THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

IMPROVING PROCESSOR EFFICIENCY BY STATICALLY PIPELINING INSTRUCTIONS

By

IAN FINLAYSON

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2012

Ian Finlayson defended this dissertation on June 20, 2012.

The members of the supervisory committee were:

David Whalley
Professor Co-Directing Dissertation

Gary Tyson
Professor Co-Directing Dissertation

Linda DeBrunner
University Representative

Xin Yuan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

For Ryan.

ACKNOWLEDGMENTS

I would like to thank my advisors David Whalley and Gary Tyson for their guidance on this work. Whether it was advice on implementing something tricky, interesting discussions, or having me present the same slide for the tenth time, you have both made me a much better computer scientist. Additionally I would like to thank Gang-Ryung Uh who helped me immeasurably on this project.

I would also like to thank the other students I have had the pleasure of working with over the past few years including Yuval Peress, Peter Gavin, Paul West, Julia Gould, and Brandon Davis.

I would also like to thank my family members, Mom, Dad, Jill, Brie, Wayne and Laura. Lastly I would like to thank my wife Caitie without whose support I never would have finished.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A new generation of mobile applications requires reduced power consumption without sacrificing execution performance. A common approach for improving performance of processors is instruction pipelining. The way pipelining is traditionally implemented, however, is wasteful with regards to energy. This dissertation describes the design and implementation of an innovative *statically pipelined* processor supported by an optimizing compiler which responds to these conflicting demands. The central idea of the approach is that the control during each cycle for each portion of the processor is explicitly represented in each instruction. Thus the pipelining is in effect statically determined by the compiler. Pipeline registers become architecturally visible enabling new opportunities for code optimization as well as more efficient instruction flow through the pipeline. This approach simplifies hardware requirements to support pipelining, but requires significant modifications to existing compiler code generation techniques. The resulting design reduces energy usage by simplifying hardware, avoiding unnecessary computations, and allowing the compiler to perform optimizations that are not possible on traditional architectures.

CHAPTER 1

INTRODUCTION

This chapter discusses the motivation for this dissertation, describes some of the key concepts involved, and gives an overview of the work. This chapter also lays out the organization for the remainder of this dissertation.

1.1 Motivation

Power consumption has joined performance and cost as a primary design constraint in the design of processors. For general-purpose processors the problem mostly manifests itself in the form of heat dissipation which has become the limiting factor in increasing clock rates and processor complexity. This problem has resulted in the leveling off of single-threaded performance improvement in recent years. Processor designers are using the additional transistors that Moore's law continues to provide to add multiple cores and larger caches instead. Unfortunately, taking advantage of multiple cores requires programmers to rewrite applications in a parallel manner, which has yet to widely occur. In order to increase single-threaded performance, the power problem must be addressed.

In the server computer domain, power is a problem as well. From 2000 to 2010, the energy usage used by server computers more than tripled so that now, worldwide, between 1.1% and 1.5% of all electricity usage went to powering server computers [16]. In the United States, this figure is between 1.7% and 2.2%. With the expansion of the cloud computing paradigm, where more services are transferred to remote servers, this figure will likely continue to increase in the future.

In embedded systems, power is arguably even more important. Because these products are frequently powered by batteries, the energy usage affects battery life which is directly related to the usefulness of the product. Moreover, portable devices such as cell phones are being required to execute more sophisticated applications which increases the performance requirements of these systems. The problem of designing embedded processors that have low energy usage to relieve battery drain, while offering high performance is a daunting one.

One approach to providing this high level of efficiency is the use of application-specific integrated circuits (ASICs). ASICs are designed and optimized for a particular task which makes them more efficient for that task than a general-purpose design. Unfortunately, the design of ASICs is an expensive and time-consuming process. The increasing complexity of

many embedded applications has compounded this problem. Additionally, many mobile devices, such as smart phones and tablets, are not simply fixed products, but rather platforms for independent mobile applications. For these reasons, a general-purpose processor with improved energy efficiency is desirable for the mobile market.

1.2 Instruction Pipelining

Many micro-architectural techniques to improve performance were developed when efficiency was not as important. For instance, speculative execution is a direct trade-off between power and execution. Embedded processors are typically less aggressive with these micro-architectural techniques in order to save energy, but many others are assumed to be efficient.

Perhaps the most common technique for improving performance of general-purpose processors is instruction pipelining. This is generally considered very efficient when speculation costs and scheduling complexity are minimized. Instruction pipelining breaks instructions into stages and overlaps execution of multiple instructions. This approach allows the time taken by each cycle of execution to be the time of the longest stage, rather than the time needed for a complete instruction to execute. Instruction pipelining greatly improves execution performance and is employed by almost every processor that needs some degree of performance.

Unfortunately, the way instruction pipelining is implemented results in several inefficiencies with respect to energy usage. These inefficiencies include unnecessary accesses to the register file when the values will come from forwarding, checking for forwarding and hazards when they cannot possibly occur, latching values between pipeline registers that are often not used and repeatedly calculating invariant values such as branch target addresses. These inefficiencies stem from the fact that instruction pipelining is done dynamically in hardware.

The goal of this work is to achieve the performance gains of instruction pipelining while avoiding the energy inefficiencies discussed earlier. We do this by moving pipelining from being done dynamically in hardware to being done statically in software. With static pipelining, the pipeline is fully exposed to the compiler and the control for each portion of the processor is explicitly represented in each instruction. When pipeline registers become architecturally visible, the compiler can manage tasks like forwarding, branch prediction and register access directly, greatly reducing the redundancy found in more conventional designs, and providing new optimization capabilities to improve pipeline instruction and data flow.

Figure 1.1 illustrates the basic idea of the static pipelining approach. With traditional pipelining, instructions spend several cycles in the pipeline. For example, the `load` instruction, which is indicated in Figure 1.1(a) requires one cycle for each stage and remains in the pipeline from cycles four through seven, as shown in Figure 1.1(b). Each instruction is fetched and decoded and information about the instruction flows through the pipeline, via the pipeline registers, to control each portion of the processor that will take a specific action during each cycle.

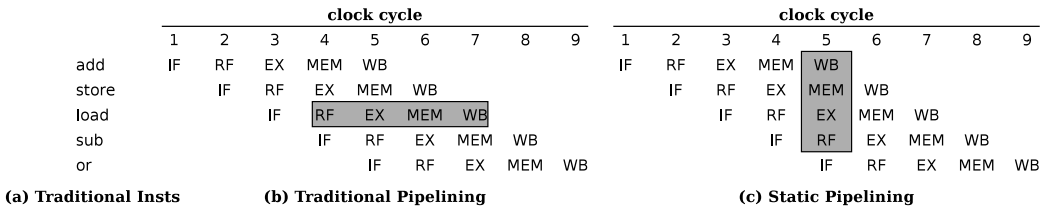

Figure 1.1: Traditionally Pipelined vs. Statically Pipelined Instructions

Figure 1.1(c) illustrates how a statically pipelined processor operates. Data still passes through the processor in multiple cycles. But how each portion of the processor is controlled during each cycle is explicitly represented in each instruction. Thus instructions are encoded to simultaneously perform actions normally associated with separate pipeline stages. For example, at cycle 5, all portions of the processor, are controlled by a single instruction (depicted with the shaded box) that was fetched the previous cycle. In effect the pipelining is determined statically by the compiler as opposed to dynamically by the hardware.

There are several benefits to this approach. First, energy usage is reduced by avoiding unnecessary and repetitive actions found in traditional pipelines. Secondly, static pipelining gives more control to the compiler which allows for more fine-grained optimizations for both performance and power. Lastly, statically pipelined processors have simpler hardware than traditional processors which should provide a lower production cost.

1.3 Contributions and Impact

The major contributions of this dissertation can be summarized as follows:

1. I have designed, implemented and evaluated a statically pipelined instruction set architecture and micro-architecture. The architecture is unique in that it is designed to expose instruction pipelining at the architectural level.

2. I have developed a compiler backend to generate optimized code for the static pipeline. Because the static pipeline is very different from most target machines, the compiler is very different both in terms of code generation and optimization. I apply traditional optimizations at a new level and have developed several new compiler optimizations including an advanced instruction scheduler.

3. I have designed a compact instruction encoding for statically pipelined instructions in order to avoid an increase in code size.

4. I have evaluated the architecture and compiler and have shown that static pipelining can achieve the performance improvements of dynamic pipelining in a much more energy-efficient manner.

This research has the potential to have a significant impact on forthcoming embedded systems. While static pipelining requires major changes in the instruction set architecture

and thus the software, it has clear benefits in terms of energy savings. Many mobile devices do not need to support binary legacy applications, and for these static pipelining may provide a significant advantage.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 discusses the design and implementation of a statically pipelined micro-architecture and instruction set architecture. Chapter 3 describes the design and implementation of a compiler targeting this architecture including code generation and optimization. Chapter 4 provides an experimental evaluation of the performance, code size and energy usage for the architecture and compiler. Chapter 5 provides an overview of related work in the area of reducing energy usage in general purpose processors. Chapter 6 outlines potential areas for further exploration. Finally Chapter 7 presents our conclusions regarding static pipelining.

CHAPTER 2

STATICALLY PIPELINED ARCHITECTURE

This chapter discusses the design of a statically pipelined architecture including both the micro-architecture and the instruction set. The architecture discussed in this chapter is one example of a statically pipelined machine; there are many other designs that are possible. The design described in this chapter, however, is used to evaluate the concept of static pipelining for the rest of this dissertation.

2.1 Micro-Architecture

This section describes the design of a statically pipelined micro-architecture. The micro-architecture is designed to be similar to a classical five-stage pipeline in terms of available hardware and operation. This was done to minimize the differences between the baseline design and our design to evaluate static versus traditional dynamic pipelining.

2.1.1 Overview

Figure 2.1 depicts a classical five-stage pipeline. Instructions spend one cycle in each stage of the pipeline which are separated by pipeline registers. Along with increasing performance, pipelining introduces a few inefficiencies into a processor. First of all is the need to latch information between pipeline stages. All of the possible control signals and data values needed for an instruction are passed through the pipeline registers to the stage that uses them. For many instructions, much of this information is not used. For example, the program counter (PC) is typically passed from stage to stage for all instructions, but is only used for branches.

Pipelining also introduces branch and data hazards. Branch hazards result from the fact that, when fetching a branch instruction, we will not know for several cycles what the next instruction will be. These hazards result in either stalls for every branch, or the need for branch predictors and delays when branches are mis-predicted. Data hazards are the result of values being needed before a previous instruction has written them back to the register file. Data hazards result in the need for forwarding logic which leads to unnecessary register file or pipeline register accesses. Experiments with SimpleScalar [1] running the MiBench benchmark suite [11] indicate that 27.9% of register file reads are unnecessary because the values will be replaced from forwarding. Additionally 11.1%

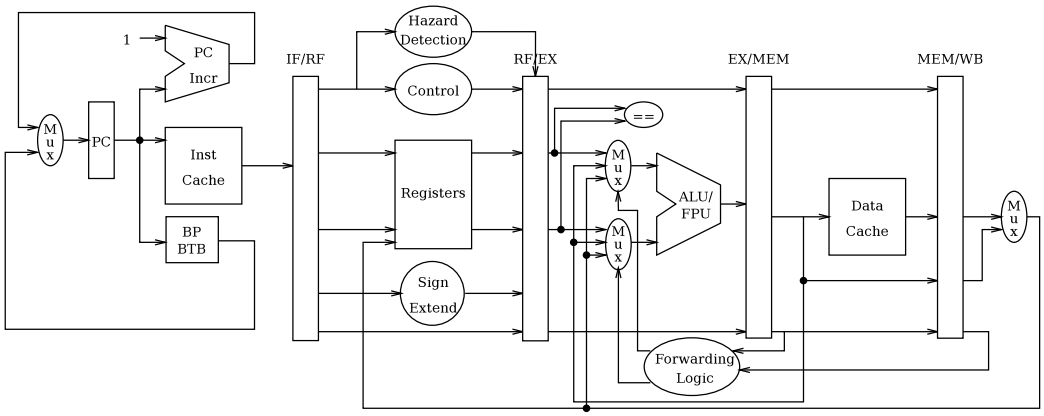

Figure 2.1: Classical Five-Stage Pipeline

of register file writes are not needed due to their only consumers getting the values from forwarding instead. Additional inefficiencies found in traditional pipelines include repeatedly calculating branch targets when they do not change, and adding an offset to a register to form a memory address even when that offset is zero.

Figure 2.2 depicts one possible datapath of a statically pipelined processor. In order to simplify the figure, the multiplexer in the lower right hand corner is shown as having three purposes. It supplies the value written to the data cache on a store operation, the value written to the register file and the value written to one of the copy registers. In actuality there are three such multiplexers, allowing for different values to be used for each purpose.

The fetch portion of the processor is mostly unchanged from the conventional processor. Instructions are still fetched from the instruction cache and branches are predicted by a branch predictor.

One difference is that there is no longer any need for the branch target buffer (BTB). This structure is used to store the targets of branches in conventional pipelines, avoiding the need to wait for the target address of a branch to be calculated to begin fetching after that branch is predicted to be taken.

The BTB is not needed in the static pipeline because the branch target is specified before the actual transfer of control and conditional branches are known when fetched as this information is indicated in the prior instruction. This is illustrated in Figure 2.3. The `NPC` status register is used to control the multiplexer that selects what address to fetch instructions from. When a transfer of control is performed in statically pipelined code, the `NPC` bit is first set to indicate the target of the transfer, along with whether or not the transfer is conditional. With this technique, there is no need to lookup the target from a branch target buffer, or to access the branch predictor for any instructions other than conditional branches.

There are more substantial differences in the processor after instructions are fetched. There is no need for pipeline registers because statically pipelined processors do not need to break instructions into multiple stages. In their place are a number of architecturally visible internal registers. Unlike pipeline registers, these internal registers are explicitly

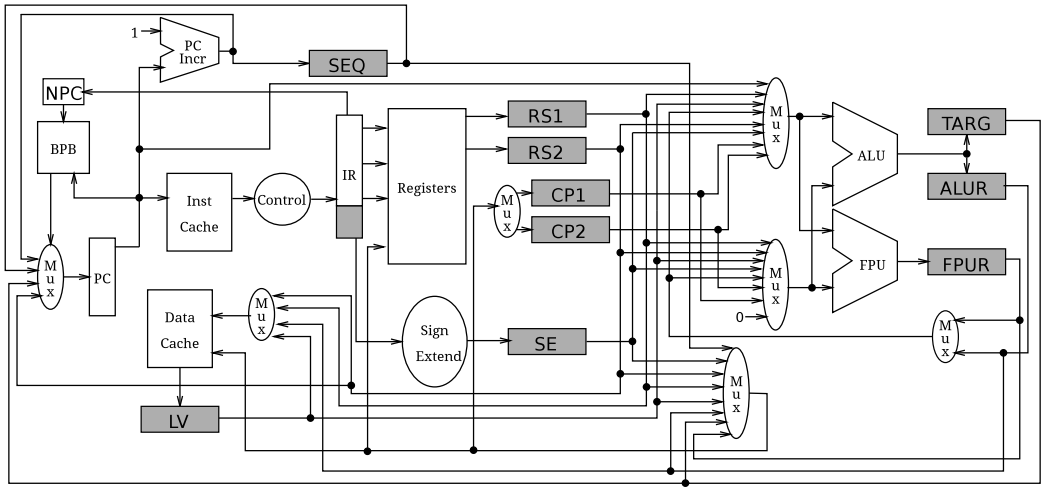

Figure 2.2: Datapath of a Statically Pipelined Processor

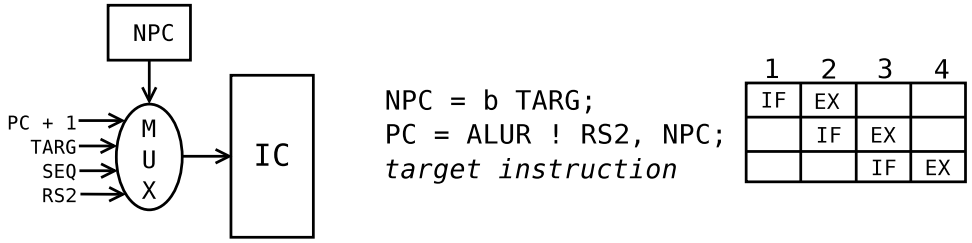

Figure 2.3: Transfer of Control Resolution

Table 2.1: Internal Registers

Name	Meaning	Name	Meaning
RS1	Register Source 1	ALUR	ALU Result
RS2	Register Source 2	TARG	Target Address
LV	Load Value	FPUR	FPU Result
SEQ	Sequential Address	CP1	Copy 1
SE	Sign Extend	CP2	Copy 2

read and written by the instructions, and can hold their values across multiple cycles.

