be recognized to be concurrently executable, and how to maintain the temporal dependencies between those statements which are not, is the subject of this study.

This design has been implemented as a LISP program which transforms a standard intermediate code representation (an abstract syntax tree) of a sequential program, to a second intermediate code representation of a parallel version of that same program. The parallel intermediate code representation has been designed to be similar to the microcode format of the DRAFT. Though the parallel code representation has been designed to be independent of many of the low-level memory management details (and more notably from the DRAFT's reconfigurability features), it has maintained many of its essential features such as a single flow of control, and the varying number of arithmetic logic unit (ALU) processors, the local memories for each of the ALU processors and, in particular, includes many of the same instructions as are found in the DRAFT microcode.

This thesis will explore the issues of automatic detection of parallelism as follows. Chapter One will provide a classification of various hardware and software paradigms relevant to program parallelism and discuss the central issues in the detection of parallelism. Chapter Two will describe this author's solution to the problem, discuss the current implementation of the parallel code generator, and the proposed implementation of a full-fledged parallelizing compiler. Chapter Three will discuss the issue of validating the performance of the resultant parallel code and provide some performance figures for the current implementation. Finally, Chapter Four will examine the relevance and performance of this implementation in the larger context of other recent work in parallel programming.

CHAPTER ONE: ISSUES IN PARALLEL PROCESSING

Parallel computing encompasses everything from dedicated processors, to pipelined methods of computation, to collections of numerous equally powerful processors. It describes, in short, any computer that has more than one processor and, hopefully, results in faster throughput than would be possible with a single processor. But such a definition is so broad as to be useless since most current computers use special input/output processors, and many use multiple processors for efficient and fault-tolerant multiprocessing. For this thesis, parallel computing will be restricted to a computer architecture dedicated to the execution of a single program in parallel, that is, where parts of the program can be executed concurrently on two or more processors. How the processors are to be physically or logically linked, how the program is to be partitioned, and with what granularity is the subject of this chapter. Issues will be discussed in general, with special regard to the DRAFT architecture. In addition, the approaches of other researchers to the problem of automatically partitioning programs will also be examined.

Hardware Issues

The processors used in a parallel architecture can be of varying or equal abilities, where the latter is probably the more common. From a scheduling standpoint, this implies that any program fragment can execute on any processor. An alternative is to have one processor act as a controller to intervene when there is contention between multiple processors for some resource such as shared memory. When memory is local rather than shared, then the controller might be in charge of the communications between the processors. The DRAFT architecture is an example of an architecture in which ultimate control is given to a single controller. In it multiple arithmetic logical units (ALU) processors are connected in a ring and are able to execute ALU instructions only. The controller, called the sequencer, executes only control instructions and never ALU instructions (see Figure 1). DRAFT computers are programmed by use of long instruction words Each machine instruction is long (256 bits or more) and consists of an instruction to the sequencer and multiple instructions to each of the ALU processors, all packed into a single long instruction word. The machine code is like a normal sequential program with jumps and loops,

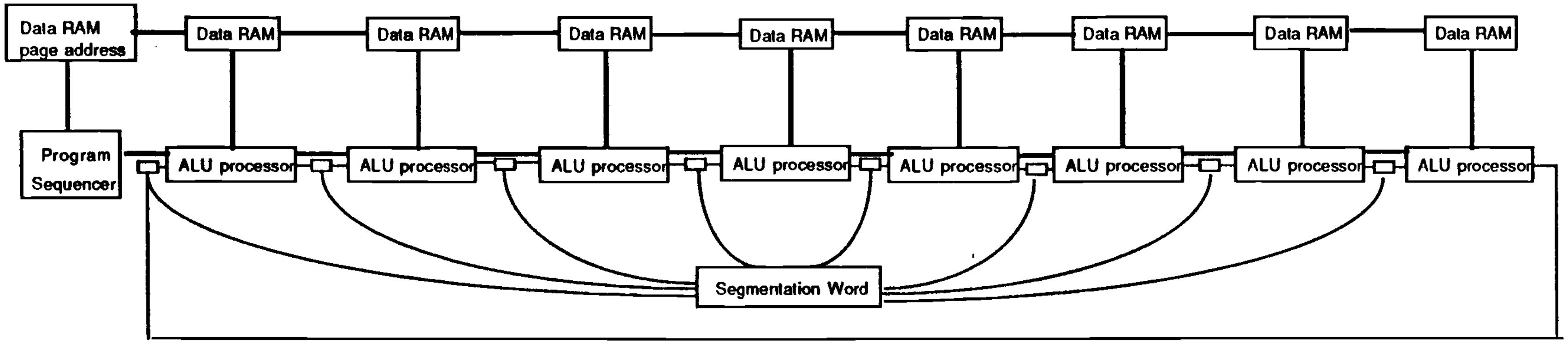

Figure 1: DRAFT Architecture

but within each single long instruction word will be 32 bit instructions which will direct the subordinate processors to execute in parallel. There is thus parallel computing power at the ALU level in combination with a single flow of control.

The issue of shared versus local memories deserves further mention. With a shared memory scheme there is always the danger that two processors will access the same memory cell at the same time, for instance, to both write a value, or one write and one read simultaneously. This can result in unpredictable values being read by this or a subsequent access to that memory cell. To solve this problem, most systems require that locks or semaphores be supported by the software, in order to "lock" out another processor when one needs protected access to it.

Parallel hardware designs using local memories, while not suffering the simultaneous access problem, encounter problems when it becomes necessary to transfer data between local memories. It is conceivable that programs could be so constructed as to have entirely independent clusters of variables. These variables would be accessed by one processor only and would never need to be transferred to another local memory. More likely is a number of variables stored in different local memories, many of which interact with only a few other variables, and some of which interact with many other variables. The hardware must support a data transfer mechanism between the registers or local memories of various processors. Hardware designers have specified many possible designs, ranging from completely connected systems to a sparse, nearest-neighbor interconnection system. One such sparse interconnection scheme has been used in the DRAFT architecture: ALU processors are connected in a ring-like manner with data being readable only from a processor's own local memory or that of its left neighbor, and the leftmost ALU can read from the rightmost local memory. The programmer can still attempt to simultaneously write to and read from one memory cell, but such errors can be detected by the microassembler. This ring-like interconnection of processors and their local memories will be the basis for automatically generating parallel code.

Software Issues

In order to execute a single program on multiple processors the program must first be partitioned into units which can be concurrently executed. In partitioning the program there are two major options: to partition at the instruction level, called fine-grain parallelism, or to partition into larger execution units which will typically be related groups of instructions such as procedures or functions. The latter, called large-grain or coarse-grain parallelism, is most suitable when there is a high degree of overhead, such as would be required with a complex memory access or interprocess communication system. If, on the other hand, overhead is low, then parallelism might as well be at the instruction level, since it is conceptually simpler to analyze the dependencies between operands when there are fewer operands involved and the operations take a known time to execute.

Regardless of the granularity of the program units, it must be decided whether those units are to be asynchronously or synchronously executed. Asynchronous execution implies that a processor can execute some assigned code to completion, then report back to another processor, particularly when that other processor is dependent on the output from the first. It is quite possible that many processors will spend their time sitting idle while waiting for such notifications. The notifications need to follow a particular protocol; typically such signals are handled by interrupts or by writing a message to a common memory location which the waiting processor must regularly check.

