Reptile: A Distributed ILP Compiler

by

Ian Bratt

B.S., Electrical and Computer Engineering, University of Colorado, Boulder 2003

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .... [illegible]
Department of Electrical Engineering and Computer Science
July 19, 2005

Certified by .... [illegible]
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Reptile: A Distributed ILP Compiler

by

Ian Bratt

Submitted to the Department of Electrical Engineering and Computer Science
on July 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The past few years witnessed a dramatic shift in computer microprocessor design. Rather than continue with the traditional pursuit of increased sequential program performance, industry and academia alike chose to focus on distributed, multi-core designs. If multi-core designs are to maintain the decades-long trend of increased single threaded performance, compiler technology capable of converting a single threaded program into multiple programs must be developed. In this thesis I present the Raw Explicitly Parallel Tile Compiler (*Reptile*), a compiler targeting the RAW computer architecture capable of converting a single threaded program into multiple threads communicating at the instruction operand granularity. On applications with sufficient amounts of parallelism *Reptile* has generated code which, on the *Raw* processor, achieves a speedup of as much as 2.3x (cycle to cycle) over an Athlon64.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to take this opportunity to thank several people that helped (in one way or another) with this thesis. I'd like to thank Dave Wentzlaff for being a terrific mentor and for all of those "indian buffet" lunches. I want to thank Rodric Rabbah for his invaluable compiler expertise and for being someone that I could talk to about my ideas. I'd like to thank Nate Schnidman for being a great officemate and helping ease my way into MIT. Mike Taylor and Walter Lee provided wonderful guidance for my project. I want to thank all of the people in the open space, Jim Sukha, Elizabeth Basha, etc. for putting up with me. I'd also like to thank Jim Sukha for help with the quadratic programming formulation of the placement problem. I'd, like to thank Bill Thies for giving me the StreamIt compiler results and Dan Connors for setting me on my current path. I'd like to thank Anant Agarwal for giving me the tremendous opportunity to join the Raw group, for keeping me focused when I got side-tracked, and for giving me the academic freedom to pursue what I find interesting. Most importantly, I'd like to thank Lisa for reminding me what is truly important.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents the Raw Explicitly Parallel Tile Compiler (*Reptile*), a compiler capable of compiling arbitrary sequential c-programs and parallelizing those programs at the instruction level, targeting the MIT Raw architecture. On applications with sufficient amounts of parallelism *Reptile* has generated code which, on the Raw processor, achieves a speedup of as much as 2.3x (cycle to cycle) over an Athlon64. *Reptile* facilitates exploration of the issues surrounding compilation for Distributed Instruction Level Parallelism (DILP) and provides a stable framework for DILP research.

The past few years witnessed a dramatic shift in computer microprocessor design. Rather than continue with the traditional pursuit of increased single threaded performance via complex centralized mechanisms, deeper pipelines and higher clock frequencies, industry and academia alike chose to focus on distributed, multi-core designs. Multi-core designs consist of several simple processors, each with a program counter, connected via a network. Motivated by conservative energy budgets and an ever increasing transistor surplus, Intel [16], AMD, IBM, SUN [6] and ARM all opted in favor of multi-core designs. Current multi-core designs offer the benefits of increased throughput, better transistor utilization, and increased power efficiency. From an energy perspective, executing two threads in parallel at a frequency of f requires less energy than executing the threads sequentially at a frequency of $2f$. A lower frequency facilitates a lower frequency voltage, allowing parallel designs to reap the benefits of the quadratic relationship between energy and voltage.

Unfortunately, most industry implementations of multi-core designs do little to increase the performance of existing single threaded applications. Continuing the decades-long trend of increased single threaded performance will require novel research at both the compiler and programming language level. Shifting the burden to the programmer requires software engineers to write multi-threaded code or learn a new, thread-friendly language. Moving the burden to the compiler requires development of compilers capable of automatically converting a single threaded program into multiple programs working together to achieve the task of the original program.

The recent transition in industry from complex single-core designs to simpler, multi-core designs foreshadows a technology trend toward chips with tens, even hundreds of cores. Academia [17], [20] [21], and to a lesser extent, industry [18], have both expressed interest in chips with tens/hundreds of cores. To successfully utilize the copious resources of future processors, development of compilation technologies exploiting parallelism *all* forms of parallelism must occur. Simply re-writing a sequential application into a multi-threaded application will not utilize the copious resources available in future processors. Exploitation of parallelism must occur at the loop level, the thread level, and the instruction level.

This thesis proposes Distributed Instruction Level Parallelism (DILP) as a means for fine grain automatic parallelization of sequential applications targeting multi-core systems. Instruction Level Parallelism (ILP) [1], involves executing independent instructions in parallel to increase performance. Modern superscalars exploit ILP by dynamically executing independent instructions in parallel, often speculating across basic block boundaries to expose more parallelism at the instruction level. Similarly, modern Very Long Instruction Word (VLIW) architectures utilize advanced compiler technology to expose ILP within a program and convey that ILP to the processor through an ISA that explicitly expresses instruction independence. Distributed ILP, like ILP, involves executing independent instructions in parallel. However, in the case of ILP, all of the instructions execute on the same physical core. In DILP, the independent instructions need not execute on the same physical core.

The major contribution of this thesis is the *Reptile* compiler, a compiler infrastructure designed for DILP exploration. *Reptile* takes as input an ANSI-c program and outputs multiple assembly files targeting a distributed architecture (see the Raw architecture in Chapter 3). The assembly files communicate at the instruction operand level, and work together to perform the same task as the original c program.

Chapter 2 of this thesis presents the background and related work. Chapter 3 introduces the Raw Microprocessor. An in depth explanation of the *Reptile* compiler is presented in Chapter 4. Chapter 7 steps through the *Reptile* compilation of a simple piece of code. Performance results for code generated with *Reptile* is presented in Chapter 8. Chapter 9 presents the conclusion and future work sections.

Chapter 2

Background and Related Work

2.1 Instruction Level Parallelism

To increase the performance of single threaded applications, many modern computer architectures exploit Instruction Level Parallelism (ILP). By executing multiple independent instructions in parallel, architectures may achieve significant performance gains. Unlike other types of parallelism, ILP exists within all types of applications, from highly sequential integer applications to dense matrix, scientific programs. Figure 2-1 shows a simple example of ILP. Part *A* of the figure depicts assembly code for a loop that copies values from one array to another. On a single-issue machine, the loop takes 5 cycles to execute. On a two-issue machine, the first two instructions and last two instructions may execute in parallel, decreasing the loop cycle time from 5 cycles to 3 cycles (Figure 2-1, Part *B*).

Almost all modern computer architectures support some form of ILP. The detection and subsequent expression of ILP to the hardware has traditionally come in two different flavors, hardware or software. Superscalars, an example of the former approach, exploit ILP via complex hardware mechanisms capable of dynamically determining instruction independence, allowing the processor to execute independent instructions in parallel. Modern VLIW/EPIC architectures support an ISA that expresses instruction independence to the hardware, allowing the compiler to statically determine instructions that should execute in parallel. Compiler/Software based ap-

```
addu    $8, $8, 4
```

```
addu    $8, $8, 4
```

Figure 2-1: (A) Sequential schedule without ILP. (B) Code exploiting ILP.

proaches to ILP have the advantage of shifting complexity from the hardware to the software. Decreasing hardware complexity helps to aid the verification process as well as cut down on design time. The trade-off, however, comes in the form of increased compiler complexity.

Compiling for increased levels of Instruction Level Parallelism was first proposed two decades ago [4]. Since, compiling for ILP has proved a popular research topic, spawning several architectures and compilers. The majority of compilation related work involves compiler/architecture techniques for increasing the number of instructions seen when scheduling. In practice, the more instructions in a block, the more ILP in a block. Trace scheduling [4] looks at traces of commonly occurring basic block patterns, scheduling code across basic block boundaries to increase the number of instructions "seen" when scheduling. Because a trace corresponds to a frequently occurring pattern of blocks, trace scheduling optimizes for the common case. Non-trace paths require patch-up code to undo incorrect speculative operations. A technique similar to traces, Superblocks [8], uses code-duplication to ensure that the "trace/superblock" contains a single entry point and multiple exit points, greatly simplifying the scheduling process. Extending the methods behind Superblock and Trace Scheduling to predicated machines, Hyperblock [13] scheduling creates large predicated regions out of commonly occurring sections of code.

2.2 Distributed Architectures

To maintain the trend of increased single threaded performance, most modern computer architecture research utilizes the increasing transistor surplus to focus on elaborate mechanisms designed specifically for increasing performance of highly sequential, single threaded applications. Examples of such mechanisms include branch prediction, speculation, out of order execution, larger caches and memory prefetching. Because of this one sided approach, existing microprocessors fail to exploit the large amounts of parallelism that exist within scientific, signal processing, and multi-threaded applications.

The reason existing microprocessors cannot support large amounts of parallelism stems from the centralized design of current architectures. Adding additional functional units to a centralized design negatively impacts clock frequency. Because modern superscalars and VLIWs rely on a centralized architecture with a unified register file, adding additional ALUs greatly increases the complexity of the register bypass paths. The diagram on the left in Figure 2-2 shows the bypass paths for a 16-way superscalar. Because the bypass paths scale quadratically with the number of ALUs, adding additional functional units will have a quadratic effect on the area of the bypass paths.

The diagram on the right in Figure 2-2 shows the traditional bypass paths replaced with a routed, point to point network. The routed network allows the addition of more functional units without affecting frequency. Physical distance between ALUs determines the number of cycles required for an operand to travel from the output of one ALU to the input of another. By designing architectures in such a fashion,

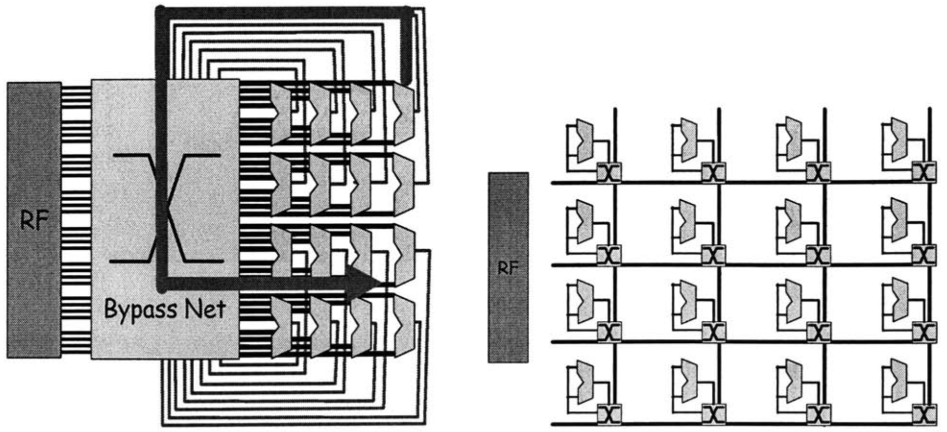


Figure 2-2: Logical diagram of the bypass paths for a traditional, centralized architecture (Left). The diagram on the right depicts a distributed architecture using a point-to-point interconnect for register operands. (Diagrams courtesy of Michael B. Taylor)

adding additional functional units will not negatively effect the clock frequency. The MIT Raw architecture [22] embodies this principle by organizing resources in *tiles*, where each *tile* contains memory and a simple processor. To add additional compute power to the Raw design, simply add more *tiles*.

Both academia [17] [20] [21], and industry [18], have proposed similar distributed architecture designs. Clustered VLIWs (for example, the TI-C6000), often found in commercial DSPs, are an example of a distributed architecture. By using two or more simple VLIWs sharing a unified register file, Clustered VLIWs cut down the bypass path complexity by only having bypass paths for a particular cluster. Values move between clusters via explicit move instructions that access a unified register file.

2.3 Distributed Instruction Level Parallelism (DILP)

Motivated jointly by the advent of Distributed Architectures and the quest for greater amounts of ILP, Distributed Instruction Level Parallelism (DILP) involves exploiting ILP on a distributed architecture. In the case of ILP, independent instructions execute in parallel on a single, multi-issue processor core. DILP, however, involves executing independent machine instructions on physically distinct processor cores. Figure 2-3 illustrates a simple DILP example where one ALU executes the loop overhead while the other ALU does the actual vector copy. Values transfer between ALUs via an abstract network. Duplication of the branch instruction allows both processor cores to follow the same control flow. The network allows propagation of the branch condition between cores.

Compiling for DILP involves not only determining instruction independence, but also assigning a physical placement for each instruction as well as orchestrating the communication between instructions on physically disparate cores. Many methods for

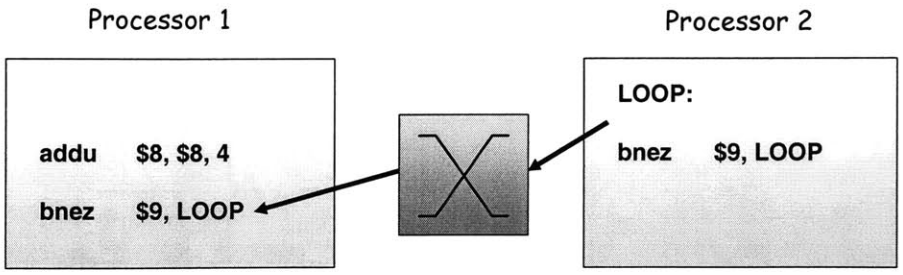


Figure 2-3: Example of Distributed Instruction Level Parallelism.

compiling for DILP exist, with most approaching the problem in terms of partitioning and placement. Partitioning attempts to cut a dataflow graph into chunks (partitions) to minimize the schedule length through the graph. Placement, following partitioning, involves assigning each partition to a physical processor core.

A small number of research groups have studied DILP compilation. Leupers et al. view the compilation phase for clustered VLIWs in terms of partitioning and placement [12]. However, clustered VLIW compilation does not involve routing over an interconnect. Instead, values are transfered between clusters via intercluster move operations. The most related work done in the field, "Space-Time Scheduling", done by Lee et al. [11], proposes and addresses the basic issues compiling for DILP. The first to view the problem in terms of partitioning, placement, and routing, Lee pioneered the area of DILP compilation. In many ways, the work presented in this thesis extends the work done by Lee. While Lee's original work focused on computation heavy kernels, this work will extend his ideas to arbitrary programs. Other research groups have proposed architectures for DILP, yet few have investigated the issues surrounding DILP compilation.

2.4 Architectural Issues Regarding DILP

Obviously, not all distributed architectures efficiently support DILP. Three main architectural features determine the feasibility of DILP: Memory model, interconnect latency, and processor issue width.

2.4.1 Memory

Virtually all programming languages assume a unified memory model with sequential consistency. This means that the state of memory must always reflect the state which would have occurred had all memory operations executed in program order. Out of order superscalars provide sequential consistency by executing memory instructions out of order and storing the addresses in case two instructions went out of order that shouldn't have (e.g., A load is promoted above a store and both instructions access the same address). Successful exploitation of DILP will require parallel execution of memory operations on physically distinct cores. Therefore, support for maintaining sequential consistency as well as memory coherence across cores must exist.

2.4.2 Interconnect

To effectively schedule multiple instructions from the same block across different cores, a DILP processor must possess a low latency network for inter-core communication. If such a network exists, it can facilitate the communication of instruction operands between physically distinct cores. The latency of the network plays a key role in determining the amount of exploitable DILP. For example, if instruction A on $core_1$ uses a result generated by instruction B on $core_2$, the number of cycles before A can execute equals the latency of B plus the latency of the network (assuming zero cost to send and receive the operand). Because basic blocks in integer code often contain fewer than 10 instructions, a latency greater than 10 cycles would prove impractical. Otherwise, scheduling instructions on multiple-cores would yield a slower schedule than scheduling all instructions to just one core.