2.1.2 Internal Registers

There are ten internal registers in our static pipeline design. Table 2.1 shows the internal registers names and meanings.

The RS1 and RS2 (register source) registers are used to hold values read from the register file. The LV (load value) register is used to hold a value loaded from the data cache. The SEQ (sequential address) register is used to hold the address of the next sequential instruction at the time it is written. This register is used to store the target of a loop branch in order to avoid calculating the target in the body of the loop. The SE (sign extend) register is used to hold a sign-extended immediate value. The ALUR (ALU result) and TARG (target address) registers are used to hold values calculated in the ALU. If the PC is used as an input to the ALU (as in a PC-relative address computation), then the result is placed in the TARG register, otherwise it is placed in the ALUR register. The FPUR (FPU result) register is used to hold results calculated in the FPU, which is used for multi-cycle operations. The CP1 and CP2 (copy) registers are used to hold values copied from one of the other internal registers. These copy registers are used to avoid additional register file accesses.

Because these internal registers are part of the machine state, they must be saved and restored with the register file upon context switches. Thus each register must be able to be stored to, and loaded from, the data cache. Some registers have a direct path, while others must be moved through a copy register or the register file. Since these internal registers are small, and can be placed near the portion of the processor that accesses it, each internal register is accessible at a lower energy cost than the centralized register file. Note that while the pipeline registers are read and written every cycle, the internal registers are only accessed when needed. Because these registers are exposed at the architectural level, a new level of compiler optimizations can be exploited as we will demonstrate in Chapter 3.

All of the internal registers are caller save or *scratch* registers, except for SEQ, CP1 and CP2. These are callee save because our optimizing compiler primarily uses these three internal registers to perform aggressive loop optimizations. If a loop has a function call in it, the compiler would disallow the use of these registers for this optimization were they caller save, as the function call in the loop could possibly overwrite the register making it effectively not loop invariant.

2.1.3 Operation

Hazards due to multi-cycle operations can easily be detected without special logic to compare register numbers from instructions obtained from pipeline registers. If during a given cycle the FPUR register is to be used as a source and the corresponding functional unit has not completed a multi-cycle operation, then the current instruction is aborted and the instruction will be reattempted on the next cycle. This process continues until the FPU has completed the operation. Misses in the data cache can be handled in a similar fashion during LV register reads.

A static pipeline can be viewed as a two-stage processor with the two stages being fetch and everything after fetch. As discussed in the next sub-section, the statically pipelined instructions are already partially decoded as compared to traditional instructions. Because all datapath operations after fetch are performed in parallel, the clock frequency for a static pipeline should able to be just as high as for a traditional pipeline. Therefore if the number of instructions executed does not increase as compared to a traditional pipeline, there should be no performance loss associated with static pipelining. In Chapter 3, we will discuss compiler optimizations that attempt to keep the number of instructions executed as low as, or lower than, those of traditional pipelines.

One benefit of static pipelining is that the branch penalty is reduced to one cycle. This is because branches are resolved only one cycle after the following instruction is fetched. Additionally, the architecture is given one cycle of notice that a transfer of control will take place along with the target address and whether or not the transfer of control is conditional. This information is placed into the NPC status register which allows us to only perform branch prediction on conditional branches as well as removes the need for a branch target buffer.

Because each portion of the datapath is explicitly controlled, there is less complexity in the operation of the micro-architecture. The logic for checking for hazards is much simpler, forwarding does not need to take place, and values are not implicitly copied through pipeline registers each cycle. Due to these factors, statically pipelined hardware should have decreased area, cost, and debugging time compared to equivalent traditionally pipelined hardware.

2.2 Instruction Set Architecture

The instruction set architecture for a statically pipelined architecture is quite different than one for a conventional processor. Each instruction consists of a set of effects, each of which updates some portion of the processor. The effects that are allowed in each cycle mostly correspond to what the baseline five-stage pipeline can do in one cycle, which include one ALU or FPU operation, one memory operation, two register reads, one register write and one sign extension. In addition, one copy can be made from an internal register to one of the two copy registers and the next sequential instruction address can optionally be saved in the SEQ register. Lastly, the NPC status register can be set to indicate an upcoming branch operation.

Figure 2.4 shows a 64-bit instruction encoding for this instruction set architecture. 64-bit instructions are not viable for use in embedded systems because the energy savings

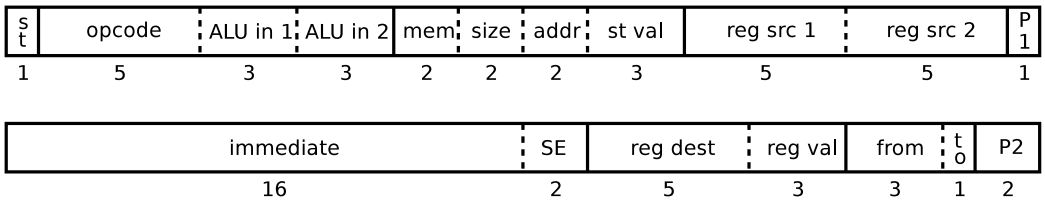

Figure 2.4: Long Encoding of a Statically Pipelined Instructions

of static pipelining would be offset by additional energy needed to store and fetch these larger instructions. As such, only a subset of the instruction effects in this long encoding can actually be specified in any 32-bit instruction. Efficiently encoding these instructions will be discussed later in this chapter.

Table 2.2 gives the legal values and corresponding meanings of each field in the 64-bit encoding. This defines the possible combinations for each type of instruction effect throughout this work such as which internal registers can be used as input to the ALU, register file write port and so on.

Several decisions here were made to keep the long encoding at 64 bits for practical implementation reasons. The statically pipelined architecture uses the same register conventions as the MIPS architecture where registers 28 and 30 are the global pointer and frame pointer respectively. These registers are never used by the VPO MIPS compiler, so we use them to indicate no read or no write for a few fields. Also, if the value of `FPUR` is written into `r[31]`, which is the return address, then the value `PC + 1` is used instead. This is used to implement saving the return address on a function call. `FPUR` is used because a floating point value should never be placed in the return address register.

2.2.1 Instruction Fields

The solid lines in Figure 2.4 separate sections of the instruction that correspond to different effects. The *st*, or status, bit is set to one for statically pipelined instructions to distinguish them from MIPS instructions. The *opcode*, and *ALU input* fields are used for the ALU operation. Table 2.3 provides a list of available opcodes along with their meanings.

The *mem* field specifies the memory operation out of load, store, unsigned load, or none. The *size*, *address* and *store value* fields are also used for memory operations. The *register source* fields are for specifying which registers should be read into the `RS1` and `RS2` internal registers. To keep the encoding within 64 bits, we have no fields that specify whether `RS1` and `RS2` should be written. Instead, `r[28]` can't be read into `RS1` and `r[30]` can't be read into `RS2` and those values specify that no value shall be read.

The *immediate* field is used to load immediate values into the `SE` register. The *sign extend* field specifies how the immediates are written: either sign extended, placed in the lower bits, placed in the upper bits, or not at all. Placing the immediate in the lower and upper bits of the `SE` register is used to form 32-bit values including function addresses. The *register destination* and *register value* fields are used to store values into the register file. A value of zero for the register destination is used to indicate no register is to be written. Also if the register destination is `r[31]`, the return address register on MIPS, the value of `PC + 1` is

Table 2.2: 64-Bit Encoded Values

Field	Meaning
ST	Status (0:MIPS) (1:Static)
opcode	See Table 2.3
ALU in1	(0:ALUR) (1:FPUR) (2:RS1) (3:RS2) (4:LV) (5:CP1) (6: CP2) (7:PC)
ALU in2	(0:ALUR) (1:FPUR) (2:RS1) (3:RS2) (4:LV) (5: CP1) (6:SE) (7:0)
R/W	(0:nop) (1:store) (2:load) (3:unsigned load)
Size	(0:byte) (1:half) (2:word) (3:double)
Addr	(0:RS1) (1:CP1) (2:ALUR) (3:LV)
S Val	(0:RS1) (1:RS2) (2:ALUR) (3:CP1) (4:SE) (5:LV) (6:SEQ) (7:CP2)
Reg Src1	Register Number to load into RS1 (28 means don't load any)
Reg Src2	Register Number to load into RS2 (30 means don't load any)
PTB1	The first bit of the PTB field, see below
Imm Val	16-bit immediate value
SE W	SE write (0: none) (1:sign extend) (2:replace LO) (3:replace HI)
Reg Dest	Register to Write to (30 means don't write any)
Reg Val	Register Value (0:RS1) (1:RS2) (2:ALUR) (3:FPUR) (4:SE) (5:LV) (6:CP1) (7:CP2)
From	Copy From (0:RS1) (1:RS2) (2:ALUR) (3:FPUR) (4:SE) (5:LV) (6:TARG) (7:None)
To	Copy To (0:CP1) (1:CP2)
PTB2	Low 2 bits of prepare branch (PTB) field - combined with P1 above: (0:nop) (1:set SEQ, no branch) (2:Jump RS2) (3:JUMP SE) (4:Jump SEQ) (5: Branch SEQ) (6: Jump TARG) (7:Branch TARG).

Table 2.3: Opcodes

Number	Opcode	Meaning	Number	Opcode	Meaning
0	nop	Nothing	16	sll	Shift Left Logical
1	add	Add	17	srl	Shift Right Logical
2	addf	Floating Add	18	sra	Shift Right Arith
3	sub	Subtract	19	slt	Set Less Than
4	subf	Floating Sub	20	sltu	Unsigned Slt
5	mult	Multiply	21	sltf	Floating Slt
6	multf	Floating Mul	22	beq	Branch IF Equal
7	div	Divide	23	bne	Branch If Not Equal
8	divu	Unsigned Div	24	itf	Int to Float
9	divf	Floating Div	25	itd	Int to Double
10	mod	Modulus	26	fti	Float to Int
11	modu	Unsigned Mod	27	ftd	Float to Double
12	and	Lodical And	28	dti	Double to Ine
13	or	Logical Or	29	dtf	Double to Float
14	xor	Logical Xor	30	neg	Negate (unary)
15	not	Not (unary)	31	negf	Floating Neg (unary)

written instead of the value specified by the *register value* field. The *from* and *to* fields are used to specify copies into the copy registers. The *from* field specifies which internal register to copy, or seven which specifies that no copy should be made. The *to* field indicates which of the two copy registers is the target.

Lastly the *PTB* field indicates whether we should set the `SEQ` register to `PC + 1` and what value, if any to place in the `NPC` status register. It can be used for these two purposes because we never store the address of a branch in the `SEQ` register as it is used to store the address of the preheader of a loop - which cannot end in a conditional branch. The value in the `NPC` status register is used to determine the target address of a branch that is predicted taken - in lieu of a branch target buffer. If the next instruction contains a conditional branch operation, the value target is only used if the branch is predicted taken, otherwise the next PC is used unconditionally.

All of the effects specified in a single instruction are independent and are performed in parallel. The values in the internal registers are read at the beginning of the cycle and written at the end of the cycle. Note that except for the effects that solely read or write a register file or data cache value, all of the effects operate solely on the internal registers. This is analogous to how RISC architectures only allow load or store instructions to reference memory locations.

In a traditional architecture, when reading a value from the register file, it is clear from the opcode whether that value will be used as an integer or floating point value. This allows the instructions to "double up" on the number of available registers by having separate integer and floating-point register files. In a statically pipelined architecture, however, a register is not read in the same instruction as the arithmetic operation that uses it. Therefore to have both integer and floating point register files, we would need

one extra bit for each register field. To avoid this problem, we use a single register file to hold both integer and floating point values. Another reason for traditional architectures to use distinct register files is to simplify forwarding logic and hazard detection which is not an issue for a statically pipelined architecture. While this may increase register pressure for programs using both integer and floating point registers, we will show that static pipelining is still able to significantly reduce the number of references to the centralized register file.

Another issue regarding the register file is how to handle double precision floating point values. In traditional architectures, it is clear when a register is read if it will be used as a single or double precision value. Double precision values typically are implemented with even/odd register pairs where both registers together make up the value. Thus when a double precision register is accessed, two are effectively accessed.

In the static pipeline, this is harder to achieve; at the time of reading a register, it is not clear how future instructions will use it. To support double precision floating point values, each register in the register file, and each internal register, is given a one bit tag that indicates whether or not the value in that register is a double precision value. When a register is read, the tag is checked. If it is set, then that register and the subsequent register are read into the internal register indicated, and the tag is propagated. In this manner, we are able to track which values are double precision throughout execution.

2.2.2 Compact Encoding

Including all possible instruction effect fields in an instruction would take 64 bit instructions. Doubling the size of each instruction would have a very negative effect on code size, as well as power consumption from the instruction cache which would negate much of the benefit static pipelining aims to achieve. Therefore we developed a compact, 32-bit encoding for the instructions.

The encoding scheme is similar to that used by many VLIW processors. Each instruction is capable of encoding a number of fields, with each field corresponding to one statically pipelined effect. The first field is the *template field* which specifies how the remaining fields should be interpreted. Note that some fields will specify more than one value in the long encoding above. For example the field corresponding to a arithmetic operation must specify the type of operation as well as the operands to use. The size of the template field dictates how many combinations of fields the encoding supports. With a larger number of combinations, there is more flexibility in scheduling, but the template field takes more space and the decoding logic is more complicated. Frequently used effects, such as an ALU operation should be present in more combinations than lesser used effects, such as copying an internal register to a copy register. Each field is present in at least one combination, or it would be impossible to use it.

The different fields are given in Table 2.4. The ALU field specifies an ALU operation including the opcode, and each operand. The LOAD field specifies one memory load operation including whether it is signed or unsigned, the size and the address to load from. The STORE field specifies one memory store operation including the size, the address to write to and which value to write. Memory operations were split into loads and stores because loads are more common and take fewer bits to specify. This is because stores must specify the internal register to be stored, while loads always load into `LV`. However, because

Table 2.4: Template Fields

Field	Description	Size
ALU	Arithmetic Logic Operation	11
LOAD	Memory Load Operation	6
STORE	Memory Store Operation	9
REGW	Register File Write	8
RS1	Register File Read into RS1	5
RS2	Register File Read into RS2	5
COPY	Copy into CP1 or CP2	4
SIMM	Small Immediate Value	7
LIMM	Large Immediate Value	18
PTB	Prepare to Branch	3

the micro-architecture is only capable of performing one memory operation at a time, no template can contain both a LOAD and STORE field.

The REGW field specifies a register write operation which consists of which internal register to write and which register in the register file to write to. The RS1 and RS2 fields specify register reads to each of these internal registers respectively. The COPY field specifies one register copy including which internal register to write and which of the two copy registers to write the value into.

The SIMM and LIMM fields specify reading and sign extending small and large immediate values respectively. Small immediate values are seven bits in length, so this field can store values from -64 to 63. Many immediate values in programs are small enough to fit in this space, so it would be wasteful to use 16-bits to store them. The number of bits for the small immediate was chosen as a trade-off between being large enough to represent most smaller immediate values and small enough to save space. Large immediate values are 16 bits of length. These are used for numbers in programs that won't fit in the seven-bit small immediate as well as for constructing the addresses of functions and other global values. The LIMM field is 18 bits because two bits are reserved for specifying whether the value should be placed in the LO or HI portion or sign extended. The PTB field specifies a prepare to branch operation which includes the value to assign to the `NPC` in preparation for a transfer of control. This field also includes a bit for specifying whether the `SEQ` register should have `PC + 1` placed in it.

Each field can represent a no-op for its given operation in some way. This means that a template does not have to match an instruction exactly, it just has to provide all of the fields the instruction needs and possibly more. For example, an instruction with just one ALU effect can be represented with any template that contains the ALU field, as the other fields can be filled with no-ops.

Decoding these fields will require some logic in the micro-architecture. Depending on the timing breakdown of the fetch and execute cycles, this logic can either be at the end of fetch, the beginning of execute, or split between them. If necessary, then we can add a stage for decoding instructions, which is discussed in more detail in Chapter 6. Note that, unlike for the MIPS architecture, register file reads happen during the execute stage, so there is no

need for a decode stage to fetch registers. The need to decode the templates could possibly decrease the clock rate, though in this dissertation we assume that it does not.

2.2.3 Choosing Combinations

There are many possible ways to combine these fields into a set of templates for the encoding. In order to choose a good set of templates, we compiled and ran the MiBench benchmark suite [11] with the wide, 64-bit instruction encoding to determine which combinations of instruction effects were run.