The synchronous model requires no communication between the processors since all execution times are enforced by a predetermined schedule. One example of a synchronous model is the pipeline architecture whereby a processor will process some data, then pass the processed data on while synchronously receiving new data. A different kind of example of synchronization is found in the long instruction word based architectures such as the DRAFT. In such architectures all ALU processors execute the instruction packed in the next long instruction word simultaneously, and there is no other relation between the instructions which the processors are executing.

The problem of one processor needing to wait for another is, however, by no means solved by using the long instruction word model. Then as before there might be a processor waiting for output from another processor, only that the waiting processor would in this case have to execute a no-operation (NOP) instruction, that is, one which does nothing at all. Because of the synchronization it will be necessary to know in advance how long each part of the program will take to execute. What this implies is twofold: that parts of the program can be assigned to processors using standard scheduling algorithms, and that such scheduling can occur at compile-time.

A third and fundamental issue in parallel software design is whether the parallelism should be explicitly programmed by the programmer by means of explicit parallel constructs, or whether parallelism between the program units can be detected automatically. The argument for the former is that only explicit constructs will provide adequate control over producing efficient parallel code. Many such explicit parallel constructs exist, for example, COBEGIN which is an organizing construct. By using it the programmer is declaring that all statements within the scope of the COBEGIN are concurrently executable. It is possible that the programmer has made a syntactic or semantic error in declaring such concurrence, and debugging a program in which there are multiple streams of instructions is difficult at best. The obvious question, is whether it is possible for the parallelism to be detected automatically, and if so, with what efficiency.

Related Research in the Automatic Detection of Parallelism

Once it has been decided that the parallelism should be detected instead of explicitly declared, it must be decided whether that detection should occur at compile-time or at run-time. Several high-speed computers, for instance the CDC-6600, use a limited form of lookahead during run-time to look for groups of instructions which might be concurrently executable by special purpose processors. While more might be known at run-time, data dependent relationships for example, run-time detection of parallelism can substantially delay execution time. For this reason compile-time analysis of the instructions is preferable. In the next section we will look at some of the fundamental issues crucial to the detection of parallelism at the instruction level. Most of these issues are also directly applicable to coarse-grained parallelism.

Central to all forms of parallelism is the limiting role played by data dependencies. If one statement operates on the output of another, there is nothing it can do but wait for that output. For example, given two assignment statements A:=1; B:=A, there is no escaping the fact that B:=A can be executed only after A:=1 is.

The first formal analysis of data dependencies was done by D. J. Kuck of the University of Illinois in Urbana-Champaign in the 1970s (cited in Kuck, Kuhn, and Padua 1981, and Kuck 1982). He asserts that there are three essential kinds of data dependencies: flow dependencies, anti-dependencies, and output dependencies. The example above demonstrates flow dependency, which says that you cannot access the value of a variable until after the variable has been given a value. This dependency is also called read-after-write dependency. Anti-dependency, also referred to as write-after-read dependency, is exemplified by the statements A:=B; B:=2. It is clear that the assignment to B cannot occur until after its prior value has been assigned to A. The third dependency type, output dependency, also known as write-after-write dependency, is a combination of the two, and is exhibited in the statements A:=0; B:=A; A:=1. In this example, the second assignment to A must succeed the first, in order that B obtains the correct value of A.

A fourth dependency proposed by Kuck is a control dependency, an example of which is given in the statement: IF A=0 THEN A:=1 ELSE A:=2. In this statement there is no advantage to executing any of the IF, THEN, or ELSE clauses concurrently. Neither the THEN nor the ELSE clause can execute concurrently with the IF clause, or with each other, since the selection of which clause is to execute is determined only after the IF clause has been executed; to attempt to execute both the THEN and ELSE clause concurrently will require that the results of executing one of those clauses will need to be undone.

Similar problems occur with a looping structure. The constraint there is that the statements in the scope of the loop must remain within the scope of the loop under parallelization. This suggests that either the entire loop construct must be assigned to a single processor, or that some or all processors be subjected to the control of the loop. The latter is consistent with the long instruction word programming paradigm presented earlier since it allows only one flow of control.

Because of the control structure constraints on the program, it is likely that parallelization can only occur with the "basic blocks" circumscribed by those control structures. A basic block is defined to be "a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end" (Aho, Sethi, and Ullman 1986). Most efforts have concentrated on the parallelization of statements within basic blocks only, since those blocks appear to impose a boundary beyond which it is not possible to execute instructions in parallel. Of course there might be attempts to execute two or more control structures simultaneously. This will work only when there are no data dependencies between them, and when they have a similar enough structure, that is, number of loops or kind of exit condition, to warrant such simultaneous execution. This process is called "loop fusion" (Kuck 1981).

The difficulty with all of these dependencies is that they encode primarily local information about the operations. This may be insufficient to result in significant run-time speedup. More than just whether two operations can execute concurrently will be the interactions between all of the operations in the program. The global interactions in the program will be the deciding factor in how much parallelization can be achieved. Again, it is Kuck who had studied this by looking for general, and more global, algorithmic patterns in order to attempt to restructure at a very high level the program into another which shows more promise for being highly parallelizable (Lee, Kruskal, and Kuck 1985). Certainly this approach is ambitious and commendable, but also highly complex based on the countless number of ways in which algorithms can be constructed. Kuck has categorized several basic high level patterns, particularly in loop structures. In several experiments, recognition of those patterns in standard non-numerical algorithms has resulted in modest speedups. About one fourth of the test programs achieved "good" speedup, which Kuck describes as about 0.3 of possible speedup linear to the number of processors. When the automatic methods of restructuring programs failed, Kuck had to rely on explicit input from the programmer in order to get a truly satisfactory level of parallelism.

Kuck's approach to automatic parallelization relies on a conventional model of a computing processor, albeit using large numbers of such processors. A radically different approach to parallelism is provided by those researchers working on data flow architectures and software as introduced by K. P. Arvind at University of California at Irvine and by W. B. Ackerman and J. B. Dennis at the Massachusetts Institute of Technology (Arvind and Iannucci 1983, Dennis 1980). In their data flow computation model there is no need for transfer of control from instruction to instruction in von Neumann style. Instead, it is claimed that all computation can be seen to be data driven. When in an assignment statement all right hand side operands have values, then the assignment to the left hand side variable can proceed. Control structures are not much more difficult. Instead of a two-tiered model where the left gets a value when the right is ready, there is, for instance in an IF-THEN-ELSE statement, a three-tiered model: when the operands in the IF clause are ready, then a result of that clause is tested, and the THEN or ELSE clause are executed accordingly. Using a Petri Net analogy, it is quite possible in a program for the firing rules of more than one statement to be satisfied. The potential for concurrent execution is high, depending again on the algorithm. Such parallel execution is enabled without excess communication or other overhead.

One particularly useful technique incorporated into the data flow methodology is to program using a single assignment language (SAL) (Gurd 1984, Glauert 1984). These languages require that each variable is allowed to be assigned a value at most once. Conventional programming languages allow assigning values to variables repeatedly in the course of a program. Translation at compile-time into SAL is possible by breaking each variable down into a series of instances of that variable. A practical example showing how concurrency is thus enabled is as follows. Assume we have the statements A:=B; C:=A; A:=D; E:=A which will take four cycles to execute according to Kuck's data dependency rules. Translated into SAL we have A0:=B0; C0:=A0; A1:=D0; E0:=A1. Note that the same group of statements can be parallelized to execute in only two cycles. What SAL does is eliminate two of Kuck's data dependencies: the write-after-read and the write-after-write dependencies, thus simplifying the problem of parallelization.