2.4.3 Processor Issue Width

In the context of a multi-core architecture, the number of independent instructions issued in parallel on a single core determines the processor issue width. For example, if a core issues at most three instructions in parallel, the issue width is three. The wider the core, the lower the benefit of distributing instructions from the same block across cores. This stems from the fact that only a finite amount of ILP exists within a segment of code. If a processor core contains architectural support for exploiting more ILP than exists within a block of code, few (if any) gains will result from distributing the instructions across multiple cores. Because of this, application type plays a significant role in which processor issue widths can effectively exploit DILP. For example, limit studies [25] have shown that most *integer* programs contain on average a realizable ILP of at most five parallel instructions per cycle. If an architecture contains two five-way cores, integer applications gain little by distributing ILP across both cores. Because this thesis attempts to exploit DILP for arbitrary applications, an ideal target architecture would have a very small processor issue width.

2.5 Why ILP?

Three main categories of parallelism exist within programs (Figure 2-4). Thread Level Parallelism (TLP) pertains to the parallelism obtained when executing two different sequential threads in parallel. This includes executing multiple threads of a multi-threaded application in parallel, as well as executing two completely independent threads in parallel. Loop Level Parallelism (LLP) consists of parallelism found across loop iterations. For example, if any portion of loop iteration i may execute in parallel with any portion of loop iteration $i + 1$, the loop contains LLP. LLP consists of two sub-classes, Full Iteration Parallelism and Partial Iteration Parallelism. Full

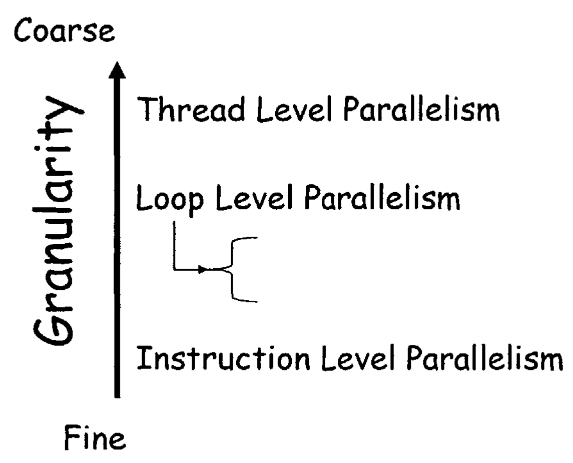


Figure 2-4: The different types of parallelism.

Iteration Parallelism occurs when two different iterations of a loop are completely independent and can execute in parallel. Often called Data Level Parallelism (DLP), Full Iteration Parallelism has historically been the target of vector machines. Partial Iteration Parallelism occurs when only a portion of one loop iteration may execute in parallel with a different portion of another iteration. This is commonly called Streaming parallelism, and corresponds to the type of parallelism exploited via Software Pipelining and by streaming machines [9, 5]. At the lowest level of granularity, Instruction Level Parallelism (ILP) consists of the parallelism found natively, within a basic block at the instruction level. Branch overhead, address calculation, etc. are examples of instructions that can often execute in parallel with the critical path of computation.

The question of what granularity of parallelism to efficiently exploit on a distributed architecture is currently a controversial topic in the research community. Many researchers advocate that distributed architectures should exploit coarser types of parallelism, leaving the single processor core to exploit ILP. However, these advocates overlook the fact that *ALL* forms of coarse parallelism can be converted into ILP. LLP can be converted into ILP via loop unrolling and software pipelining. Therefore, if a distributed architecture can support ILP, it can support all types of parallelism, facilitating a more *general purpose* architecture. Because of this, DILP is a promising type of parallelism and a good target for a parallelizing compiler.

Chapter 3

The Raw Microprocessor

3.1 Overview of Raw

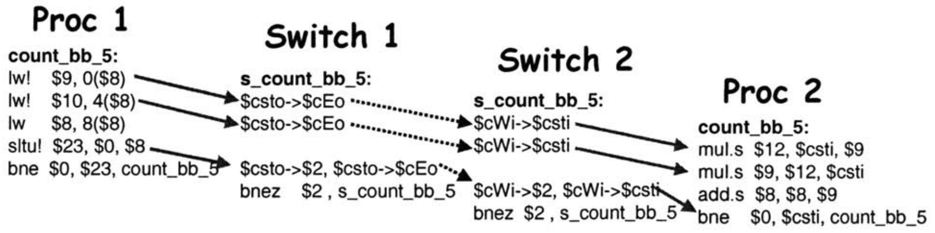
Traditional single chip processors strictly adhere to what is known as the *Von Neumann* architecture. The *Von Neumann* architecture consists of centralized instruction and data memory, execution units and register files. Additionally, the *Von Neumann* architecture assumes a single flow of control (one program counter), providing a clear notion of sequential ordering and time. The elements of a *Von Neumann* machine communicate off chip through a single bus. As technology continues to scale and the number of functional units on a chip increases, large, centralized memory systems and register files will negatively impact clock frequency.

The Raw microprocessor [24] (Figure 3-2), developed at MIT, attempts to address this problem by spatially distributing the architectural resources of the processor. The Raw architecture consists of 16 identical *tiles*, each tile containing a simple compute processor, register file, and instruction and data memory. Raw *tiles* communicate via a *register mapped point-to-point network* optimized for word-sized operands. A programmable switch routes operands between *tiles*.

3.2 The Raw SoftNetwork

The register mapped ports allow an instruction to place a value on the network with no overhead. Similarly, instructions using values from the network simply read from the register mapped ports. The programmable switches bear the responsibility of routing operands through the network.

Figure 3-1 shows an example of a 2-tile raw program where the two tiles are located next to each other in a horizontal row. The code contains processor code and switch code for each tile. Both tiles and switches have the same control flow, with one tile calculating the branch condition and using the switches to propagate the condition to both switches and the other tile. Solid edges represent data dependences between a processor and the local switch. The dashed edges represent data dependences between the two switches. In the figure, instruction opcodes containing a ! symbol write to the register mapped network port (called *$csto*). Physically, *$csto* is implemented



```
count_bb_5:
lw!   $9, 0($8)
lw!   $10, 4($8)
lw    $8, 8($8)
sltu! $23, $0, $8
bne  $0, $23, count_bb_5
```

```
s_count_bb_5:
$csto->$cEo
$csto->$cEo

$csto->$2, $csto->$cEo
bnez    $2 , s_count_bb_5
```

```
s_count_bb_5:
$cWi->$csti
$cWi->$csti

$cWi->$2, $cWi->$csti
bnez  $2 , s_count_bb_5
```

```
count_bb_5:
mul.s  $12, $csti, $9
mul.s  $9, $12, $csti
add.s  $8, $8, $9
bne    $0, $csti, count_bb_5
```

Figure 3-1: Example of a 2-tile Raw program. Dashed edges represent dependences between switch instructions, and solid edges represent dependences between a processor and switch.

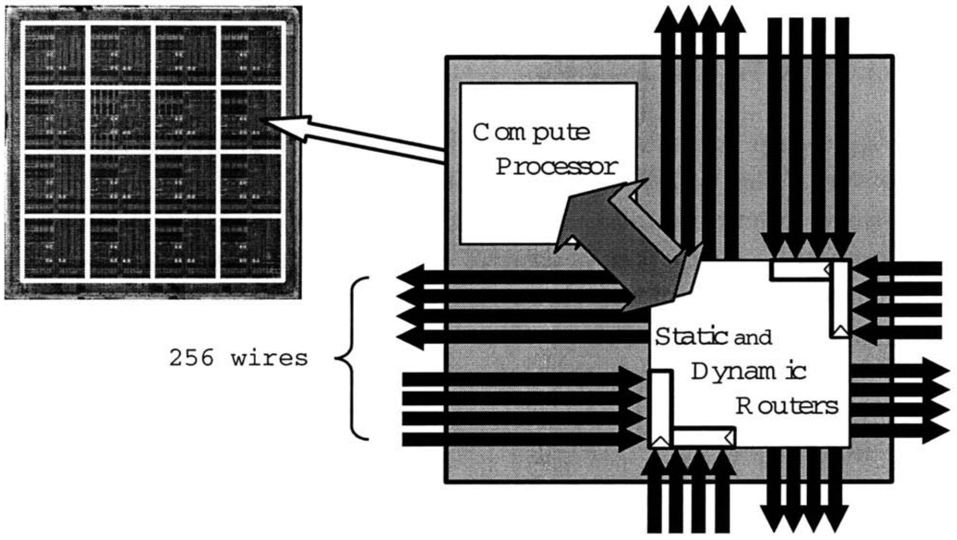


Figure 3-2: The Raw Microprocessor. Die photo (left) and abstract view of one compute tile (switch and processor).

via a queue. The processor inserts words into the queue and the switch reads words from the queue. The switch may place values read from the *$csto* queue onto the network or into a local switch register. The queue contains eight words of storage space allowing the processor to write up to eight words to the register mapped port before the switch reads any values.

The switch instructions may route operands already on the network (for example, reading from the north and sending south), inject operands from the processor into the network or drain operands from the network and send them to the processor or to a local register. The switch contains a 4-way crossbar, with each way corresponding to one of the cardinal directions. If the switch wants to pull the incoming value from the north and send it west, the instruction looks like *$cNi*$\rightarrow$*$cWo*, where the i and o represent input and output, and the N and W represent North and West, respectively (Note that in practice, the above instruction is prefixed with a *route* opcode. For brevity and space, the *route* opcode was excluded from the example). The input ports (*$cNi, $cWi, $cEi, $cSi*), allow for four words of storage. To send a value from the network to the processor, the switch must read from one of the input ports or a local register and write to *$csti.* The *$csti* queue allows for the switch to write four words of data before the processor reads any values from *$csti.* To read a value from the switch, the processor simply reads from *$csti.*

3.3 Physical Implementation

The Raw microprocessor prototype was implemented in IBM's 180 nm, 6-layer standard-cell ASIC process. The Raw group developed a Raw prototype motherboard for the Raw chip. Programs are run on the Raw board via a host machine. More details of the Raw microprocessor may be found in [22].

Chapter 4

The Reptile Compiler

4.1 Infrastructure

To successfully generate parallelized code at the instruction level and exploit DILP, *Reptile* leverages the OpenImpact [19] and Trimaran [23] compiler infrastructures. Both compilers target VLIW-like machines, therefore, they contain optimizations that increase the amount of ILP within a basic block of code (loop unrolling, speculation, etc.). Leveraging these two compilers, the *Reptile* compiler can take an arbitrary C-program and create multiple programs communicating at the operand level and map those programs to the Raw architecture. *Reptile* uses *newlib* for library calls and generates Raw assembly code compatible with the default Raw toolchain.

The Reptile compiler is composed of three main phases, as shown in Figure 4-1.

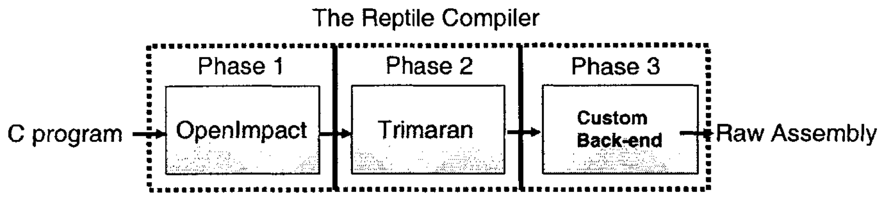


Figure 4-1: High level view of the *Reptile* compiler.

4.1.1 OpenImpact

Developed in Wen-mei Hwu's Impact research group at the University of Illinois, the OpenImpact compiler helped to pave the way for ILP compilation research. Many of the techniques and mechanisms developed by the Impact group ended up in the Intel IA-64 architecture [15]. Currently, the OpenImpact compiler generates IA-64 code of higher quality than that generated by *gcc*.

The OpenImpact front-end supports ANSI-C C-programs. The first phase of compilation parses the C-code and performs inter-procedural alias analysis. For profiling, OpenImpact generates an instrumented binary. Running the binary produces execution weights for all of the basic blocks in the program. Optimizations targeting traces/regions of code use the profile-generated weights to determine which basic blocks to merge into regions. Similarly, profile information helps identify the most frequently executed portions of code. OpenImpact converts the program into an Intermediate Representation (IR), called *lcode*. The *lcode* IR has may similarities to a RISC, 3-operand ISA. All classical optimizations operate on *lcode*. The *lcode* has a textual representation containing all of the information required to reconstruct the original *lcode* IR. Several code generator backends for OpenImpact exist which read in the *lcode* text files, perform register allocation, scheduling and peep-hole optimizations.

4.1.2 Trimaran

Developed jointly by HP labs, the University of Illinois and New York University, the Trimaran compiler extends the Impact compiler infrastructure to target the HP Playdoh architecture. Created for VLIW/EPIC architecture exploration, the Playdoh architecture supports software pipelining and rotating register files. Unfortunately, the Playdoh architecture never made it to silicon. However, many of the ideas explored with Playdoh made their way into the Itanium processor.

The version of Trimaran used for the *Reptile* research project originated from Scott Mahlke's *CCCP* research group at the University of Michigan. Mahlke's group modified Trimaran to target clustered VLIWs, making it quite useful for the *Reptile* project. The *CCCP* version of Trimaran contains support for distributed register files, limited bypass paths and arbitrary numbers of clusters.

4.2 *Reptile* Overview

The goals of developing the *Reptile* compiler include automatic parallelization of arbitrary C-programs as well as creation of a stable framework for DILP compilation research. The *Reptile* compiler currently targets the Raw architecture, generating code for up to 16 Raw tiles. The *Reptile* compiler contains three main phases.

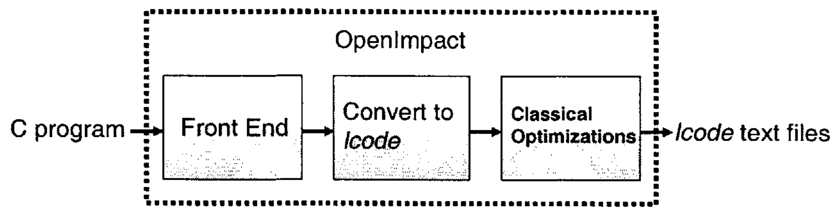


Figure 4-2: Phase 1 of the *Reptile* compiler.

1. The first phase of *Reptile* compilation involves use of the OpenImpact compiler. This phase of compilation supports input in the form of an arbitrary C-program. After parsing the C-program with the front-end, the compiler performs a suite of classical optimizations. The final output of the OpenImpact compiler, a slightly modified textual representation of the *lcode* IR, conveys all the information needed to begin phase two of the compilation. Figure 4-2 displays phase 1 of the *Reptile* compiler, where the front end parses the C-program, optimizes the code, and prints out a textual representation of the OpenImpact IR.

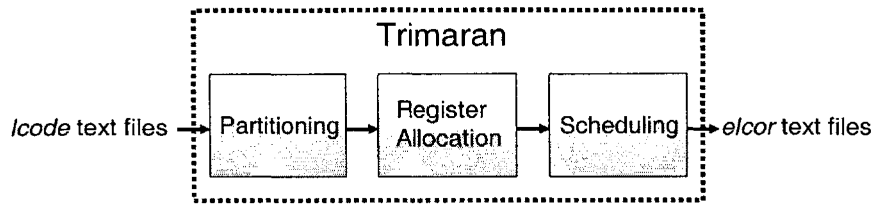


Figure 4-3: Phase 2 of the *Reptile* compiler.