Tables 2.5 and 2.6 show the most commonly generated instruction combinations. The first column of this table shows the template fields used in the instruction. The second column gives the size in bits. Note that if the size is larger than 32 bits, there is no way to encode the instruction in 32 bits. Because we need space for the template field, the size must be less than 32 to be encoded. The third column gives the percentage of dynamic instructions that this instruction accounts for. Lastly, the fourth column shows the cumulative percentage - the percentage of dynamic instructions accounted for by this combination and all previous combinations.

These 74 instruction combinations shown make up over 95% of dynamic instruction usage for the benchmarks studied. In total, there are 286 instruction combinations that are ever used out of 576 possibilities. There are 576 possible combinations because there are six field that can either be present or not be present, and two fields that have three possibilities: the memory field can be a LOAD, a STORE, or not be present, and the immediate field can be a SIMM, a LIMM, or not be present. This gives us $2^6 * 3^2 = 576$. Despite these large numbers of combinations, the top nine most popular combinations account for over half of instructions executed. Furthermore, many of these combinations can be covered by a single template. For example, ALU by itself is the most frequent instruction, but this can be covered by any template which contains an ALU field. Because of this, we are able choose a set of templates that covers most of the dynamic instructions.

To choose the templates we use, we went down the list in Tables 2.5 and 2.6 in order of frequency of execution. For each instruction, if its size would allow it to fit in a template, we first look through the templates we currently have selected. If it fits in one of them, then we move on as this instruction is already covered. If not, we see if one of the templates can be augmented to support the instruction while still being small enough. If not, we add the instruction as a new template. By repeating this, we arrive at a set of templates that allows the most frequent instructions to be encoded.

We performed this task with different numbers of templates and determined that 32 templates was best. 16 was too few to capture relatively common instructions while 64 did not provide many additional common instructions. Because 32 templates were chosen, the *template* field takes up 5 bits, leaving 27 for the different fields. Table 2.7 shows the 32 templates chosen with this technique. With these templates, we are able to encode the 90 most commonly executed instructions that are able to fit in the 27 bits available (in addition to many of the others). These 90 instructions together provide 90.4% of the overall dynamic instructions, allowing the encoded instructions to capture most of the performance of the long instruction encoding in half of the space.

Table 2.5: Combinations Under Long Encoding I

Field	Size	Percentage	Cumulative
ALU;	11	14.885	14.885
ALU; RS1;	16	8.521	23.405
ALU; SIMM;	18	6.487	29.892
LOAD;	6	4.525	34.417
ALU; REGW;	19	4.064	38.482
ALU; REGW; SIMM;	26	3.837	42.318
ALU; REGW; RS1;	24	3.268	45.587
ALU; RS2;	16	3.123	48.710
REGW;	8	2.549	51.259
ALU; PTB;	14	2.473	53.732
ALU; RS1; SIMM;	23	1.982	55.714
LOAD; RS1;	11	1.662	57.376
ALU; REGW; PTB;	22	1.590	58.966
RS1; SIMM;	12	1.549	60.515
RS1;	5	1.493	62.008
STORE; REGW; RS1;	22	1.469	63.477
ALU; LOAD; SIMM;	24	1.362	64.839
ALU; REGW; RS1; SIMM;	31	1.312	66.152
LOAD; REGW;	14	1.296	67.447
RS1; RS2; SIMM;	17	1.208	68.656
ALU; STORE; RS1;	25	1.203	69.859
ALU; LIMM;	29	1.184	71.043
RS2;	5	1.049	72.092
REGW; PTB;	11	1.015	73.107
REGW; RS1;	13	0.946	74.053
REGW; SIMM;	15	0.938	74.991
ALU; LOAD; RS1;	22	0.936	75.927
ALU; RS1; RS2;	21	0.908	76.834
ALU; PTB; SIMM;	21	0.890	77.724
ALU; LOAD;	17	0.871	78.595
ALU; RS1; COPY;	20	0.842	79.437
ALU; REGW; LIMM;	37	0.830	80.267
ALU; LOAD; REGW;	25	0.801	81.068
STORE; RS1;	14	0.771	81.839
ALU; RS1; PTB;	19	0.758	82.597
ALU; REGW; RS2;	24	0.709	83.306
ALU; LOAD; RS1; COPY;	26	0.689	83.995

Table 2.6: Combinations Under Long Encoding II

Field	Size	Percentage	Cumulative
LOAD; REGW; RS1;	19	0.642	84.637
ALU; LOAD; RS1; SIMM;	29	0.638	85.275
ALU; RS2; SIMM;	23	0.575	85.850
SIMM;	7	0.511	86.360
LIMM;	18	0.489	86.849
LOAD; PTB;	9	0.488	87.337
NOOP;	0	0.477	87.814
ALU; REGW; RS1; PTB;	27	0.458	88.272
ALU; STORE; RS1; RS2; LIMM;	48	0.377	88.648
ALU; RS1; LIMM;	34	0.355	89.003
RS2; SIMM;	12	0.354	89.357
ALU; COPY;	15	0.305	89.662
ALU; STORE; SIMM;	27	0.296	89.958
COPY;	4	0.295	90.253
ALU; REGW; RS1; RS2;	29	0.285	90.538
ALU; LOAD; PTB;	20	0.279	90.818
ALU; STORE; RS1; SIMM;	32	0.271	91.089
REGW; LIMM;	26	0.264	91.353
ALU; REGW; RS1; LIMM;	42	0.264	91.618
RS1; LIMM;	23	0.251	91.869
ALU; RS1; RS2; SIMM;	28	0.240	92.109
ALU; REGW; PTB; SIMM;	29	0.238	92.347
ALU; STORE; RS1; RS2; SIMM;	37	0.216	92.562
STORE;	9	0.214	92.776
RS1; RS2;	10	0.213	92.989
ALU; LOAD; REGW; RS1; LIMM;	48	0.207	93.196
ALU; RS2; LIMM;	34	0.205	93.401
LOAD; RS2;	11	0.199	93.600
ALU; LOAD; REGW; RS1; SIMM;	37	0.195	93.795
LOAD; RS1; COPY; PTB;	18	0.180	93.975
REGW; RS1; SIMM;	20	0.163	94.139
LOAD; RS1; RS2;	16	0.147	94.286
ALU; STORE; RS1; LIMM;	43	0.147	94.432
ALU; RS1; COPY; PTB;	23	0.145	94.578
STORE; LIMM;	27	0.144	94.722
ALU; COPY; SIMM;	22	0.142	94.864
ALU; RS2; PTB;	19	0.141	95.005

Table 2.7: Selected Templates

Template Number	Fields
0	ALU; REGW; SIMM;
1	ALU; REGW; RS1; PTB;
2	ALU; RS2; LOAD; RS1;
3	ALU; RS1; SIMM; COPY;
4	STORE; REGW; RS1; RS2;
5	ALU; LOAD; SIMM; PTB;
6	ALU; LOAD; REGW;
7	RS1; RS2; SIMM; REGW;
8	ALU; STORE; RS1;
9	ALU; REGW; RS2; PTB;
10	ALU; LOAD; RS1; COPY;
11	LOAD; REGW; RS1; COPY; PTB;
12	ALU; RS2; SIMM; COPY;
13	ALU; STORE; SIMM;
14	REGW; LIMM;
15	RS1; LIMM; PTB;
16	ALU; RS1; COPY; PTB;
17	STORE; LIMM;
18	ALU; RS1; RS2; PTB;
19	ALU; LOAD; PTB; SIMM;
20	LOAD; LIMM; PTB;
21	STORE; RS2; RS1; COPY; PTB;
22	LOAD; REGW; SIMM; RS1;
23	ALU; REGW; RS2; PTB;
24	LOAD; REGW; RS1; RS2; PTB;
25	ALU; RS2; PTB; SIMM;
26	STORE; REGW; SIMM; PTB;
27	ALU; STORE; PTB; COPY;
28	RS2; LIMM; PTB;
29	RS1; RS2; SIMM; PTB; LOAD;
30	REGW; LOAD; SIMM; COPY;
31	STORE; RS2; SIMM; RS1;

In order to evaluate the architecture for different encoding schemes, the assembler, simulator and other tools use the long instruction format exclusively. Programs are passed through an encoder program which encodes the program into the 32 bit format using the set of selected templates. If any instructions cannot be encoded, an error is reported. The program is then decoded again with a decoder program. The output from this is then compared to the original long encoding. If the two match, then the program can be encoded into 32 bits without any loss of efficiency. As discussed in Chapter 3, the compiler also makes use of the set of selected templates in order to schedule instructions correctly.

CHAPTER 3

COMPILING FOR A STATICALLY PIPELINED ARCHITECTURE

This chapter discusses compilation issues relating to the static pipeline. We give an overview of the compilation process including code generation and optimization issues. For a statically pipelined architecture, the compiler is responsible for controlling each part of the datapath for every cycle, so effective compilation optimizations are necessary to achieve the performance and energy goals of static pipelining. Likewise, because the instruction set architecture for a statically pipelined processor is quite different from that of a RISC architecture, many compilation strategies and optimizations have to be reconsidered when applied to a static pipeline. Lastly we give a detailed example of how our compiler generates optimized code for the statically pipelined architecture.

3.1 Overview

Figure 3.1 shows the steps of our compilation process which is based on the VPO compiler [3]. First, C code is input to the frontend. This consists of the LCC compiler frontend [9] combined with a *middleware* process that converts LCC's output format into the Register Transfer List (RTL) format used by VPO. This format directly represents machine instructions on the target architecture.

These RTLs are then input into the VPO MIPS backend. This compiler process performs many compiler optimizations including control flow optimizations, loop invariant code

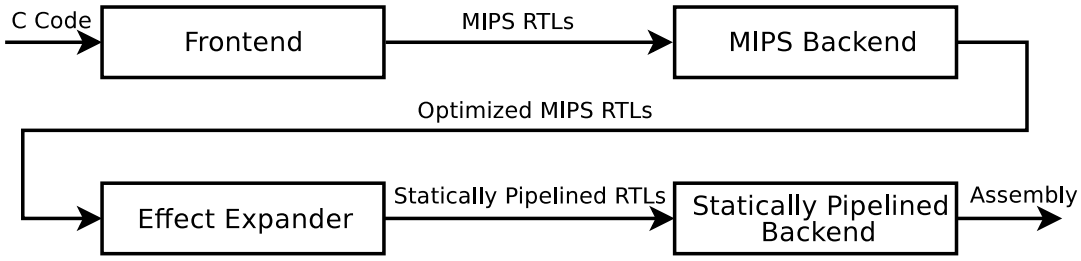

Figure 3.1: Compilation Process

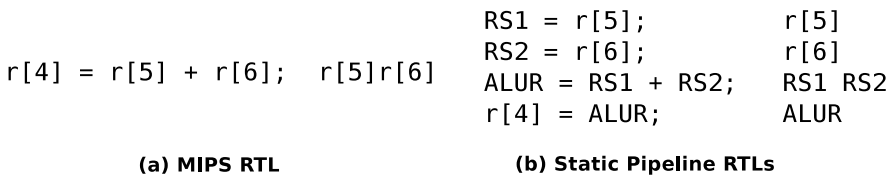

Figure 3.2: Effect Expansion Example

motion, register allocation, and data flow optimizations. These optimizations are done before converting the instructions to those for the statically pipelined architecture because some of these optimizations are easier to apply at the higher MIPS instruction level. Register allocation in particular is difficult to perform directly on statically pipelined instructions due to the need to have either `RS1` or `RS2` available to load any registers. Additionally this strategy allows us to concentrate on optimizations specific to the static pipeline as all higher level optimizations are already performed.

A couple of changes were necessary to the MIPS compiler for this to work. First, because our target machine has one register file for both integer and floating point values, the register allocation of the MIPS compiler must reflect this. In the unmodified MIPS architecture there are separate register files so, for example, `r[6]` and `f[6]` are two distinct registers. We modified the MIPS VPO port such that these refer to the same physical register to match the static pipeline. This strategy avoids incorrect code being generated due to a conflict between different values in the same register.

The second change that had to be performed is that the output of the compiler had to be changed slightly. Normally, there are a number of meta-data lines that are processed by the compiler and are not present in the output. This meta-information include lines that signify a register being trashed by a function call or a register getting a value from a function's return, lines that give information on local and global variables being used, a line indicating the register types being used and so on. Also some instructions generated for function prologue and epilogues had to to be suppressed because they had to be done at the time of final code generation by the statically pipelined backend. These changes make the output of this compilation stage tenable for the second compilation pass that follows.

Next the instructions are broken down into statically pipelined instruction effects by the *effect expander*. Lastly these instructions are fed into the statically pipelined compiler backend, also based on VPO, which applies additional optimizations and produces final assembly code. Each of these two processes will be discussed in detail.

3.2 Effect Expansion

The effect expander breaks the MIPS instructions into instruction effects that are legal for the static pipeline. This process works by looking at each MIPS RTL and generating a sequence of statically pipelined RTLs that perform the same computation.

Figure 3.2(a) depicts an example MIPS RTL. The fact that registers `r[5]` and `r[6]` are on the end of the line signify that they are dead on that line - that is that that RTL is the last one

```
                        SE = JumpTo(L7);
                        TARG = PC + SE;
PC = r[4] ! 0, L7;      RS1 = r[4];
                        SE = 0;
                        PC = RS1 ! SE, TARG (L7);
```

(a) MIPS RTL **(b) Static Pipeline RTLs**

Figure 3.3: Statically Pipelined Branch

to use those registers before they are overwritten. Figure 3.2(b) shows how this instruction is expanded into multiple statically pipelined RTLs. First the source registers, `r[5]` and `r[6]` have to be loaded into the internal registers `RS1` and `RS2` respectively. Next these are added together and the result is placed into the `ALUR` register. Lastly, the `ALUR` internal register is written back to `r[4]` in the register file. The effect expander also must produce the dead register lists for each instruction. Dead register lists must take into account not only the registers in the register file, but the internal registers as well.

This process works by matching the forms of the different types of RTLs that the VPO MIPS backend produces. These RTLs include ALU operations, loads and stores, branches, calls and returns, and type conversions. The patterns are parameterized so that they work for as many cases of RTLs as possible. For example, the add instruction in Figure 3.2(a), is matched to a pattern of `r[dest] = r[op1] op r[op2]`. The patterns parameters (in bold) are then used in the generated statically pipelined RTLs.

3.3 Code Generation

In traditional architectures, branch instructions include both the comparison (if necessary) along with the target of the transfer. In a statically pipelined architecture, however, these are broken into separate instructions. This presents a code generation challenge for a number of reasons.

Figure 3.3 shows how a typical conditional branch is broken into separate statically pipelined instructions. With the MIPS RTL, the target of the branch is included in the branch instruction itself. This is important for two reasons. First, the compiler must know the target of each transfer of control in order to accurately represent the control flow of the program. Secondly, the assembler must be able to replace the label with the correct PC-relative offset.

In the statically pipelined instructions, the target address is calculated separately from the transfer of control. In order for the compiler to know the target of each transfer of control, the transfers are annotated with the target label. This is the `(L7)` on the last line of Figure 3.3(b). The annotation is not used for executing the branch, it is just there for the compiler's sake.

In order for the assembler to be able to put in the correct offset, it has to do a little more work. Because the offset must be the difference between the target of the transfer and the point of the transfer, the assembler must find not only the label, but the point at which the offset is added to the PC as well. In order to do this, it scans forward from the point in

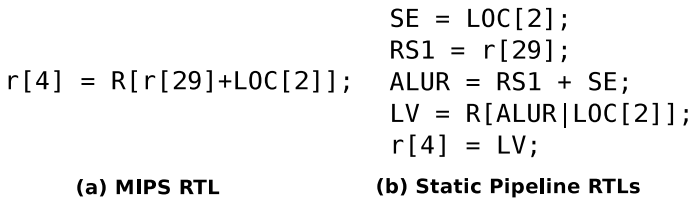

Figure 3.4: Statically Pipelined Load from Local Variable

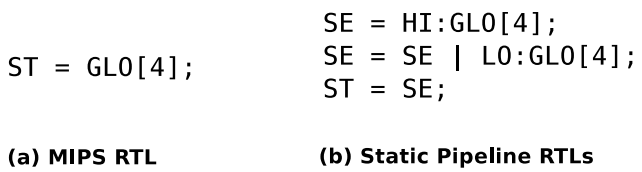

Figure 3.5: Statically Pipelined Function Call

which it sees the *JumpTo* macro until it sees an instruction that adds the PC and SE. This is usually the very next instruction, but it could be farther down due to scheduling decisions.

Figure 3.4 shows how a load of a local variable is broken into separate statically pipelined instructions. The compiler keeps track of the local variables (if any) to which each memory address is associated. This tracking is done in order to disambiguate memory references and to determine if memory references are indirect or not when possible. This can be seen on the left side of the figure where we specify that the memory address for the load is the stack pointer (`r[29]`) plus the local number two.