Control structures in SAL are more difficult to handle. An example of the problem is as follows: assume we have the program fragment IF A=0 THEN B:=0; C:=1 ELSE B:=1. The danger in translating this fragment to SAL, that is, to IF A0=0 THEN B0:=0; C0:=1 ELSE B1:=1, is that a subsequent expression which wishes to refer to the last value of variable B will not know whether to refer to B0 or to B1. SAL's solution is to modify all control structures so that they are functions which provide return values. In the case of the IF-THEN-ELSE the new structure would be something like B0,C0 := IF A0=0 THEN 0,1 ELSE 1,nil. While such a modified construct has a certain amount of elegance, it also has some drawbacks. If the scope of the IF-THEN-ELSE was large, and especially if the THEN and

ELSE clauses were poorly balanced, then the function would become unreadable. If variable C had been assigned a previous value and it was a large variable (say, an array or a record), then making redundant copies of it could become wasteful. This is a problem shared by all functional language approaches.

In a strict sense, data flow is a special kind of architecture with a marked asynchronism. A more general definition is that it is an interface between parallel hardware and sequential software. Data flow can be viewed as a methodology by which a high level program is transformed into a special data dependency graph. This graph allows the efficient determination of operand availability, and is very much like a conventional abstract syntax tree.

A third approach to the automatic detection of parallelism is provided by Josh Fisher who designed and is building the ELI-512 computer (Fisher 1983). This Very Long Instruction Word (VLIW) architecture is much like the DRAFT's except that the VLIW has up to 16 ALU processors, and uses a shared memory. Fisher's student, John Ellis, designed and implemented a compiler which generates parallel code for the ELI-512 computer with minimal help from the programmer (Ellis 1986). Fisher and Ellis claim to have a parallelizer which produces code that has a factor of eight speedup and has successfully broken the basic block barrier using a method called trace scheduling, which is to trace all possible control flow paths through the program, and schedule them onto the long instruction words. in order of probability of their occurrence. It is in deciding the probability of the control flow paths that the programmer is asked to be explicit. Because of this need for the programmer to estimate probability of execution, the ELI-512 project has restricted itself to the translation of scientific programs, where they believe such probabilities are easy to assess. To further simplify the parallelization effort, they ask that the programmer specify how many times a loop should be unrolled.

To summarize, the unifying issue for all methods of automatic detection of parallelism is in the resolution of data dependencies; recognizing how they force some instructions to be executed in a particular order, while allowing other instructions or groups of instructions to be executed concurrently. Using the single assignment form for the instruction increases prospective parallelism. The limiting factor is the inherent nature of the input program, the way such programs are structured, and how interdependent the constituent operands are. Program restructuring seems a most promising tool for eventually being able to achieve greater parallelism, but is still in the research stage. Trace scheduling, by contrast, is severely limited in usefulness by the complexity of the algorithm of the input program and verges on not providing automatic translation to the parallel form.

In the terms presented above, let us now provide an overview of the task at hand for this parallel code generator.

We start with the hardware constraint that there is a single sequence controller and that there are multiple ALU processors. In the first implementation of the DRAFT architecture, The Factoring Machine, there are at most eight such equal-sized processors. In this software modeling of the DRAFT architecture, however, there may be any number of ALU processors, all of which are arranged in a ring. The second hardware constraint is that there are only local memories, and that data must explicitly be transferred by means of transfer instructions which each take an entire cycle, but can be executed in parallel with other instructions. Parallelism is at the instruction level, and there are no explicit parallel constructs.

Because of the long instruction word format of the target language, there is no need for communication between the processors, and all assignments of instructions to processors are done at compile-time.

Our parallelizing effort begins with the standard abstract syntax tree representation of a sequential program. We then preprocess the tree in order to transform the program into single assignment form with normalized control structures. Note that, unlike the traditional SAL method, our design does not require the explicit use of special control structures, since the normalization can be effected automatically. The actual parallelization is accomplished as the normalized abstract syntax tree becomes incrementally transformed into a parallel intermediate code representation which is very similar to the DRAFT microcode. The incremental transformation occurs as a scheduling procedure of each of the instructions, monitoring that all data dependency and basic block constraints are followed. In order to ensure that the scheduling results in a reasonably small number of long instruction words overall, a lookahead algorithm is employed. We turn now to the design of our parallelizer.

CHAPTER TWO: DESIGN AND IMPLEMENTATION OF THE PARALLEL CODE GENERATOR

In this chapter we discuss the design and implementation of a parallel code generator which translates from sequential to parallel code. We show how sequential code can be automatically converted to parallel code by means of scheduling, discuss several algorithms used in scheduling, and show how this parallel code generator can be integrated into a compiler.

Parallel Code Generation via Scheduling

Translation between programming languages has long been an automatable task due to the regularity of programming language syntactics. The main challenge in designing a translator--be it a compiler, interpreter, assembler, or parallel code generator--is to have the translator produce efficient code in the target language. Though it is possible to translate directly from the source to target language, it is generally considered useful to first parse the source program into an intermediate code representation. This allows for further analysis and optimization of the code, as well as some degree of machine independence. The most common intermediate code representation is the abstract syntax tree (AST) which represents the code in prefix normal form (Figure 2). Once built, the AST can be traversed in a depth first manner, enabling an exact reconstruction of the source program. This process guarantees that in translating from one sequential language to another there is thus never a change in the functionality of the source program during the translation process.

Consider what is involved when the translation is from sequential to parallel code. Because multiple program operations can execute simultaneously in the parallel language, there is a danger that the originally sequential operations can be executed in an incorrect order, thereby altering the semantics of the source program. From Kuck (see Chapter 1) we know that the semantics of the sequential program can be kept intact so long as the data and control dependencies are either eliminated or strictly adhered to. The output dependencies and anti-dependencies between operations can be eliminated by translation into single assignment language (SAL) format, while operations which are ordered because of flow and control dependencies must be kept in a specific order. This can be accomplished by inserting dependency arcs between operations, or by assigning distinct execution cycle numbers to each operation. If, for example, the first operation in a

```
((A:=2)
 (B:=0)
 (IFTHENELSE (A = B)
             ((C:=A+B))
              ((C:=A-B))))
```

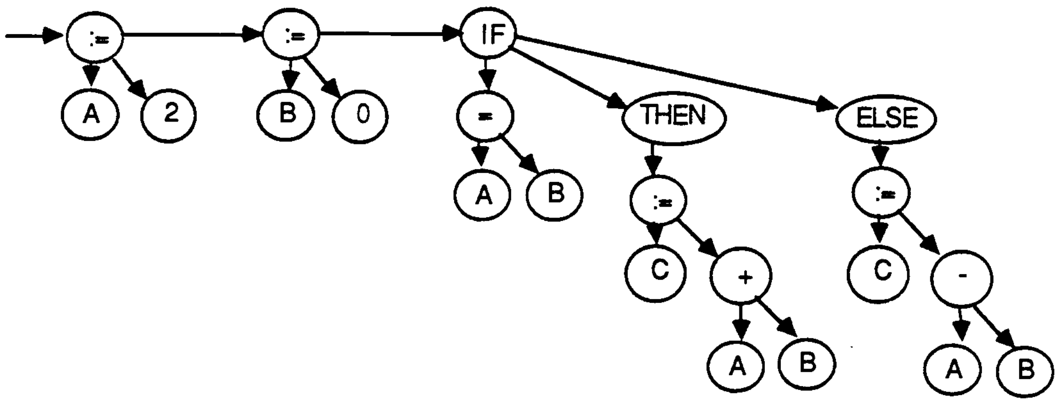

Figure 2: Abstract Syntax Tree

source program is assigned to some processor at cycle 1, then the second might be assigned to either cycle 2 or to cycle 1 depending on a dependency relation existing between the two operations. Maintaining this strict order between somehow dependent operations is crucial to parallel code generation.