2. The second phase of compilation uses the *CCCP* version of the Trimaran compiler. This phase reads in the *lcode* text files and converts them to the Trimaran IR (*Rebel*). Once converted, *partitioning* assigns each instruction to a virtual "partition" with the intent of later assigning each partition to a physical Raw *tile*. After *partitioning*, register allocation takes place for each partition, assuming one register file per partition. Insertion of inter-partition moves allows instructions on differing partitions to access the same values. Instruction scheduling occurs after register allocation. The output of the second phase of compilation, a textual representation of the Trimaran IR, conveys instruction-to-partition assignments as well as register allocation information and basic block execution frequency. Figure 4-3 illustrates phase 2 of the *Reptile* compilation. Section 5.2 describes phase 2 in greater detail.

3. Phase three of the compilation, Figure 4-4, reads in the files representing the Trimaran IR and outputs assembly compatible with the Raw toolchain. A mapping between the Trimaran IR and the *Reptile* IR allows for generation of Raw assembly files. A placement phase assigns each virtual "partition" to a physical Raw tile. After placement, a routing phase generates dead-lock free switch code orchestrating operand communication between tiles. After routing, a peep-hole optimization pass performs Raw specific optimizations as well as post codegen scheduling. Section 6 discusses the specific modules of phase 3.

4.2.1 Memory

Because the Raw architecture does not contain any form of memory coherence, the compiler must statically place memory objects on different tiles and ensure that all

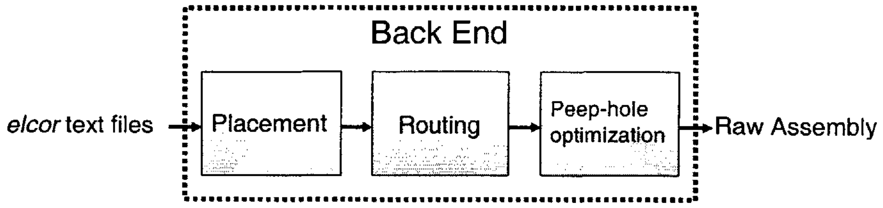


Figure 4-4: Phase 3 of the *Reptile* compiler.

instructions accessing those objects are scheduled to the appropriate tile. Currently, the *Reptile* compiler memory alias analysis is not sophisticated enough to provide sufficient memory parallelism on the Raw architecture. Therefore, all memory operations generated by the *Reptile* compiler are scheduled to the same *tile*.

Chapter 5

Reptile: Phase 1 and 2

5.1 *Reptile* Phase 1

As seen in Figure 4-2, Phase 1 of the *Reptile* compilation process consists of three modules. This section will only briefly discuss Phase 1 of *Reptile* due to the fact that the OpenImpact compiler is well documented and prevalent in academia.

Front-End The Front end parses arbitrary C-programs turning them into a high level representation called *pcode*.

Lcode Conversion The *pcode* produced is later turned into the main impact IR, *lcode*, which corresponds loosely to a RISC ISA, supporting the standard load-store architecture and three operand instructions.

Classical Optimizations The standard suite of classical optimizations are performed, guided by profile data.

5.2 *Reptile* Phase 2

Figure 4-3 shows the three modules of the second phase of *Reptile* compilation; partitioning, register allocation and scheduling.

5.2.1 Partitioning

The partitioning module takes the assembly corresponding to an entire program and breaks it up into p partitions, where partitions communicate at the instruction operand level. At the lowest level, the partitioning phase consists of taking a basic block of code and assigning each machine instruction within the block to a partition. Therefore, one may view partitioning as taking a basic block of code and turning it into p basic blocks, one for each partition, with the control flow replicated across all new blocks (see Figure 5-1). Partitions communicate via an abstract, ideal network capable of routing operands (register values) back and forth between the partitions. In later phases of compilation, the "abstract network" in Figure 5-1 is replaced by

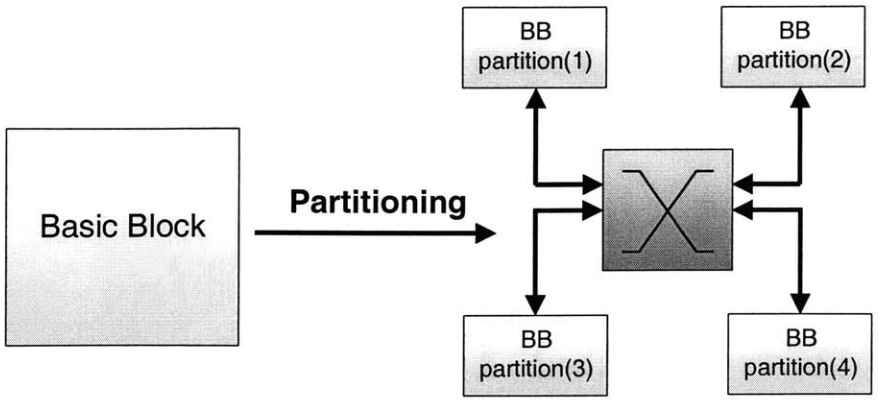


Figure 5-1: High level view of partitioning. A single basic block is turned into 4 basic blocks, working together to execute the code of the original block. The blocks communicate instruction operands via an operand network.

the Raw Scalar Operand Network. The *placement* module, described in Section 6.1, maps each partition to a physical processor.

Exploitation of DILP occurs by partitioning the code at the instruction level. The tradeoff, however, comes in the form of communication between partitions. If the code contains no ILP, then no gains will arise from partitioning. A good partitioning algorithm attempts to exploit parallelism and minimize communication. Not all programs contain enough parallelism to justify partitioning of instructions. Therefore, a good partitioning algorithm must not attempt to partition code when no parallelism exists and when a partitioned schedule would result in a longer schedule than had all instructions been scheduled to a single core.

In addition to binding instructions to partitions, the partitioning module binds operands to partitions as well. If an instruction accesses an operand on a different partition, the network communicates the operand value between partitions.

The complete *Reptile* partitioning module consists of the following steps:

1. For each function in the program, sort all basic blocks according to execution frequency.

2. Starting with the most frequently occurring block:

   (a) Partition the block via simulated annealing, adding inter partition moves for pre-bound operands. See Appendix A for a background on simulated annealing.

   (b) Bind all instructions to their corresponding partition.

   (c) Bind all unbound operands used in the block to a partition based upon which instructions in the partition access the operand.

   (d) Repeat the above for all remaining basic blocks.

```
for (i=0; i<stop_outer; i++){
   for(j=0;j<stop_inner;j++){
      //Randomly assign an instruction to a different partition
      perturb(T);

      //Schedule the Code and determine the schedule length
      cost=getCost();
      deltaC = cost-oldcost;

      //If the cost increased
      if(deltaC > 0){

         //reject if rand() > e^(-(cost-oldcost)/T)
         if((1.0/exp((deltaC)/T)) < ((double)rand()/RAND_MAX)){
            //Reject the most recent perturbation
            reject(); cost=oldcost;
         }
      }
      oldcost=cost;
   }

   //Decrease the temperature one step
   T=nextT(T);

   //Break out if the Temp is really low
   if(T < .0000000001) break;
}
```

Figure 5-2: C-code from Partitioning module in the *Reptile* compiler.

Partitioning blocks in order of most frequently occurring allows binding of operands to those partitions which access the operands most frequently. The OpenImpact compiler provides the execution frequency for each basic block via profiling.

The majority of execution time takes place in the simulated annealing partitioning module. Figure 5-2 shows the main loop for performing the simulated annealing algorithm to partition a basic block. The first step of the partitioning creates a random assignment from instructions to partitions. The code consists of two loops, with the inner loop repeatedly perturbing the system and updating the state accordingly, while the outer loop decreases the "temperature".

The following subsections explain the simulated annealing parameters in terms of the partitioning problem.

Temperature

An initial heuristically determined temperature, T_0, provides the starting temperature for the algorithm. After completion of the inner loop in Figure 5-2, update of the temperature occurs according to $T[i+1] = \alpha T[i]$, where $\alpha < 1$ and chosen heuristically. Trial and error was used to determine values for α and T_0. T_0 was chosen to provide a large enough initial temperature for sufficient randomness to occur, while still maintaining a reasonable running time. Similarly, α was chosen to decrease the temperature at a slow enough rate to avoid quenching, yet still provide a manageable execution time.

Inner Loop Count (stop_inner)

The inner loop count, *stop_inner*, determines how many different perturbations occur at a given temperature. The choice of *stop_inner* greatly affects execution time. Increasing *stop_inner* will yield better results at the cost of execution time. Similarly, decreasing *stop_inner* will decrease execution time, however, it will also decrease the probability of the algorithm finding the global minimum. The *Reptile* compiler sets $stop\_inner = i * p$, where i represents the number of instructions in the block and p represents the number of partitions. This was shown to work well in practice.

Perturbation (*perturb*())

Within the inner loop, the perturbation of the existing partition occurs by randomly choosing an instruction and mapping that instruction to a randomly chosen partition (excluding the partition the instruction was previously assigned to).

Cost (*getCost*())

The cost of a particular partitioning may correspond to several different metrics. For example, one approach schedules the partitioned code and reports the schedule length as the cost. Other approaches include defining a cost associated with inter-cluster moves, load balancing, and an estimate of register pressure. Many different approaches were tried with the *Reptile* compiler, and no approach proved a clear winner.

Figure 5-3 displays an instruction dataflow graph after partitioning. Each node in the graph corresponds to a single machine instruction. The graph represents the inner loop of a 2-tap complex fir filter. The different colored nodes correspond to different partitions. Edges between nodes represent data dependences. Notice that some edges travel between nodes on the same cluster while other edges connect nodes on different clusters. The cross cluster edges correspond to the dataflow that must utilize the network to send data between tiles.

5.2.2 Register Allocation

After *partitioning*, the *Reptile* compiler register allocates each virtual partition, assuming that each partition has access to a certain number of registers. One could argue that register allocation should take place at the same time as partitioning. However, adding a register pressure cost to the partitioning algorithm to drive the simulated annealing has a similar effect. This is something that could easily be added to *Reptile*. *Reptile* uses a graph coloring based register allocator developed by the *CCCP* research group at the University of Michigan.

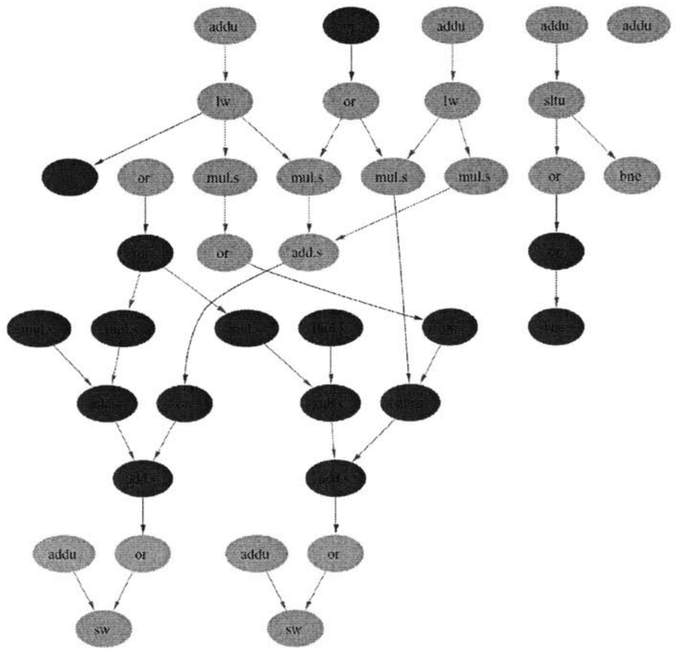


Figure 5-3: Dataflow graph for a 2-tap complex fir filter partitioned for two partitions. Each node represents one machine instruction. Nodes of similar color belong to the same partition.

5.2.3 Scheduling

In the *Reptile* compiler, scheduling occurs after partitioning and register allocation. Scheduling takes place at the basic block level, assuming that all partitions enter a basic block at the same time. The scheduler assumes unified control flow in the sense that if one partition executes a branch, all partitions branch. The unified control flow model differs from the Raw model and is accounted for in the final phase of *Reptile* compilation, where a post-codegen scheduling pass cleans up the code and compensates for the fact that the original scheduling assumes a VLIW like model and global control flow.

Chapter 6

Reptile: Phase 3

As seen in Figure 4-3, Phase 3 of the *Reptile* compilation process consists of three main modules; placement, routing and peep-hole optimizations. The input to Phase 3 of the *Reptile* compiler, a textual representation of the Trimaran IR, conveys instruction-to-partition mappings. Additionally, register allocation for each partition has occurred. The register allocation assumes all instructions within a partition access the same register file. The instructions communicate between partitions via explicit register to register move instructions. These move instructions read from the local register file and write to *any* other register file.

6.1 Placement

The placement phase of compilation maps virtual partitions to physical processor cores (tiles) (see Figure 6-1). The complexity of the placement problem can be grasped by noticing that if there are 16 partitions and 16 tiles, there exists 16! possible mappings of partitions to tiles. In the figure, each node represents a partition. An edge between two nodes indicates communication between two partitions. The label on the edge, $c_{i,j}$ represents a metric for the communication between partitions i and j *for the entire program.* For example, if two nodes, i and j, communicate often and would benefit from being placed physically close to each other, then $c_{i,j}$ is large. If the two nodes rarely communicate, $c_{i,j}$ is small. Several different methods could be used to construct $c_{i,j}$, but for this thesis, we construct $c_{i,j}$ as defined in Equation 6.5. *Reptile* sets $c_{i,j}$ equal to the number of words sent from partition i to j. Another possible definition sets $c_{i,j}$ equal to the number of words sent from partition i to j that are on the critical path of computation.

The placement problem may be formalized as follows: Let $\mathbf{P}$ be the set of partitions and $\mathbf{T}$ the set of tiles. For simplicity, we will let $|\mathbf{T}| = |\mathbf{P}| = n$ (there are an equal number of tiles and partitions). The goal of the placement problem is to find a correspondence (a one-to-one, onto function), $\mathbf{x}$, mapping $\mathbf{P}$ to $\mathbf{T}$:
We define the function $\mathbf{x}$ as:

$$\mathbf{x} : \mathbf{P} \longrightarrow \mathbf{T} \tag{6.1}$$

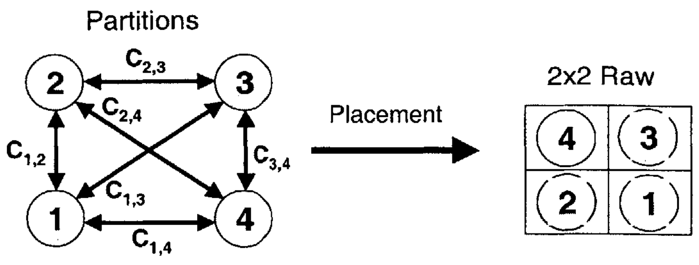


Figure 6-1: The *placement* module takes a group of n virtual partitions (left) and maps them to n Raw tiles (right).