As in the case of conditional branches, this proves more challenging in the statically pipelined architecture. The offset must be placed into the `SE` register before the actual load takes place. Rather than try to scan backwards from each memory reference to ascertain whether a local variable is associated with it, we add an annotation to the memory operation in the statically pipelined code. This is the "`|LOC[2]`" in the 4th instruction in Figure 3.4(b). This annotation is not used for actual computation, but for the compiler to easily keep track of the local variable references.

In the MIPS architecture, function calls are accomplished with the `jal` (jump and link) instruction. This instruction has six bits for the opcode and twenty-six bits for the function address. In the static pipeline architecture, we have no instruction type that can fit such a large address, so we must load the address in two parts.

Figure 3.5 shows how a call instruction is broken into statically pipelined instruction effects. `GLO[4]` is defined previously in the code as being a reference to a function. `ST` stands for stack and indicates the point of the function call. On the right, we see that the address is formed in two instructions by a low and high half. The second instruction specifies that the low portion of the function address is to replace the low portion of the `SE` register. The bitwise or operator only describes what happens; no ALU operation actually takes place with that instruction. This technique is also used for global data values and constants that are too large to fit into the 16-bit immediate field.

Another effect that must occur during a function call is that the return address must be placed in `r[31]` so that the function can properly return. When the compiler is generating the final assembly code, it writes an instruction effect to save the value of `PC + 1` into `r[31]` for each call. During scheduling, the compiler also ensures that no register write is scheduled at the same time as a function call because the architecture only supports writing one register at a time.

Typically, once the code in Figure 3.5(b) is generated, it would be up to the linker to place the correct values into the executable file. However, to avoid having to modify the linker, we used a workaround solution for this study. What we did is create a table of all the functions, globals and large constants in a MIPS assembly file. Instead of placing the value of the constant in the instruction, we then place the index to the global table entry corresponding to that constant. The compiler automatically inserts a MIPS `la` (load address) instruction at the beginning of the main function that loads the address of the start of the global table into a register. The global table file is linked in with the other assembly files to produce an executable. Because the global table and the first instruction of main are in MIPS assembly, the linker resolves that address correctly.

During simulation, the address of the global table is saved into a variable in the simulator at the start of main. When instructions load a high or low value into the `SE` register, the simulator then transparently loads the correct value from the correct entry in the global table. These accesses of the global table do not affect the performance results calculated by the simulator. In this way the correct values are loaded without having to modify the linker or sacrifice the correctness of our results.

There were several other minor code generation issues that had to be addressed for the static pipeline. For example, the VPO compiler often assumed that certain operations took one instruction because that was the case for every other architecture. In particular, the VPO compiler assumed that return instructions required one instruction so it inserted some epilogue code before the last instruction of a function. Since that code was inserted between the two instructions needed for a return in the static pipeline, it caused bugs until this assumption was corrected. The VPO compiler also assumed in several instances that a register move could be accomplished in one instruction which is also not the case for the static pipeline. This illustrates another motivation for performing as many optimizations as possible in the MIPS compilation stage: to avoid violating assumptions in as many compilation processes as possible.

3.4 Optimizations

The statically pipelined compiler backend applies a number of optimizations to the statically pipelined code. Some of these are traditional compiler optimizations that are able to achieve new effects when applied to the lower level of statically pipelined architecture, whereas others are optimizations that target this architecture specifically.

Note that the code that is input to the statically pipelined compiler backend has already had some optimizations applied to it in the MIPS compiler backend phase. These include register allocation, data flow optimizations, control flow optimizations and loop invariant code motion. This means that virtually all of the optimizations we apply here are things that

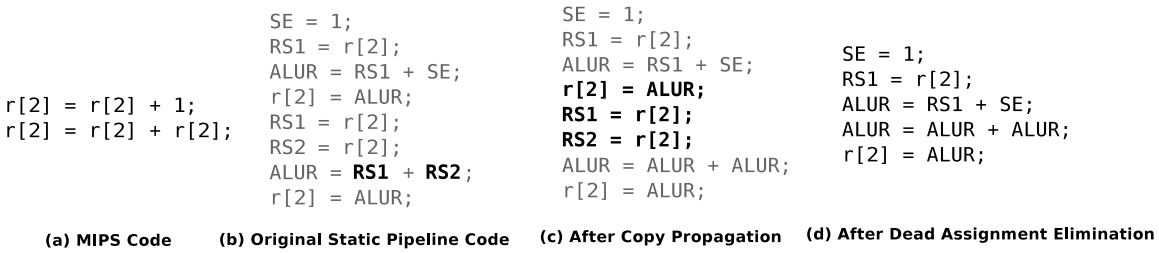

Figure 3.6: Simple Data Flow Optimizations

are specific to the statically pipelined architecture. As we will demonstrate, by providing the compiler with a more detailed view of the hardware, we are able to optimize the code in ways that were previously not possible.

3.4.1 Traditional Optimizations

The traditional optimizations that we apply consist of data flow optimizations such as common sub-expression elimination, copy propagation, dead assignment elimination and control flow optimizations such as branch chain removal, empty block removal, and unreachable code removal. These optimizations did not need to be specifically written for the statically pipelined architecture, but are able to achieve additional effects beyond what they could at the RISC level.

Figure 3.6 demonstrates how traditional data flow optimizations can be employed for the static pipeline. On the left is a sequence of traditional instructions that increment a register and then double it. Figure 3.6(b) shows how these two instructions are expanded into statically pipelined instruction effects.

Next we apply copy propagation. This is an optimization which takes into account instruction effects that copy a source to a destination and creates equivalence classes among state elements that have the same value. It then replaces uses of any member of the equivalence class with the oldest member of the class with the same or cheaper access cost where possible. Figure 3.6(c) shows the result of applying this optimization. In this case, the only change is the second to last instruction. We replace the usage of `RS1` and `RS2` with `ALUR`. This is done because `ALUR` is copied into `r[2]` and then into the registers `RS1` and `RS2`, creating the equivalency.

The copy propagation optimization is not useful on its own, but it is helpful in that it enables other data flow optimizations, such dead assignment elimination. Dead assignments are those that write a value into a register or a memory location that is never used. The two reads of `r[2]` into `RS1` and `RS2` are now dead assignments as we no longer reference those values. This causes the assignment `r[2] = ALUR` to become dead in turn as that value is now never referenced. Figure 3.6(d) shows the result of removing these dead assignments.

These two optimizations combine to remove a register write and two register reads. Instead of having to pass the result of the first addition through the register file, we directly reference the output register which is not possible with traditional architectures.

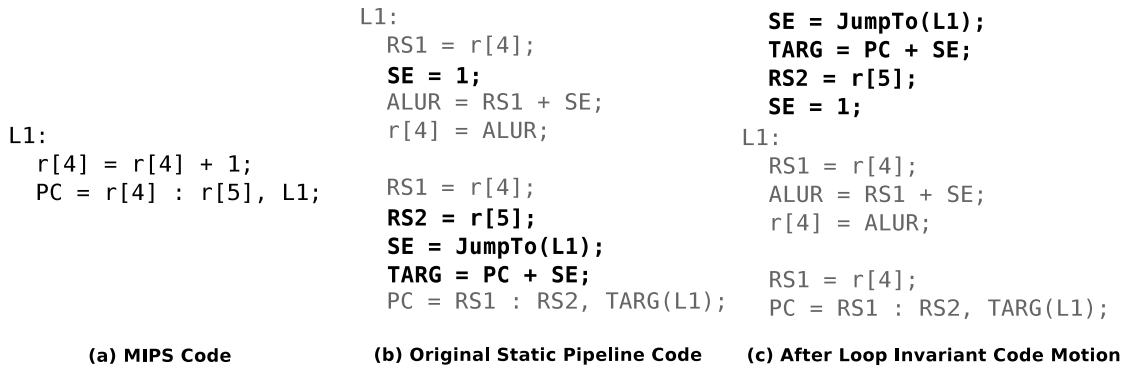

Figure 3.7: Simple Loop Invariant Code Motion

By providing the compiler with the ability to directly reference internal registers, we allow it to apply these optimizations at a lower level and achieve new results.

The control flow optimizations are only useful when other optimizations enabled them. For example if the data flow optimizations produced an empty block or a branch chain by removing instructions, then they will be removed by the appropriate compiler optimization. These optimizations are only useful in these cases, because they were already successfully applied by the MIPS compiler.

3.4.2 Loop Invariant Code Motion

Loop Invariant Code Motion is an optimization in which instructions inside of a loop are moved out of a loop when the results do not change across loop iterations. It is beneficial because loops dominate the execution time of programs, so reducing the number of instructions that are executed for each iteration improves performance and reduces power consumption.

Like the traditional optimizations discussed above, the traditional loop invariant code motion was able to optimize the code in ways that are impossible for traditional architectures. This is due to the finer granularity of instructions. When a RISC-level instruction instruction is broken into separate statically pipelined instruction effects, each effect that is loop invariant can be moved outside of the loop even if the original instruction, as a whole, is not loop invariant.

An example of this can be seen in Figure 3.7. On the left is a trivial loop in RTLs for the MIPS architecture. The loop simply increments the value of `r[4]` until the value is equal to `r[5]`. Neither of these MIPS instructions is loop invariant, so this loop cannot be improved with loop invariant code motion.

Figure 3.7(b) shows how this loop is expanded into statically pipelined instructions. This loop could be improved using copy propagation as described in the previous section, but here we will only demonstrate the loop invariant code motion. The first four instructions correspond to the addition and the last five correspond to the conditional

```
function move_out_sequential_address(loop):
  move_up = false

  if loop has no preheader:
    make a preheader block

  for each block b in the loop:
    if b branches to the loop header:
      if b uses "TARG" in branch:
        move_up = true
        replace "TARG" with "SEQ" in branch

  if move_up:
    insert "SEQ = PC + 1" into preheader
done
```

Figure 3.8: Sequential Address Loop Invariant Code Motion Algorithm

branch. While neither set of instructions can be entirely moved out of the loop, some of the individual instruction effects can be.

The value of the branch target is loop invariant, so the two instructions that calculate it are moved out of the loop. Because branch targets are necessarily a part of branch instructions in the MIPS architecture, it is not possible to move them out of loops - even though the computation always results in the same target. With the statically pipelined architecture, however, they can be moved out of loops with loop invariant code motion.

Likewise the value of `r[5]` is loop invariant, so the read of that register can be moved out of the loop, as can the sign extension of the constant value one. In the MIPS architecture, register reads and sign extensions cannot be moved out of the instructions that reference them even if they are loop invariant. In the statically pipelined architecture they can. Note that, due to scheduling, moving an instruction out of a loop may not affect the performance of the code. It will, however, always have a beneficial effect on energy usage as it causes less work to be done for each loop iteration. By providing the compiler with finer grained control over what happens in each clock cycle, static pipelining allows for greater application of loop invariant code motion than with traditional architectures.

In addition to simple loop invariant code motion providing new benefits for a statically pipelined architecture, we have developed two additional loop invariant code motion optimizations that directly target the static pipeline. In the first, depicted in Figure 3.8, we hoist the calculation of the target address of the top of the loop. As discussed in Chapter 2, the `SEQ` register is used for this purpose.

Figure 3.9 shows an example of this optimization. On the left is MIPS code for a loop that calculates the sum of all numbers from one to `r[4]`. In the center is the statically pipelined code after most other optimizations have been applied. Notice that the target address of the loop is calculated with the two instructions `SE = L6` and `TARG = PC + SE`.

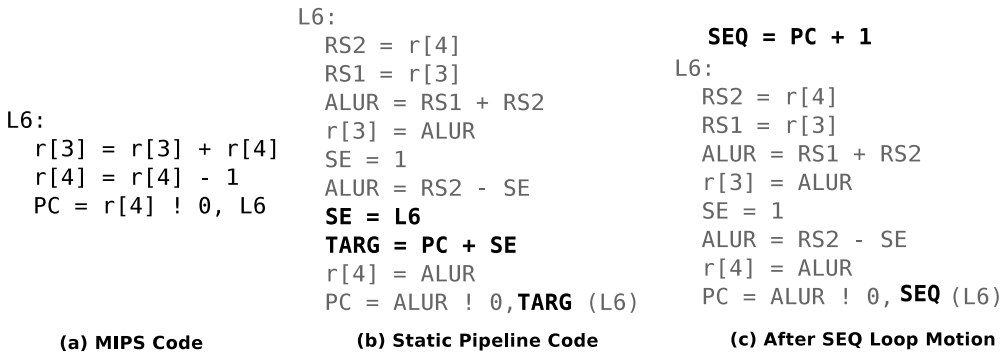

Figure 3.9: Sequential Address Loop Invariant Code Motion Example

Figure 3.9(c) shows the code after applying this optimization. Rather than calculate the target address in the loop, we simply store the address at the top of the loop in the SEQ register, and reference SEQ rather than TARG in the loop. The instruction SEQ = PC + 1 does not actually cause an addition to occur. It simply instructs the architecture to place that value (which is computed each cycle anyway) into the SEQ register.

This optimization is only attempted on innermost loops. This is because there is only one SEQ register, so it is used where it will have the most impact. In order to accomplish this optimization for a given loop, we first identify the preheader block of the loop. The preheader is a basic block that is executed before the header of the loop. If there is no preheader, we create one. For this optimization, the preheader must also be physically before the header of the loop because the instruction to save the branch address in SEQ must be immediately before the address itself. It would be possible to perform this optimization using a basic block that is physically before the loop header, but wasn't a preheader. However this situation is very rare and would complicate the optimization.

Once a preheader has been found, we loop through each block in the loop and check if it branches to the header. If so, we replace the use of TARG in the block's branch with SEQ. If we make that substitution anywhere in the loop, then we add the SEQ = PC + 1 instruction to the preheader of the loop. This optimization itself doesn't remove the instructions that calculate the branch address in TARG, it relies on dead assignment elimination to perform that task. Another benefit of this optimization is that, if there is exactly one other conditional branch in the loop, VPO can then hoist that target calculation out of the loop using loop invariant code motion.

The second loop invariant code motion optimization targeting the static pipeline attempts to move register file accesses out of loops. If a register is used inside of a loop, we can move the usage of it out of the loop and replace the value with a copy register. The algorithm for this optimization can be seen in Figures 3.10 and 3.11. The first figure is an algorithm that either replaces the uses of a register value with a copy register, or checks if a replacement can be done. The algorithm in the second figure makes use of this to perform the optimization.

Like the SEQ optimization discussed previously, this optimization is only attempted for innermost loops. It starts by checking if a preheader exists and, if not, creating one. It then

```
function replace_uses_of_reg(loop, block, reg,
    cp_reg, replace_rs1, replace_rs2, check):

  if block is null, block is not in loop, or we have seen this block:
    return true

  for each instruction i in block:
    if i reads reg into RS1:
      replace_rs1 = true
    else if i reads reg into RS2:
      replace_rs2 = true

    else if i reads another register into RS1:
      replace_rs1 = false
    else if i reads another register into RS2:
      replace_rs2 = false

    else if i uses RS1 and replace_rs1:
      if check and we cannot replace RS1 with cp_reg in i:
        return false
      else:
        replace RS1 with cp_reg in i
    else if i uses RS2 and replace_rs2:
      if check and we cannot replace RS2 with cp_reg in i:
        return false
      else:
        replace RS2 with cp_reg in i

  if RS1 is not an output of block:
    replace_rs1 = false
  if RS2 is not an output of block:
    replace_rs2 = false

  result = true
  for each successor of block:
    if not replace_uses_of_reg(loop, successor, reg,
        cp_reg, replace_rs1, replace_rs2, check):
      result = false

  return result
done
```

Figure 3.10: Register File Loop Invariant Code Motion Algorithm I

```
function move_out_register_reads(loop):
  if loop has no preheader:
    make a preheader block

  if CP2 is not used in loop, and not an output of preheader:
    cp_reg = CP2
  else if CP1 is not used in loop, and not an output of preheader:
    cp_reg = CP1
  else:
    return 0
  if RS1 is not an output of preheader:
    rs_reg = RS1
  else if RS2 is not an output of preheader:
    rs_reg = RS2
  else:
    return 0

  for each block b in the loop:
    for each instruction i in b:
      if i writes a register reg:
        if reg is live exiting loop:
          set reads of reg to -1
        else:
          mark reg as written
      if i reads a register reg:
        if reg has been written:
          set reads of reg to -1
        else:
          if replace_uses_of_reg(loop, loop.header,
              reg, cp_reg, false, false, true)
            increment the reads of reg
          else:
            set reads of reg to -1

  find the register reg that has the maximum reads
  insert "rs_reg = reg" into preheader
  insert "cp_reg = rs_reg" into preheader
  replace_uses_of_reg(loop, loop.header, reg, cp_reg, false, false, false)
  if reg is written in the loop:
    replace reg with cp_reg in the write
done
```

Figure 3.11: Register File Loop Invariant Code Motion Algorithm II

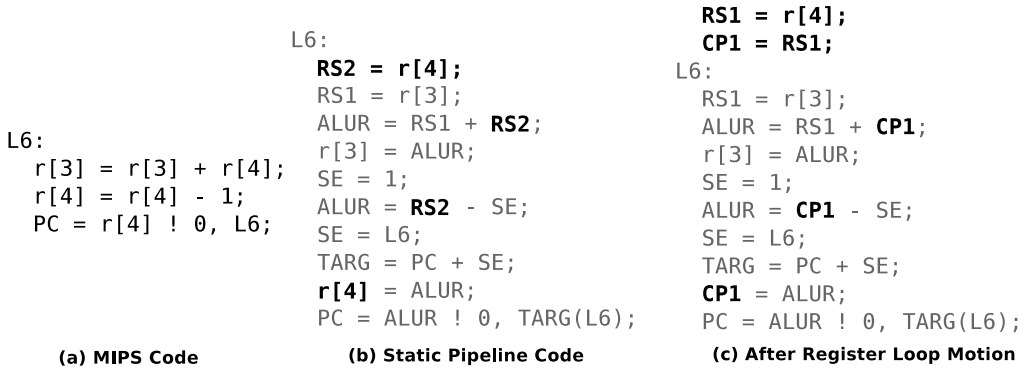

Figure 3.12: Register File Loop Invariant Code Motion Example

checks if there is an available register source and copy register. To do this, the compiler firsts checks if either copy register is currently unused in the loop, and is not live exiting the preheader block. Next it checks if either of the register source registers are live exiting the preheader block. If there aren't available registers, the optimization cannot be applied.