A second issue in parallel code generation applies only when the target code uses a long instruction word format. Such architectures require different treatment of control structures and ALU operations. When an operation is a control structure, it must be assigned to the sequencer. When it is an ALU operation the decision must be made to which of the multiple ALU processors to assign the operation to. It is here that the ordering of statements becomes very important.

We have designed a second intermediate code representation in order to simplify this decision. This parallel intermediate code representation (parcode) contains an execution cycle number, and has room for multiple ALU operations and for a sequencer operation. Figure 3 shows a sample of DRAFT microcode, and Figure 4 our corresponding parcode. In both the parcode and the microcode the processors are numbered from left to right,

ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
CLEAR R0	SUB R0, R1	NOP	INCB R1,R2	EX
INCA R0	NOP	NOP	MOV D0,R1	JP

Figure 3: DRAFT Microcode

	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
7	ZERO(NUM1)	ONE:= ONE-TWO	--	INC(A,B)	--
8	NUM1:=NUM1+1	--	--	MOVE(A)	JUMP

Figure 4: Intermediate Parallel Code Representation

and the rightmost column is an instruction to the sequencer. The default values for the processors are to execute no-op instructions, while the default for the sequencer is to continue on to the next instruction. Notice that the operand names in the parcode are still in symbolic form while the micro-code requires bound register and memory addresses. Using a second inter-mediate representation allows the treatment of parallelism independent of machine dependent issues. Actual code generation can easily be imple-mented in a later step.

Generating parallel code can thus be seen to be a data structure transformation, from a sequential abstract syntax tree to a parallel, yet still abstract, multiple-queue-like structure. The translation is initiated by traversing the AST as in a sequential program translation, and is completed by assigning each ALU operation to an appropriate parcode queue at a posi-tion in the queue which does not violate temporal dependency rules. This is essentially a scheduling algorithm for operations onto parcode.

Memory architecture must also be considered in scheduling. In a shared memory architecture it matters only WHEN a variable is defined, but with local memories it also becomes important to know WHERE, that is, by what processor, the variable is defined. (Hereinafter the combination of ALU processor plus local memory plus local register bank will be referred to as a segment). If it becomes necessary to transfer a variable to another segment, a program instruction which can transfer the variable must be inserted into the target code. Since data transfer is accomplished by operations, they too must be scheduled. We will denote data transfer operations as MOVE operations.

Consider the following example for a four segment machine. Assume that we want to schedule the statements X:=2; Y:=X. The first operation, X:=2, could be assigned to any of the segments. Assigning it to segment 3 produces

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	X:=2	--	--	--	--

while assigning it to segment 2 produces

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	--	X:=2	--	--	--

which are two equally good solutions. Assume we choose the latter. Suppose now that we want to add the operation Y:=X. Adding this statement to segment 2 will result in

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	--	X:=2	--	--	--
2	--	Y:=X	--	--	--

while adding it to segment 3 will require an explicit data transfer:

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	--	X:=2	--	--	--
2	--	MOVE(X)	--	--	--
3	Y:=X	--	--	--	--

Because of the circular arrangement of processors in the DRAFT architecture, a MOVE from segment 3 transfers data to segment 0, hence scheduling Y:=X to segment 0 would require a total of four instructions

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	--	X:=2	--	--	--
2	--	MOVE(X)	--	--	--
3	MOVE(X)	--	--	--	--
4	--	--	--	Y:=X	--

while assigning it to segment 1 would require five instructions. Notice that if the operation to schedule had been Y:=3 instead of Y:=X then the operation could have been assigned to any segment other than 2 without needing to add an additional long instruction word. Assigning Y:=3 to segment 3, for example, would result in

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	Y:=3	X:=2	--	--	--

This is a simple way to schedule: to test-assign each operation to every one of the ALU processor segments, assess the results of such test-assignments, and then assign the operation to the segment with the best results. Note that in the above examples what we have considered to be the "best results" is lowest number of long instruction words in the parcode after the operation has been assigned to each of the segments. Since effective parallel execution would require an even distribution of operands, an alternative cost measure might be the number of operands already assigned to the local memories of each segment. Proper balance will have a strong effect on the run-time performance of the target code. But for programs with a large number of operands such a method would quickly degrade into assigning operations to processors without regard for the locations of their constituent operands and ultimately result in inefficient code.

Looking only at the immediate consequences of assigning operations to segments presents a problem. In the previous example we saw how the assignments which resulted in the least number of instructions were those that involved the least amount of data transfer, implying that assignments of operations to segments involving data transfer will be avoided, perhaps causing the operands to group together in few segments. In a case in which there are many data dependencies, it is conceivable that all operations would be assigned to a single segment, thus providing no parallelism whatsoever. Our expectations of a highly parallel program are that operations and operands are evenly spread out over the processors, yet the high cost of data transfer appears to prevent that.

The answer to this dilemma is that we haven't looked far enough. Instead of looking only at the immediate effects of an operation placement, we should modify the approach to look at the effects on future operation assignment. We will call such an approach a lookahead method. In our terminology, a "lookahead of N" means that we will postpone our decision where to assign an operation until we have looked at the consequences of assigning all next N ALU operations to all possible segments. Consider the potential difference which a lookahead of two could have over the simple lookahead described so far. Assume that we want to schedule the statements Y:=3; IF X=Y THEN..., and that the parcode currently is:

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	X:=2	--	--	--	--

Using a lookahead of one the first operation to be scheduled, Y:=3, could be assigned to any of the segments. Let us assume that it is assigned to segment 1. The best segments to which the next operation, X=Y, can be assigned are segments 1 and 3 because two moves are required in either case. Assigning it to segment 3 results in

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	X:=2	--	Y:=3	--	--
2	--	--	MOVE(Y)	--	--
3	--	MOVE(Y)	--	--	--
4	X=Y	--	--	--	--

A better choice, recognized with a lookahead of two, would have been to assign Y:=3 to segment 2 or 3, resulting in the more efficient parcode fragment

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	X:=2	Y:=3	--	--	--
2	--	MOVE(Y)	--	--	--
3	X=Y	--	--	--	--

Lookahead is a way of making more informed judgments about operation placement by looking at the implications for future operations. It is intuitive that looking ahead at the next three or four operations when deciding where to place an operand will result in more efficient parcode than would be produced by just looking at the next one operation. A look-ahead of all remaining operands would result in an optimal assignment of operations to segments, since every possible combination of operations and segments would be looked at, but that solution is much too computationally intensive. For this reason we use a relatively local lookahead of up to five. There is no obvious way to cut down on the complexity since assigning operations to any of the segments could result in the best parcode even if that is not immediately evident.

It is not the case that every degree of lookahead results in more efficient parcode. For simplicity, let us consider code generated for a two processor machine that has been built so far to contain:

Cycle	ALU 1	ALU 0	Sequencer
1	A:=1	B:=2	--
2	C:=A	--	--

and where we are to schedule the following operations: D:=B+C; E:=A. Before we can schedule the first operation it gets converted to two-operand form, so that we have three operations to schedule: D:=B; D:=D+C; E:=A. With a lookahead of one, the first operation will get assigned to segment 2, and the best result of scheduling all three operations will be:

Cycle	ALU 1	ALU 0	Sequencer
1	A:=1	B:=2	--
2	C:=A	D:=B	--
3	E:=A	MOVE(D)	--
4	D:=D+C	--	--

A lookahead of 2 would recognize that all three operands B, C, and D are interrelated, and so would move B over to segment 1 first. The irony is that this cleverness would result in the less efficient parcode:

Cycle	ALU 1	ALU 0	Sequencer
1	A:=1	B:=2	--
2	C:=A	MOVE(B)	--
3	D:=B	--	--
4	D:=D+C	--	--
5	E:=A	--	--

Current Implementation of the Parallel Code Generator

The scheduling and lookahead mechanisms described above have been implemented in the form of a Franz LISP program which consists of two primary modules: a preprocessor to the parcode generator which reads in and modifies the AST, and the actual code generator, which traverses the modified AST and builds the parcode. In this section we provide an in-depth discussion of each of these modules.