And $\mathbf{x}$ has the following properties:

$$\mathbf{x}(i) \neq \mathbf{x}(j) \;\; \forall \;\; i \neq j \tag{6.2}$$

$$\forall \;\; i \in \mathbf{P} \;\; \exists \;\; k \in \mathbf{T} \;\; s.t. \;\; \mathbf{x}(i) = k \tag{6.3}$$

We define the latency l, between two tiles, k and q as:

$$l_{k,q} = \text{the number of cycles to transfer a word between tile } k \text{ and } q \;\; (k, q \in \mathbf{T}) \tag{6.4}$$

And the communication metric between two partitions, i and j as:

$$c_{i,j} = \text{Number of words sent between partition } i \text{ and partition } j \;\; (i, j \in \mathbf{P}) \tag{6.5}$$

A useful placement is one that minimizes communication. Therefore, we attempt to find the function $\mathbf{x}$ that minimizes the total number of cycles in which words are present on the network. This is equivalent to determining the $\mathbf{x}$ that minimizes the following function:

$$\min \sum_{i \in \mathbf{P}} \sum_{j \in \mathbf{P}} c_{i,j} l_{\mathbf{x}(i),\mathbf{x}(j)} \tag{6.6}$$

6.1.1 Placement: Simulated Annealing

The *Reptile* compiler uses simulated annealing to determine the above function, $\mathbf{x}$, for placement. While the simulated annealing approach does not guarantee an optimal result to Equation 6.6 , in practice the optimal is not needed. Additionally, it is not always the case that a placement minimizing the number of cycles in which data is sent on the network(Equation 6.6) yields the smallest execution time. For example, consider the case where partition i sends a large number of operands to partition j, but partition j does not send any values to partition i. Because the communication is one-sided, the communication may be pipelined, and the partitions *placed* physically far apart. To remedy this, future versions of *Reptile* will explore different definitions of the communication metric ($c_{i,j}$). The placement phase consists of the following two steps.

1. Examine the entire program and construct the communication parameters, c, where $c_{i,j}$ is defined in Equation 6.5.

2. Use simulated annealing to determine the function $\mathbf{x}$ as described in Equation 6.1.

The first step of the placement chooses a random assignment for the x variables, subject to the constraints described in equation 6.3. The following subsections describe the simulated annealing variables used for the *placement* module.

Temperature

The temperature is handled the same as for the *partitioning*. An initial heuristically determined temp provides the starting temperature. Temperature update occurs according to $T[i+1] = \alpha T[i]$, where $\alpha < 1$ and chosen heuristically. Again, the starting temperature and α were chosen to provide a high enough starting temperature and slow enough cooling schedule to generate respectable results, without significantly impacting running time.

Inner Loop Count (inner_loop)

The inner loop count, L, determines how many different perturbations occur at a given temperature. Different values of L were tried, and it was found that setting $L = |\mathbf{T}| * |\mathbf{P}| = n^2$, where n is the number of tiles/partitions.

Perturbation (*perturb*())

The perturbation switches the assignment of two different partitions. For example, if current partitioning maps partition i to tile k and partition j to tile q, a valid perturbation maps i to q and j to k, where i and j were chosen randomly.

Cost (*getCost*())

The cost corresponds to the function described in Equation 6.6.

6.1.2 Placement: Integer Quadratic Programming

Another solution to the minimization in Equation 6.6 is Integer Quadratic Programming. Unlike Simulated Annealing, a quadratic programming approach is guaranteed to yield the optimal solution to Equation 6.6. Unfortunately, no reasonable bounds exist on the running time of current Integer Quadratic Programming algorithms.

A Quick formulation of the Integer Quadratic Programming solution follows: For simplicity, we will assume an equal number of processors and partitions (n), and that the processors and partitions are numbered consecutively from 0 to $n-1$. One may represent the function $\mathbf{x}$, as an nxn permutation matrix, $\mathbf{X} \in \{\mathbf{0}, \mathbf{1}\}^{(\mathrm{nxn})}$, with elements $x_{i,j}$ defined below:

$$x_{i,k} = \begin{cases} 1 & , \text{ If partition } i \in \mathbf{P} \text{ is mapped to tile } k \in \mathbf{T} \\ 0 & , \text{ otherwise} \end{cases} \tag{6.7}$$

In order to ensure that $\mathbf{X}$ represents a one-to-one, onto, mapping, $\mathbf{X}$ is subject to the following constraints:

$$\sum_{k \in \mathbf{T}} x_{i,k} = 1 \ \forall \ i \in \mathbf{P} \text{ and } \sum_{i \in \mathbf{P}} x_{i,k} = 1 \ \forall \ k \in \mathbf{T} \tag{6.8}$$

One may construct the vector $\hat{\mathbf{x}}$ such that the first n elements are $(x_{0,0}, x_{0,1}, \cdots x_{0,n-1})$ and the last n elements are $(x_{n-1,0}, x_{n-1,1}, \cdots x_{n-1,n-1})$. Obviously, $\hat{\mathbf{x}} \in \{\mathbf{0}, \mathbf{1}\}^{(\mathrm{n}^2)}$.

Similarly, one may construct the matrix, $\mathbf{A} \in \mathbf{N}^{\mathrm{n}^2 \mathrm{xn}^2}$, such that

$$A_{i,j} = c_{(\frac{i}{n}),(\frac{j}{n})} l_{(i \bmod n),(j \bmod n)} \tag{6.9}$$

Where c and l are the constants defined in Equations 6.4 and 6.5 and the division in the subscript of c is integer division. We can then rewrite Equation 6.5 in the following form:

$$\min \sum_{i \in \mathbf{P}} \sum_{j \in \mathbf{P}} \sum_{k \in \mathbf{T}} \sum_{q \in \mathbf{T}} c_{i,j} l_{k,q} x_{i,k} x_{j,q} = \min \hat{\mathbf{x}}^{\mathbf{T}} \mathbf{A} \hat{\mathbf{x}} \tag{6.10}$$

Minimizing the above quadratic form subject to the constraints in Equation 6.8 will yield the optimal values of $\mathbf{X}$ (optimal placement) according to our cost function. Matlab code implementing the above quadratic program was implemented, unfortunately, the excessive running time made it impractical for use in *Reptile*.

6.2 Routing and Switch Code Generation

After partitioning and placement, *Reptile* generates switch code to orchestrate communication between physical processors. Refer to Section 3.1 for a brief overview of the Raw architecture. On input to phase 3 of the *Reptile* compiler, the code contains no notion of network topology. The code assumes a perfect, all-to-all network, allowing code in any partition to send values to any other partition. Explicit send and receive instructions move operands from the network to the local register file. The *Reptile* compiler must not only generate switch code for the send/recv begin and end points of communication, but must also choose a route for the operand to travel between tiles as well as generate switch code for the route.

6.2.1 Deadlock

Before explaining the switch code generation step, the following section will first discuss how deadlock might occur in the Raw static network. To facilitate discussion of deadlocks issues, this thesis will adopt the terminology first developed by Holt [7]. We will frame the deadlock problem in terms of an abstract graph composed of nodes (*agents* and *resources*) joined via *waits* and *holds* edges. In the context of the Raw static network, the switch instructions serve as agents and the switch input buffers represent the resources. Let *waits*(A,R) represent a waits edge between agent A and resource R. This edge implies that before agent A can perform its specified action, it must first obtain resource R. The waits edge is also defined from agent to agent. For example, *waits*(A1,A2) means that agent A1 must wait for agent A2 to complete before it may complete. In the context of the Raw static network where agents are instructions, *waits*(I1, I2) implies that instruction I1 cannot execute until instruction I2 is complete. Similarly, let *holds*(A,R) represent a holds edge between agent A and resource R. The holds edge implies that agent A is using resource R and preventing other agents from using the resource.

Figure 6-2 shows an example of Raw switch code with added waits and holds edges. In the figure, the initial send instructions on both switches (I1 and I5) have already

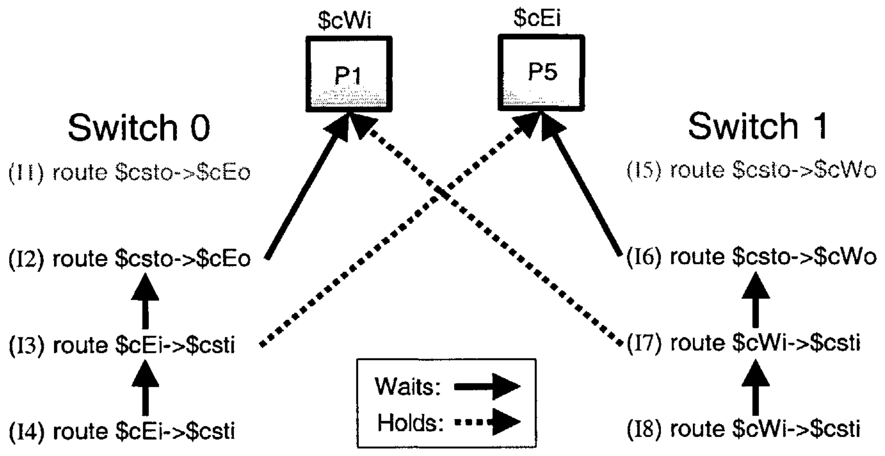


Figure 6-2: Switch Code and Switch input buffers with the corresponding waits and holds edges. The edges were drawn assuming the input buffers ($cWi and $cEi) each have one word of storage and that I1 and I5 have already executed.

executed, placing a word in each of the input buffers (P1 in $cWi and P5 in $cEi). The waits and holds edges were drawn assuming one word of storage for each input buffer. Because the Raw switch is an in order issue machine, each instruction must wait on the preceding instruction before issuing. Therefore, a waits edge connects each instruction to its preceding instruction (for example, *waits*(I3,I2)). The agents I2 and I6 are both waiting to place a word in the resources ($cWi and $cEi) respectively (the resources are currently holding the operands from instructions I1 and I5). The resources will not be released until instructions I3 and I7 have read the values from the input buffer. This is represented by the holds edges in Figure 6-2. The operand currently in resource $cWi will not be removed (and the resource freed) until agent I7 executes (*holds*(I7,$cWi)).

A property of the holds and waits edges is that if agent A1 is waiting on a resource R, and agent A2 is holding resource R, then agent A1 is waiting on agent A2. Formally:

$$\textit{waits}(A1, R) \text{ and } \textit{holds}(A2, R) \Longrightarrow \textit{waits}(A1, A2) \tag{6.11}$$

For example, in Figure 6-2, the edges *waits*(I2,$cWi) and *holds*(I7,$cWi) implies that instruction I2 is waiting on instruction I7 (*waits*(I2,I7)). Similarly, the edges *waits*(I6,$cEi) and *holds*(I3,$cEi) implies that instruction I6 is waiting on instruction I3 (*waits*(I6,I3)). We can now add the additional waits edges to the graph (shown in Figure 6-3).

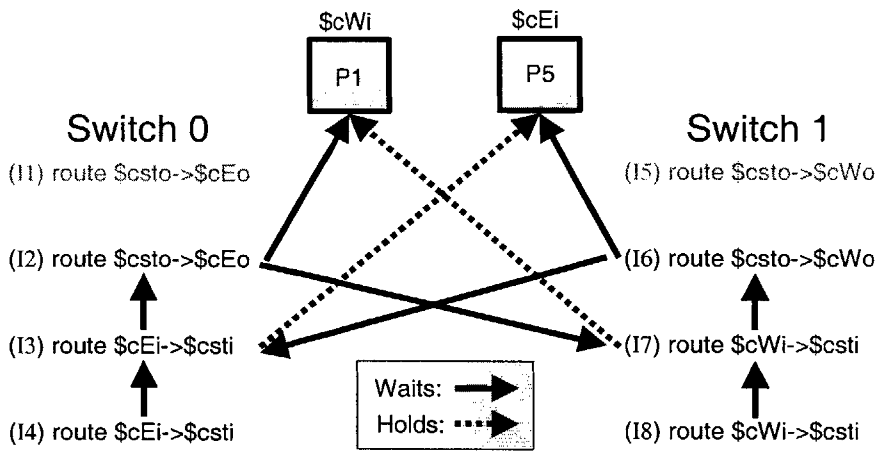


Figure 6-3: Code from Figure 6-2 with additional waits edges.

Inspection of Figure 6-3 reveals a cycle in the graph constructed from the waits edges (I3,I2,I7,I6,I3). Because a cycle exists in the waits graph, the schedule will result in deadlock. This can be grasped intuitively by inspecting the schedule. In the schedule, I3 is waiting on I2 to execute. I2 is waiting for the resource $cWi to be freed, while $cWi is waiting on I7 to execute before it can be freed. I7 must wait on I6 before it can execute, and I6 is waiting on the resource $cEi which is waiting on I3 to free it. The deadlock could be avoided by breaking the cycle in the waits graph. This is done by scheduling I3 above I2 or I7 above I6, removing either *waits*(I3,I2) or *waits*(I7,I6). Adding more storage to the network buffers also breaks the cycle, removing both *waits*(I2,I7) and *waits*(I6,I3) by removing the holds edges.

6.2.2 Switch Code Generation

To prevent the deadlock shown in Figure 6-3, phase 3 of the *Reptile* compiler generates switch code assuming only a single word of storage exists in the network. This can be done because *Reptile* starts with code containing no *crossover* patterns (*i.e.*, no communication routes that require more than a single word of network storage for proper execution). A crossover pattern is defined as the following: Let partition i and partition j both send and receive a word from each other. If the send occurs in both

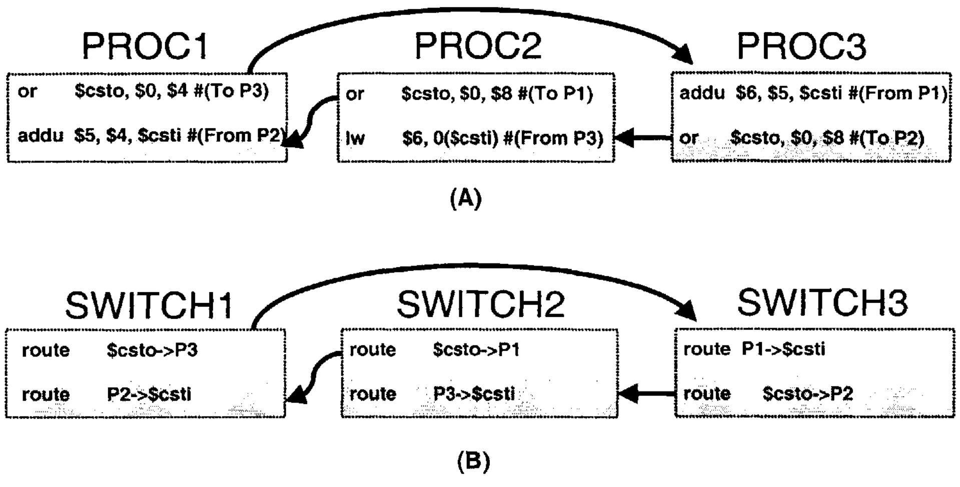

PROC1
or $csto, $0, $4 #(To P3)
addu $5, $4, $csti #(From P2)
PROC2
or $csto, $0, $8 #(To P1)
lw $6, 0($csti) #(From P3)
PROC3
addu $6, $5, $csti #(From P1)
or $csto, $0, $8 #(To P2)
(A)
SWITCH1
route $csto->P3
route P2->$csti
SWITCH2
route $csto->P1
route P3->$csti
SWITCH3
route P1->$csti
route $csto->P2
(B)

Figure 6-4: (A) 3-Tile processor code. (B) The corresponding switch code assuming an all-to-all network. Note that the switch instructions have Partitions IDs as operands rather than routing directions

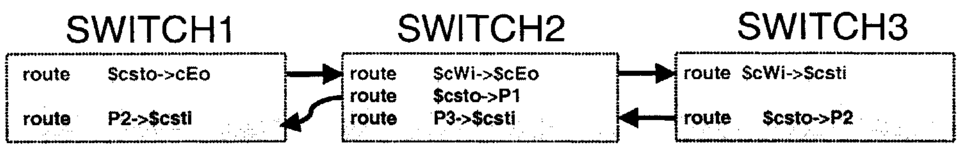


Figure 6-5: Switch code after routing the first send/receive pair.

partitions i and j before the receive, a crossover pattern occurs. The send/receive pairs in Figure 6-3 are all *crossover* routes.