Next, the loop is analyzed for register usage. The compiler looks for registers that are read, but are either never written (and thus loop invariant), or written only once, at the end of the loop. If the register is written at the end of the loop, it cannot be live exiting the loop because then the correct value must actually be in the register file and not a copy register. If the loop has a function call in it, this optimization cannot be applied to any registers that are caller save or *scratch* registers because the compiler must conservatively assume that they are written in the called function. The compiler also tests each register to see if replacing its usage with a copy register is feasible. Some usages of the register source registers cannot be replaced with the copy registers. For example, `RS2` can be used as the target of a jump, but the copy registers cannot. From the set of registers that meet this criteria, the compiler chooses the one that is read most frequently as it will provide the most benefit.

Once the compiler has chosen a register to move out of the loop, it inserts the two instructions to copy the register's value into the copy register in the preheader. Next it replaces all the uses of the register value with the copy register. In order to do this, it keeps track of whether or not `RS1` or `RS2` have the value of the register. The compiler scans each block in the loop recursively. If it finds a read of the given register, it marks the appropriate register source register as holding the register value. If it finds a use of a register source register that is marked as holding the register value, the compiler replaces it with a use of the copy register. If the compiler finds a read of a different register into `RS1` or `RS2`, then that register source register is marked as not holding the register value.

Figure 3.12 shows an example of this optimization. Parts (a) and (b) show the same MIPS code as in the previous example, while part (c) shows the code after applying this optimization. We have utilized the copy register to hold the value of `r[4]`. In order to do this, we insert code before the loop to load the value of `r[4]` into `RS1`, and then copy this value into the copy register `CP1`. All uses of the value of `r[4]` (used through either `RS1` or `RS2` are then replaced with `CP1`. In this example, those uses are the add and subtract ALU

operations. We also replace the write of `r[4]` with a write of `CP1` at the end of the loop. This removes the register read instruction out of the loop. It also replaces one register file write with an internal register write which uses less energy.

3.4.3 Instruction Scheduling

Because statically pipelined architectures achieve good performance by executing multiple pipeline effects in parallel, good instruction scheduling is vital. Our instruction scheduler first schedules instructions locally in each basic block. To do this, the scheduler starts by constructing a data dependence graph for the instructions in the block to represent dependencies between instructions. Each instruction in the block corresponds to one node in the data dependence graph. Edges represent different types of dependencies between instructions.

There are true, or read after write (RAW) dependencies. There are also false dependencies which are either write after read (WAR) or write after write (WAW) dependencies. In addition to these data flow dependencies, we keep track of memory dependencies. A memory store cannot be moved above a load or store of that same location. Because we cannot always distinguish between different memory addresses, we must often assume that memory instructions access the same locations. We are able to disambiguate between memory references that use the same base register with different offsets, or different global addresses.

Transfer of control dependencies must also be taken into account. An instruction cannot be moved if it is a transfer of control, whether it's a branch instruction at the end of the block or a call instruction.

Figure 3.13 shows an example basic block along with a data dependence graph for that block. The thicker, solid lines represent the true dependencies. For example, there is a true dependence between instructions one and two because instruction one calculates the value `RS1` which is used in instruction two.

The thinner dashed lines represent the false dependence's. For example, there is an write after read (WAR) anti-dependence between instructions two and four because instruction four writes the value `RS1` which is used in instruction two. Also, there is a write after write (WAW) output dependence between instructions six and nine since they both write into the `ALUR` register.

There is one memory dependency between instructions two and five because we cannot disambiguate the addresses and must assume that the store in instruction five could overwrite the value loaded in instruction two. They are not shown in the diagram for clarity, but there are also transfer of control dependencies between every instruction and instruction eleven. This prevents the branch from moving up ahead of any other instruction.

Once the data dependence graph is constructed, nodes are selected from it to be scheduled. Choosing the nodes in the original order will always produce a correct schedule, however it is not always the most efficient. It is more efficient to schedule longer chains of instructions first as the longer chains form the critical path through the block. Most instruction schedulers choose the longest chains by counting false dependencies the same as true dependencies. However for the static pipeline, we have far more

```
1  RS1 = r[13];
2  LV = R[RS1];
3  FPUR = LV * CP1;
4  RS1 = r[11];
5  R[RS1] = FPUR;
6  ALUR = RS1 + CP1;
7  r[11] = ALUR;
8  RS1 = r[13];
9  ALUR = RS1 + CP1;
10 r[13] = ALUR;
11 PC = ALUR ! RS2, SEQ;
```

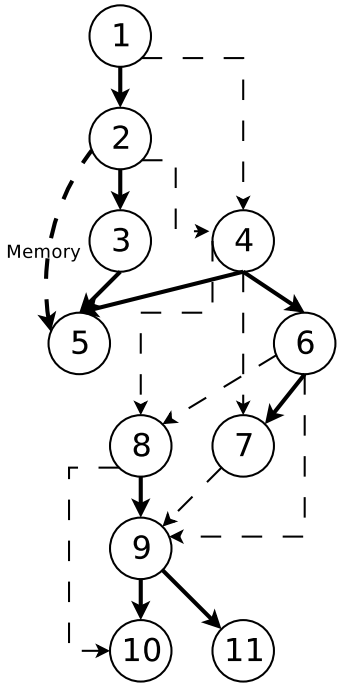

Figure 3.13: Data Dependence Graph

false dependencies than traditional architectures due to the use of internal registers for prescribed purposes. Additionally, instruction schedulers for traditional architectures can avoid false dependencies by renaming registers. With static pipelining, however, renaming registers is much more difficult due to the restrictions placed upon the use of internal registers.

To get around this problem, we attempt to ignore the false dependencies in choosing the longest instruction chain. The scheduler looks at each node in the data dependence graph and finds the one with no true dependencies that prevent it from being scheduled with the longest chain of dependent instructions. We then pick that node to be scheduled next and add any register it sets to a list of live registers. No node that writes one of the live registers can be selected for scheduling. We also must recalculate any false dependencies it has with unscheduled nodes that were originally before it, as these will change once the selected node is scheduled first.

With this method, however, we can arrive at a situation where no nodes can be selected for scheduling because they either have unmet dependencies, or would overwrite live variables. When this happens, the scheduler forgoes attempting to schedule longest chains first and simply schedules the instructions in the block in order. The scheduler is only able to use the more advanced algorithm on approximately 28% of blocks, but since it improves performance of those blocks, it was retained. A dynamic programming approach using backtracking would be able to find a suitable schedule and attempt to schedule longest chains first for all cases, however it would greatly increase compilation time.

```
RS1 = r[13];
LV = R[RS1];            RS1 = r[11];
FPUR = LV * CP1;
ALUR = RS1 + CP1;       R[RS1] = FPUR;  RS1=r[13];
ALUR = RS1+CP1;         r[11] = ALUR;
PC = ALUR ! RS2, SEQ;   r[13]=ALUR;
```

Figure 3.14: Scheduled Code

To schedule the nodes from the data dependency graph, we maintain a resource reservation table. This structure is a two dimensional table where we construct our new schedule. Each column represents one instruction type on the target machine, such as the ALU operation, memory operation and so on. Each row represents one new instruction to be scheduled together. Initially the resource reservation table is empty.

Each time we schedule a node, we start at the end of the table and scan backwards as far as the dependencies will allow. Once we have reached a row in the table where one of the elements has a dependency with the instruction we are scheduling we stop. We then start scanning down the resource reservation table looking for an empty spot to put the instruction. If the dependency that prevented us from going further in the table is an false dependence or a transfer of control dependence, then we can start looking for a spot in the row with the dependent instruction, otherwise we start in the next one.

As we scan down the resource reservation table, we look for a row that has an empty slot in the column corresponding to the type of instruction that we are scheduling. We also need to check if adding the instruction to that row will allow that row to be encoded with one of the available templates. Once we find a suitable row in the resource reservation table, we insert the instruction there and continue on to the next node. Once all nodes have been scheduled into the resource reservation table, we replace the instructions in the block with the scheduled ones in the resource reservation table. Figure 3.14 shows the code from Figure 3.13 after scheduling has been completed.

After all of the blocks are initially scheduled, the compiler attempts to move instructions from the top of each block into all of its predecessors. The algorithm for this cross block scheduling optimization can be seen in Figures 3.15. This improves performance because we only insert instruction effects into a predecessor block if it can be inserted into an existing instruction. First, the block is examined to see if this is possible. If the block is the first block of the function, the optimization is not attempted as it has no predecessors. Also if the block has a predecessor that is more deeply nested in a loop, the optimization is not attempted as it would increase energy usage to move instruction effects into a loop.

Once it has been determined that the block is a candidate for cross-block scheduling, each instruction effect is considered in turn. If the effect is an instruction type that can cause a run-time exception, or cause a stall it is not moved up since it may not be executed along every code path leading to an exception or performance loss. These instructions include floating point operations and memory accesses. Branches and sets of the `SEQ` register also cannot be moved out of blocks as they must be in their original block for control flow to

```
function try_to_insert(e, b, insertions):
  place = null
  for each instruction i in b in reverse:
    if e conflicts with any effect in i:
      return place
    if e conflicts with any effect to be inserted at i:
      return place
    if e can merge into i:
      place = i
done

function cross_block_scheduling( ):
  for each block b:
    change = false
    if b has no predecessors:
      continue
    if b has a predecessor that is more deeply nested than itself:
      continue

    for each instruction i:
      for each instruction effect e:
        if e is a long-running operation or branch:
          continue
        if e conflicts with any previous instructions in b:
          continue
        can_moveup = true
        insertions = []

        for each predecessor p of block b:
          if e conflicts with an output of p:
            can_moveup = false
            break
          insertion_point = try_to_insert(e, p, insertions)
          if not insertion_point
            can_moveup = false
            break
          else
            add (e, insertion_point) into insertions

        if can_moveup:
          for (e, i) in insertions:
            append effect e into instruction i
          change = true
    if change:
      reschedule b
done
```

Figure 3.15: Cross Block Scheduling Algorithm

remain correct. The compiler also checks whether the instruction effect can be moved ahead of all of the other effects in the block - that there is no dependency problem.

For each effect that meets these qualifications, the compiler looks at each predecessor block. If the effect sets any register that is live exiting the predecessor block, the instruction cannot be moved into it, and we move on to the next instruction effect. Next, the compiler scans the predecessor blocks instructions looking to add in the given effect. If the effect can be added to the instruction given the set of encoding templates, the instruction is marked as being the point to append the effect. If an instruction has a dependency issue with the effect, the process ends. If any instruction was marked as being the point to append the effect, the effect is appended, otherwise the effort to move the instruction effect is abandoned.

If the instruction effect can be added into every predecessor block, then the changes are committed, and we move on to the next instruction effect until done. If any effects were able to be moved out, then the block is re-scheduled using the method described above. The ability to schedule instruction effects across basic block boundaries significantly improves performance and code size. Due to the way in which we expand instructions into pipeline effects, basic blocks of statically pipelined code typically start with register file reads, and sign extensions. Likewise they typically end with register file writes, memory operations and branches. Also, until the cross-block scheduling is performed, it is rare to have internal registers live exiting a basic block. For these reasons, there are often many opportunities for the cross-block scheduling algorithm to compact the code.

3.5 Example

This section provides a complete example of the compilation process from source code to assembly. The code produced at each stage is the actual code produced by the compilation stage in question.

Figure 3.16(a) shows the C source code of a simple loop to add a value to every element of an array. The first step in compiling this code for the static pipeline is to generate the optimized MIPS code which can be seen in Figure 3.16(b). Here `r[9]` is used as a pointer to the current element of the array, `r[6]` holds the value of the constant `m`, and `r[5]` has had the value `a + 400` loaded into it which is the last value of `r[9]` in the loop.

The next step is to expand the effects of the MIPS code into statically pipelined instructions. The result of this step can be seen in Figure 3.16(c). Blank lines separate instruction effects corresponding to different MIPS instructions. This increases the number of instructions in the loop from five to nineteen.

Figure 3.17 shows the operation of the copy propagation optimization. On the left is the code before the optimization, and on the right is the code after. This optimization is applied three times in this loop. The first propagates the copy of `LV` into `r[3]` and then into `RS1`. The use of `RS1` is replaced with a use of `LV` in the sixth instruction of the loop. Likewise copies of `ALUR` are propagated through registers in the register file and on into source registers.

Figure 3.18 shows how dead assignments can be removed as a result of copy propagation. The lines in darker text on the left are now dead assignments. Because the use of the registers calculated are no longer used, the instruction computing them are not needed. In

```
for(i = 0; i < 100; i++)
  a[i] += m;
```

(a) C Source Code

```
L9:
  r[3] = R[r[9]];
  r[2] = r[3] + r[6];
  R[r[9]] = r[2];
  r[9] = r[9] + 4;
  PC = r[9] ! r[5], L9;
```

(b) MIPS Code

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  r[3] = LV;

  RS1 = r[3];
  RS2 = r[6];
  ALUR = RS1 + RS2;
  r[2] = ALUR;

  RS1 = r[9];
  RS2 = r[2];
  R[RS1] = RS2;

  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;

  SE = L9;
  TARG = PC + SE;
  RS1 = r[9];
  RS2 = r[5];
  PC = RS1 ! RS2, TARG(L9);
```

(c) Expanded Statically Pipelined Code

Figure 3.16: Source And Initial Code Example

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  r[3] = LV;
  RS1 = r[3];
  RS2 = r[6];
  ALUR = RS1 + RS2;
  r[2] = ALUR;
  RS1 = r[9];
  RS2 = r[2];
  R[RS1] = SE;
  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS1 = r[9];
  RS2 = r[5];
  PC = RS1 ! RS2, TARG(L9);
```

(a) Before Copy Propagation

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  r[3] = LV;
  RS1 = r[3];
  RS2 = r[6];
  ALUR = LV + RS2;
  r[2] = ALUR;
  RS1 = r[9];
  RS2 = r[2];
  R[RS1] = ALUR;
  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS1 = r[9];
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Copy Propagation