The main purpose of the preprocessor is to put the AST into a form suitable for scheduling. Three basic transformations must be accomplished by the preprocessor: modifying the operands to be in single assignment form; normalizing the statements to compensate for the conversion to SAL; and converting operations from three-operand to two-operand form.

All program operands are modified. Because the DRAFT microassembler does not allow literal constants, all literals must be given symbolic names. No two literals will be assigned the same symbolic name even when they share a common value in order to attain greater parallelism. Similarly, no two operands on the left hand side of an assignment statement will have a common name. These various instances of the variable are assigned distinct symbolic names by appending instance numbers onto the variable name, changing variable A, for example, into A0, A1, A2, etc.. The new symbols for the constants and variables are entered into a hashing symbol table along with fields indicating their type, and either their "parent" variable name in the case of variables, or their literal value in the case of constants.

This translation into SAL is not without problems when dealing with control structures. Recall the example statement in the previous chapter:

IF A=0 THEN B:=0; C:=1 ELSE B:=1

which, translated into SAL, became

IF A0=0 THEN B0:=0; C0:=1 ELSE B1:=1.

The problem was that a subsequent reference to variable B would not know whether to refer to B0 or B1. SAL's solution was to create an entirely new control structure which required all variables defined in an IF-THEN-ELSE statement to be explicitly listed much like return values in a function. Our solution is a modification of this idea in which the translator will check when there is a problem and remedy the problem when necessary. In the case of an IF-THEN-ELSE statement there is a problem when a variable is defined in both of the THEN and ELSE clauses. The solution is to merge the last definitions in each of the clauses into a single instance. In the example above, we add the operation B2:=MERGE(B0,B1). The MERGE operation is not an explicit construct in either the source or target language, instead it is used for internal purposes only, as a directive to the register allocator.

The problem with loops is also easy to solve. Consider the operations:

A:=0; REPEAT A:=A+6 UNTIL A=36

which in SAL are

A0:=0; REPEAT A1:=A0+6 UNTIL A1=36.

The problem is that A1 is never allowed to accumulate a running total since it is repeatedly assigned the original value of A plus 6. More rigid variations of a loop structure could be introduced, but we find that it is possible for the translator to scan the code to determine if there is a problem. There is a problem and a need for corrective action only when a variable is both used and defined within a loop such that the usage precedes the definition. The corrective action is to add a statement to the loop which reassigns such variables. In the above example the solution is to add the statement A0:=A1 resulting in

A0:=0; REPEAT A1:=A0+6; A0:=A1 UNTIL A1=36.

A final transformation that must occur prior to scheduling is the conversion from the three-operand form typical of ASTs into the two-operand instruction form used by most low-level languages including DRAFT microcode. This conversion should occur prior to scheduling for efficiency, since the two-operation conversion results in additional operations. The disadvantage of translating to two-operand form at this point is that a clean separation of machine-independent versus machine-dependent changes to the program is lost. But because scheduling involves very tight packing of operations into the long instruction words, adding extra operations after scheduling would only result in extraneous and underutilized long instruction words. Hence it is desirable to convert into two-operand form prior to scheduling.

A similar argument also applies to machine-dependent optimizations. Consider the simple statement A:=B+1; which becomes A:=B; A:=A+1 when converted to two-operand form. A more efficient translation is to INC(A,B), a DRAFT microcode instruction which increments the value of B and assigns the result to variable A. It is better to detect such a savings before the longer number of operations are packed into the parcode, hence this kind of machine-dependent optimization is also performed in advance of the scheduling.

We turn now to the primary module which generates parallel code. There are three essential aspects to the parcode generator: the manner in which operations are scheduled, the manner in which an informed scheduling decision can be made via lookahead, and the way in which parcode is built. Each of these will now be discussed.

Parallel code is generated by traversing the AST in depth first order, scheduling each operation to a specific segment and cycle coordinate in the parcode, and adding new long instruction words when necessary. Each time a control structure is encountered during the traversal, appropriate operations (such as JUMP CONDITIONAL or JUMP UNCONDITIONAL) and labels are inserted into the sequencer column of the parcode. When an ALU operation is encountered during the traversal, it is assigned to one of the ALU columns. Selection of the columns is determined by:

1. determining the cost of assigning the operation to each of the segments, and
2. assigning the operation to the segment with the lowest cost.

The cost function is the number of long instruction words in the parcode after the assignment of the operation.

Step 1 above refers to a call to the lookahead function. For each level of lookahead, operations will be "trial-assigned" to the parcode. A copy of the parcode is built exactly as the actual parcode will be. Though ALU columns in the trial-parcode are built as ALU columns in the actual parcode, it is not necessary to build up the sequencer column with such precision. Instead, it is sufficient to insert a marker indicating a basic block boundary. The algorithm for lookahead is recursive and is shown here:

```
while there is an operation left to trial-schedule
  -if there is only one operation left to trial-schedule,
  -then trial-assign it to each segment and return the
    segment number which results in the lowest cost for
    the trial-parcode,
  -else trial-assign the operation to each segment of
    the trial-parcode, and call lookahead with the
    remaining operations.
```

Note that there will be as many operations to "trial-schedule" as there are degrees of lookahead. It is this lookahead function which will dominate the run-time of the parcode generator. If N is the number of operations in the program in two-operand form, S the number of segments, and L the degree of lookahead, the complexity of the parcode generator will be

$$NS^L$$

This complexity clearly limits both the number of segments and the degree of lookahead if the translator is to generate code in any kind of efficient manner.

Finally, we build the parcode. What the lookahead returns is: an operation to schedule, a segment number to which to assign an operation, and a number (possibly 0) of data transfer operations which must also be attached to the parcode. Packing that operation or group of operations onto the parcode is accomplished by the following algorithm:

```
for each operation to attach to the parcode
  -determine the earliest possible cycle at which the
   operation can be inserted based on data and
   control dependencies, and
  -attempt to insert the operation at that segment and
   cycle. If that spot is taken try the next cycle,
   and then the next, and so on. If necessary, create
   another long instruction word to append onto the
   parcode.
```

Because this algorithm is called both during scheduling and during look-ahead trial-scheduling, this algorithm has been implemented free of side effects.

The parcode generator is capable of translating a number of generic high-level language operations into parallel code. All control structures are handled in this parcode generator: IF-THEN, IF-THEN-ELSE, WHILE, REPEAT, and FOR. Relational operators include EQUAL, NOT EQUAL, GREATER THAN, GREATER THAN OR EQUAL, LESS THAN, and

LESS THAN OR EQUAL. Boolean operators AND, OR, and NOT are implemented. Arithmetic expressions include PLUS, BINARY MINUS, UNARY MINUS, MULTIPLICATION, MOD, DIV, and EXPONENTIATION. The last four operators will be scheduled differently than the others because these operations are handled by macro calls in DRAFT microcode. There is no major difference between these and other arithmetic operations except that scheduling macro-operations will involve reserving an entire long instruction word, instead of using only a single cycle and segment coordinate. Compound boolean and compound arithmetic expressions can be translated, but combining the two in mixed-mode expressions has not been implemented.