Starting with code containing no *crossover* patterns, switch code generation occurs by first creating correct switch code for an all-to-all network. Figure 6-4 shows an example of processor code, (A), and corresponding switch code, (B), assuming an all to all network. Taking into account physical placement, the switch code generation portion of *Reptile* routes values by routing an entire send/receive pair at once (generating switch code for the end points as well as the code for the route the operand takes). The *Reptile* compiler uses dimension ordered routing to determine operands routes. Routing considers send/receive pairs for scheduling only if all preceding switch instructions for the send and receive have been scheduled. The constraint that the communication exhibit no *crossover* guarantees the existence of such a send/receive pair.

Figure 6-5 shows the switch code in Figure 6-4 after scheduling one send/receive pair (the first send on switch 1 and the first receive on switch 3). Note the addition of the added through route instruction on switch 2. Figure 6-6 shows the switch code after scheduling the second send/receive pair, and Figure 6-7 shows the switch code after scheduling all send/receive pairs.

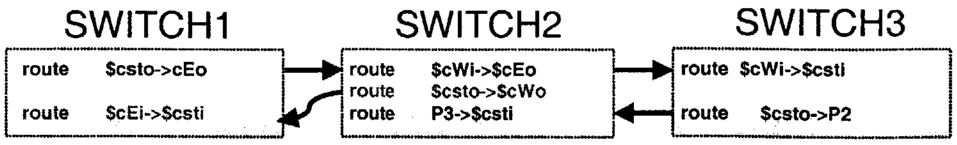


Figure 6-6: Switch code after routing the second send/receive pair.

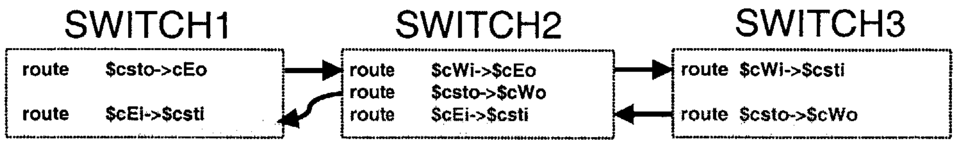


Figure 6-7: Switch code after routing the final send/receive pair.

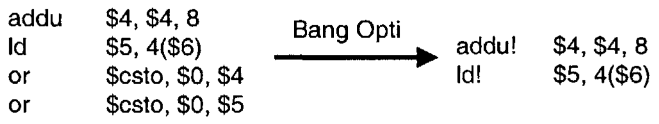
The switch code schedule obtained in Figure 6-7 does not contain any *crossover* routes, and only requires a single word of network storage for proper execution. Inspection of the original processor code in Figure 6-4 reveals that the scheduler could move the second instruction on switch 3 above the first. Similarly, the scheduler could reorder the first two instructions on switch 2. Both of the above schedule modifications would yield a better schedule, hiding the latency of the network by buffering values between switches. However, both suggested modifications to the schedule result in *crossover* routes, possibly creating deadlock. The following section will explain Queue Allocation, a scheduling technique allowing the reordering of instructions reading and writing from the same queue/buffer (creating *crossover* routes) without creating the deadlock seen in Figure 6-3.

6.3 Back End Optimizations

The *Reptile* compiler contains three simple post codegen peep-hole optimizations: *Bang* opti, *Use* opti, and Queue Allocation (*QA*).

6.3.1 *Bang* Opti

The *Bang* optimization simply sets the bang bits on instructions. The bang bit allows the instruction to effectively write to multiple locations, writing the result to a local register as well as to the network mapped registers. Figure 6-8 illustrates a sequence of code before and after *Bang* optimization. Notice that two instructions were removed, setting the bang bit on the first two instructions. *Bang* optimization occurs at the basic block level.



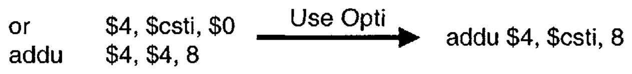
```
addu    $4, $4, 8
ld      $5, 4($6)
or      $csto, $0, $4
or      $csto, $0, $5
```

```
addu!   $4, $4, 8
ld!     $5, 4($6)
```

Figure 6-8: Example of code before the *Bang* opti (left), and after *Bang* opti (right).

6.3.2 *Use* Opti

The *Use* optimization modifies instructions to read directly from the network rather than first moving the value from the network into a register, then using the register. Figure 6-9 shows code before and after *Use* optimization.



```
or      $4, $csti, $0
addu    $4, $4, 8
```

```
addu $4, $csti, 8
```

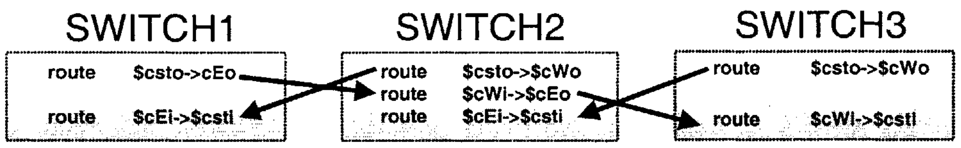
Figure 6-9: Example of code before the *Use* opti (left), and after *Use* opti (right).

6.3.3 Queue Allocation

Queue allocation reschedules the processor and switch code for all tiles. The queues are modeled by the algorithm, allowing simultaneous scheduling of instructions that communicate via queues along with regular instructions. This allows the reordering of send and receive instructions on the switch and processor, allowing *crossover* routes and better schedules by utilizing the storage present in the network. An example of the code in Figure 6-7 after *QA* may be seen in Figure 6-10 (the Queue Allocation was performed assuming two words of storage in the input queues).

6.4 *Reptile* ABI

The *Reptile* compiler designates one tile as the *critical tile.* The critical tile is responsible for managing the stack as well as parameter passing.



```
route   $csto->cEo
route   $cEi->$csti
```

```
route   $csto->$cWo
route   $cWi->$cEo
route   $cEi->$csti
```

```
route   $csto->$cWo
route   $cWi->$csti
```

Figure 6-10: Version of the switch code shown in Figure 6-7 after Queue Allocation.

6.4.1 Function Calls

Because of the parallel nature of code produced by *Reptile*, care must be taken when dealing with function calls. For example, consider the case of a 4-tile program. If a program running on all four tiles calls a function that was compiled with *Reptile* for 4-tiles, then all four tiles (and switches) must execute a function call. However, if the function was not compiled with *Reptile* (i.e. it is a library call like *printf*), or it was compiled for only one tile, then only one tile must execute the function call. This causes problems with function pointers, because the compiler cannot statically determine if a function call should be parallel or execute on just one tile. Currently, *Reptile* does not allow the use of function pointers.

6.4.2 Parameter Passing

Parameter passing is handled by the critical tile. The first four arguments are passed in registers 4-7, with the rest passed on the stack starting at address $sp + 16$ (see Figure 6-11). *Reptile* does not support 64 bit operands (longs or doubles), so the first four parameters are always passed in registers, regardless of type.

6.4.3 Stack Usage

The use of the stack (Figure 6-11) matches that of the MIPS ABI. Spatially, the stack memory is placed on the critical tile and stays on the critical tile for the entire execution of a program. This has several drawbacks, the first being the fact that no memory parallelism may exist. Additionally, when registers on other tiles need to be spilled, they are spilled on the critical tile. Future work will create a small stack for each tile, allowing local register spills.

6.4.4 The Switch

Control on the switch matches that of the processor. The local switch register, \$2 is used to store the branch condition. Additionally, switch register \$3 is used to store the switch return address. Both these registers are spilled upon entry to a function by storing them in the processors memory, and filled upon function return.

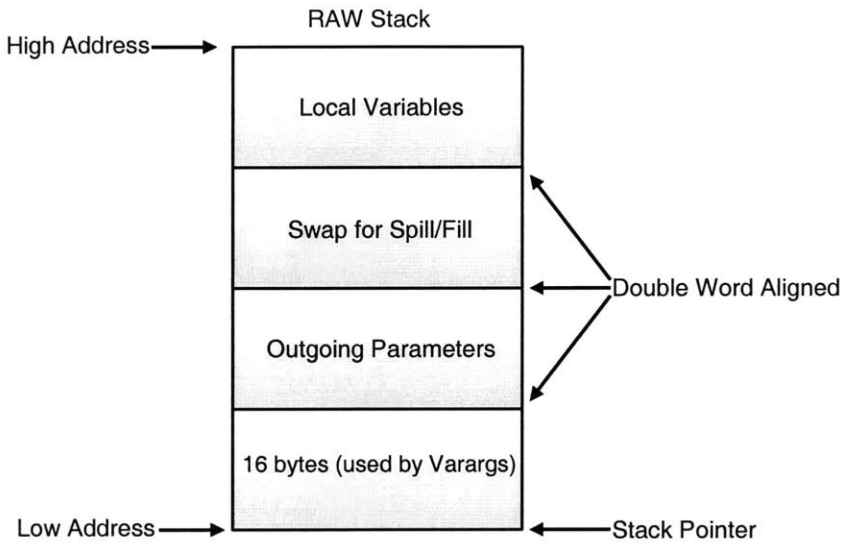


Figure 6-11: The Raw stack.

Chapter 7

Case Study: bzip2

7.1 Bzip2 C-code

To illustrate the *Reptile* compilation process, the following section will step through the compilation steps for a section of code taken from the benchmark *256.bzip2*, found in the SPEC [2] benchmark suite.

```
for (i = gs; i <= ge; i++) {
    UInt16 icv = szptr[i];
    cost0 += len[0][icv];
    cost1 += len[1][icv];
    cost2 += len[2][icv];
    cost3 += len[3][icv];
    cost4 += len[4][icv];
    cost5 += len[5][icv];
}
```

Figure 7-1: C-code taken from the *sendMTFValues* function in *256.bzip2*.

The C-code in Figure 7-1 was taken from the *sendMTFValues* function in *256.bzip2*. According to profile data, if all basic blocks in the *256.bzip2* program were ranked by the amount of execution time spent in the blocks, the basic block represented by the C-code in Figure 7-1 would rank highest. Profile information shows the program enters the loop thousands of different times, each time executing approximately 50 iterations. The accumulator variables ($costX$) are 16 bit values. Because the 2D array within the loop lacks structure, the loop does not parallelize well. For each iteration of the loop, the index to the second subscript of the array is loaded from another array, making it impossible to statically analyze. Additionally, the six consecutive loads from the 2D array do not access memory sequentially. A distance the size of the second dimension of the array separates each load. Any parallelism that exists within the loop is simply at the instruction level. No loop unrolling may occur and, because of the noncontiguous nature of the 2D array access, no data parallelism may be exploited.

```
(cb 93 3778304.000000 [(flow 1 93 3704212.000000)(flow 0 94 74092.000000)
  (op 254 ld_uc2    [(r146 i)] [(r670 i)(i0)]
  (op 256 zxt_c2    [(r144 i)] [(r22 i)])
  (op 259 ld_uc     [(r148 i)] [(r146 i)(l_g_gp _len)]
  (op 261 add_u     [(r149 i)] [(r144 i)(r148 i)])
  (op 262 zxt_c2    [(r22 i)] [(r149 i)])
  (op 266 zxt_c2    [(r154 i)] [(r23 i)])
  (op 268 zxt_c2    [(r156 i)] [(r146 i)])
  (op 270 ld_uc     [(r159 i)] [(r157 i)(r156 i)]
  (op 272 add_u     [(r160 i)] [(r154 i)(r159 i)])
  (op 273 zxt_c2    [(r23 i)] [(r160 i)])
  (op 277 zxt_c2    [(r165 i)] [(r24 i)])
  (op 281 ld_uc     [(r170 i)] [(r168 i)(r156 i)]
  (op 283 add_u     [(r171 i)] [(r165 i)(r170 i)])
  (op 284 zxt_c2    [(r24 i)] [(r171 i)])
  (op 288 zxt_c2    [(r176 i)] [(r25 i)])
  (op 292 ld_uc     [(r181 i)] [(r179 i)(r156 i)]
  (op 294 add_u     [(r182 i)] [(r176 i)(r181 i)])
  (op 295 zxt_c2    [(r25 i)] [(r182 i)])
  (op 299 zxt_c2    [(r187 i)] [(r26 i)])
  (op 303 ld_uc     [(r192 i)] [(r190 i)(r156 i)]
  (op 305 add_u     [(r193 i)] [(r187 i)(r192 i)])
  (op 306 zxt_c2    [(r26 i)] [(r193 i)])
  (op 310 zxt_c2    [(r198 i)] [(r27 i)])
  (op 314 ld_uc     [(r203 i)] [(r201 i)(r156 i)]
  (op 316 add_u     [(r204 i)] [(r198 i)(r203 i)])
  (op 317 zxt_c2    [(r27 i)] [(r204 i)])
  (op 321 add       [(r3 i)] [(r3 i)(i1)])
  (op 1153 add      [(r670 i)] [(r670 i)(i2)])
  (op 323 br ile    [(r3 i)(r713 i)(cb 93)]
```

Figure 7-2: *lcode* generated by the OpenImpact compiler.

7.2 Bzip2: *Reptile* phase 1

Phase 1 of the *Reptile* compiler parses the C-code, profiles, optimizes, and outputs a textual representation of the Impact IR. For a more in-depth explanation of this phase, see Section 4.2. The textual representation of the Impact IR, called *lcode* corresponds roughly to a RISC-like 3-operand ISA. Figure 7-2 shows the Impact *lcode* generated from the C-code seen in Figure 7-1. The *lcode* assumes an infinite number of registers. The operations in the code are numbered, with the first operation being op 254. There are 7 loads in the code, with the first load determining the value for the second index in the 2D array. Inspection of the code in figure Figure 7-2 reveals that register 670 contains the value of *szptr+i*. Every iteration of the loop, register 670 is increased by 2 because the array *szptr* is composed of half words. The other 6 loads in the code correspond to the 2D array accesses. The rest of the instructions in the block are needed to perform address computation overhead, and clear out the upper 16 bits of the half-word operands.

7.3 Bzip2: *Reptile* phase 2

Phase 2 of the *Reptile* compilation process reads in the *lcode* from the previous section, partitions the code, register allocates, and schedules. For brevity, the actual textual output of this phase of the compilation will not be included.

