MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1377 July 1993

A Parallelizing Compiler Based on Partial Evaluation

Rajeev Surati

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097, and by the National Science Foundation under grant number MIP-9001651.

A Parallelizing Compiler Based on Partial Evaluation

by

Rajeev J. Surati

Submitted to the Department of Electrical Engineering and Computer Science
on May, 1992, in partial fulfillment of the
requirements for the degree of
Bachelor of Science

Abstract

This thesis demonstrates a compiler that uses partial evaluation to achieve outstandingly efficient parallel object code from very high-level source programs. The source programs are ordinary Scheme numerical programs, written abstractly, with no attempt to structure them for parallel execution. The compiler identifies and extracts parallelism completely automatically; nevertheless, it achieves speedups equivalent to or better than the best observed results achieved by previous supercomputer compilers that require manual restructuring of code.

This thesis represents one of the first attempts to capitalize on partial evaluation's ability to expose low-level parallelism. To demonstrate the effectiveness of this approach, we targeted the compiler for the Supercomputer Toolkit, a parallel machine with eight VLIW processors. Experimental results on integration of the gravitational n-body problem show that the compiler, generating code for 8 processors, achieves a factor of 6.2 speedup over an almost optimal uniprocessor computation, despite the Toolkit's relatively slow interprocessor communication speed. This compares with an average speedup factor of 4.0 on 8 processors obtained at University of Illinois using manual code restructuring of a suite of benchmarks for the Cray YMP.

Thesis Supervisor: Hal Abelson
Title: Professor of Computer Science andEngineering

To my parents, Sudha and Jayantilal, and my brother, Sanjeev.

Acknowledgements

I would like to thank Andrew Berlin for his support and encouragement. He was the originator of the idea for this thesis project.I am most grateful to him for this and the countless hours he spent helping me with it. I would also like to acknowledge Prof. Hal Abelson, my thesis advisor, for being so supportive.

Most of the ideas contained within are the result of an effort to build the Supercomputer Toolkit. Gerry Sussman, Hal Abelson, Jacob Katzenelson, Willy McAllister, Guillermo Rozas, and Andrew Berlin were all involved with that effort and have all been a tremendous amount of help in helping develop the parallelizing compiler. Many of the ideas relating to scheduling discussed herein, implemented by me, are the product of work done by Andy Berlin and Guillermo Rozas. Prof. Sussman contributed the test application for the compiler.

I appreciate the time Prof. Hal Abelson, Andrew Berlin, and Elmer Hung placed in aiding me in writing this document and providing me with valuable advice.

I would also like to acknowledge Brian LaMacchia, Henry Wu , Chris Hanson, Arthur Gleckler, Franklyn Turbak, Vijay Balusubramanian, and Mark Friedman for all the help they have given me in answering my questions about Scheme, Unix, Latex and Computer Architecture.

Hewlett-Packard is to be thanked for donating the computer equipment on which this work was performed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the major challenges faced by supercomputer compilers is the question of how to identify and exploit the underlying parallelism in a computation. Most numerical code has quite a bit of inherent parallelism. However, this parallelism is often not apparent in complex programs where the actual parallelism may be hidden within the quirks of the original source code. Currently, the most widely used methods for extracting such parallelism involve a lengthy combination of profiling computations, identifying processes that can be run in parallel, and *manually restructuring* the original source code to expose the parallelism in the computation. Since these computations are fundamentally parallelizable, however, there must be a way to automatically extract the computations that can be done in parallel. The reason why people have not succeeded with this in the past is that most compilers today optimize based on the *structure* of a program. Basically this means that these compilers attempt to produce the best object code that does most everything the original program does (within limits). The problem is that this method of compilation also reproduces inefficiencies present in the original program. For example, the original source code might contain inefficient methods for creating and manipulating various data structures. These inefficiencies might, in turn, hide parallelism that might be present in the underlying computations. Thus, for many numerical programs, optimization based on the structure of a program is not strong enough to expose the inherent parallelism in a computation.

What is needed instead is a compiler that asks "What are the actual computations being expressed by this program?" and attempts to parallelize the computation based on any inherent parallelism. Partial evaluation is a promising compiler technique that can do just that. Partial evaluation collapses all the data structures and data manipulations in a program into the relevant computations that must be done in order for the program to produce the desired output. Thus, it automatically sifts through the complex data structures of a program, so that it is readily apparent which computations can be done in parallel. Thus, partial evaluation is able to expose the inherent parallelism in a program much more efficiently than ordinary compilation techniques.

This thesis demonstrates a compiler that uses partial evaluation to achieve outstandingly efficient parallel object code from very high-level data independent source programs. The compiler that we implemented attains parallel execution and overall performance equivalent to or better than the best observed results from the manual restructuring of code. Although partial evaluation has been used successfully to compile efficient sequential code for uniprocessor machines, this thesis represents one of the *first* attempts to capitalize on partial evaluation's ability to expose low-level parallelism. New static scheduling techniques are developed to utilize the fine grained parallelism on a multiprocessor machine. The compiler accepts ordinary Scheme programs as source, and generates code for the Supercomputer Toolkit, a parallel computer with 8 VLIW processing nodes. The compiler maps the computation graph resulting from partial evaluation onto the Toolkit's architecture.

On a scientific program written in Scheme that integrates the trajectories of the planets in the solar system, commonly referred to as an n-body problem, the compiler was able to automatically parallelize the computation onto an eight-processor configuration of the Supercomputer Toolkit and achieves a factor of 6.2 speedup over a uniprocessor version which is running code that is executing a floating point operation(FLOP) on 99% of the cycles. The speedup is impressive because the Supercomputer Toolkit has a low communication bandwidth. A value can be transmitted from any one processor every eighth cycle. The latency is also quite high for a statically scheduled architecture. Each transmission has an ALU to ALU latency of 6 clock cycles.

An example of typical speedups for manually restructured(hand optimized) code is given with the Perfect Benchmarks [7]. This set of benchmarks is provided by the Center for Supercomputing Research and Development at the University of Illinois at Urbana Champaign. They report that by manually restructuring there benchmarks and using the Cray YMP compilers, they can achieve an average speedup factor of 4 for an 8 processor Cray YMP over a uniprocessor Cray YMP. The compiler demonstrated here can achieve similar speedups automatically.

By reconstructing the data dependencies of a computation expressed by a program, partial evaluation succeeds in "exposing the low level parallelism in a computation by eliminating inherently sequential data-structure references." [5] This is crucial for the the exploitation of parallelism across a multiprocessor. Partial evaluation eliminates all of the data independent conditional branches in a program and thus produces huge sequences of easily parallelizable straight-line code [3]. A basic block is essentially a sequence of operations in a computation that must be executed once the sequence of instructions is entered. These huge sequences of straight line numerical code would be considered basic blocks. The large blocks produced by partial evaluation are several thousands of instructions long. In more traditional compilers, basic blocks are normally 10 to 20 instructions long. Huge blocks are important because their predictability makes them easy to parallelize.

On multiprocessor systems, basic blocks are usually executed serially because they are usually quite small. To properly exploit the fine-grained parallelism available in a large basic block, the basic block should be scheduled across a multiprocessor instead.

The partial evaluation parallelization technique is compared with other more traditional optimization methods like trace scheduling and software pipelining. Since the technique can eliminates sequential data-structure references which the other methods do not take advantage of, it can only serve to enhance the already excellent performance of the traditional methods.

Presented in the following chapters are the methods of construction and results from utilization of this compiler in the context of the Supercomputer Toolkit. Chapter 2 begins by discussing the general structure of the compiler. It then describes each of the elements of the compiler in greater detail. Chapter 3 presents the manner in which the compiler takes advantage of the Supercomputer Toolkit Architecture. Chapter 4 presents the experimental results of the compiler on some scientific applications related to the n-body problem. Chapter 5 compares and contrasts this novel compilation technique against other common techniques. Finally, in Chapter 6, the conclusions of this thesis are presented along with suggestions for future work.

Chapter 2

The Compiler

The compilation process has four phases: partial evaluation, division into regions, assignment of regions to processors, and the scheduling of instruction. This process is depicted in Figure 2-1.

The Scheme source program must represent a computation which is data-independent. The computation may not change based on the input data. Computing a croos product is an example of a data-independent computation since the computation remains the same even though the input vector data may change. The partial evaluator produces a data dependency graph that represents a computation at the operator(+, - , * , sqrt, etc...) level. The data dependency graph is too fine grained to divide on a node-by-node basis because of the communication latency. Its granularity is a little greater than one cycle per operation whereas a communication takes slightly more than six cycles. The granularity is made slightly coarser by dividing the graph into regions which couple computations which should occur on the same processor because of the communication cost. The region dependency graph is then divided amongst various processors using a graph multisection technique similar to list scheduling. Finally the individual regions are scheduled at the instruction level onto the architecture to form the parallel object code for the Toolkit. The inner workings of each of the phases is presented in the rest of the chapter.