The result is a parallel code representation of the sequential code, plus the exact segment and cycle coordinates of each operand stored in a symbol table. Figure 5 shows a short sample program which calculated the DIV and MOD of two numbers by repeated subtractions. The program is shown at three stages: as it is read into the preprocessor, after it is converted by the preprocessor, and after its operations have been scheduled onto the parcode.

AS READ INTO THE TRANSLATOR:

```
((:= W 0)
 (:= B 3928490)
 (:= A 4993)
 (:= R B)
 (WHILE (>= R A)
        ((:= R (- R A))
         (:= W (+ W 1)))
```

AFTER PREPROCESSING:

```
((ZERO W0)
 (:= B0 CON0)
 (:= A0 CON1)
 (:= R0 B0)
 (WHILE (>= R0 A0)
        ((:= R1 R0)
         (- R1 A0)
         (INC W1 W0)
         (:= W0 W1)
         (:= R0 R1))))
```

AFTER PARALLELIZATION:

Cycle	ALU 3	ALU 2	ALU 1	ALU 0	Sequencer
1	ZERO(W0)	B0:=CON0	A0:=CON1	--	--
2	--	R0:=B0	MOVE(A0)	--	START_LOOP
3	--	R0<A0	--	--	--
4	INC(W1,W0)	R1:=R0	--	--	EXIT_ON_TRUE
5	W0:=W1	R1:=R1-A0	--	--	--
6	--	R0:=R1	--	--	--
7	--	--	--	--	END_LOOP

W = WHOLES
R = REMAINDER
A = DIVISOR
B = DIVIDEND
CON = CONSTANT

Figure 5: Sample Program

Proposed Implementation of a DRAFT Compiler

We will now show how this parallel code generator can be integrated into a complete compiler for a high-level language. At the front end of such a compiler is a lexical scanner, a syntactic parser, a semantic checker and optionally an optimizer, in short, all parts which have to do with the analysis of the source code. The back end is responsible for all aspects having to do with synthesizing the target code. It handles all target code dependent optimizations, allocates registers to most commonly used operands, and deals with other memory management details. Our parallel code generator could easily fit between the two.

The parcode generator expects its input to be an AST in the form of a LISP s-expression. The syntax tree was chosen to be the input representation precisely because it is the common representation output by a compiler front end. A front end could thus be programmed to accept a program written in a subset of any high-level procedural language, such as Pascal, FORTRAN or C. The output from the front end, provided it was in the proper AST form could then be directly input into the parcode generator.

The output from the parallel code generator is likewise in a form suitable for linking directly to a compiler back end. A back end for a DRAFT compiler is responsible for completing the translation process into DRAFT microcode by resolving all memory references and by scanning the parcode in a final peephole optimization phase. There is one difficulty with resolving memory references in the DRAFT architecture, for only a single memory page can be accessed by the ALU processors at any one time. This requires coordinating memory accesses by the processors, thus requiring scheduling operands to be stored in main memory to be on specific pages. The difficulty of handling memory accesses in this way will determine just how general the programs can be which are to run on a DRAFT computer. Whether additional high-level features such as procedures and functions can be implemented also remains to be seen, as these are also constrained by this kind of memory access as well as by the limited stack size (9 levels) of the current implementation of this architecture.

If a high-level language were to be designed for the DRAFT architecture, it would no doubt require the use of arrays. The problem with automatically parallelizing arrays is that it is undesirable to schedule them as instances because of their size. The answer could be to schedule each array cell separately, but this introduces another problem. If one were to schedule array cells, then it would be necessary to limit the way arrays are subscripted, namely to allow only literal values as subscripts. Array cells which are referenced using run-time determined subscripts cannot be scheduled at compile-time for it will not be known until run-time to which physical memory location they refer. This severely restricts the use of arrays.

Finally, we would like to be able to take more advantage of some of DRAFT's architectural features. One of the most interesting features of a DRAFT computer is its ability to reconfigure its word size. Ideally we would like the high-level language programmer to be able to take advantage of this feature by allowing the declaration of variables in varying sizes, for example, int (32 bits), longint (64 bits), verylongint (128 bits), and perhaps even veryverylongint (256 bits). Reconfigurability could then be very useful. If, for example, A and B were declared as int and C as longint, then we could have C:=A*B without danger of overflow. The difficulty with incorporating reconfigurability into our scheduling method is that it undermines the generality of the scheduling scheme. An implementation restriction of the first DRAFT computer is that it is not possible to compute with a variable while it resides in segments on each end of the row of ALU processors, for instance, extending over the leftmost and rightmost segments, but not in the segments in between. Using varying sized operands and reconfiguring the segments would only work if operands of those varying sizes were prescheduled, requiring substantial changes to the current parallel code generator.

There is another interesting feature in the DRAFT architecture, namely its ability to look at the result of boolean expressions being executed in several segments simultaneously and then take action either when all of them evaluate to TRUE (e.g., JUMP on AND), or when a single one of them evaluates to TRUE (e.g., JUMP on OR). We could take advantage of this feature by transforming all boolean expressions into either conjunctive normal form or disjunctive normal form.

CHAPTER THREE: PERFORMANCE DATA

The performance of any translator is measured both by how efficiently it executes, as discussed in the last chapter, and by the efficiency of the code it produces. In this chapter we first discuss different ways of testing the efficiency of automatically generated parallel code, then look at ways to predict how well sequential programs can be parallelized. The remainder of the chapter will present and discuss results obtained from testing our parallel code generator.

One standard against which to measure the efficiency of code produced by our parcode generator is to compare it to code produced by other similar translators. The most similar translator is the compiler written for the ELI-512, but there are several reasons that comparisons with its benchmark programs are inappropriate: they all use arrays, they include explicit assertions about branching probabilities and how far to unroll loops, and the input programs were rewritten and restructured by hand in order to obtain best results (Ellis 1986). Until there is some established definition of how automatic an "automatic" parallel code generator should be, and a standard method of testing such generators, comparisons between different systems are of limited use.

Another potential method for assessing the performance of our generator would be to compare its translations to translations done by a programmer. Though this would be very interesting, it is unclear that anything could be concluded from such an experiment. Factors such as how experienced the programmer was, whether the experience was with sequential and/or parallel code, whether the programmer was familiar with the algorithm being translated, and whether unlimited time should be given to translate the code would have to be considered. The emphasis, in other words, is on aspects other than on the parallel code itself.

The most common method of assessing the efficiency of parallel code is by comparing the execution times obtained by running a program on multiple processors to running it on a single processor (Hwang and Briggs 1984). This is called the speedup of the program and is most effective when the tests are run using the same code generator to ensure uniformity. Speedup is calculated by:

$$Speedup = \frac{\textit{number of instruction cycles using 1 processor}}{\textit{number of instruction cycles using } N \textit{ processors}}$$

This is the method which we will use to check the efficiency of our code. By varying the number of segments for which code can be generated we can see whether there is an apparent optimal number of processors according to our data.

In addition to varying the number of segments we can also vary the amount of lookahead used by our code generator during scheduling. Running that kind of test will determine whether the extra time spent looking ahead during scheduling pays off in more efficient target code. Before we present the results of our tests, we explore the possibility of being able to predict how parallelizable some program is.