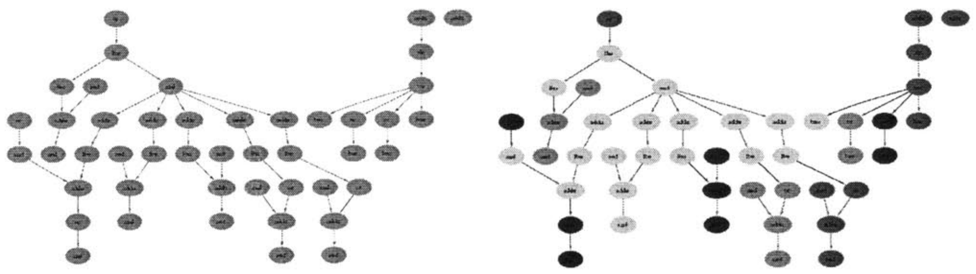


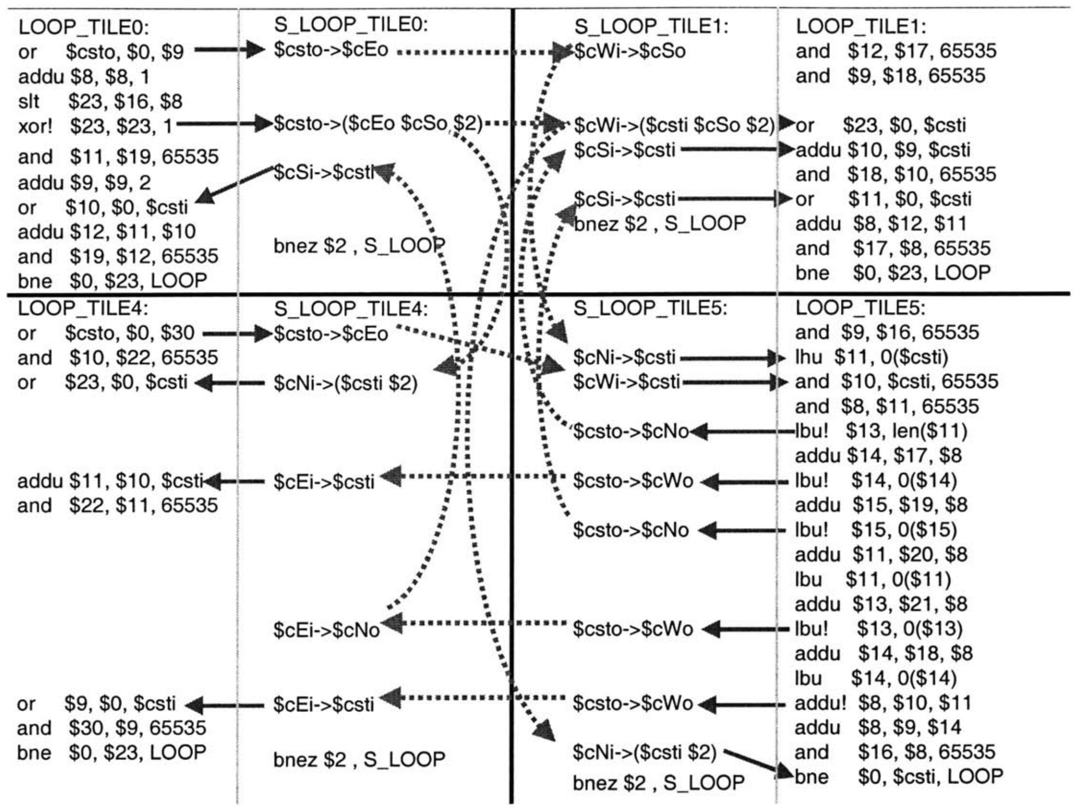
Figure 7-3: Dataflow graph before partitioning (left) and after partitioning (right).

Figure 7-3 displays the dataflow graph of the compiled code before partitioning (left), and after partitioning (right). The different colored nodes in the graph on the right represent different partitions. Each partition will eventually be placed on a single Raw *tile*. Dataflow edges between partitions must be routed via the static network. It is obvious from the graph that one partition contains more nodes than the others. This is a result of the fact that all memory instructions must be scheduled to the same partition. Because the example code contains 7 memory instructions, the partitions containing the memory instructions ends up larger than the rest. After partitioning, the code is scheduled assuming an all-to-all network. Register allocation occurs assuming each partition has access to a local register file.

7.4 Bzip2: *Reptile* phase 3

Phase three of the *Reptile* compiler takes in the partitioned code from the previous section, places the partitions to physical Raw tiles, generates switch code to route operands between partitions and performs peep-hole optimizations on the generated assembly.

Figure 7-4 includes assembly code from the final output of the *Reptile* compiler for a 4-tile square configuration. The figure is divided into four quadrants, one for each tile. Each quadrant contains two assembly schedules, one for the processor (assembly on the outside), and one for the switch (assembly on the inside). Solid arrows denote reads and writes between the processor and the switch. Similarly, dashed arrows represent dependences between the switches. Note that no dashed arrows exist between tile 0 and tile 5. This is because tile 0 cannot communicate directly with tile 5, instead, values must be routed through tile 4 or tile 1 and then onto their final destination. Similarly, values may not travel directly from tile 4 to tile 1. Instead, they must be routed through either tile 0 or tile 5. Each tile executes its own control flow, with one tile computing the branch condition (in this case the fourth instruction of tile 0) and sending it to the other tiles. Additionally, the control flow on the switches mirrors that of the processors. When a branch condition is routed from one switch to the next, it is also stored in the local switch register ($2). At the end of the code on the switch, the switch executes a conditional branch based on the register containing the branch condition. Because *Reptile* currently does not support memory parallelism, all loads occur on the same tile (in this case, Tile 5).



```
LOOP_TILE0:
or    $csto, $0, $9
addu $8, $8, 1
slt   $23, $16, $8
xor!  $23, $23, 1
and   $11, $19, 65535
addu $9, $9, 2
or    $10, $0, $csti
addu $12, $11, $10
and   $19, $12, 65535
bne   $0, $23, LOOP
```

```
S_LOOP_TILE0:
$csto->$cEo
$csto->($cEo $cSo $2)
$cSi->$csti
bnez $2 , S_LOOP
```

```
S_LOOP_TILE1:
$cWi->$cSo
$cWi->($csti $cSo $2)
$cSi->$csti
$cSi->$csti
bnez $2 , S_LOOP
```

```
LOOP_TILE1:
and   $12, $17, 65535
and   $9, $18, 65535
or    $23, $0, $csti
addu $10, $9, $csti
and   $18, $10, 65535
or    $11, $0, $csti
addu  $8, $12, $11
and   $17, $8, 65535
bne   $0, $23, LOOP
```

```
LOOP_TILE4:
or    $csto, $0, $30
and   $10, $22, 65535
or    $23, $0, $csti
addu $11, $10, $csti
and   $22, $11, 65535
or    $9, $0, $csti
and   $30, $9, 65535
bne   $0, $23, LOOP
```

```
S_LOOP_TILE4:
$csto->$cEo
$cNi->($csti $2)
$cEi->$csti
$cEi->$cNo
$cEi->$csti
bnez $2 , S_LOOP
```

```
S_LOOP_TILE5:
$cNi->$csti
$cWi->$csti
$csto->$cNo
$csto->$cWo
$csto->$cNo
$csto->$cWo
$csto->$cWo
$cNi->($csti $2)
bnez $2 , S_LOOP
```

```
LOOP_TILE5:
and $9, $16, 65535
lhu $11, 0($csti)
and $10, $csti, 65535
and $8, $11, 65535
lbu!  $13, len($11)
addu $14, $17, $8
lbu!  $14, 0($14)
addu $15, $19, $8
lbu!  $15, 0($15)
addu $11, $20, $8
lbu   $11, 0($11)
addu  $13, $21, $8
lbu!   $13, 0($13)
addu  $14, $18, $8
lbu    $14, 0($14)
addu! $8, $10, $11
addu  $8, $9, $14
and     $16, $8, 65535
bne     $0, $csti, LOOP
```

Figure 7-4: Raw assembly for a 4-tile version of the example code (tiles are numbered, starting in the upper left and proceeding clockwise, 0,1,5,4). Each quadrant corresponds to a Raw tile. The two pieces of assembly in the upper left are the processor and switch code for tile 0. Similarly, the assembly in the upper right is the processor and switch code for tile 1. The solid edges represent data dependences between the switch and the processor. The dashed arrows represent data dependences between switches.

Chapter 8

Results

8.1 Experimental Methodology

To assess the *Reptile* compiler, two different applications were chosen. The first application, a complex finite impulse response filter (*cfir*), was chosen to see how well *Reptile* exploits applications with large amounts of parallelism. The *cfir* was modified for 2-tap, 4-tap, 8-tap and 16-tap filters. The second application, a kernel from the *256.bzip2* benchmark (see Section 7.1), was chosen to assess *Reptile* performance for unstructured code with low amounts of parallelism.

Performance numbers for the benchmarks compiled with *Reptile* were obtained from the Raw simulator (*btl*). To compare performance results for the *Reptile* generated programs several other architectures were evaluated. Both benchmarks were compiled and run on a Pentium 3, athlon64, and Itanium 2. However, simply comparing against other architectures does not only assess the *Reptile* compiler, but the Raw architecture as well. Therefore, to fully evaluate the quality of the code generated by *Reptile*, results were gathered for the *cfir* benchmark implemented in the Streamit language. The Streamit language is a language developed at MIT for targeting signal processing applications. Streamit explicitly expresses pipeline parallelism at the language level, allowing the Streamit compiler to generate high quality code targeting the Raw architecture. By comparing *Reptile* generated code to Streamit generated code, we may assess the extent to which *Reptile* can automatically find the parallelism that natively exists within a c-program.

A straight comparison of Raw to other architectures is complicated by the fact that the Raw hardware only supports software based icaching. Nothing intrinsic to the Raw design requires software based icaching, it was simply included in the Raw prototype to explore the issues around software icaching and to facilitate research. The Raw simulator (*btl*), however, does support hardware icaching. Before claiming that the results gathered from the simulator accurately represent the Raw hardware, the simulator must first be validated by the hardware. By experimentally showing that the simulator (when using software icaching), generates results faithful to those from the hardware, one can make the claim that the addition of an accurate hardware icaching scheme to the simulator would yield cycle counts representative of a Raw architecture with hardware based icaching.

8.1.1 Simulator Validation

The various configurations of the *cfir* benchmark were compiled with *Reptile* and run under software icaching on both the simulator and the hardware. Figure 8-1 displays the results for all configurations. The four plots correspond to 2-tap, 4-tap, 8-tap and 16-tap versions of the filter. With the 2-tap version being in the upper left, the 4-tap in the upper right, etc. The x-axis on all plots represents the number of tiles used by the program. The bars on the left correspond to cycle counts received from the Raw simulator with software icaching, while the bar on the right corresponds to cycle counts taken from the actual Raw hardware. The difference between the two numbers is fairly negligible, with the hardware on average reporting a 3% larger execution time. Notice that as the benchmark size increases (from 1-16 taps), the error between hardware and simulator decreases, implying that longer running benchmarks show little performance variation between the hardware and the simulator. Because the *btl* simulator accurately reflects the Raw hardware when running in software icaching code, we will use a carefully augmented version of the simulator to generate results for a theoretical Raw machine supporting hardware icaching.

8.2 Performance Results

The following sections report performance results for code generated via the *Reptile* compiler. Numbers were gathered using the Raw *btl* simulator with hardware icaching.

8.2.1 Complex FIR

To explore how well *Reptile* parallelizes applications with large amounts of parallelism, a *cfir* benchmark was compiled with for tap sizes. Results were gathered from the Raw simulator with hardware icaching. Additionally, cycle counts for the *cfir* benchmark on a Pentium 3, athlon64 and Itanium 2 were included as well as results for Streamit generated code.

To understand the parallelism within the *cfir* benchmark exploited by the *Reptile* compiler, Figure 8-2 displays the dataflow graph for a 2-tap version of *cfir*. The different node colors correspond to different partitions. Nodes of the same color execute on the same physical Raw tile. Additionally, Figure 8-3 displays the dataflow graph for a 4-tap version of *cfir*. Note that increasing the number of taps in *cfir*resulted in a wider dataflow graph without increasing graph depth. The increase in width without lengthening the graph implies that increasing the number of taps in the filter increases the parallelism within the program, almost linearly.

Figure 8-4 shows performance results for the various *cfir* filters taken from the Raw simulator using hardware icaching. The y-axis reports speedup relative to a single tile version while the x-axis corresponds to the Raw configuration (2, 4, 8, and 16 tiles).

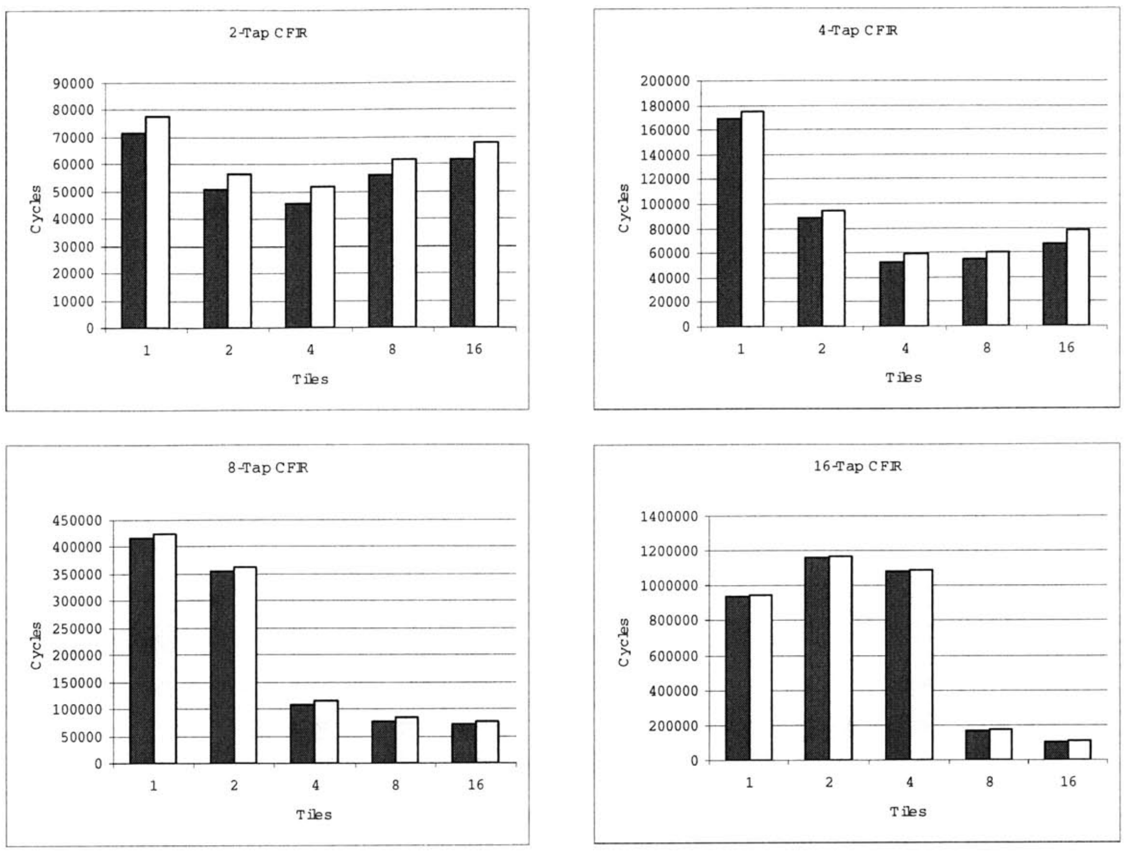


Figure 8-1: Cycle counts for the various versions of *cfir* compiled with the *Reptile* compiler. The x-axis represents different numbers of Raw tiles. The bar on the left corresponds to cycle count running on the *btl* simulator with software icaching. The bar on the right corresponds to running on the Raw motherboard with software icaching.