2.1 The Partial Evaluator

The partial evaluator is used to eliminate data abstractions and compound data structures at compile time. This leaves only the numerical computation data dependency graph. It also results in an order of magnitude speedup of scientific codes [5]. The partial evaluator utilized by this compiler was written by Andrew Berlin. A more thorough discussion of the partial evaluator is contained in [5].

Berlin accomplishes partial evaluation through a technique that uses placeholders to propagate intermediate results. The placeholders are also used to represent data which is not known at compile time in the input data structures. It is then possible, by using these placeholders in the place of actual data, to symbolically evaluate the computation with respect to the input data. An operation is computed if the input data is actually available. Otherwise, a new placeholder is created to symbolically represent the result of that computation and the evaluation may proceed. A data dependency graph of the computation is constructed by keeping track of all the operations which are performed on the data and the intermediate values. A simple example[1] to illustrate this follows:

[1] This example appears in [5]

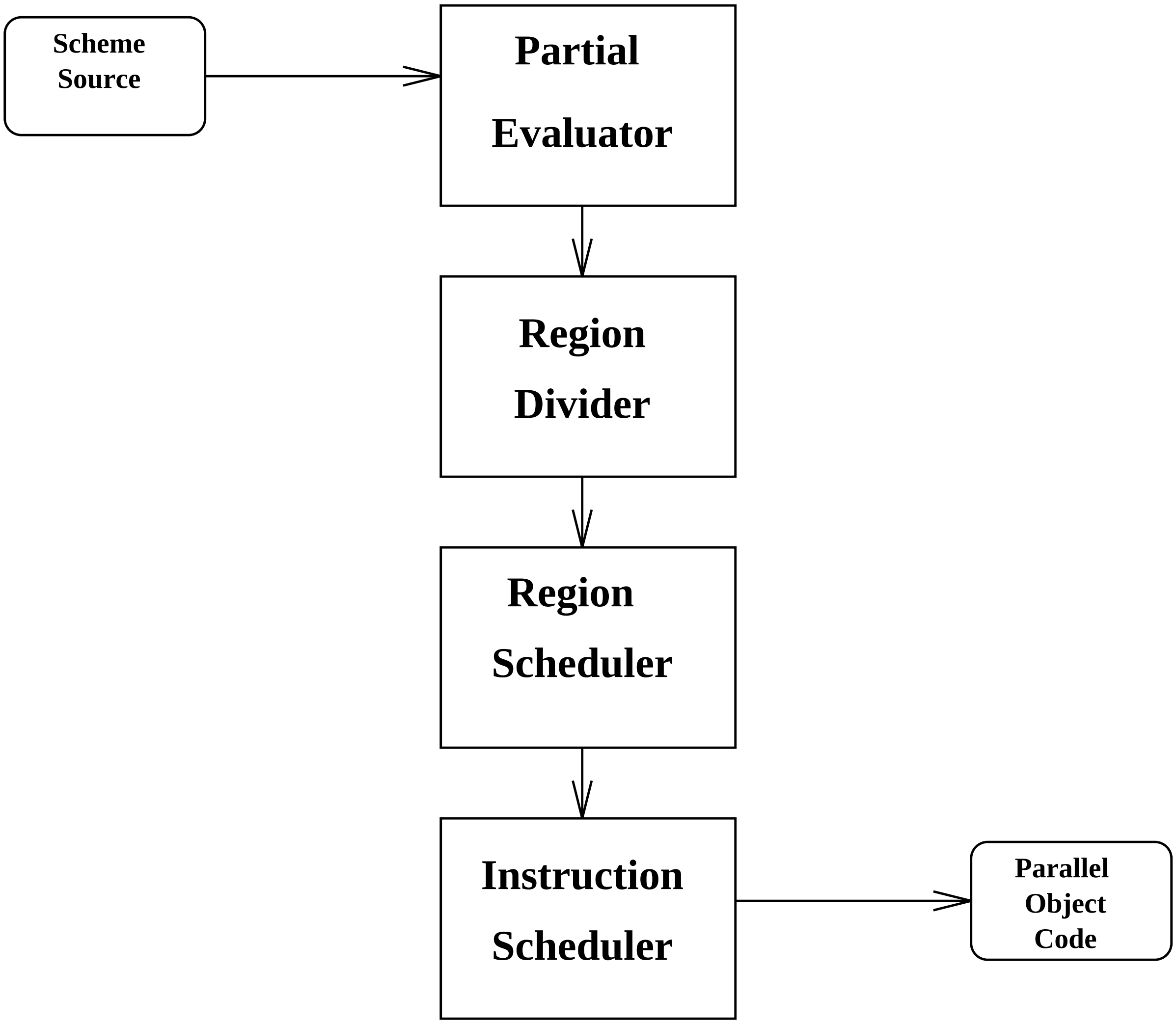

Figure 2-1: Four phase compilation process that produces parallel object code from Scheme source code.

```
(define (square x)
        (* x x))
(define (sum-of-squares L)
        (apply + (map square L)))
```

```
(sum-of-squares (list (make-placeholder 'a)
                      (make-placeholder 'b)
                      3.14))
```

In the above code the sum of the square of three numbers, one of which is known, is computed. The data dependency graph of the computation that is produced by the partial evaluator is shown in Figure 2-2. The partial evaluator eliminates the data abstraction and reduces the computation to the minimum number operations necessary: two adds and two multiplies(3.14 is a known input, its square is computed at compile time.)

In addition to producing the computation's data dependency graph, the partial evaluator employs a number of other optimizations that are now possible because of the elimination of data structures. Examples of this are dead code elimination and constant folding. Dead code elimination removes operations from a computation if they do not contribute to the net result of a computation. Constant folding might reduce an expression like[2]

```
(* 10 x 5)
```

to:

```
(* 50 x)
```

[2] In Scheme a multiplication with multiple arguments is commutative.

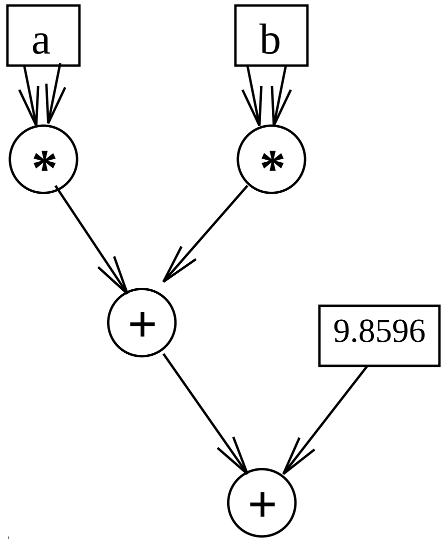

Figure 2-2: The data dependency graph of a computation which takes the sum of the squares of three numbers, one of which is 3.14.

The end result of all of the partial evaluation is a data dependency graph which represents the actual numerical operations needed to compute the results based on the input information and program presented to the compiler.

2.2 Region Division

The cost of communications on the Supercomputer Toolkit is effectively six clock cycles. The data dependency graph's granularity is such that most instructions are computed in one cycle. The granularity is too fine, because it is not implicit that some operations should be computed on the same processor.[3] In order to make such things implicit, a coarser grain graph called a region dependency is created. Operations in the data dependency graph are collapsed into regions. A region is a computation which ends with a transmission. The only things that should be transmitted are

[3] One attempt at addressing this issue is discussed in [10].

values which are inputs to more than one operation.

A simple algorithm creates a region dependency graph from a data dependency graph. A region ends in an operation whose result is used by more than one other operation. A region has only one such operation. A course grained region dependency graph may be created out of a data dependency graph by simply labeling each operation node in the data dependency graph as a region and then combining each of the operations (temporarily labeled as a region) with a single dependent into the region of that dependent. This leaves a region for each operation that either has multiple dependents or results in an output. output. Each region has dependencies on regions that contain operations that the operations the region encompasses have dependencies on.

An example is shown in Figures 2-3 and 2-4, where the data dependency graph for the following code is shown and turned into a region dependency graph.

```
(define (simple-example A B C D)
        (let ((E (/ B C)))
             (* (- (* A B) E)
               (+ D E))))
```

The algorithm places the multiplications, additions and subtractions into one region. The division operation is placed into another region because multiple operations are dependent upon it. The granularity of the graph is made closer to the desired coarseness through region division.