It has been argued that the potential for generating parallel code from a sequential program relies strongly on the nature of that program, and yet it is unclear what the characteristics are which make a program suitable, or unsuitable, for parallelization. It is easy to count the number of instructions, but the number of instructions is independent of the program's ability to be parallelized. Because we restrict ourselves to parallelization within basic blocks, it might be useful to characterize a program by how many basic blocks it has, or by how many ALU operations it has per basic block.

But any measure must consider how entangled the operations in the basic blocks are. One is tempted to think that the number of arcs in a data dependency graph of the program would provide some metric. But a transitive closure over such a graph will determine that there is a certain amount of entanglement over the entire program. What we need is a less global metric, and more of a local assessment of operation interdependency. A proposed local metric is to sum over all basic blocks the smallest number of instructions in which to execute each block imagining the best possible parallelism. Adding to that sum the number of sequencer instructions which cannot be executed in parallel with ALU instructions, we end up with an optimal speedup for each program:

$$\textit{optimal speedup} = \frac{\textit{number of instruction cycles using 1 processor}}{\textit{optimal number of instruction cycles}}$$

Note that this metric does not consider the data transfer which would be necessary in a local memory system such as ours.

For our tests we have seven programs taken from introductory programming textbooks. All programs use only integer arithmetic and basic control structures. The programs have varying characteristics such as size, number of ALU operations per basic block, and operation interdependencies. In what follows we briefly describe the test programs and their characteristics. Because we hope to find a correlation between program characteristics and suitability for parallelization, we also provide an average of all speedups obtained using multiple processors. When a table entry is blank, that means that no test was performed on that particular combination of segments and lookahead.

program FACT--calculates the factorial of an integer.

number of two-operand instructions--10
two-operand instructions per basic block--3
optimal speedup--1.80
average speedup--1.31

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.29	1.50	1.13	1.50	1.50
	4	1.50	1.13	1.13	1.29	--
	8	1.50	1.13	1.13	--	--

program DIVISION--simulates DIV and MOD by subtraction.

number of two-operand instructions--13
two-operand instructions per basic block--4
optimal speedup--1.50
average speedup--1.25

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.50	1.09	1.34	1.34	1.20
	4	1.50	1.09	1.09	1.20	--
	8	1.50	1.09	1.09	--	--

program ISPRIME--determines if an integer is a prime number.

number of two-operand instructions--16
two-operand instructions per basic block--2
optimal speedup--1.36
average speedup--1.23

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.25	1.25	1.15	1.25	1.15
	4	1.25	1.25	1.15	1.25	--
	8	1.25	1.36	1.15	--	--

program MULTIPLY--multiplies by repeated addition.

number of two-operand instructions--20
two-operand instructions per basic block--1.9
optimal speedup--1.50
average speedup--1.36

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.13	1.20	1.13	1.38	1.38
	4	1.50	1.38	1.50	1.38	--
	8	1.50	1.38	1.50	--	--

program BALANCE--follows the balance of a charge account as monthly payments are made.

number of two-operand instructions--30
two-operand instructions per basic block--6.25
optimal speedup--2.50
average speedup--1.30

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.25	1.37	1.19	1.47	1.25
	4	1.39	1.19	1.25	1.39	--
	8	1.39	1.19	1.25	--	--

program ROOT--uses the Bisection method to compute the root of a function within a certain interval and with an error not exceeding a given tolerance.

number of two-operand instructions--46
two-operand instructions per basic block--3.5
optimal speedup--1.41
average speedup--1.10

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.17	1.05	1.08	1.14`	1.14
	4	0.95	1.08	1.14	1.17	--
	8	1.11	1.07	1.14	--	--

program PALINDROME--produces numerical palindromes from integers by repeatedly adding its reversals to itself.

number of two-operand instructions--83
two-operand instructions per basic block--3.6
optimal speedup--2.05
average speedup--1.28

		degrees of lookahead				
		1	2	3	4	5
	1	1.00	--	--	--	--
segs	2	1.16	1.32	1.32	1.32	1.28
	4	1.16	1.32	1.32	1.32	--
	8	1.16	1.32	1.32	--	--

To see what the effects are when changing the number of segments we average the results of all data obtained with a lookahead of three:

	number of segments			
	1	2	4	8
speedup	1.00	1.19	1.23	1.21

The best performance is obtained using four segments. This suggests that using fewer than four processors might not adequately take advantage of inherent parallelism in the programs. Since the test programs have fewer than four instructions per basic block on the average, it is not surprising that using eight processors does not improve the performance. That speedup decreases with an increased number of processors implies that at least one operation was assigned to an extraneous processor, probably at a time when the lookahead was too limited to assess the implications of that assignment, and that this assignment later necessitated otherwise superfluous data transfer.

Maintaining a constant number of segments, say two, and varying the degree of lookahead we obtain:

	degrees of lookahead				
	1	2	3	4	5
speedup	1.25	1.24	1.19	1.34	1.27

There is, to our surprise, little or no improvement in speedup obtained with increased lookahead. That it is possible for a greater lookahead to result in a worse performance was discussed in the last chapter. A lookahead of one turns out not to be as bad as we had anticipated. This is probably due to the fact that even when scheduling cannot look forward it must still take into account what the parcode looks like so far, and that the look backward might overshadow the look forward. Nonetheless we see a slight improvement in performance with higher lookahead.

We don't think that any general conclusions can be drawn about the speedup being in all cases no better than 1.5. In his study of automatically parallelizing non-numeric programs, Kuck found that most test programs resulted in a speedup of between 1.5 and 2 times, even when up to thousands of processors were used (Lee, Kruskal, and Kuck 1985). A few of the test programs resulted in speedups of up to several hundred times when executed on such huge parallel computers, again suggesting that it is the nature of the program which has the most effect on how well it can be parallelized.

Can parallelism be predicted from the nature of the program? We attempted to correlate the overall speedup for each program with our two proposed metrics--instructions per basic block and parallelism potential based on the optimal speedup--using the formula for a correlation coefficient found in Shneiderman, 1980. * Surprisingly, we found no correlation whatsoever between the average number of instructions per basic block and average speedup ($r = -0.03$). The correlation between average speedup and parallelism potential was more promising at $r = 0.40$, suggesting that this metric is on the right track to capturing how well a program can be parallelized. Where both metrics fail is that they do not consider that operands might be located in separate local memories.

These tests are by no means exhaustive, but they do suggest a paradigm for continued testing. More complete testing could be accomplished with a larger sample size, experimenting more with the number of processors and degree of lookahead, and attempting to devise a metric which has

* The range of values which the correlation coefficient can have is from a negative correlation of -1.0 to no correlation at 0 to a high correlation at 1.0.

a better correlation with actual speedup. The next chapter explores how some of the underlying methodology of the algorithms can be altered.

CHAPTER 4: CONCLUSIONS

This thesis has explored the question whether translation from sequential code into long instruction word parallel code can be automated. Our findings indicate that automatic parallelization of this kind is feasible and practical using the scheduling method. In this chapter we discuss long instruction word programming, local memory architectures, and the use of lookahead in scheduling. We conclude by reexamining the debate about the use of explicit parallel constructs versus automatic detection of parallelism.

Long instruction word programming is a viable method of translating at the instruction level. Its advantages are its synchronism and its parallelism at the ALU level, both of which have simplified our task. That long instruction word operations are synchronized is the very reason we are able to schedule at compile-time. That parallelism is at the ALU level is appropriate to our parallelizing within basic blocks. The advantages of long instruction word programming are also its disadvantages. Its synchronous nature means that there will be some data references that simply cannot be resolved at compile-time. ALU level parallelism prevents program structures larger than ALU operations from being concurrently executable.