All versions were generated with the *Reptile* compiler. The graph in the upper left (a 2-tap *cfir*) shows a max speedup for the 4-tile Raw configuration. This is due to the fact that the 2-tap version only contains a finite amount of parallelism. Adding additional tiles yields no performance improvement. Additionally, the 8 and 16 tile versions of the program actually show a decrease in performance relative to the 4 tile version. This reflects the fact that the *Reptile* partitioning algorithm is imperfect, and will sometimes parallelize too aggressively, resulting in a worse schedule than had fewer tiles been used. In the case of the 2-tap *cfir* version (upper right in Figure 8-4), the 4-tile Raw configuration yields the largest speedup. Both the 8-tap and 16-tap versions achieve the largest speedup with 16-tile Raw configurations. This implies that the 8-tap and 16-tap *cfir* benchmarks have enough parallelism to utilize at least 16 Raw tiles. Notice that in the case of the 8-tap version, moving from 1-tile to 2-tiles shows very little performance improvement. This is an artifact of the way *Reptile* currently handles spill code. Because *Reptile* assumes only a single tile contains the stack, register values from other tiles are spilled into the single stack. This means

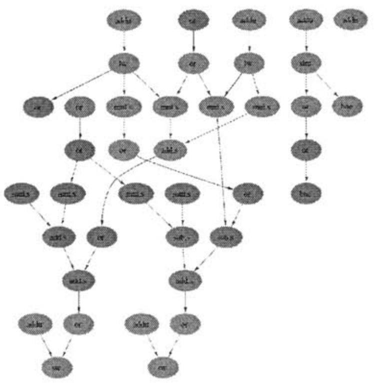


Figure 8-2: Dataflow graph of a two tap *cfir*.

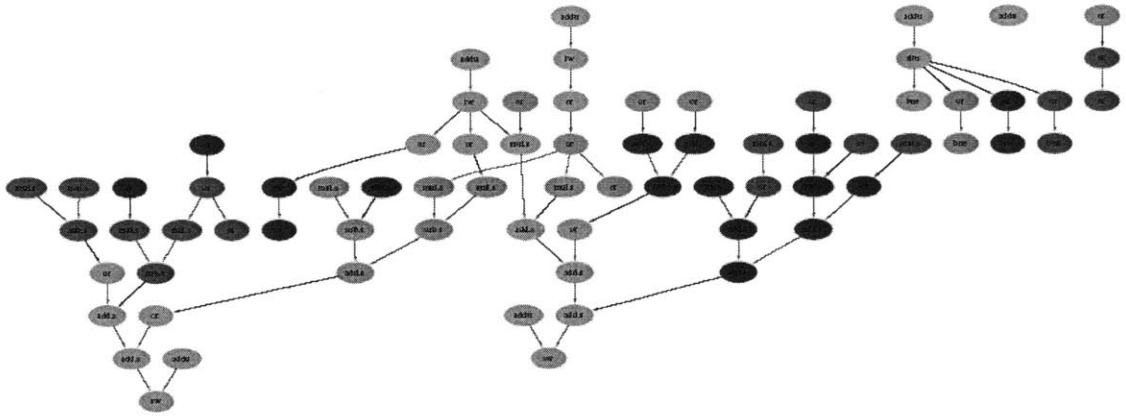


Figure 8-3: Dataflow graph of a 4-tap *cfir*.

that register spill instructions generate communication on the static network. In the case of the 8-tap version of *cfir*, a four tile version provides enough registers that no values need to be spilled, avoiding the communication costs of register spills. Future versions of *Reptile* will have a distributed stack model, allowing each tile to spill to their local memory space.

Figure 8-5 shows the performance results for the 2-tap, 4-tap, 8-tap and 16-tap versions of *cfir* compared to other architectures. The best Raw configuration was used to represent performance for Raw (2x2 for 2-tap and 4-tap, 4x4 for 8-tap and 16-tap). The results for the Pentium 3, Athlon 64, and Itanium 2 were obtained by compiling the various versions of *cfir*for each architecture, with the most aggressive optimizations supported by *gcc* (for the Pentium and athlon) and *ecc* (for the itanium) enabled. The numbers reported are processor cycle counts. The Streamit results were obtained by compiling a Streamit version of the program for the Raw architecture. Each cycle count was normalized to the Pentium 3 cycle count. The *Reptile* compiler outperforms the Pentium 3 and athlon 64 for the 4-tap, 8-tap and 16-tap versions of *cfir*. This implies that the *Reptile* compiler is capable of finding and exploiting ILP if the application contains enough parallelism. *Reptile* performs within a factor of 2-3x of the Streamit compiler, implying that *Reptile* does not find *all* of the parallelism within the benchmark, and significant work still needs to be done. The Streamit versions continually show the largest speedup over the Pentium 3, from 2.4x for the 2-tap case to 5.6x for the 16-tap version. Additionally, the Streamit numbers reported were generated with the "old" Streamit backend. It is believed that the "new" Streamit backend would have achieved significantly better results.

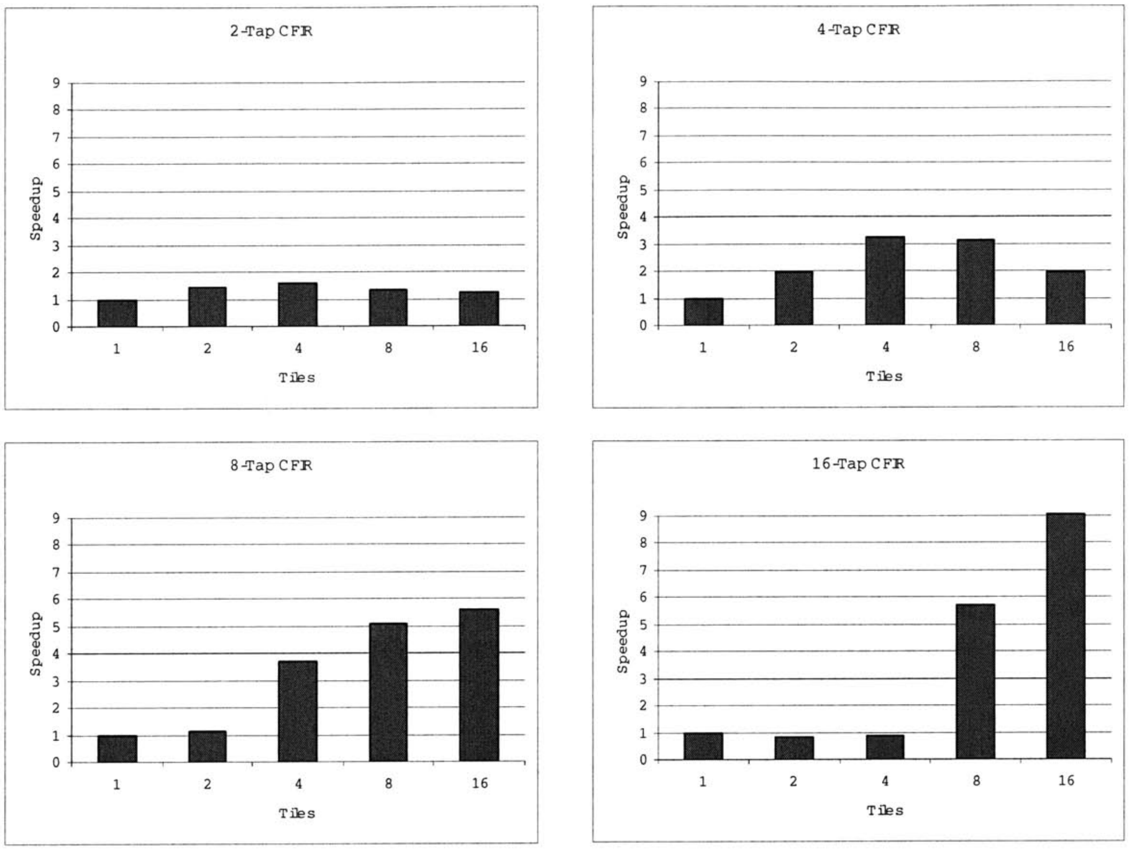


Figure 8-4: Speedups relative to a single Raw tile for variously sized versions of *cfir* compiled with *Reptile*. The x-axis corresponds to different tile configurations. The numbers were generated via the *btl* simulator augmented with hardware icaching.

8.2.2 *256.Bzip2* kernel

The *256.bzip2* benchmark kernel was described thoroughly in Section 7.1. Figure 8-6 contains the dataflow graph for the kernel taken from *256.bzip2*. The section of code contains significantly less parallelism than a 4-tap *cfir*(Figure 8-3). Results for the *256.bzip2* kernel were obtained by compiling the code with *Reptile* and choosing the tile configuration yielding the best performance (in this case, a 2x2). The *Reptile* generated code was run on the *btl* simulator with hardware icaching. The same code

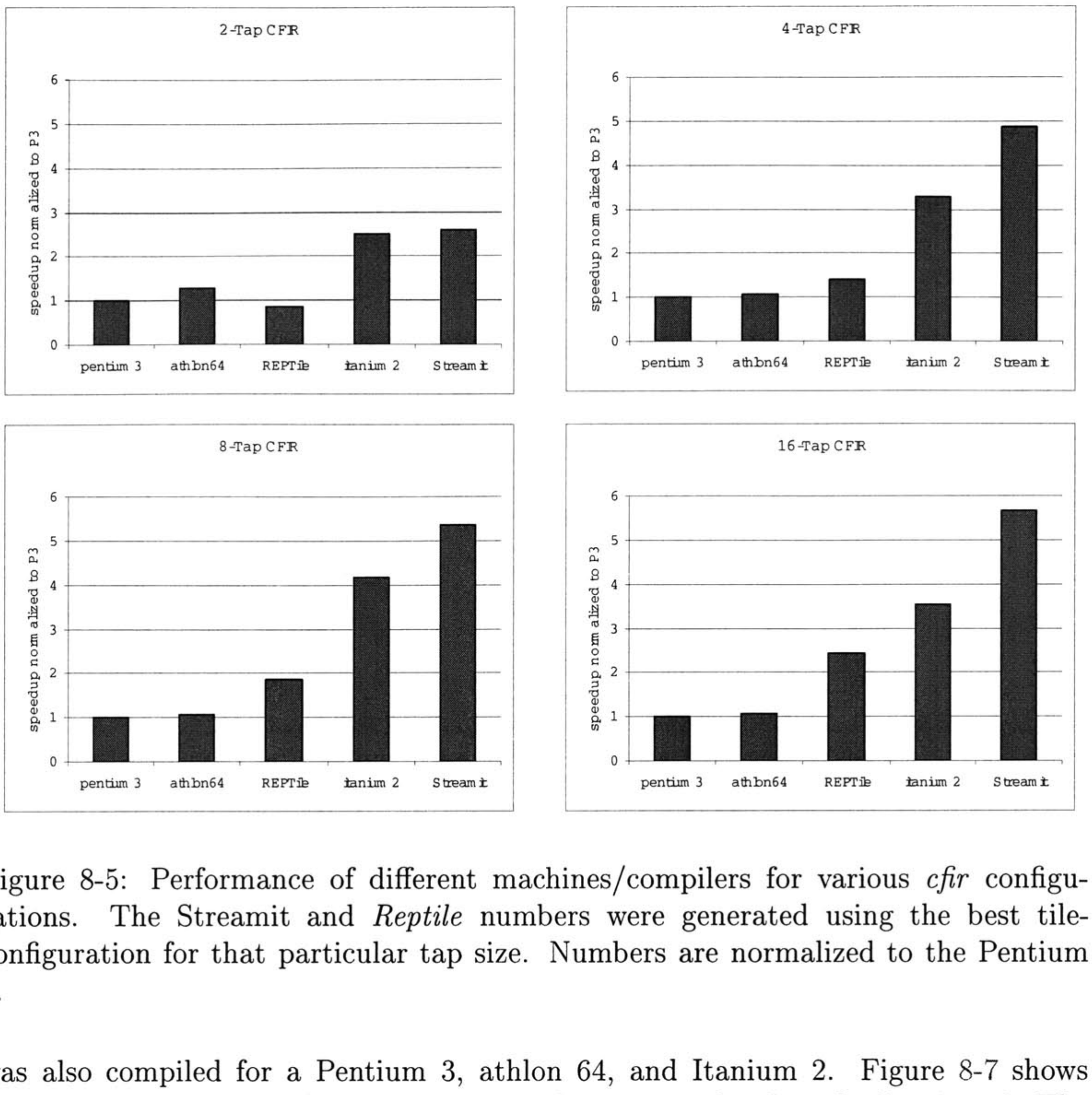


Figure 8-5: Performance of different machines/compilers for various *cfir* configurations. The Streamit and *Reptile* numbers were generated using the best tile-configuration for that particular tap size. Numbers are normalized to the Pentium 3.

was also compiled for a Pentium 3, athlon 64, and Itanium 2. Figure 8-7 shows the performance results for the various machines normalized to the Pentium 3. The Itanium 2 achieves nearly a 10x speedup over the Pentium 3. Inspection of the generated Itanium assembly reveals that this speedup is achieved via aggressive software pipelining. The *Reptile* code obtains a speedup of 2.1x over the Pentium 3, but only comes within a factor of 2 of the athlon 64.

8.3 Placement Sensitivity

In the *Reptile* compiler, after a program is partitioned into several virtual partitions, each partition must be mapped to a physical Raw tile. Section 6.1 described a method for placement based upon simulated annealing, as well as a method based upon integer quadratic programming. To understand how placement affects performance, the *Reptile* compiler generated two different versions of the various versions of *cfir*. One version places partitions for Raw in an effort to minimize the amount of communication (*good* placement), while the other version places partitions on Raw with the

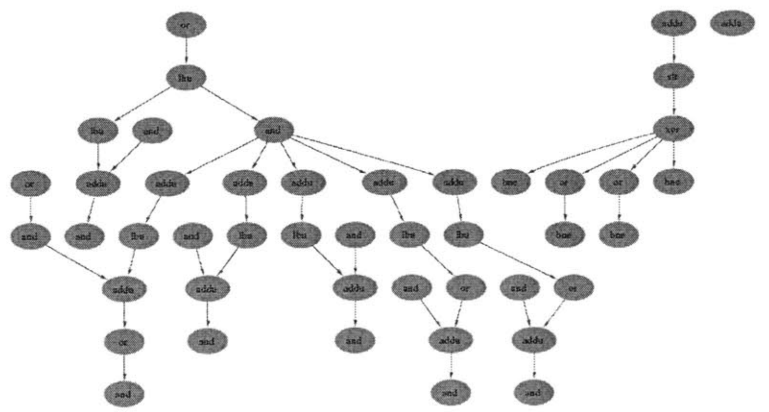


Figure 8-6: Dataflow graph of the *256.bzip2* kernel

intent of maximizing communication (*bad* placement). For example, if two partitions communicate quite often, the first *good* partitioning will place the partitions physically close to each other where as the *bad* partitioning will place the partitions as far apart as possible.

Figure 8-8 shows the results of the *good* and *bad* placement algorithms. The simulated annealing algorithm was used, with the metric $c_{i,j}$ representing the number of words sent from virtual partition i to virtual partition j over the Raw static network. The benchmarks were run on the Raw simulator with hardware icaching. The cycles counts were normalized to the *bad* partitioning. Figure 8-8 illustrates that as the size of the benchmark increased, the sensitivity to placement also increased. For example, both the 8-tap and 16-tap versions of *cfir* use a 4x4 tile configuration. However, the 8-tap version achieves a 1.47x speedup by partitioning to minimize communication where as the 16-tap version achieves a 1.56x speedup. An interesting observation is that the *bad* placement for the 2-tap version actually does better than the *good* placement. This is most likely due to the fact that the placement objective function is imperfect. While minimizing total communication on the network does well in most cases, it is not necessarily the best metric for all programs. For example, one could consider the case where partition i sends a large amount of words to partition j, but partition j does not send any words to partition i. In this case, placing the two partitions close together achieves no performance improvement over placing them far apart. This is because of the fact the because there are no cycles (i to j, then j to i), the communication can be pipelined.