2.3 Region Scheduling

After the data dependency graph is collapsed into a coarser grained region dependency graph, it is possible to schedule the regions onto a multiprocessor. This is the

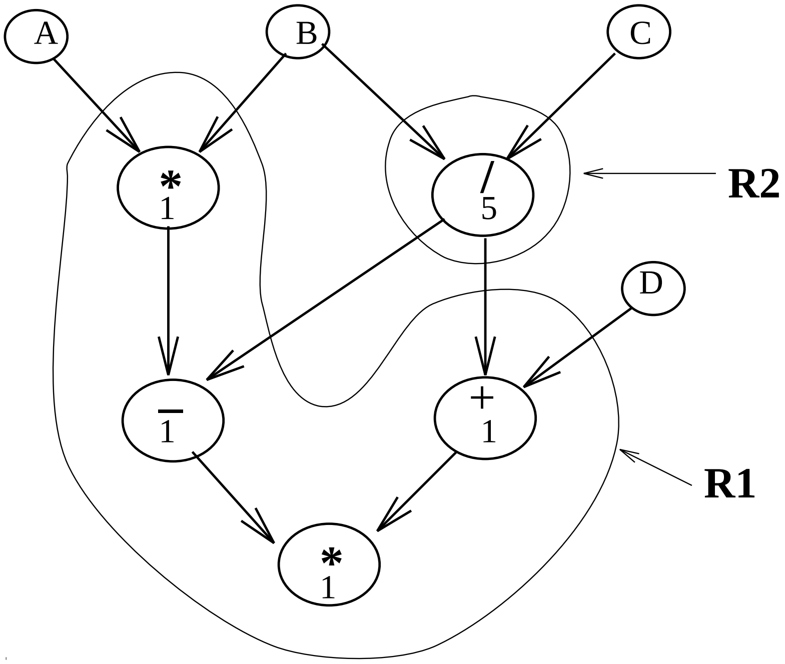

Figure 2-3: A data dependency graph for simple-example with its regions circled.

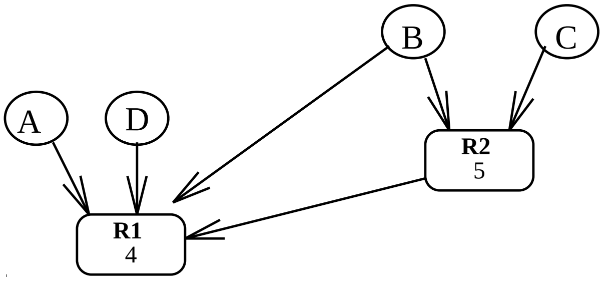

Figure 2-4: The region dependency graph representation of this data dependency graph.

traditional multiprocessor scheduling problem of scheduling tasks on processors such that execution time is minimized. This is known to be a “strong” NP-hard problem [16]. Purely heuristic methods are justified on such a problem, as long as they do well on average. The heuristic used here relies on a critical path weighting Scheme and very akin to list scheduling. There are two steps to this heuristic:

1. Each region is assigned a weight which is the latency of the longest path from the region to the regions which end the graph. This is the sum of the latencies of the regions along that path. The latency of the region is the sum of the operations it contains, since they will all occur on the same processor.

2. Schedule the regions

    - If there are no more regions to be scheduled, quit.
    - Compute the ready regions and order them by weight. The ready regions are the ones that are not only ready to be executed, but have a weight that is approximately equivalent to the weights of the the regions ready to execute with the largest weights.
    - If there are more ready regions than processors not executing a region, take a processor and schedule the region which requires the least amount of communication to execute on that processor.
    - If there are less regions then processors, schedule the region on the processor on which it requires the least amount of communication to execute.
    - Continue scheduling.

The communication cost of a region on a processor is the number of regions which that region is dependent on whose results are not in the processor’s memory.

A set of regions ordered in sequence of execution is produced for each processor. When a region’s result value has been computed, it is necessary to transmit the value to the other processors which have regions waiting to be executed dependent on this result. If all of the dependent regions happen to be on the same processor, the transmission is unnecessary. Otherwise, it will cost six cycles to transmit the result to the other processor.

The next step is to schedule the individual instructions within the regions themselves. Before going into explicit detail about the scheduling of instructions, an assumption made during region scheduling must be made clear. The assumption is that a region's resultant value that is transmitted will be available as soon as it is computed. To closely approximate this, the transmissions have the highest priority in scheduling. As soon as an operation that produces a value that should be transmitted is scheduled, the transmission is immediately scheduled on the earliest cycle possible.

2.4 Instruction Scheduling

The instruction scheduler maps instructions in each scheduled region onto each processor at the instruction level. In the case of the Supercomputer Toolkit, Very Long Instruction Words(VLIW) must be generated for each processor. This task is not trivial, since it requires the scheduler to order the numerical operations onto the architecture so that the total execution time is minimized. This is tough to do because the ordering of operations can effect the number of cycles necessary to complete the program. For example, suppose there is a value required by several other operations on other processors. The later the value is produced, the later the other operations can occur. This can be a big problem on a parallel processor since it is possible that a processor will waste cycles while waiting for one of these values. Another effect is more subtle. The registers in a machine are used to store temporary results. The more often a particular value in a register is used while it is there means fewer loads and stores may be necessary from and to memory, thereby reducing the chance that the processor will become idle waiting for memory transactions.

Most compilers for VLIW machines attempt to minimize execution time by considering either of the issues mentioned above, but not both simultaneously. The instruction scheduler deals with both of these issues through operation reordering and a technique for register allocation that attempts to minimize memory references. Two phases of scheduling are required. During phase one an instruction ordering is suggested and a plan for register use is created for the minimization of instruction references. During phase two the plan developed in phase one is followed, and instructions are reordered to better match the architecture.

During phase one, an instruction ordering is generated within the bounds of the region imposed ordering. Regions couple computations which contain intermediate results which will be used only once. This is because the operations encompassed in a region have a single dependent. Placing these instructions close together in the code is good because it guarantees that the intermediate values of each region will never have to be stored and loaded to memory. The ordering goes a long way toward minimizing instruction stores and loads as it is and is a good first order solution to the problem.

Traditional register allocation is performed during phase one. The region ordered instruction ordering is followed precisely without any consideration being made to a pipeline or other architectural specific features. Register instruction groups are created which indicate what instructions were scheduled that use a value while that value was in a register. Each time the value is placed into a register, a new register instruction group is added to that value's set of groups. The groups are used by the second phase to determine which registers are free to use on a given cycle as well as which register has the value whose earliest use is farthest in the suggested instruction ordering. This is useful when determining which register to place a value in when all the registers are occupied by other values that are needed by operations still waiting to be executed. The instructions groups are the plan that is followed to load and store registers during phase two. The stores are known as register "spilling."

An example of an instruction register group might be helpful. Suppose B is a result which is an input operand to three operations numbered 20, 21, and 300 (where a greater number implies the later it should be scheduled) Suppose B is placed in a register on cycle 19 and is spilled during the register planning allocation in phase one between instruction 21 and 300. The instruction register instruction groups for B would be {20 21} and {300}.

Phase two takes phase one's instruction ordering and optimizes it for the architecture. It schedules in the suggested instruction ordering, looking ahead only when the instruction that should be scheduled according this ordering is not ready to be executed on that cycle. This reduces execution time because it fills in what would have been NOPs(No Operation) cycles. There are two reasons there might be empty NOPs. One reason is that the dependencies of regions may require such a delay. The other is that architectural issues like pipelining may leave a result inaccessible for a cycle and this wasn't a consideration in phase one.

The advantage can be explained better with an example that show one way phase two is able to optimizes. In Figure 2-5 are four regions which are part of a larger region dependency diagram. The regions are being scheduled onto a two processor configuration over the course of 8 cycles. R4 is composed of three instructions which each take a cycle to execute, one is dependent on R1 and the others on R2. Figure 2-6 shows the schedule if the instructions were scheduled in exactly the ordering imposed by regions, since the other two instructions in R4 are dependent only on R2 finishing, they may be executed in the two free cycles after R2 finishes, thereby possibly reducing the execution time on processor 2. Figure 2-7 shows this optimization. Thus it is

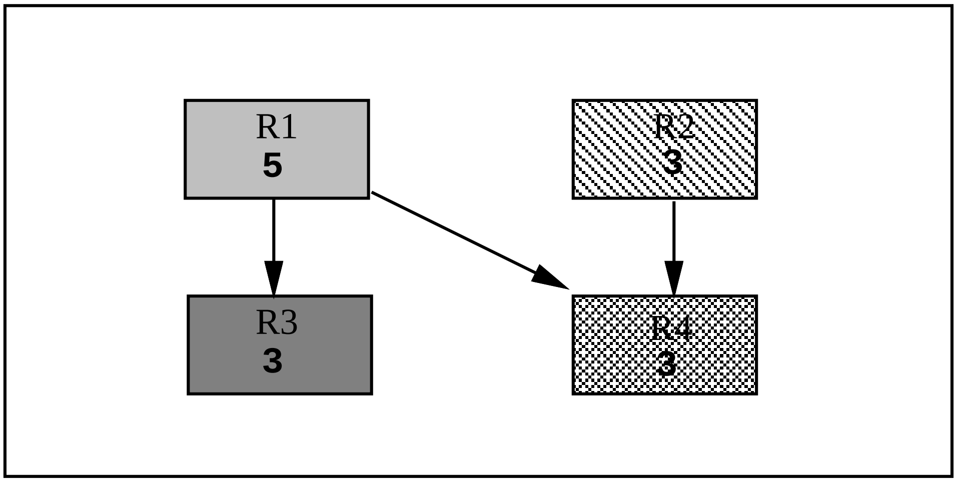

Figure 2-5: A portion of an region dependency graph with 4 regions.