Figure 3.17: Copy Propagation Example

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  r[3] = LV;
  RS1 = r[3];
  RS2 = r[6];
  ALUR = LV + RS2;
  r[2] = ALUR;
  RS1 = r[9];
  RS2 = r[2];
  R[RS1] = ALUR;
  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS1 = r[9];
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(a) Before Dead Assignment Elimination

```
L9:
  RS1 = r[9];
  LV = R[RS1];


  RS2 = r[6];
  ALUR = LV + RS2;

  RS1 = r[9];

  R[RS1] = ALUR;
  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;

  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Dead Assignment Elimination

Figure 3.18: Dead Assignment Elimination Example

the example, the reads into `RS1` and `RS2` are removed first which causes the two reads into the register file to become dead.

The resulting code can be seen in 3.18(b). In addition to removing five instructions, we have completely eliminated the use of two registers in the register file to hold intermediary values. In a traditional pipeline, all data values circulate through the centralized register file. This example demonstrates how static pipelining can avoid register file accesses by giving the compiler access to internal registers.

Figure 3.19 shows the removal of redundant assignments. Because the MIPS code needs to read a value from a register every time it needs it, it is common to repeatedly load the same value. When generating statically pipelined code, however, the compiler can simply retain a value in one of the internal registers. In Figure 3.19(a), the value of `r[9]` is read into `RS1` three times without the values changing between, so the compiler removes the last two of them.

In VPO, copy propagation, dead assignment elimination and redundant assignment elimination are all actually handled along with common sub-expression elimination, but are separated here for clarity.

Figure 3.20 depicts how loop invariant code motion can move individual instruction effects out of the loop. Because the branch target does not change, the two instructions that calculate it can be moved out of the loop. Likewise, the constant value four does not change, so it too can be moved out of the loop. This improves both performance and energy usage. With traditional architectures, these computations are loop invariant, but can not be moved out with compiler optimizations due to the fact that these computations cannot be decoupled from the instructions that use them.

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  RS2 = r[6];
  ALUR = LV + RS2;
  RS1 = r[9];
  R[RS1] = ALUR;
  SE = 4;
  RS1 = r[9];
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(a) Before Redundant Assignment Removal

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  RS2 = r[6];
  ALUR = LV + RS2;

  R[RS1] = ALUR;
  SE = 4;

  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Redundant Assignment Removal

Figure 3.19: Redundant Assignment Elimination Example

```
L9:
  RS1 = r[9];
  LV = R[RS1];
  RS2 = r[6];
  ALUR = LV + RS2;
  R[RS1] = ALUR;
  SE = 4;
  ALUR = RS1 + SE;
  r[9] = ALUR;
  SE = L9;
  TARG = PC + SE;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(a) Before Code Motion

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
L9:
  RS1 = r[9];
  LV = R[RS1];
  RS2 = r[6];
  ALUR = LV + RS2;
  R[RS1] = ALUR;
  ALUR = RS1 + SE;
  r[9] = ALUR;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Code Motion

Figure 3.20: Loop Invariant Code Motion Example

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
L9:
  RS1 = r[9];
  LV = R[RS1];
  RS2 = r[6];
  ALUR = LV + RS2;
  R[RS1] = ALUR;
  ALUR = RS1 + SE;
  r[9] = ALUR;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(a) Before Register Code Motion

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
L9:
  LV = R[CP1];
  RS2 = r[6];
  ALUR = LV + RS2;
  R[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Hoisting r[9]

Figure 3.21: Register Invariant Code Motion I

Figure 3.21 depicts the specialized loop invariant code motion pass targeting register accesses can move further effects out of the loop. Recall that this optimization pulls register reads out of loops and uses a copy register to store the value. Here the optimization is performed on `r[9]`. The register is read into `CP1`, and its uses are replaced with `CP1` as well. Because `r[9]` is written at the end of the loop, that write is replaced with a write into `CP1` instead.

Figure 3.22 shows a second application of the loop invariant code motion optimization discussed above. Because we have a remaining copy register, and there are further register reads in the loop, the optimization is applied again. Here it is applied on the register `r[6]`. It is hoisted out of the loop by assigning it to `RS1` and then `CP2`. The use of the value is then replaced by `CP2`. This optimization enables the standard loop optimization to then pull out the read of `r[5]` due to the assignment of `RS2` now being loop invariant. Between the two applications of this transformation, the compiler is able to hoist three register reads and one register write out of the loop which has a significant impact on the energy usage of this loop kernel.

Figure 3.23 depicts the optimization pass that replaces the calculated branch target with the value saved in `SEQ` at the top of the loop. The instruction that saves the next sequential address at the start of the loop is inserted, and the loop branch is modified to jump to `SEQ` instead of `TARG`. The two instructions that calculate the value of `TARG` are then eliminated by the dead assignment elimination optimization.

Figure 3.24(a) shows the code thus far. The optimizations to this point have reduced the original nineteen instructions in the loop to only six. Figure 3.24(b) shows the code after scheduling is applied. This example also shows the branch being split into two instructions, which happens right before scheduling. First the `NPC` status register is set to specify the type and destination of the branch, and then the comparison is done. This is because, as discussed in Chapter 2, branches must be specified ahead of time to avoid performing branch predictions on every instruction. This transformation is done just before scheduling

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
L9:
  LV = M[CP1];
  RS2 = r[6];
  ALUR = LV + RS2;
  M[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  RS2 = r[5];
  PC = ALUR ! RS2, TARG(L9);
```

(a) Before Hoisting r[6] and r[5]

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
  RS1 = r[6];
  CP2 = RS1;
  RS2 = r[5];
L9:
  LV = M[CP1];
  ALUR = LV + CP2;
  M[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  PC = ALUR ! RS2, TARG(L9);
```

(b) After Hoisting r[6] and r[5]

Figure 3.22: Register Invariant Code Motion II

```
  SE = L9;
  TARG = PC + SE;
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
  RS1 = r[6];
  CP2 = RS1;
  RS2 = r[5];
L9:
  LV = M[CP1];
  ALUR = LV + CP2;
  M[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  PC = ALUR ! RS2,TARG (L9);
```

(a) Before Using SEQ REgister

```
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
  RS1 = r[6];
  CP2 = RS1;
  RS2 = r[5];
  SEQ = PC + 1;
L9:
  LV = M[CP1];
  ALUR = LV + CP2;
  M[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  PC = ALUR ! RS2, SEQ (L9);
```

(b) After Using SEQ Register

Figure 3.23: Sequential Address Invariant Code Motion Example

```
  SE = 4;
  RS1 = r[9];
  CP1 = RS1;
  RS1 = r[6];
  CP2 = RS1;
  RS2 = r[5];
  SEQ = PC + 1;
L9:
  LV = M[CP1];
  ALUR = LV + CP2;
  M[CP1] = ALUR;
  ALUR = CP1 + SE;
  CP1 = ALUR;
  PC = ALUR ! RS2, SEQ(L9);
```

(a) Before Scheduling

```
  SE = 4;      RS1 = r[9];
  CP1 = RS1;   RS1 = r[6];
  CP2 = RS1;   RS2 = r[5];    SEQ = PC + 1;
L9:
  ALUR = CP1 + SE;       LV = M[CP1];
  ALUR = LV + CP2;       CP1 = ALUR;    NPC = b SEQ(L9);
  PC = ALUR ! RS2, NPC;  M[CP1] = ALUR;
```

(l) After Instruction Scheduling

Figure 3.24: Scheduling Example

Table 3.1: Example Loop Results

Metric	MIPS	Static Pipeline
Instructions	5	3
ALU Operations	5	3
Register Reads	8	0
Register Writes	3	0
Branch Calculations	1	0
Sign Extensions	2	0

so as to have the previous optimizations and analyses not have to know that branches are done in multiple instructions.

The scheduler is able to compress the six instructions in the loop down to only three. In this case it schedules the second write to `ALUR` before the first one. The ability to schedule across false dependency boundaries enables the compiler to produce a better schedule. The instructions before the loop are also shown scheduled, but in the actual code, they are merged with existing instructions in the block before the loop.

Table 3.1 shows a comparison of the MIPS code for this example and the code produced by the statically pipelined compiler. The code for the static pipeline improved upon the performance of the MIPS code by reducing the number of instructions in the loop from five to three. It also has eliminated all accesses to the register file inside the loop. Because of the relatively high energy cost of register accesses, this should result in substantial energy savings. The statically pipelined code also reduced the number of ALU operations by not adding zero when calculating a memory address, and eliminated the branch calculations and sign extensions.

CHAPTER 4

EVALUATION

This chapter presents an experimental evaluation of the statically pipelined architecture described in this dissertation. First we discuss the experimental setup used for the evaluation. Next we will present the results including those for performance, code size and register accesses. Lastly we will estimate the energy savings achieved by static pipelining.

4.1 Experimental Setup

We use 15 benchmarks from the MiBench benchmark suite [11], which is geared especially for embedded applications. This benchmark suite includes the following different categories: *automotive*, *consumer*, *network*, *office*, *security* and *telecomm*. The benchmarks used are shown in Table 4.1. For our experiments we used at least two benchmarks from each category.

We extended the GNU assembler to assemble statically pipelined instructions and implemented a simulator based on the SimpleScalar suite [1]. In order to avoid having to compile all of the standard C library and system code, we allow statically pipelined code to call functions compiled for the MIPS. As described earlier, a status bit is used to indicate whether it is a MIPS or statically pipelined instruction. After fetching an instruction, the simulator checks this bit and handles the instruction accordingly. On a mode change, the simulator will also drain the pipeline.

For all benchmarks, when compiled for the static pipeline, over 90% of the instructions executed are statically pipelined ones, with the remaining MIPS instructions coming from

Table 4.1: Benchmarks Used

Category	Benchmarks
automotive	bitcount, qsort, susan
consumer	jpeg, tiff
network	dikstra, patricia
office	ispell, stringsearch
security	blowfish, rijndael, sha
telecom	adpcm, CRC32, FFT

calls to standard library routines such as *printf*. All cycles and register accesses are counted towards the results whether they come from the MIPS library code or the statically pipelined code. Were all the library code compiled for the static pipeline as well, the results would likely improve as we would not need to flush on a mode change, and also we would have the energy saving benefits applied to more of the code.

For the MIPS baseline, the programs were compiled with the original VPO MIPS port with all optimizations enabled and run through the same simulator, as it is also capable of simulating MIPS code. We extended the simulator to include branch prediction with a simple bimodal branch predictor with 256 two-bit saturating counters, and a 256-entry branch target buffer. The branch target buffer is only used for MIPS code as it is not needed for the static pipeline. The simulator was also extended to include level one data and instruction caches. These caches were configured to have 256 lines of 32 bytes each and are direct-mapped.

Each of the following graphs represent the ratio between static pipelining code to MIPS code. This is done because the different benchmarks have drastically different running times. So a ratio of 1.0 means that the value was the same for the MIPS and static pipeline. A ratio less than 1.0 means that the static pipeline has reduced the value, while a ratio over 1.0 means that the static pipeline has increased the value.

Each bar represents a different simulation, with some benchmarks having several simulations. For example the security benchmarks such as *blowfish* have an encode and decode process. For the averages, the ratios are averaged rather than the raw numbers. This is to weight each benchmark evenly rather than giving greater weight to those that run longer. When a given benchmark had more than one simulation associated with it, we averaged the figures for all of its simulations and then took that as the figure for that benchmark. We did this averaging to avoid weighing benchmarks with multiple runs more heavily.

4.2 Results

Figure 4.1 shows the simulation results for execution cycles. Many of the benchmarks in MiBench are dominated by fairly tight loops. This means that the performance difference is largely determined by how well the static pipeline compiler does on that one loop. That is the primary reason for the relatively large deviation among benchmarks. For example, our compiler does quite well with the main loops in *FFT* and the *tiff* color conversion programs which leads to the substantial speedups. On the other hand, *bitcount* and *adpcm* has more instructions in the main loop leading to execution time increases for those benchmarks. On average, the statically pipelined code performed just slightly better than the MIPS code.

Figure 4.2 shows the compiled code size for the benchmarks. Some of the different simulations under the same benchmark, such as *susan*, have the same code size because the compiled code is the same while others, such as *jpeg*, have small differences due to having different source files for each program. The static pipeline compiler produces nearly the same code sizes as the MIPS compiler with an increase of 2.8% on average.

Figure 4.3 shows the simulation results for register file reads. Because the static pipeline is able to use values in internal registers directly, it is often able to bypass the centralized

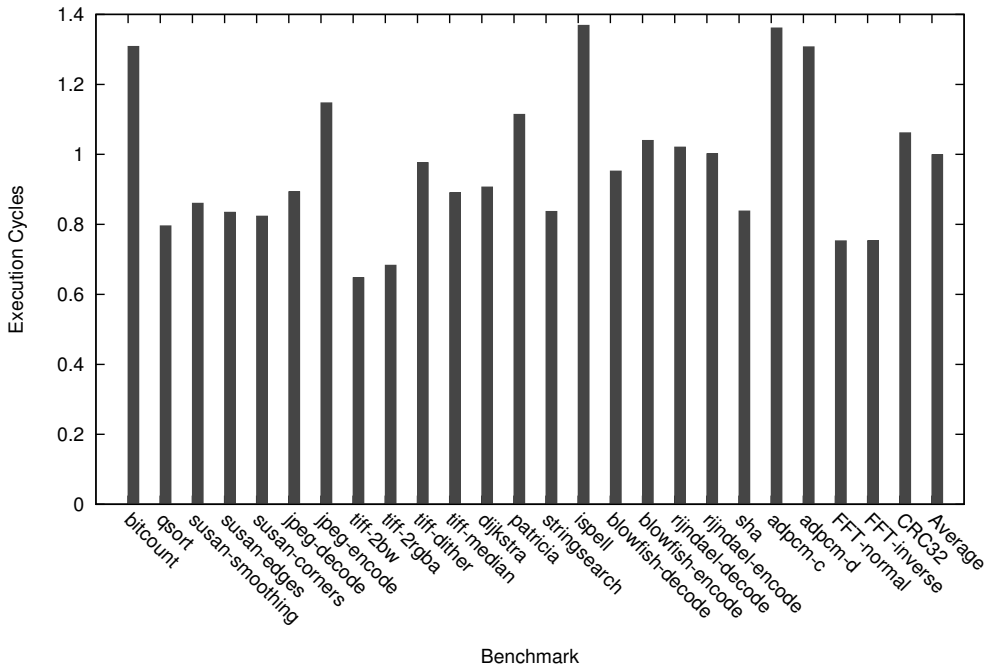

Figure 4.1: Execution Cycles

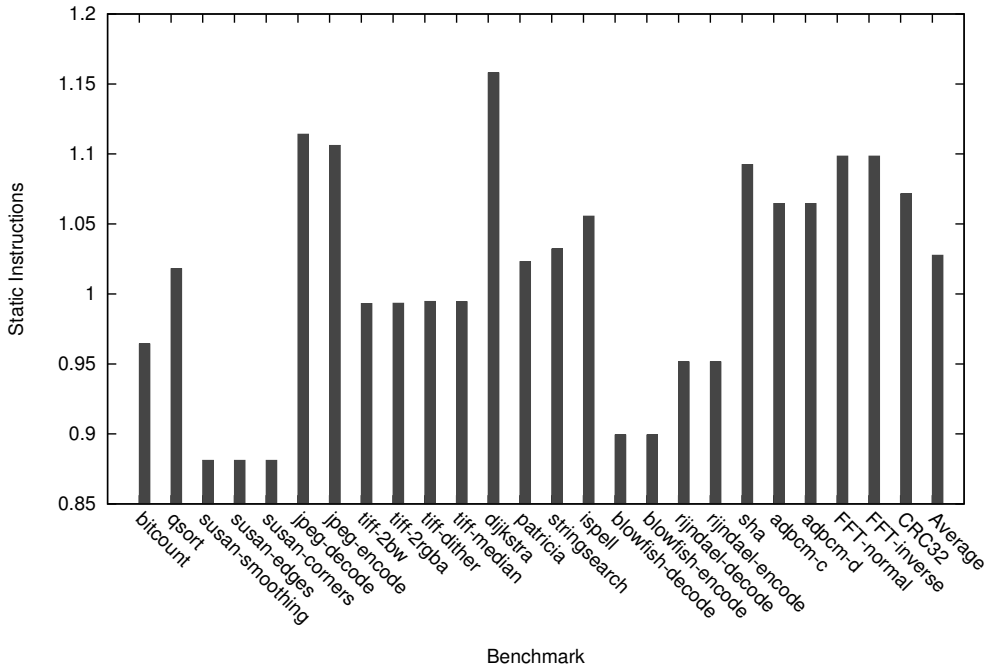

Figure 4.2: Code Size

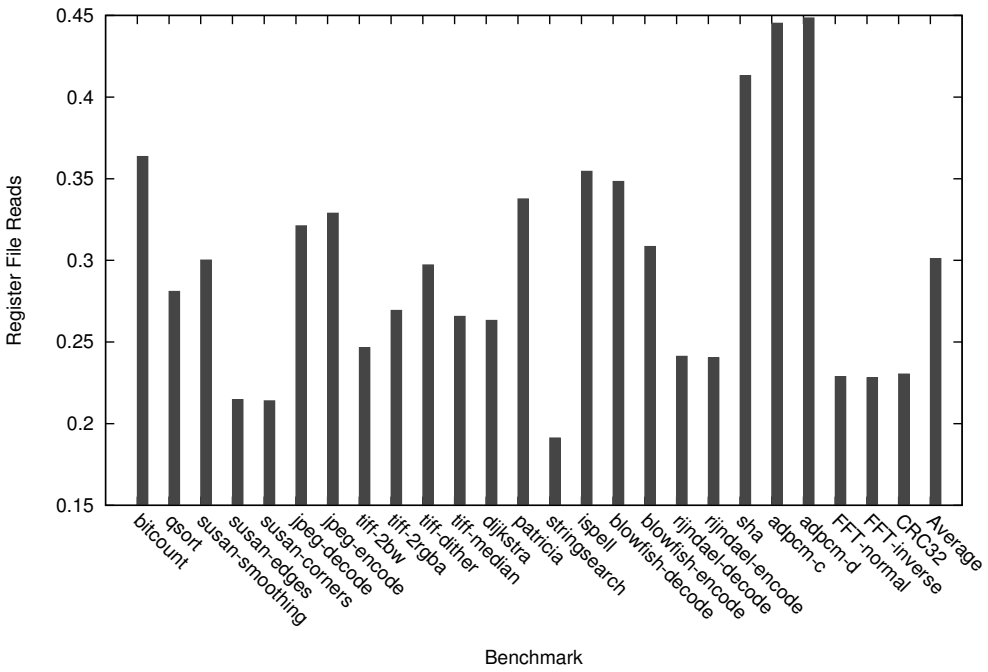

Figure 4.3: Register File Reads

register file as discussed in Chapter 3. For this reason, we are able to remove nearly 70% of the register file reads. For the MIPS baseline pipeline, we only count register file reads when the instruction actually references the register file.