8.4 Peep-hole Optimizations

Section 6 described the three peep-hole optimizations implemented by *Reptile*. Figure 8-9 shows the effect of the different peep-hole optimizations on performance. The 2-tap, 4-tap, 8-tap and 16-tap versions of the *cfir* benchmark were compiled with *Reptile* with different peep hole optimizations turned on. The results show the in-

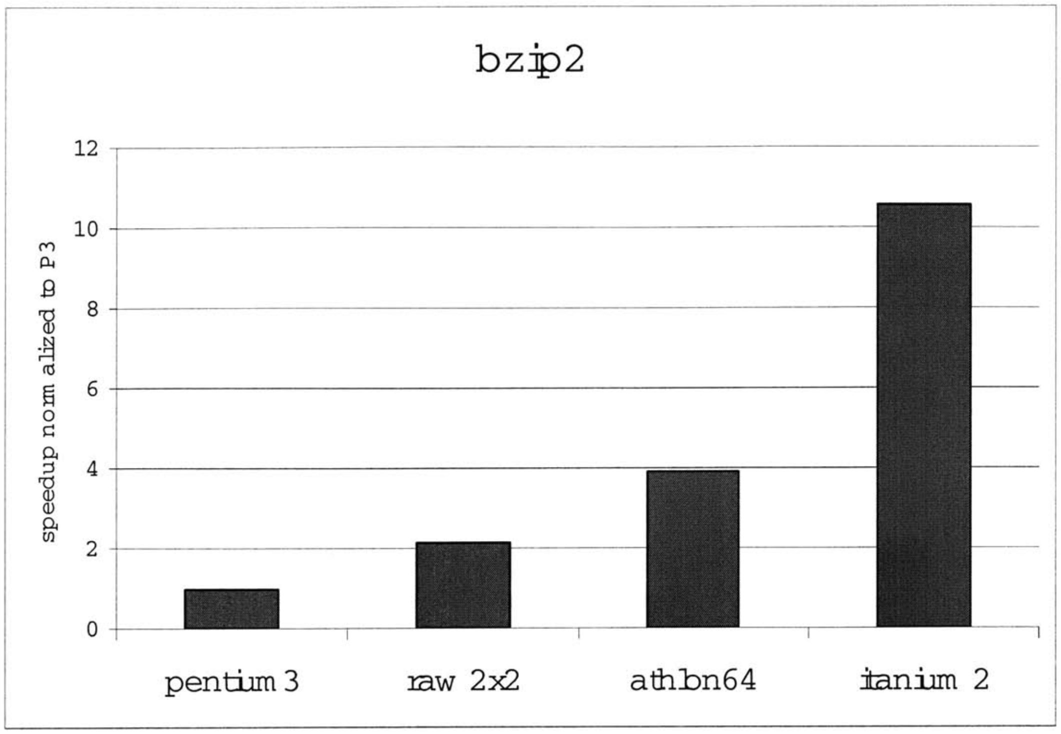


Figure 8-7: Speedup of different machines on the *256.bzip2* benchmark kernel relative to the Pentium 3.

crease in performance as the *Bang* opt, *Use* optimizations, and Queue Allocation are performed. The results were generated from *btl* using hardware icaching. According to the results, the two larger versions of the *cfir* benchmark gain less from queue allocation than the smaller versions. The most likely explanation for this behavior is that for the larger versions of the benchmark the amount of computation relative to communication is larger than for the smaller versions of *cfir*, meaning that the 8-tap and 16-tap versions benefit less from reordering the communication.

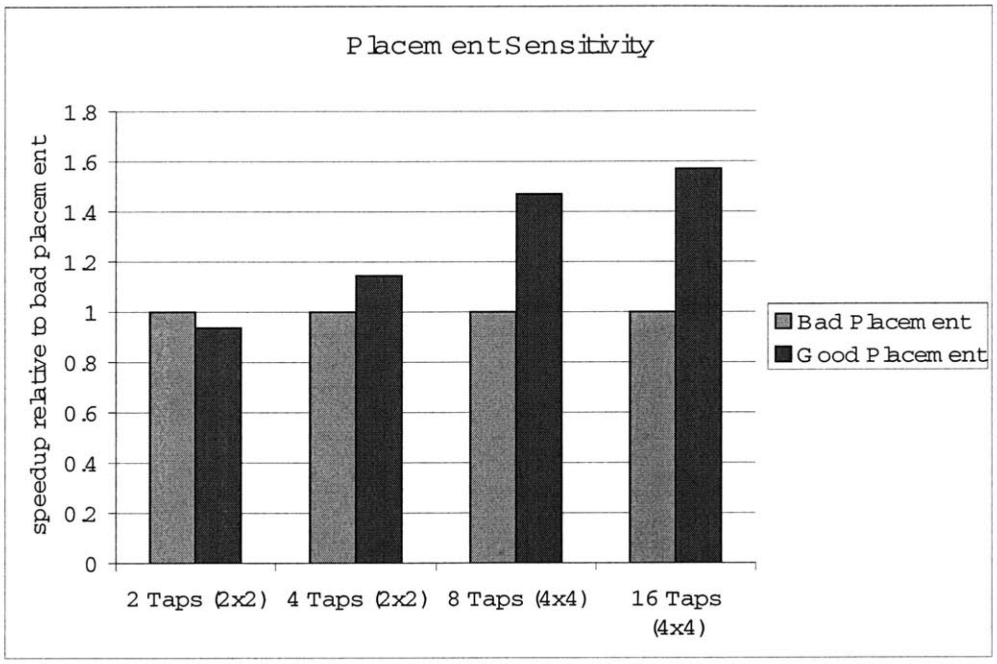


Figure 8-8: Placement maximizing communication (bar on left), placement minimizing communication (bar on right) for 4 implementations of *cfir*

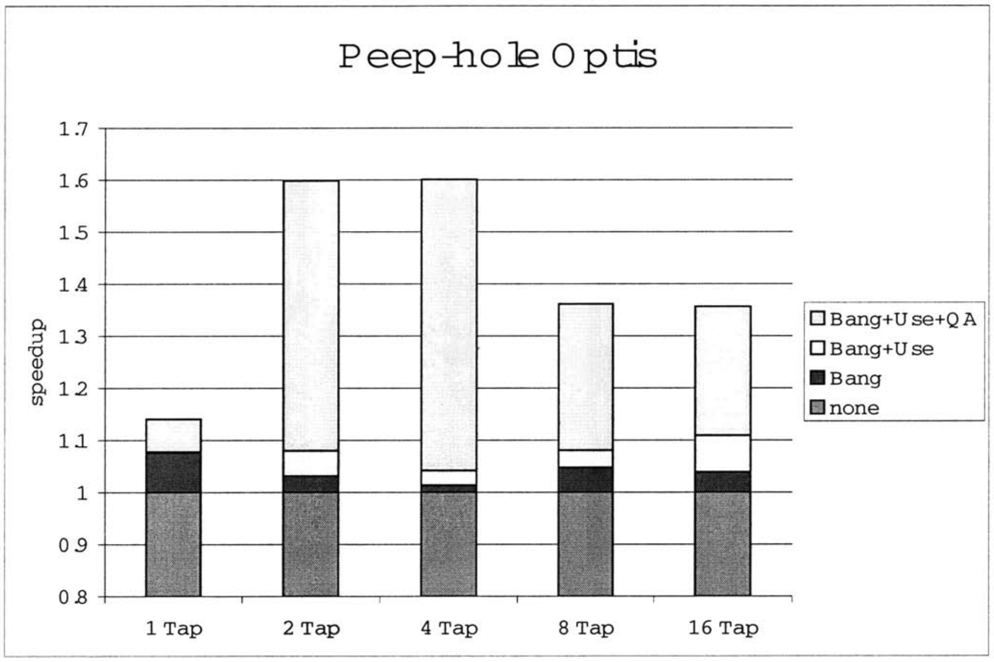


Figure 8-9: The speedup obtained via various peep-hole optimizations on the *cfir* benchmark.

Chapter 9

Summary

9.1 Conclusion

The *Reptile* compiler parallelizes arbitrary ANSI-c programs at the instruction level and maps the parallel computation to the Raw architecture. Simulated Annealing is used to partition the instruction dataflow graph. After partitioning, simulated annealing is again used to determine a mapping from partition to physical Raw tile. Once each partition is placed to a Raw tile, switch code is generated to route operands across the network. The switch code contains no crossover routes and assumes an all-to-all network. Legal switch code is generated, using dimension ordered routing to determine the operand routes. After processor and switch code generation, a suite of peep-hole optimizations is performed to increase code quality. On applications with sufficient amounts of parallelism, *Reptile* is able of achieving significant performance increases, outperforming modern out of order superscalars. *Reptile* is able to come within a factor of 2-3x of code generated by the Streamit compiler, implying that *Reptile* is not capturing all of the parallelism inherent to the code. On an unstructured integer example, the *Reptile* compiler is capable of achieving a performance within 2x of a modern superscalar.

9.2 Future Work

Future work for the *Reptile* project includes increasing the robustness of the existing *Reptile* compiler as well as utilizing the *Reptile* infrastructure to explore new issues in DILP compilation.

The following projects will increase the stability and performance of *Reptile*:

- Implement an intelligent, post codegen scheduling pass that unifies switch and processor code scheduling.
- Implement a suite of post-codegen classical optimizations.
- Integrate the Suif equivalence class alias analysis into the *Reptile* compiler.
- Modify the *Reptile* stack model to handle multiple stacks, one for each tile.

The *Reptile* compiler can be used to explore the following DILP compilation issues:

- Explore the issues surrounding an ABI supporting parallel library calls and parallel indirect function calls.
- Use Trace scheduling to increase the performance of *Reptile* on integer applications.
- Investigate the issues surrounding code duplication. When can code be duplicated across tiles to avoid communication?
- Explore spatial software pipelining.

Appendix A

Simulated Annealing

This section gives a quick background on Simulated Annealing, a technique used extensively in the *Reptile* compiler.

Simulated Annealing belongs to the class of stochastic optimization algorithms. Stochastic optimization algorithms utilize randomness to obtain approximate solutions to difficult optimization problems.

The name "Annealing" applies to the process of heating up a solid until the atoms are in a sufficiently random state, then allowing the solid to cool such that the atoms within the solid reach a state of minimum energy at the final temperature. If the solid does not reach a high enough temperature initially, or if the cooling occurs too quickly (a process called quenching), the final state will not have minimal energy.

Simulated Annealing, first proposed by Metropolis [14], leverages computer simulation methods to "simulate" the annealing process. The simulation occurs in the following steps. Let $\mathbf{Q_i}$ represent the position of the atoms at step i. Similarly, let E_i represent the energy of the configuration $\mathbf{Q_i}$. To simulate the randomness of the atoms, random perturbation of the atoms results in a new configuration, $\mathbf{Q_j}$. After perturbation, a new energy, E_j, describes the energy of the perturbed system. If the new energy, E_j proves to be smaller than the previous energy, E_i, $\mathbf{Q_j}$ becomes the new state of the system. If the perturbed system has more energy than the previous system, $(E_j > E_i)$, then the new state becomes the current state according to a probability given by:

$$e^{\left(\frac{E_i - E_j}{k_B T}\right)} \tag{A.1}$$

Where T corresponds to the simulated "temperature" and k_B to Boltzmann's constant. Notice that the probability of accepting a higher energy state decreases as the temperature decreases. By accepting changes that increase energy, the simulated annealing algorithm "escapes" from local minima. At very low temperatures, the simulation accepts only changes in configuration that decrease system energy (analogous to a hill climbing algorithm). Simulated Annealing consists of many repetitions of the above algorithm, decreasing the simulated "temperature" after a fixed number of repetitions. The process repeats, stopping the simulation when the temperature reaches a value sufficiently close to zero.

The above algorithm applies not only to literally simulated "annealing", but also to difficult combinatorial optimizations problems [10]. For example, consider the Traveling Salesman Problem (TSP), known to be NP-complete. The TSP problem attempts to find an optimal route for a traveling salesman that must visit many destinations. The "optimal" route minimizes travel time. To apply Simulated Annealing to the TSP problem, the energy becomes the travel time of the salesman, and the route the salesman takes becomes the configuration of particles. Random perturbations switch the ordering of visited cities, thereby changing the travel time (energy) of the system. Of course, simulated annealing provides no guarantee of optimality, however, many real world problems only require an approximate solution. The pseudo-code below implements the simulated annealing algorithm.

```
$\mathbf{Q}$ = random_initial_guess()
$T = startT$
Begin Outer Loop over $T$

    $L = startL$
    Begin Inner Loop over $L$
        $\mathbf{Q}'$ =perturb($\mathbf{Q}$)
        $oldCost = getCost(\mathbf{Q})$
        $newCost = getCost(\mathbf{Q}')$
        if($(oldCost - newCost) > 0$)
            $\mathbf{Q} = \mathbf{Q}'$
        else
            if($e^{\left(\frac{oldCost-newCost}{T}\right)} > rand[0:1]$)
              $\mathbf{Q} = \mathbf{Q}'$
            endif
        endif
        $L = getNextL()$
    End Inner Loop
    $T = getNextT()$
End Outer Loop
```

For the case of the TSP, the state $\mathbf{Q}$ described in the above pseudo code corresponds to a particular route visiting all cities. The cost of the route, $getCost(\mathbf{Q})$, corresponds to the traveling time required for that particular route, $\mathbf{Q}$.

No "correct" values exist for the starting temperature *startT* and the bound on the inner loop. The choice for these parameters varies upon the problem as well as the desired running time. Therefore, common implementations of simulated annealing leverage a heuristics based approach for determining the values of *startT* and the inner loop bound. A good choice for the inner loop bound in the case of the TSP might be to iterate the inner loop as many times as there are cities to visit. The "cooling schedule", the function that determines the next temperature, is also chosen heuristically.

Bibliography

[1] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley and Sons Ltd., 1989.

[2] Standard Performance Evaluation Corporation. The SPEC benchmark suites. *http://www.spec.org/.*

[3] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann, 2004.

[4] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[5] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[6] Jason Hart, Swee Yew Choe, Lik Cheng, Chipai Chou, Anand Dixit, Kenneth Ho, Jesse Hsu, Kyung Lee, and John Wu. Implementation of a 4th-generation 1.8ghz dual-core sparc v9 microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 186–187,592, February 2005.

[7] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[8] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective structure for VLIW and superscalar compilation. Technical report, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[9] Kapasi et al. The Imagine Stream Processor. In *Proceedings of ICCD*, 2002.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[11] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–54, October 1998.

[12] Rainer Leupers. Instruction scheduling for clustered vliw dsps. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[14] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6), 1953.

[15] Samuel D. Naffziger and Gary Hammond. The implementation of the next-generation 64b Itanium microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 344–345,472, February 2002.

[16] Samuel D. Naffziger, Blaine Stackhouse, and Tom Grutkowski. The implementation of a 2-core multi-threaded itanium-family processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 182–183,592, February 2005.

[17] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of Grid processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, pages 40–51, December 2001.

[18] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang ad J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and imlementation of a first-generation cell processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 184–185,592, February 2005.

[19] John W. Sias, Sain zee Ueng, Geoff A. Kent, Ian M. Steiner, Erik M. Nystrom, and Wen mei W. Hwu. Field-testing impact epic research results in itanium 2. In *Proceedings of the International Symposium on Computer Architecture*, pages 26–37, 4004.

[20] Steve Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the International Symposium on Microarchitecture*, pages 291–302, December 2003.

[21] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[22] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffman, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[23] TRIMARAN: An infrastructure for research in instruction level parallelism. http://www.trimaran.org.

[24] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997.

[25] David W. Wall. Limits of instruction-level parallelism. Technical Report Technical Note TN-15, Digital Western Research Laboratory, December 1990.