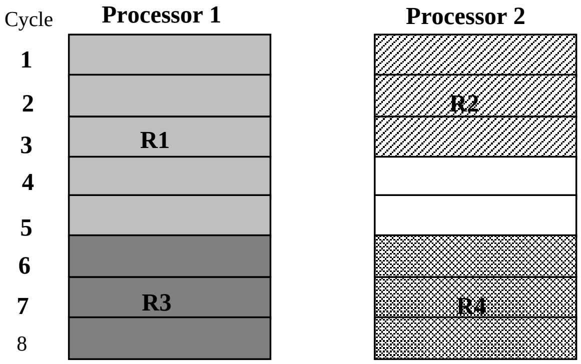

Figure 2-6: The instruction schedule if the region ordering is maintained.

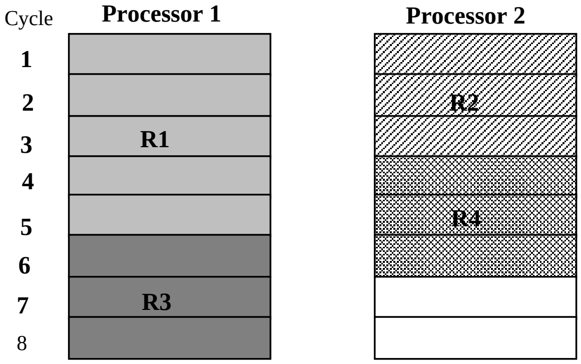

Figure 2-7: The instruction schedule if lookahead is used.

easy to fill in these NOPS with instruction further down in the ordering.

When instruction reorderings occurs, the planning contained in the register instruction groups becomes useful. Register spilling is scheduled for all values which the the register planner spilled. This means a value is immediately stored in memory as soon as it is produced if it was spilled in phase one's preallocation. Thus, any spilling that occurs in excess to this is due to the instruction reordering and the values it produces. Register groups provide a means to figure out which of such values in the registers should be spilled. The other register are filled with values which are intended to be there by the phase one allocation and that should will remain there. It is only the remaining registers from which a value must be spilled. This can be done because the register groups of a value are dynamically updated to reflect the execution of an operation each time an operation is performed The register groups thus contain up to date information about when an operand will be needed in the phase one ordering. Instruction executed out of order are eliminated from the groups as soon as they are executed. Thus, one can spill the register that is used the latest in the old instruction ordering of the remaining instructions to be executed. In these cases, two memory cycles are lost(one to store one to load). At worst one NOP is caused because of this loss of memory cycles and the gain of a FLOP cycle from having prescheduled the operation that produced this value is lost. It may mean that a memory operation is gained because some other instructions using that value have already occurred while that value was in a register. One less reference to the value is made and this allows a register to become free sooner than it was in the region ordered instruction. Execution time is thus shortened by taking advantage of the holes in the region imposed instruction ordering that was used in an effort to try to minimize memory references.

2.5 Summary

In this section a method of parallelization based on partial evaluation was presented. The method's compilation process results in some highly compacted parallel object code that executes a basic block across a parallel computer to try and take advantage of fine grain parallelism.

Chapter 3

The Supercomputer Toolkit

The purpose of this chapter is to provide an overview of the Supercomputer Toolkit so that the compilation results may be understood. The Supercomputer Toolkit is not a general purpose computing machine. It is optimized heavily for the static and data-independent nature of numerical problems. Thus, the Toolkit has no operating system and is a backend processor for a workstation, much like WARP [6]. The Toolkit is an 8 processor MIMD machine. It is composed of eight separate VLIW processing nodes. A thorough explanation of the technical details of the Supercomputer Toolkit may be found in [1] A detailed explanation of the compiler's view of the toolkit processor boards , the interconnection network, and the synchronization mechanism follows.

3.1 The Toolkit Processing Nodes

Figure 3-1 shows the architecture of each processing node. It is symmetric and designed to take advantage of a lot of instruction level parallelism. Each node has a 64-bit-floating-point chip set, a five-port 32x64-bit register file, two separately addressable data memories, two address generators for those memories, two I/O ports, a

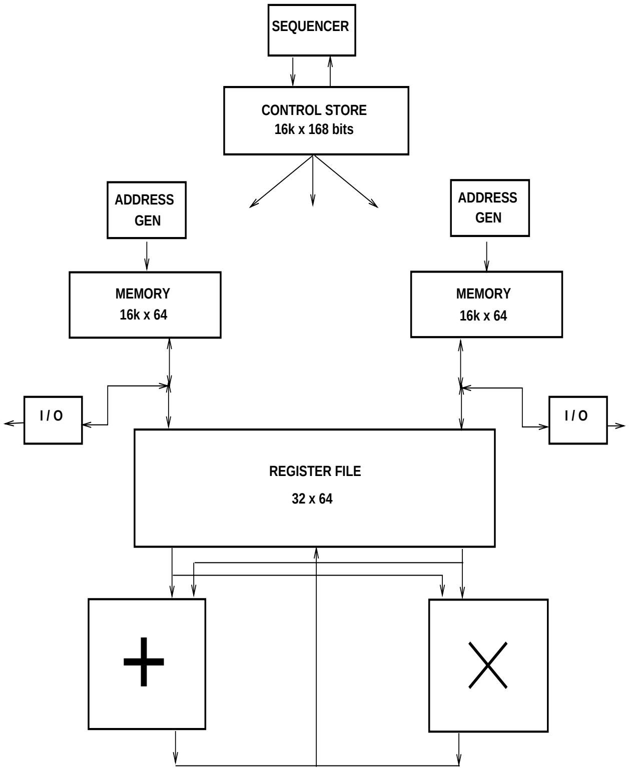

Figure 3-1: This is the overall architecture of a Supercomputer Toolkit processor node, consisting of a fast floating-point chip set, a 5-port register file, two memories, two integer alu address generators, and a sequencer.

sequencer, and a separate instruction memory. A Toolkit Processing Node is pipelined and thus capable of executing the following instructions in parallel: a left memory-I/O operation, a right memory-I/O operation, an FALU operation, an FMUL operation, and a sequencer operation, all on a single clock cycle. The Toolkit is completely synchronous and clocked at 12.5 Mhz. When both the FALU and FMUL are utilized, the Toolkit is capable of a peak rate of 200 Megaflops, 25 on each board. The compiler as it is currently written can only harness 1/2 of this capability because it utilizes either the FMUL or FALU, but not both, on any cycle. When the compiler is used, the peak computation rate is 100 Megaflops

The compiler's interpretation of the 32 register file is that 26 are available for scheduling computations. An additional two of the registers are reserved for communication purposes. The remaining are reserved for hardware purposes and are thus unavailable.

The floating point chips can compute many different functions, the ones utilized by the compiler are:

FLOP	Latency
+	1
-	1
*	1
/	5
sqrt	9

The floating point chips have a three stage pipeline whereby if an operation is scheduled on cycle N, the result must be latched on cycle N+L(where L is the latency of the computation) and can then be placed in a register on any of the following cycles up until the the next latch on that 1/2 of the chipset. There are feedback paths for the chips which allow operands produced while in the pipeline to be fed back in on the next cycle. The compiler takes advantage of these feedback mechanisms and finds them particularly useful for the intermediate values which have only one dependent. If the path is utilized no register needs to be used to store the value. This can save memory cycles.

A single basic block is scheduled by the compiler. This means there is no control flow. Thus the compiler can simply schedule sequencer instructions which increment the program counter on each node.

Since partial evaluation eliminates data structures in a computation, the only way to address a value is its memory location on a Toolkit Processing Node. Thus the address generators are simply used to generate the hard coded addresses for these values on any instruction.

The compiler's notion of memory management is simply to put the inputs and constants of a computation at the bottom of memory. There are copies of them on both sides making it easier for these values to be accessed as there are thus two paths for a value to the register file. Everything above the constants and inputs are intermediate values and outputs. Spills due to phase one scheduling alternate between memories. It should also be noted that on any one side, either a memory load or store, or an I/O transmission or reception on any one cycle may be scheduled.

3.2 Interconnection Network and Communication

The toolkit allows for flexible interconnection among the boards through its two I/O ports. The interconnection scheme is not fixed and many configurations are possible. The compiler, however, currently views this network as two separate buses: a left and a right bus. Each toolkit is connected to these buses through its left and right I/O ports. This configuration was chosen given the number of processors as a reasonable network to evaluate the compiler on.

Here is an example of the statically scheduled communications transactions that are possible on the toolkit. A value is sent from Processor **A** to Processor **B** on clock cycle 1. Processor **B** will execute an instruction that receives that value on clock cycle 3. Thus, the latency of any communication, once it is sent, is always 3 clock cycles. During the interim cycle(2) when the transmission is sent no other transmission on that bus may occur.