Figure 4.4 shows the simulation results for register file writes. Like register reads, the compiler is able to remove a substantial number of these accesses, around 63% on average. As in the example in Section 3.5, some loops had nearly all of the register accesses removed such as *rijndael* and *CRC32*. Because the register file is a fairly large structure that is frequently accessed, these register access reductions should result in substantial energy savings.

Figure 4.5 shows the simulation results for internal writes. For the MIPS programs, these internal accesses are the number of accesses to the pipeline registers. Because there are four such registers, and they are read and written every cycle, this figure is simply the number of cycles multiplied by four. For the static pipeline, the internal accesses refer to the internal registers. Because the static pipelining code explicitly instructs the architecture when to access an internal register, we are able to remove a great deal of these accesses, over 60% on average. Also note that the MIPS pipeline registers each hold many different values, and so are significantly larger than the internal registers of a static pipeline.

Figure 4.6 shows the simulation results for internal reads. It should be noted that reads of the statically pipelined internal registers or the MIPS pipeline registers do not actually use any energy. They are each single registers, so there is no logic needed to index them, as there is for the register file reads. They are just included here for completeness.

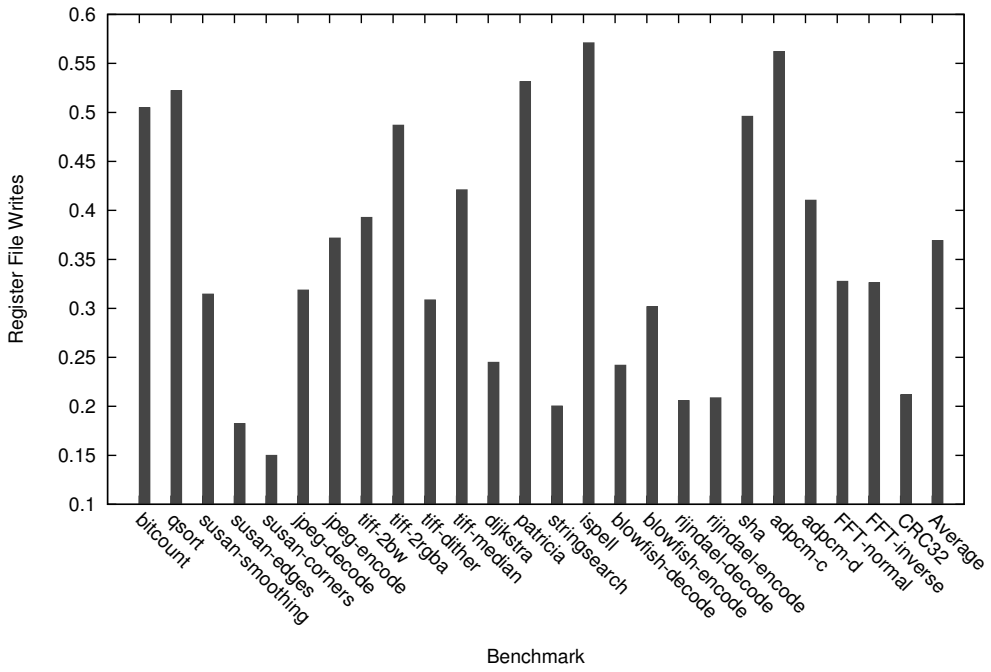

Figure 4.4: Register File Writes

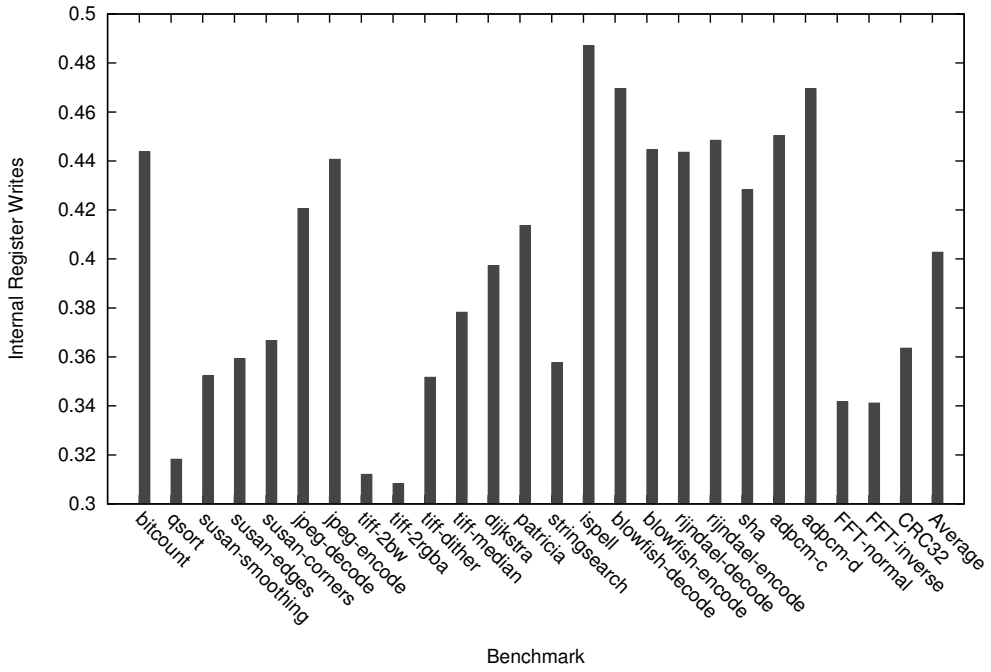

Figure 4.5: Internal Register Writes

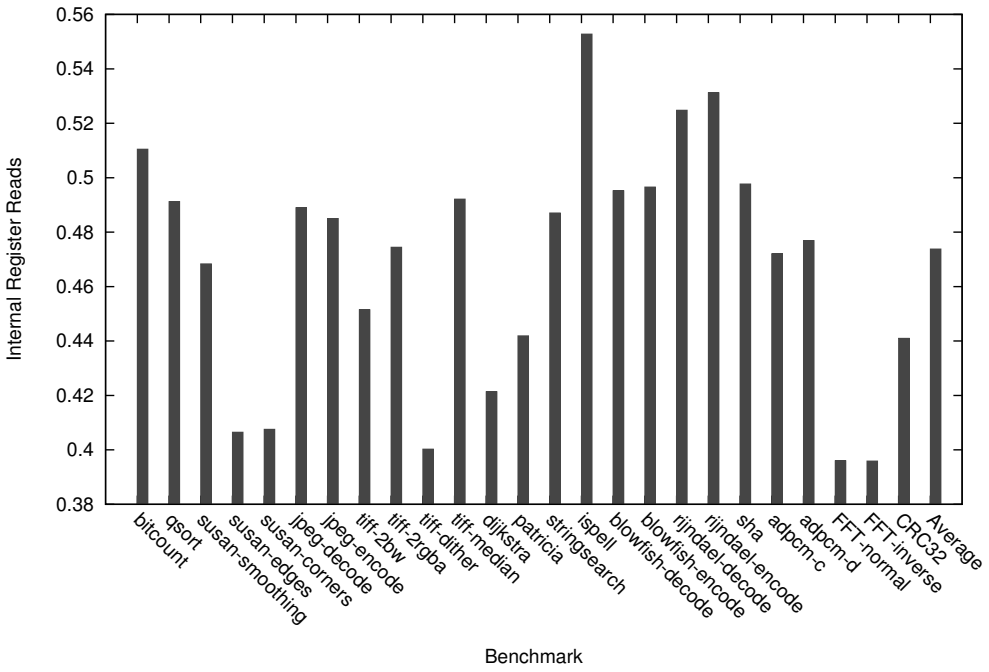

Figure 4.6: Internal Register Reads

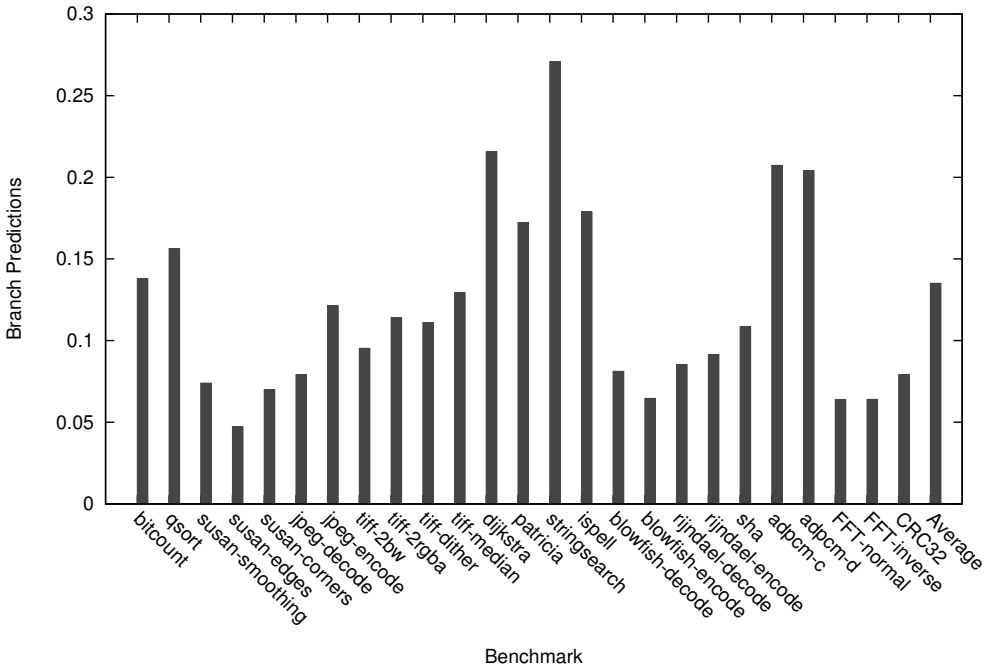

Figure 4.7: Branch Predictions

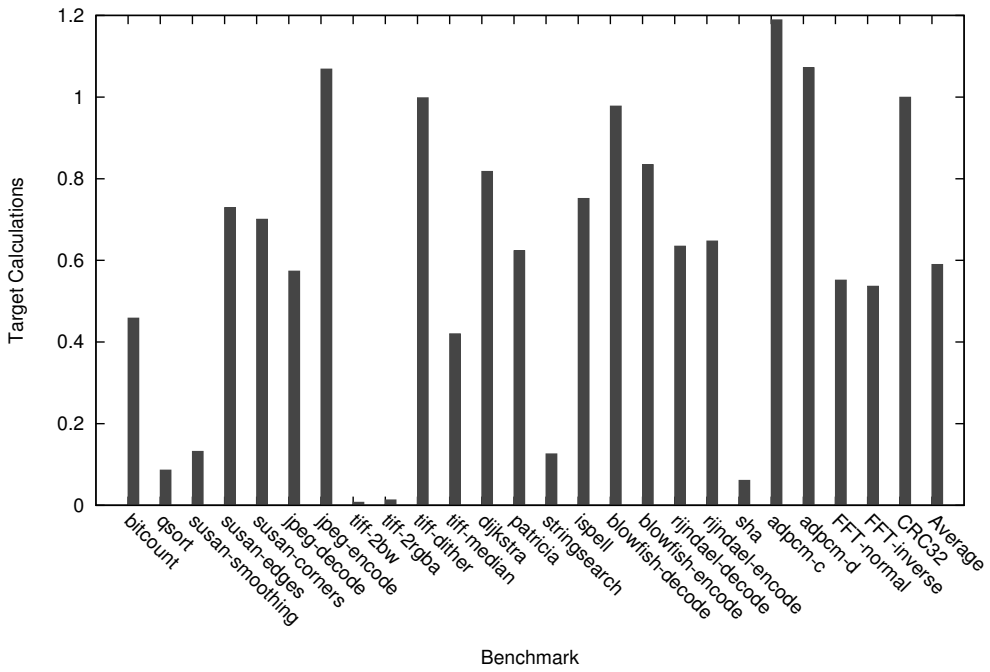

Figure 4.8: Branch Target Calculations

Figure 4.7 shows the simulation results for branch predictions. Recall from Chapter 2 that the static pipeline specifies when a conditional branch will occur one cycle ahead of time. This eliminates the need to predict branches except when the instruction actually is a conditional branch. This results in a substantial decrease in the number of branch prediction buffer accesses, over 86% on average.

Figure 4.8 shows the simulation results for branch target calculations. Because the static pipeline has the ability to avoid calculating branch targets for innermost loops by saving the next sequential address at the top of the loop, and by hoisting other branch target address calculations out of loops as invariant, we are able to reduce the number of branch target calculations substantially at over 40%. Notice that some benchmarks, such as the *tiff* color conversion benchmarks or *sha*, have had nearly all of the branch calculations removed by the compiler. This is due to the fact that the kernel loops have no branch target calculations in them any longer.

Table 4.2 summarizes the average results from the graphs above. As can be seen, we have significantly reduced the number of register file accesses, internal register accesses, branch predictions, and branch target address calculations. At the same time, we have slightly decreased the number of execution cycles, with a small increase to code size.

Table 4.2: Summary of Results

Metric	Average Static Pipeline to MIPS Ratio
Execution Cycles	0.999
Code Size	1.028
Register Reads	0.301
Register Writes	0.369
Internal Reads	0.474
Internal Writes	0.403
Branch Predictions	0.135
Target Calculations	0.590

Table 4.3: Pipeline Component Relative Power

Component	Relative Access Power
Level 1 Caches	1.66
Branch Prediction Buffer	0.74
Branch Target Buffer	2.97
Register File Access	1.00
Arithmetic Logic Unit	4.11
Floating Point Unit	12.60
Internal Register Writes	0.10

4.3 Processor Energy Estimation

This section presents an estimation of the processor energy savings achieved by the static pipelining approach. This estimate uses the simulated counts of events such as register file accesses, branch predictions and ALU operations along with estimations of how much power is consumed by each event.

The SRAMs within the pipeline have been modelled using CACTI[30]. Other components have been synthesized for a 65nm process, then simulated at the netlist level to determine average case activation power. We have normalized the power per component to a 32-entry dual-ported register file read, because the power per component are dependent on process technology and other implementation dependent issues. The ratios between component power are also somewhat dependent on process technology, however these differences should not have a qualitative impact on the final estimates. The resulting total energy estimate is a linear combination of the number of activations and the power attributions per component. The relative power per activation we attribute to each component is given in Table 4.3.