We now consider the effects the local memory scheme has on our scheduling algorithm and compare it to the effects of a shared memory scheme. Having local memories adds an additional constraint on scheduling: to be accessible, an operand must be defined, and also resident in the segment in which it is to be referenced. This requires additional work both in tracking the locations of operands, and in inserting data transfer nodes into the parcode, but the work is not unmanageable. What is a serious problem is how quickly data transfer operations result in the formation of several long instruction words. Consequently operations tend to avoid segments that are far away and stick to just a few, causing an uneven distribution of operations to segments.

Consider the effects of a different interconnection scheme on scheduling. Assume that each ALU processor had access to not just one, but two or three neighboring segments. Operation scheduling would proceed exactly as it does in the current ring interconnection scheme, since each segment would continue to be checked. The only difference is that fewer data transfer operations would be needed and costs of even distant segments would remain low. Even if the architecture was completely connected the scheduling mechanism would not change, and at most one move operation would be needed to access an operand. If the local memory scheme were abandoned in favor of shared memory--which can be simulated by a full interconnection scheme--then there would be no data transfer nodes at all.

The resultant gain in performance would not be without a cost. The higher the interconnection scheme, the greater the danger that processors attempt to simultaneously access some memory cell, hence the need for a system of locks or semaphores. Note that the problem of operand availability exists for both the local memory and shared memory systems. While in the former, the processor might have to wait for an operand because the operand still needs to be transferred, in the latter the processor might have to wait until another processor has released its exclusive access to some memory location. Which of these schemes is better remains open to question.

Next we address ourselves to our algorithms and implementation given the restrictions posed by the DRAFT architecture. In running tests, our parcode generator often produced optimal parallel code, but it was not always because of a smart lookahead. Sometimes a seemingly random operation placement decision turned out to be better than a more informed decision. There are several possible ways the parcode generator could be changed. Below we list three such ways.

Our first option for improving our parcode generator is to streamline our program and its functions to a point which enables the scheduler to use a higher degree of lookahead. This assumes that our lookahead mechanism was correct but hadn't gone deep enough. The lookahead tree might, for instance, get pruned by cutting off the trial-scheduling as soon as the cost of the path exceeds the cost of some other already completed path.

A second option in changing our generator is to change the functionality of the lookahead module. Up until now scheduling was a matter of methodically constructing the trial-parcode with the exact building blocks which would later be used. For this reason it was necessary to look at and trial-schedule all operations, even if they had nothing to do with the operation being scheduled. It is tempting to think that this exactness is unnecessary, but abandoning an exhaustive lookahead undermines the whole idea of packing operations onto the parcode. It is nonetheless possible that a heuristic could be found which would allow an informed decision to be made with only a few select operations. It is not obvious how to go about devising such a heuristic.

Our third option is to automatically restructure the sequential code so as to improve its parallelizability. This option is clearly a ways off from being understood.

What can we conclude from our efforts? In the debate between automatic detection of parallelism and use of explicit parallel constructs, it is clear that both sides will win. As has been often said, there are billions of lines of code written in sequential high-level languages. If the aim is to execute these on parallel computers, then the code can and must be automatically translated into parallel code. It is conceivable that the code produced will be code parallelized at the instruction level, but it is unlikely that better than a factor of four speedup can be achieved at that level due to the basic block restriction. Whether large-grained parallelism can do better remains to be seen. Automatic translation into large-grained units starts out with the same dependency constraints as in our instruction level translation. Presumably the highest level of parallelism between the large-grained units can likewise be obtained only when the global problem solving method of programs are better understood and when we can restructure the code in advance of its being parallelized. The great advantage of a larger-grained parallelism is that having a disproportionate amount of overhead is no longer a problem.

Besides looking back at old programs we are also creating new languages and programs and it is there that we have the option of introducing new parallel constructs. We must not think too narrowly when we choose those constructs, for hardware designs will continue to be changing. A good start has been made by the trend towards encapsulating data. As for explicit parallel constructs we think that the large-grain is an appropriate place for them. Program design is a top-down endeavor wherein a problem becomes decomposed into functional, temporal, or otherwise coherent modules. We feel that it should be evident to the programmer at design time which of the modules are concurrently executable. By contrast fine-grained parallel constructs are both tedious and near-sighted. Because programs can be parallelized automatically at the instruction level, we see no reason for explicit constructs at that level. Whatever the grain-size it should be possible for language designers to coordinate on constructs for parallel machines.

There is no doubt that the debate whether parallelism should be implicit or explicit will continue. There is a tradeoff between the amount of work the compiler does and the amount the programmer does, and, more than likely, the programmer will come up with better code given enough time and patience. Perhaps the ideal situation would be to provide options for the programmer to either let the compiler do everything, or let the programmer have complete control, or a third option which might let the compiler make a first pass over the code, and then let the programmer fine tune it, perhaps allowing the compiler to comment on the programmer's changes. In light of the many advances being made in architecture, it is appropriate to try to explore using new programming constructs which take full advantage of those architectures. But the changeover will be gradual, so we should attempt to make the changeover gradual for the programmer. The automatic translation method shown in this thesis is one such attempt.

BIBLIOGRAPHY

Aho, A. V., Sethi, R., and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1986.

Anklam, P., Cutler, D., Heinen, Jr., R., and M. D. MacLaren, *Engineering a Compiler: VAX-11 Code Generation and Optimization,* Digital Press, Bedford, MA, 1982.

Arvind, K. P. and R. A. Iannucci, "A Critique of Multiprocessing von Neumann Style," in *The 10th Annual International Symposium on Computer Architecture Conference Proceedings,* Computer Society Press, 1983.

Chiarulli, D. M., *A Horizontally Reconfigurable Architecture for Extended Precision Arithmetic,* Ph.D.Thesis, Louisiana State University, May 1986

Chiarulli, D. M., Rudd, W. G., and D. A. Buell, *DRAFT: A Dynamically Reconfigurable Processor for Integer Arithmetic,* Technical Report # 84-027, Department of Computer Science, Louisiana State University, Baton Rouge, LA, 1984.

Dennis, J. B., "Data Flow Supercomputers," Computer, Vol. 13, No. 11, pp. 48 - 56, November 1980.

Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures,* MIT Press, Cambridge, MA, 1986.

Fisher, J. A., "Very Long Instruction Word Architectures and the ELI-512," *The 10th Annual International Symposium on Computer Architecture Conference Proceedings,* Computer Society Press, 1983.

Glauert, J. R. W., "High Level Dataflow Programming," pp. 43 - 53 in Chambers, F. B., Duce, D. A., and G. P. Jones, eds., *Distributed Computing,* Academic Press, London, 1984.

Gurd, J. R., "Fundamentals of Dataflow," pp. 3 - 19 in Chambers, F. B., Duce, D. A., and G. P. Jones, eds., *Distributed Computing,* Academic Press, London, 1984.

Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1984.

Karp, A. H., "Programming for Parallelism," Computer, Vol. 20, No. 5, pp. 43 - 57, May 1987.

Kuck, D. J., "High speed machines and their compilers," pp. 195 - 214 in *Parallel Processing Systems,* D. T. Evans, ed., Cambridge University Press, Cambridge, MA, 1982.

Kuck, D. J. , R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Eighth Annual ACM Symposium on the Principles of Programming Languages,* ACM, Williamsburg, VA, 1981.

Lee, G., C. P. Kruskal, and D. J. Kuck, "An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors," IEEE Transactions on Computers, Vol. C-34, No. 10, October 1985.

Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishers, Inc., Cambridge, MA, 1980.