The compiler does all of the static scheduling and operates within the constraints of the toolkit. It also adds the extra constraint of storing all of the values that are transmitted immediately after the value is received This ensures that the register allocation and instruction scheduling strategies are not interfered with by communication. The latency of a communication is thus effectively 6 cycles from ALU to ALU. It take 6 cycles from the time a values is produced, put in a register, and sent on the bus until it is available in one of the computation registers of another processor. Also, because there are 8 processors and two busses that each take two cycles to transmit over the effective bandwidth available to a processor is one send every eight cycles. This is an extremely low bandwidth machine.

3.3 Synchronization

In order to coordinate processors to execute a basic blocks within the constraint of synchronized instructions, a mechanism is necessary to get the processor to operate in lockstep. The processors are operating on a single global clock, this does not guarantee that they are operating in lockstep however. They need to be synchronized precisely so the static transactions with implicit send and receive protocol will work. The toolkit provides a global flag and subroutine that allows the boards to be brought into lockstep. The compiler uses this mechanism to get the processors operating in lockstep at the start of the basic block. Since the blocks are so large, any cycle wasted on synchronization are statistically irrelevant.

3.4 Summary

A detailed description of the Supercomputer Toolkit hardware and its capabilities as utilized by the compiler was presented. In the next chapter the result of using this compiler for the Supercomputer toolkit are illustrated on the n-body problem.

Chapter 4

Experimental Results

The performance of the compiler has been evaluated on the Supercomputer Toolkit by compiling two scientific applications. These two scientific applications are simulations of the n-body problem. The compiler is able to achieve substantial speedups despite the low bandwidth interprocessor communications of the Toolkit. In this chapter, I present the theoretical parallelism possible for each application and the compiler measured exploitation of that parallelism on the Supercomputer Toolkit. The region scheduling compiler technique though suitable for small multiprocessors is shown not to scale well.

4.1 The n-body Problem

The n-body problem is the computation of trajectories of n particles with each particle exerting $\frac{1}{r^2}$ central force on each of the other the bodies. Numerical simulation of the n-body problem is important for a number of research applications [2]. Though the two application represent simulations of the same problem, they represent significantly different numerical computations. This is because they utilize two different numerical integrators. One integration method is known as Stormer and the other as Runge Kutta. They both represent data independent computations. Both applications calculate the positions of planets in the solar systems. Thus, the masses of the bodies are known at compile time. The programs are essentially integration steps that need to be iterated over and over again. Each integration step produces new positions and velocities of the planets which are then used as inputs for future steps. Simulations that take hundreds of hours of CPU time are often performed using programs like these.

4.2 Theoretical Parallelism

A parallelism profile of a 9 body stormer integration and a 9 body 4th order Runge Kutta integration are shown in Figures 4-1 and 4-2. Both figures represent the maximal parallelism in these problems. They show how quickly the computations could be computed if there were an infinite number of processors , infinite communication and memory bandwidth, and instantaneous communication among processors. Because the number of processors utilized on each cycle is greater than 10 in these profiles, there is plenty of underlying fine grain parallelism in the actual computation that could be exploited by this compiler on an eight processor machine like the toolkit. The major difference between the parallelism profiles of the two computations, is that the Stormer integration has substantially more parallelism available at the start of the computation.

4.3 Results

Four different computations have been compiled in order to measure the performance of the compiler: a 6 body stormer integration(ST6), a 9 body stormer integration(ST9), a 12 body stormer integration(ST12), and a 9 body fourth order Runge Kutta integration. The speedup measured is the single processor execution time of the computation divided by the total execution time on the multiprocessor. The number of single processor cycles are compared with the eight processor number of cycles in Table 4.1 along with the number of NOP cycles and the efficiency of utilization. Because of the partial evaluation, the single processor efficiency figures are extremely close to optimal.

Program	1 Processor cycles	NOP Cycles	Single Processor efficiency	Eight Processors cycles	Speedup
ST6	5811	16	99.7 %	954	6.1
ST9	11042	32	99.7%	1785	6.2
ST12	18588	32	99.8%	3095	6.0
RK9	6329	15	99.7%	1228	5.2

Table 4.1: Table of Speedups of applications running on 8 processors

Such efficiency indicates that the speedup measurement shows precisely the gain in actual floating point computation by scheduling onto a multiprocessor like the Supercomputer Toolkit. The gain due to these techniques which automatically parallelized the computation are very much in line with what one expects when running computations on an 8 processor machine like the Supercomputer Toolkit.

Figures 4-3 and 4-4 show the speedups that were attained on toolkit configurations with different number of processors. As indicated above, the speedups are fine for an eight processor machine since the graphs seem to show reasonable gains up to about eight processors. It is clear in the graphs that the scheduler is not doing too well for more processors than that. There are two reasons that account for this drop off. One is that the Supercomputer Toolkit has an extremely low interprocessor communication bandwidth. The other reason is that the region scheduling does not scale well beyond eight processors.

Bandwidth is a problem because the amount of communication necessary tends to increase as the computations are spread out over more processors. With a bandwidth such that of each processor is only allowed a send every eight cycles, the speedups are very impressive. To address the bandwidth issue, bus utilization data was collected for all the programs. The results are shown for ST9 in Figure 4-5 and are characteristic of the other programs. The bus utilization measurement indicates the percentage of cycles the buses are busy. It is the sum of the cycles that each bus is busy divided by the twice the total number of cycles executed (twice because there are two buses). The bus utilization graph coupled with the speedup graph of this computation suggest that the two bus architecture is indeed quite inadequate after about 10 processors. If the busses are utilized more than 90% of the time there is an extremely high probability that sends which were instantaneously scheduled by the region scheduler are being delayed a lot. This is bad because the region scheduler assumed instantaneous communication. In the bus utilization diagram, the drop off in speedup occurs when there is about 70% bus utilization. Interestingly, in the data for the RK7 and ST6 and ST12 this is also true. This may suggest that 70% utilization makes the bus busy enough so that transmissions suffer from longer delays until transmission than when less processors were being scheduled.

Another problem is that region scheduling does not seem to work well for more than eight processors. The region scheduler partitions the regions and turns a 6329 cycle RK9 into 854 cycle RK9 on an 11 processor ideal machine(ideal because it has instantaneous communications). This is a very big problem because that 854 cycles represents the best that can be done by the region scheduler if all the values are available as soon as they are produced. That is only a factor of 7.8 speedup for 11 processors. There is more parallelism available than that. This can be seen quite clearly in Figure 4-1. Luckily, the instruction scheduler is able to reorder the instructions suitably such that the effect is reduced and RK9 is turned into a 780 cycle computation. Nonetheless, for more processors than eight, the region scheduling doesn't seem to work well. It is unable to extract the fundamental parallelism as demonstrated by these computations far below where it should for more processors.Compiling for larger computers than the Toolkit this could be a very big problem.

4.4 Summary

By compiling two applications it has been shown that the compiler is more than adequate for compiling basic blocks on an eight processor machine like the Supercomputer Toolkit. The compiler, however, has difficulty on larger multiprocessors. There are two things that lead to this difficulty: the architecture imposed low bandwidth communications and the inability of the region scheduling method to work well on larger multiprocessors.

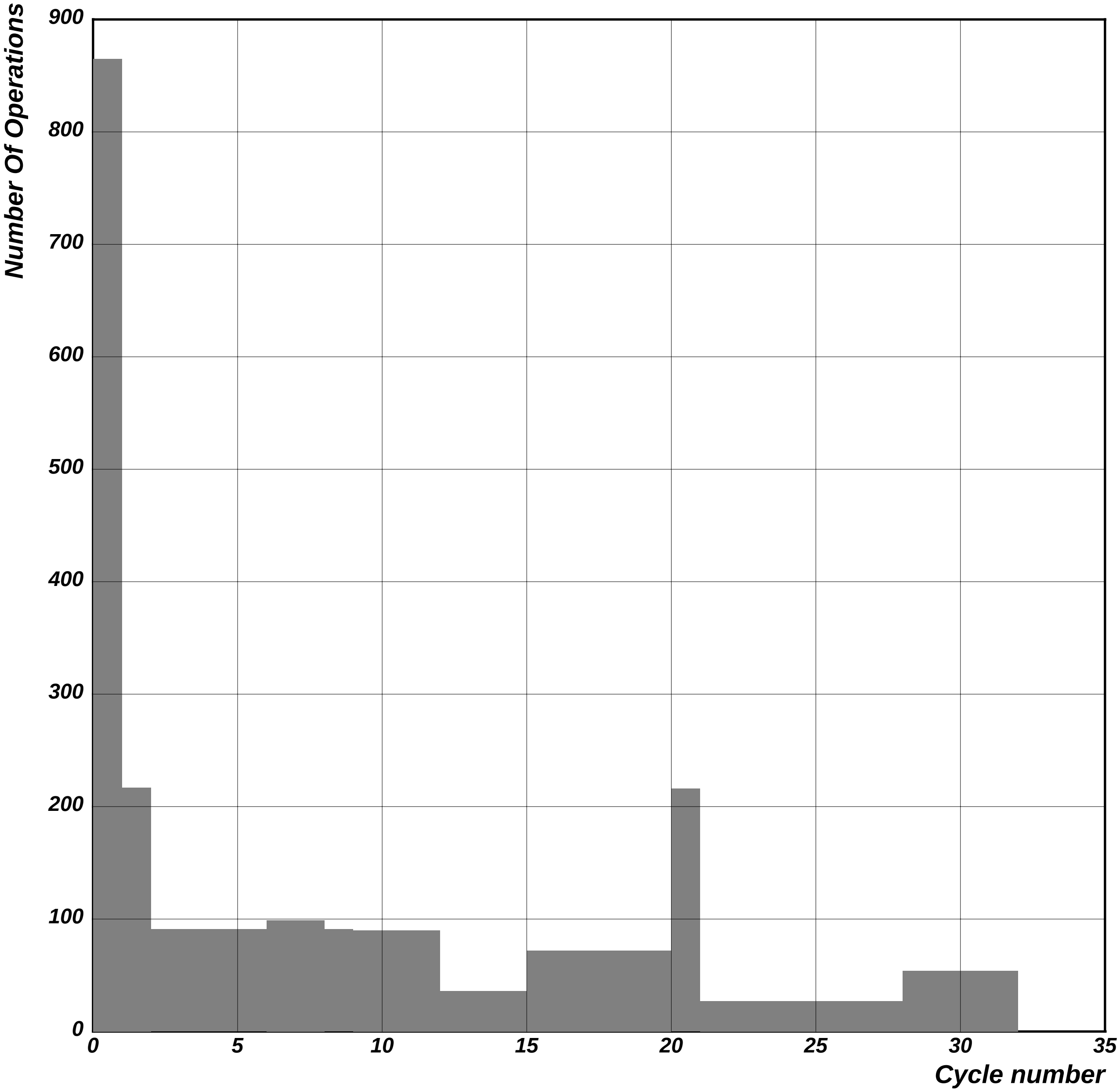

Figure 4-1: Parallelism profile of a 9 body Stormer integration[4]

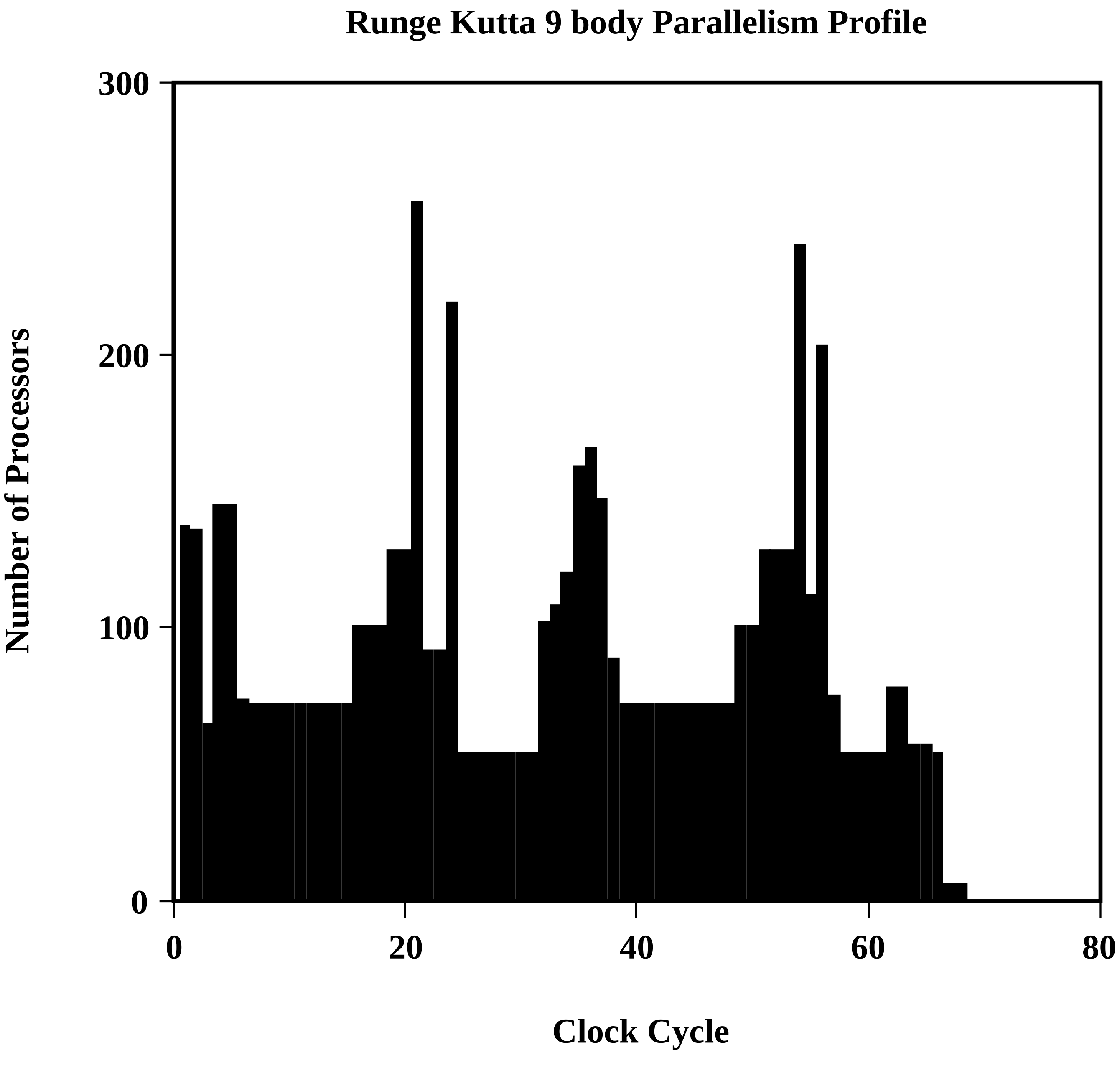

Figure 4-2: Parallelism profile of a 9 body 4th order Runge Kutta integration

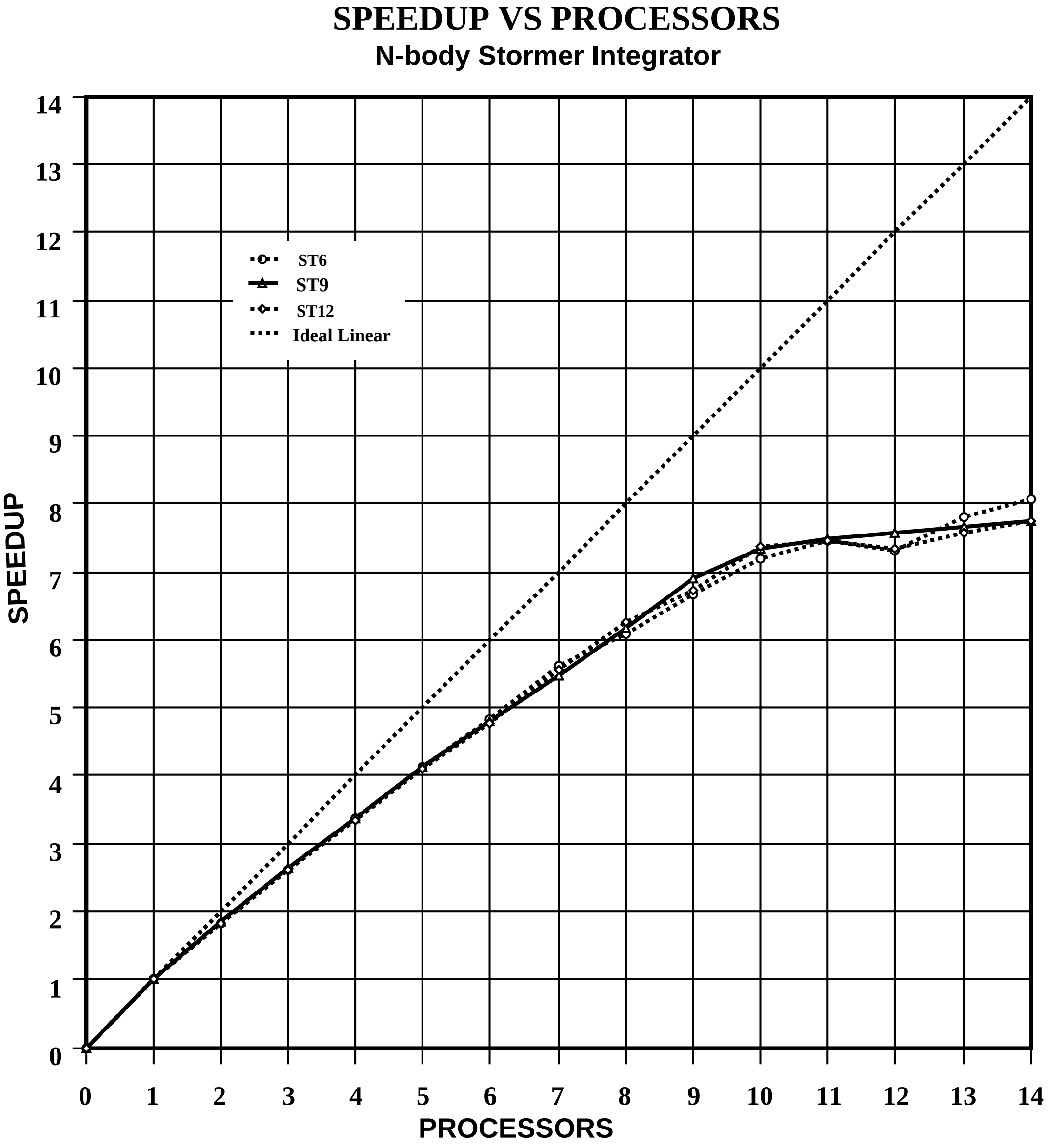

Figure 4-3: Speedup graph of Stormer integrations.

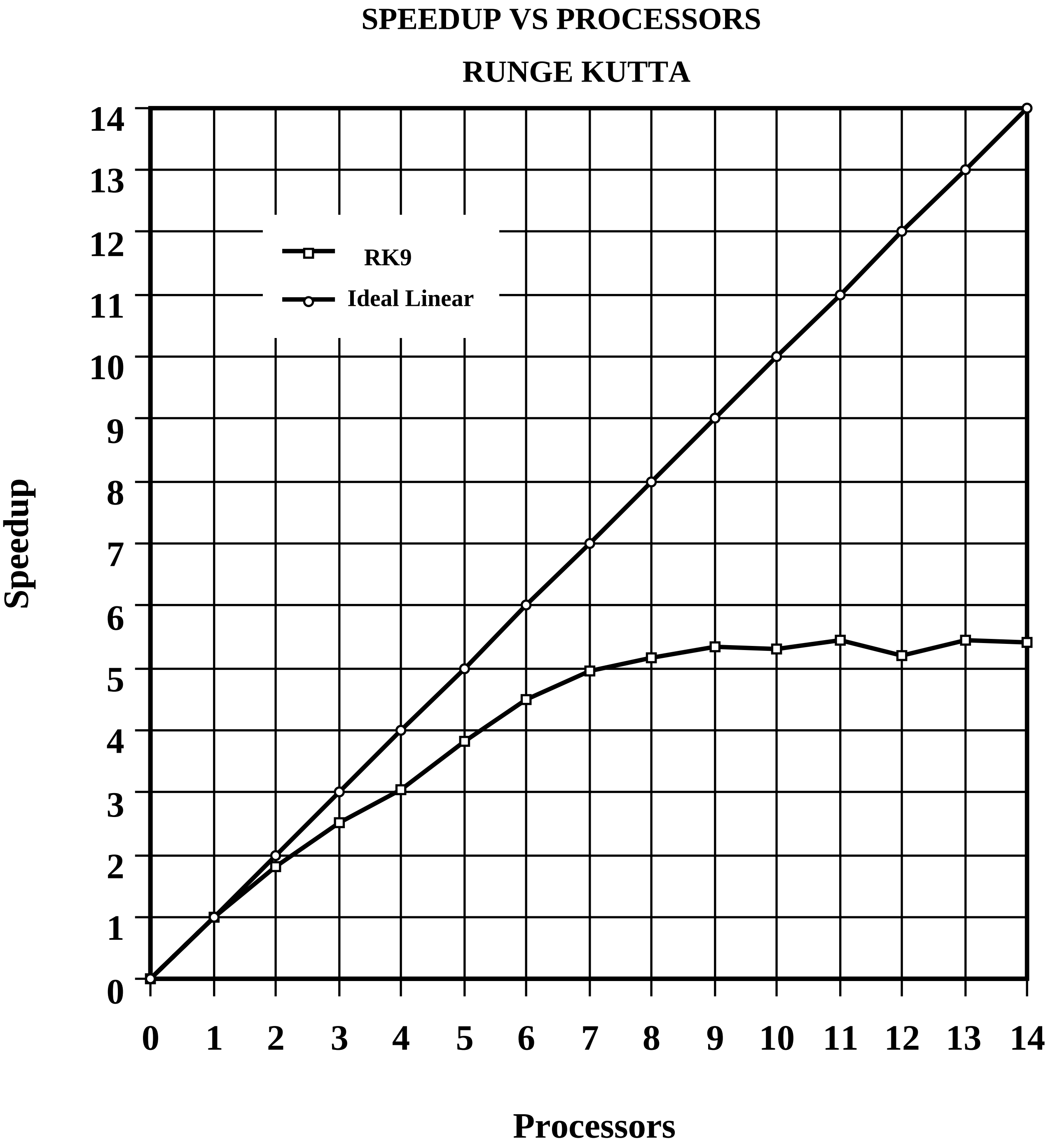

Figure 4-4: Speedup graph of Runge Kutta integrations.

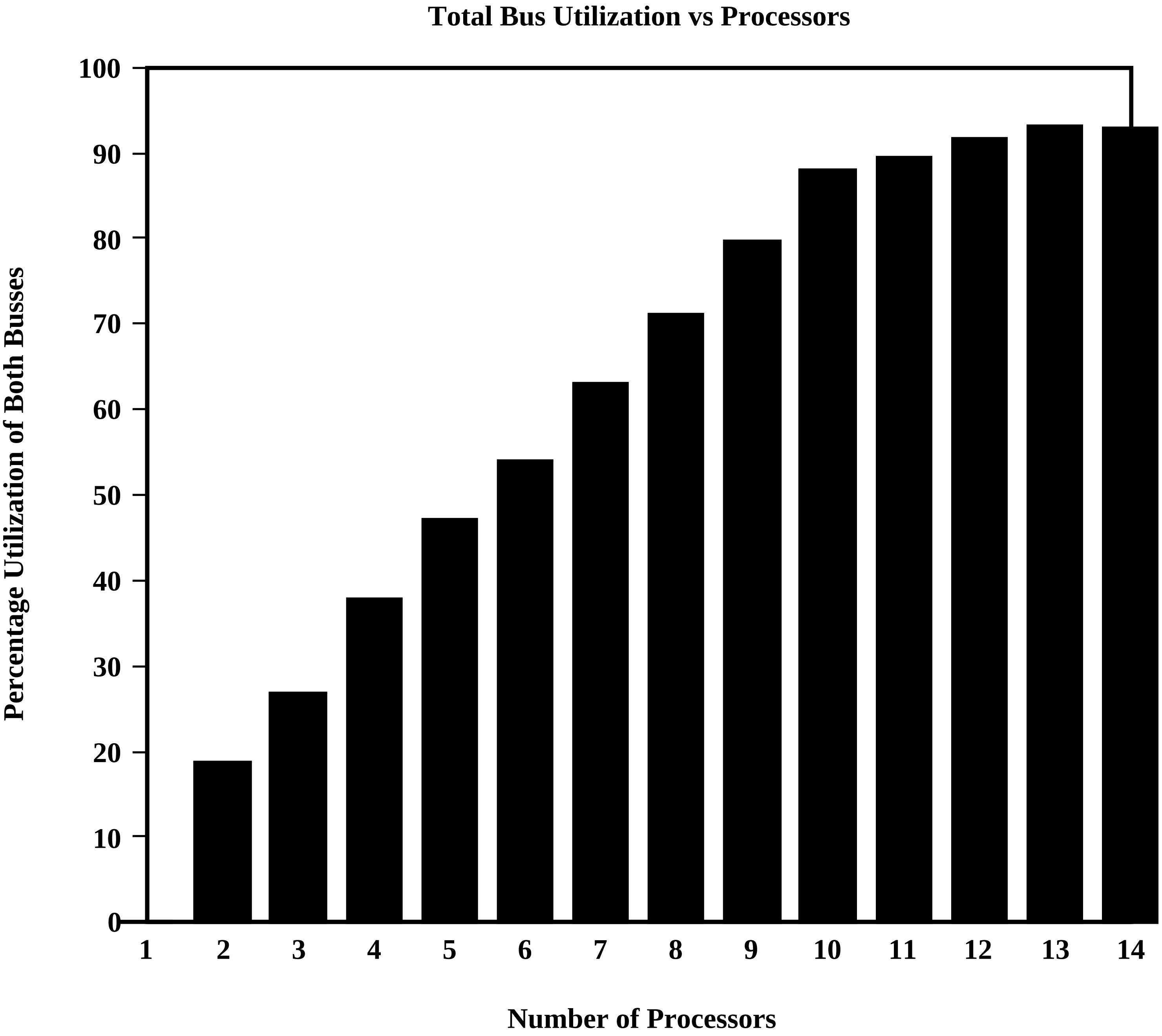

Figure 4-5: Bus Utilization vs Processors for ST9.

Chapter 5

Comparison With Other Work

This compiler's approach to parallelizing numerical programs is fundamentally different from the approach taken by other compilers. This compiler specifically optimizes the computation contained within a program. Other compilers are more general and are designed to optimize the execution of the program. In order to put this work into perspective, five different approaches including trace scheduling, software pipelining, vectorizing and iterative restructuring are all compared and contrasted with this compiler's methodology.

5.1 Trace Scheduling

Trace scheduling [9] is a popular technique used by parallelizing compilers. The technique creates traces of the most frequently used path of basic blocks in the control structure of a program. The basic blocks are typically on the order of 10 to 20 instructions. Run time information that keeps track of the various traces through the program is used to determine which trace should be optimized. This trace is then heavily optimized as if it were a huge basic block. What this approach does not take into account is that many of the branches it selects are data independent and can be predicted based on compile time information. These branches can be eliminated. The partially evaluating parallelizing compiler approach is able to collapse data-independent portions of the program into large basic blocks without these branches. The partially evaluating compiler can guarantee that the right set of branches in these portions of code are taken by simply eliminating them. This is better than trying to probabilistically determine the branch direction. Another shortcoming of the trace scheduling approach is that it lacks partial evaluation's ability to remove inherently sequential data-structure references. This means that the trace scheduling technique by itself will not be able to take advantage of the all the inherent parallelism in a computation.

One thing that trace scheduling is good at is optimizing data dependent branches. Run time information can be used to reliably predict which way the branches typically go and substantial optimization may be performed on the resulting trace. A good strategy would be to couple both techniques. Partial evaluation would do a good job optimizing data independent portions of the computations, whereas trace scheduling would do well with the data dependent portions.

5.2 Software Pipelining

Software pipelining [11] optimizes a particular fixed size loop structure so that several iterations of the loop are started on different processors at constant intervals in time. This increases the throughput of the computation. Using partial evaluation on such a loop structure would result in the loop being completely unrolled with all the data structures references removed and the total parallelism of the operations executed in that loop becoming available and visible for parallelization.

5.3 Vectorizing

Vectorizing is a commonly used optimization for vector supercomputers. Matrix multiplies are an example of computations which can be done quickly on machines like these. Vectorizing compilers look for specific operations on arrays of numbers in memory. The compiler can then vectorize to execute these operations in parallel on the numbers in the arrays. These computations need to be expressed in a particular manner so that the compiler can identify the vectors which can be operated on in parallel. These machines and compilers do very well when the structures of the programs for computations match the architecture they are written for. Computations not structured in this manner do very poorly on these architectures. It would be very hard to get a partially evaluating compiler to identify vectorizable computations because memory location is not a notion the partial evaluation computation graphs give a sense of. An interesting thing that may be said about the parallelizing partially evaluating compiler, however, is that it is good at scheduling fine grained parallelism on MIMD like architectures where it is possible to utilize this fine grained parallelism

5.4 Iterative Restructuring

Iterative restructuring represents the manual approach to parallelization. Remarkably there are now many utilities for profiling and analyzing parallelism that allow programmers to find bottle necks in their code. One such utility is known as Max-Par [7] which essentially deduces the data dependency graph after the computation is completed and shows the parallelism available and that being exploited in various portions of the programs. A user can then use this to deduce which routines are parallelizable and may then rewrite the program so the compiler can identify and exploit this parallism.

The example given in the introduction of the Perfect Benchmark performance on the Cray-YMP should be noted, because this type of manual optimization was done in order to get those benchmarks into a form that the Cray YMP compilers could exploit parallelism on. The compiler in this thesis can do these things automatically. In the compiler introduced here, the data dependency graph does not ever need to be seen by the programmer. It is *automatically* generated and used by the compiler as an effective tool for exploiting the underlying parallelism in a computation.

5.5 Handcoding

Hand produced code for a computation will look much different from the compiler's code. The hand coding will localize many related computation in a particular piece of code. This may or may not occur on the compiler which spreads out the computation across the processors during a cycle. This is arguably better than hand coding because handcoding a complex computation on these statically scheduled architectures would undoubtedly drive someone nuts. The compiler, in its infinite patience, can search for open slots on a processor and spread out the computation across the processors.

5.6 Summary

In this section it has been shown that the use of partial evaluation in a parallelizing compiler in comparison to other techniques represents some definite advantages in order for the exploitation of underlying parallelism in numerical computations. Other methods do not seem to be able to exploit the underlying parallelism basically because using their methods, they can't find some of it. Thus partial evaluation should be coupled with some of the already good techniques so that the compiler can identify all of the underlying parallelism in a computation and exploit it. Some manual methods were also shown. One was surprisingly similar to what the partially evaluating parallelizing compiler tries to do automatically.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Automatic parallelizing compilers for supercomputers would benefit greatly if they included partial evaluation as part of their optimization. Besides providing an order of magnitude improvement for sequential code, the technique exposes inherent parallelism in a program by recreating the data dependency graph for the computation in the program. By utilizing the newly exposed parallelism, it has been shown here that parallelizing compilers utilizing this technique can achieve performance as good as or even better than that achieved by manual means.

We have implemented a basic block compiler which utilizes partial evaluation and static scheduling techniques to show how the resulting fine grain parallelism may be exploited. The exploitation techniques have been evaluated on two different highly abstracted programs written in Scheme which simulate n-body problems which are important in the fields of celestial mechanics and particle physics. The results reveal that it is possible to automatically achieve a factor of 6.2 speedup on an eight-processor configuration of the Supercomputer Toolkit from a single processor version of the program. This is impressive because the Supercomputer Toolkit utilized by the compiler has extremely low bandwidth, allowing a processor to send a value effectively every 8 cycles with a latency of 6 cycles. It was also found that the simple heuristic technique of region scheduling does not scale well for larger parallel processors, though it does work well on a computer the size of the Supercomputer Toolkit.

Other techniques utilized by parallelizing compilers do not include a mechanism that allows the compiler to examine the computations data dependency graph in order to figure out how to parallelize the computation. These other techniques could easily be complemented by partial evaluation resulting in dramatic speedups of a computation, even data dependent ones. It is believed that all automatic parallelizing compilers should have a mechanism to view the underlying parallelism in a computation.

6.2 Suggestions for Future Work

There are two ways to improve the compiler. One way involves extending the compiler's capabilities. The compiler could be extended to handle branches and subroutines so that it may handle data dependent computations. The other way is to increase the level of optimization that is performed. A possible optimization is to find a better method of exploiting the fine grain parallelism than region division that will work well on larger architectures. Perhaps a method like task fusion [10] should be attempted. Another optimization that could be added involves computing values redundantly across processors because it is cheaper than transmitting these values in some cases.

Bibliography

[1] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G. Sussman, "The Supercomputer Toolkit and its Applications," MIT Artificial Intelligence Laboratory Memo 1249, Cambridge, Massachusetts.

[2] J. Applegate, M. Douglas, Y. Gürsel, P. Hunter, C. Seitz, G.J. Sussman, "A Digital Orrery," *IEEE Trans. on Computers*, Sept. 1985.

[3] A.V. Aho, R. Sethi and J.D. Ullman,*Compilers: Principles, Techniques and Tools* Addison Wesley, 1988

[4] A. Berlin and D. Weise, "Compiling Scientific Code Using Partial Evaluation," to appear in IEEE Computer. Also see MIT Artificial Intelligence Laboratory Memo number 1145, July, 1989.

[5] A. Berlin, "Partial Evaluation Applied to Numerical Computation", in proceedings of the 1990 ACM Conference on Lisp and Functional Programming. Also see "A Compilation strategy for numerical programs based on partial evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1144, July, 1989.

[6] S. Borkar, R. Cohen, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-speed Parallel Computing," *Supercomputing '88*, Kissimmee, Florida, Nov., 1988.

[7] G. Cybenko, J. Bruner, S. Ho, “Parallel Computing and the Perfect Benchmarks.” Center for Supercomputing Research and Development Report 1191., November 1991

[8] J. Ellis, *Bulldog: A Compiler for VLIW Architectures.* PHD thesis, Yale University, 1985.

[9] J.A. Fisher, “Trace scheduling: A Technique for Global Microcode Compaction.” *IEEE Transactions on Computers*, Number 7, pp.478-490. 1981.

[10] Kasahara, Hironori, Honda, Hiroki, Narita, Seinosuke, “Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR”, *Supercomputing 90*, pp 856-864, 1990

[11] Monica Lam, “A Systolic Array Optimizing Compiler.” Carnegie Mellon Computer Science Department Technical Report CMU-CS-87-187., May, 1987.

[12] C. Heinzl, “Functional Diagnostics for the Supercomputer Toolkit MPCU Module”, S.B. Thesis, MIT, 1990.

[13] P. Hut and G.J. Sussman, “Advanced Computing for Science,” *Scientific American*, vol. 255, no. 10, October 1987.

[14] H. Printz, “Automatic Mapping of Large Signal Processing Systems to a Parallel Machine,” Carnegie Mellon Computer Science Department Technical Report CMU-CS-91-101., May, 1991.

[15] G. J. Sussman and J. Wisdom, “Numerical Evidence that the Motion of Pluto is Chaotic,” *Science*, Volume 241, 22 July 1988.

[16] J. D. Ullman, “NP-Complete Scheduling Problems”, *Journal of Computer and System Sciences*,vol. 10 (1975),pp 384-393.