Figure 4.9 shows the results of this analysis. On average, static pipelining reduces processor energy usage by 45%. This savings comes primarily from the reduced register file accesses, branch prediction table accesses and the fact that we do not need a BTB. Of course these results are also affected by the relative running time of the benchmark

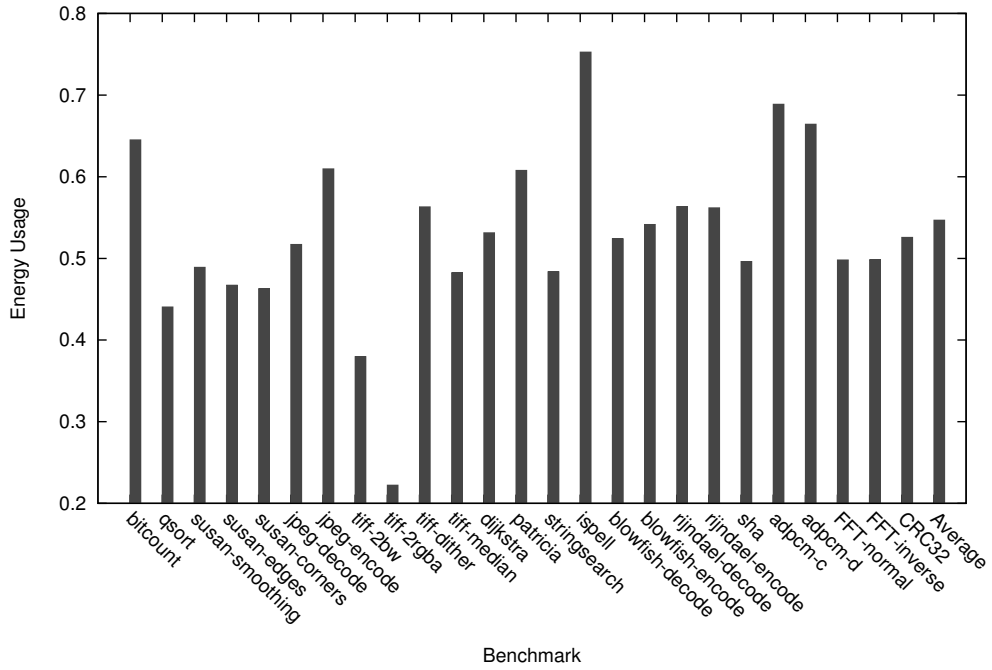

Figure 4.9: Estimated Energy Usage

as that has a direct effect on instruction cache usage and static power consumption. This estimation does not take into account certain other areas of energy savings such as simpler hazard detection logic or the fact that MIPS pipeline registers are much larger than the static pipeline's internal registers.

While this is only an estimation, it was done conservatively and suggests significant energy savings. As discussed in Chapter 6, we intend to more accurately estimate energy benefits by doing full pipeline netlist simulations for both the MIPS and static pipelines.

CHAPTER 5

RELATED WORK

There has been much research focused on reducing power consumption in general purpose processors. First we will discuss instruction set architectures that reduce hardware complexity. Second we will present micro-architectural strategies for reducing energy usage. Lastly we will examine compilation and encoding techniques for improving energy efficiency.

5.1 Instruction Set Architectures

Instruction set architectures have a great impact on the performance and energy efficiency of the underlying micro-architecture. Many ISAs have been proposed over the years that have focused on improving performance, or energy-efficiency. This section will describe some of these ISAs.

Computers in the late 1960s and 1970s used complex instructions that were implemented in terms of low-level instructions. These are called CISC instructions and micro-instructions respectively. Reduced instruction set computers (RISCs) [22] were inspired by micro-instructions in that the instructions were simple and took a small, predictable amount of time to execute. RISC computers were also simpler to implement and took advantage of pipelining to increase performance.

Very Long Instruction Word (VLIW) architectures [8] were designed to provide instruction level parallelism with simpler hardware. Rather than relying on hardware to figure out which instructions could be executed in parallel, VLIW architectures specify this information explicitly. Explicitly Parallel Instruction Computing (EPIC) [26] architectures were developed as an extension to VLIW that gave the compiler even more control over execution. Transport Triggered Architectures (TTAs) [6] are similar to VLIW architectures, except that they allow values to be explicitly passed between the ports of functional units. This removes the necessity of routing all values through large, multi-ported register files. By removing many register reads and writes, and decreasing the number of register ports, TTAs save energy while still allowing high performance as in VLIW.

No Instruction Set Computer (NISC) architectures [23] have no fixed instruction set that bridges the gap between software and the micro-architecture. Instead the compiler generates control signals directly. This allows for hardware/software co-design and improved performance. Coarse-Grained Reconfigurable Architectures (CGRAs) [20] consist of a grid

of functional units and register files. Programs are mapped onto the grid by the compiler, which has a great deal of flexibility in scheduling. The FlexCore processor [31] also exposes datapath elements at the architectural level. The design features a flexible datapath with an instruction decoder that is reconfigured dynamically at runtime.

These architectures rely on multiple functional units and large register files to improve performance at the expense of a significant increase in code size. In contrast, static pipelining focuses on improving energy usage without adversely affecting performance or code size.

Several instruction set architectures have focused on providing both 32-bit and 16-bit instructions in order to reduce code size and fetch energy. Examples of such architectures are Arm/Thumb [10] and MIPS16 [15]. With these architectures, there is a trade-off between the small code size of the 16-bit instructions and the high performance of the 32-bit instructions. The task of choosing between them falls to the compiler.

One important axis for comparison between instruction set architectures (ISAs) is the amount of details that are encoded in the instructions. ISAs that provide more information have more complications in the compiler, but fewer in the micro-architecture. Less complexity in the micro-architecture implies more energy efficiency. On the other hand, more information in the instruction set often increases code size and the energy needed to fetch instructions. Static pipelining is able to provide detailed instructions that allow for simpler hardware without significantly increasing the code size.

5.2 Micro-Architecture

There has been much research that attempts to decrease power consumption through micro-architectural modifications. In most architectures, the primary way of communicating information between instructions is through the register file, creating an energy bottleneck in this structure. Because of this, many of the techniques we will examine focus on the register file.

Tseng and Asanovic [32] present several ways to reduce the power consumption of register files. *Precise read control* checks to see whether the register file operands are needed for the instruction before reading them. *Bypass R0* avoids actually reading register zero, and just passes a literal zero to the next pipeline stage. *Read Caching* is based on the observation that subsequent instructions will often read the same registers. When a register is read, they cache the value. If it is read the next cycle, they read the small register instead of the large register file. *Bypass Skip*, avoids reading operands from the register file when the result would come from forwarding anyway. Park et. al. [21] build on Tseng's work on bypass skip. In this work, they modify a compiler to schedule instructions in such a way as to make more values come from forwarding. By doing this, they aim to avoid even more register reads. Static pipelining can remove these register file accesses without the need for special hardware logic which negates some of the benefit.

Zyuban and Kogge [35] argue that the register file is a serious energy bottleneck and that the way to alleviate this is to split the register file into multiple ones. In order to do this, they dispatch instructions to clusters based on which registers it references. These

clusters each have their own register file with a subset of the total registers. This will cut down the size of each register file and also the number of ports.

Kalambur and Irwin [14] make the case for using a CISC-style memory-register addressing mode for an embedded architecture. The idea is to use instructions that read a register and a memory location and write to a register to cut down on register file power consumption. Additionally it will result in smaller code size which will reduce power dissipation in the instruction cache as well. However, this technique will decrease the frequency at which the processor can run.

Sami et. al. [24] present a way to save register file power by exploiting the fact that forwarding leads to useless reads and writes. They identify these wasteful reads and writes statically in the compiler and set bits in the instructions to signify that these accesses should not be done. They target VLIW architectures where register file power is an even greater concern than in traditional processors due to the increased number of register ports. Static pipelining achieves this benefit without the need to check if a register file access has been disabled.

Sassone et. al. [25] present a technique to improve processor efficiency by optimizing sequences of instructions called strands. A strand is a set of instructions that has some number of inputs and only one output. The key idea here is that if a strand is treated as one instruction, then the intermediate results do not need to be written to the register file. They modify a compiler to identify strands, schedule them together and tag them. When a strand is fetched, it is placed in a strand accumulation buffer until the whole strand is there. They then dispatch it as a single instruction where it is executed on a multi-cycle ALU which cycles its outputs back to its inputs.

Scott et. al. [27] present details of the Motorola MÂůCore architecture. They use 16-bit instructions on a 32-bit datapath. The ISA is a load-store architecture which also includes predication and special doze, wait and stop instructions to facilitate power management.

Yang and Orailoglu [34] present a technique to eliminate the wastefulness of accessing the branch predictor every cycle. They do this by statically calculating the length of each basic block in order to know which instructions need branch prediction. When the branch predictor is not needed, they don't access the it and also put the tables in a drowsy mode and wake them up before being needed. This technique does require a counter to keep track of the position within a block, which uses some energy.

Davidson and Whalley [7] present a technique for reducing the cost of branches by decoupling the calculation of branch targets from the branch operations. The branch addresses are stored in special branch registers. This reduces branch stalls because the branch address is calculated before the actual transfer of control. Also, because branch targets are almost always constant, these calculations can be moved out of loops. Static pipelining implements a similar concept, but at a lower level instruction set.

Mathew et. al. [19] developed a processor to perform perception algorithms with low power consumption. Their design aims to be a middle ground between an application-specific and general-purpose processor This is done by allowing the compiler more control over movement of data, clock gating and looping. The compiler also controls the use of a scratch-pad memory in lieu of a cache.

Canal et. al. [4] have presented a technique for reducing power consumption in many parts of a pipeline. The technique is based on the observation that, while most datapaths

are capable of dealing with 32 or more bits at a time, frequently most of these are not significant. For calculations such as additions, they only operate on the significant bytes. For reading values, they store the number of significant bytes along with each value to reduce the number of bytes accessed.

Hiraki et. al. [13] present a pipeline design that reduces energy by keeping a loop buffer of decoded instructions. When encountering a special *begin loop* instruction, the processor begins to fill the buffer with decoded instructions while doing the first iteration and sets a loop counter to the number of loop iterations. Subsequent iterations need not be fetched from the instruction cache or decoded until the counter runs out at which point the processor begins fetching normally again.

Shimada et. al. [29] present an alternative to dynamic voltage scaling. The reason for pipelining is to increase the clock, but employing pipelining, and then reducing the clock is counterproductive. They propose a pipeline where stages are unified when the frequency is decreased. This reduces power by gating pipeline registers and also reduces branch penalties. The pipeline stages are combined by simply passing the control signals between stages instead of through the latched pipeline registers.

Most of these micro-architectural techniques aim to reduce energy usage by removing energy waste in some portion of the pipeline. Static pipelining achieves most of these results, and more, by allowing the compiler to explicitly instruct the actions of the hardware at a low level. This has the benefit that the micro-architecture doesn't need special logic to look for energy saving opportunities, which uses energy in itself.

5.3 Compilation

There has been much research on encoding and compilation techniques for embedded systems, most of which have been aimed at reducing code size. This is an important goal because it reduces the size of program memory and has a positive impact on energy due to reduced demand on the instruction cache, though energy may also be consumed decompressing the instructions. There has also been some work, however, on compiler and encoding techniques aimed specifically at reducing energy usage. We discuss some of these techniques in this section.

Cheng and Tyson [5] present a way to encode instructions efficiently. It uses a programmable decoder to encode an application-specific, 16-bit instruction set for a general-purpose processor. This work profiles instructions used in an application in order to devise the specific instruction set to use. This approach allows for higher code-density and less power consumption without sacrificing performance.

Hines et. al. [12] present a technique for reducing program size, power consumption and execution time by storing commonly used instructions in an instruction register file. They allow instructions to reference multiple entries in the instruction register file. They modified a compiler so that it can use heuristics and a greedy algorithm to pack instructions into the IRF.

Woo et. al. [33] present a method for encoding instructions so as to reduce fetch power consumption by minimizing the hamming distance (the number of bits that differ) between successive instructions. First they rename registers so that the register fields have minimum

switching. Next they set unused portions of instructions to match the bits of adjacent instructions. Lastly they encode the opcodes in such a way as to minimize switching between successive opcodes.

Lee et. al. [17] present another work that focuses on decreasing the fetch power by minimizing the switching between successive instructions. However this one focuses on rearranging instructions rather than re-encoding them. They modify a VLIW compiler in order to schedule instructions to minimize the hamming distance between subsequent instructions. They modify the scheduler to rearrange effects in instructions (horizontal) and instructions within a sequence (vertical).

Ayala et. al. [2] present a way to reduce register file power consumption by having the compiler power down sections of the register file that are not needed. If the compiler sees a set of registers is inactive for a period of time, it can put them into a *drowsy* state which uses less power, but retains the values. This is accomplished with a new instruction that specifies which registers should be drowsy.

CHAPTER 6

FUTURE WORK

This chapter describes some potential avenues for future exploration on the topic of static pipelining. We discuss the goal of building a physical implementation of a statically pipelined processor, a few further optimizations and refinements that could be implemented for the architecture and compiler, as well as applying the idea of static pipelining to higher performance architectures.

6.1 Physical Implementation

As discussed in Chapter Four, our current estimates on energy savings are only estimates. While our results were estimated conservatively, and are still significant, it would increase the strength of this work to have more accurate results. Our current estimates are based on counting the number of times different events happen in the micro-architecture and estimating the energy costs of each event. This method does not allow us to take into account other areas of energy savings such as the fact that we no longer need to do forwarding and that hazard detection is much simpler. In order to evaluate the energy saved from these sources, we plan to construct a physical implementation using a hardware design language such as Verilog or VHDL.

A physical implementation would also answer the question, as discussed in Section 2.2 of whether or not the decoding of instructions should be done as part of either the fetch or execute stages, and how much, if any, it would reduce the clock rate. Having a physical implementation would also allow us to measure the area and timing of the processor accurately.

6.2 Additional Refinements

The software pipelining compiler optimization could be applied to further improve the performance of statically pipelined code. This optimization is a technique used to exploit instruction-level parallelism in loops [28]. Loops whose iterations operate on independent values, typically in arrays, provide opportunities for increased parallelism. Software pipelining overlaps the execution of multiple iterations and schedules instructions in order to allow the micro-architecture to take advantage of this parallelism. Software pipelining would have little benefit for the baseline MIPS, except when long latency operations, such

as multiply and divide, are used. However, for a statically pipelined machine, software pipelining could be applied in order to schedule innermost loops more efficiently. Software pipelining, however, can also have a negative effect on code size.

In the statically pipelined architecture, we encode instructions in order to attain reasonable code size, however this does have a negative impact on performance. In order to compromise these conflicting requirements, we could allow both 32-bit and 64-bit instructions in different situations. Like the *Thumb2* instruction set that is a part of the ARM architecture [18], we would use the 64-bit instructions in inner-most loops, where performance is paramount, and the 32-bit instructions elsewhere to retain most of the code size benefits of the smaller instructions.

6.3 Static Pipelining for High Performance

The last area of future work is the possibility of applying the concept of static pipelining to higher performance processors. These machines have less of an emphasis on power consumption, so the decreased energy usage of static pipelining may not be quite as important. However, the limiting factor for increasing the clock rate for current high performance processors is the heat density of the chip. The simplified datapath of statically pipelined processor, along with reduced switching in registers may help to ameliorate this problem.

The design of a high performance, statically pipelined processor would likely include more internal registers, along with more functional units. This would mean that the instructions would have additional different types of effects, possibly leading to an issue with code size, though larger code sizes are generally less of an issue with general purpose processors than with embedded ones.

CHAPTER 7

CONCLUSIONS

Mobile computers such as smart phones and tablets are booming in popularity, and will likely supplant traditional desktop and laptop computers as the primary way in which most people use computers. Due to this change, the already important problem of decreasing power consumption in embedded processors has become vital. Additionally, more and more complex applications are being written for these platforms which require a greater degree of performance.

Instruction pipelining is a great technique for increasing the performance of microprocessors. As we have shown, however, traditional dynamic pipelining of instructions at run-time can be quite wasteful with regards to energy usage. This is primarily caused by unnecessary register file accesses, repeatedly calculating invariant values in loops along with the overhead of maintaining the pipeline state.

This dissertation has introduced an alternative way to implement instruction pipelining, statically at compile-time. We have designed a statically pipelined micro-architecture and instruction set. The instruction set architecture is different than most existing architectures in that it exposes the details of the micro-architecture to the compiler. We have also shown how these additional details can be efficiently encoded in 32-bits.

This dissertation next discussed the problem of compiling for the statically pipelined architecture. We showed how the compiler produces code by breaking traditional instructions into their individual pipeline stages. Due to the differences between the statically pipelined architecture and more traditional architectures, several code generation problems had to be addressed in this effort.

We then discussed how conventional compiler optimizations can be applied to statically pipelined code. Because the instruction set gives the compiler more access to the internal workings of the processor, these optimizations can achieve affects that are not possible for traditional architectures. We then discussed several optimizations targeting the statically pipelined architecture specifically including loop invariant code motion optimizations and instruction scheduling.

We then performed an experimental evaluation of static pipelining by compiling and simulating 15 MiBench benchmarks for the dynamically pipelined MIPS architecture and our statically pipelined architecture. We demonstrated that static pipelining can match and even exceed the performance of dynamic pipelining, significantly reduce the number of register file accesses, branch predictions and branch target calculations, and not

significantly increase code size. We then performed a conservative estimation of the energy savings of static pipelining that indicate static pipelining would reduce energy usage by 45%. These results are conservative given the limitations of our energy model and should improve with more complete evaluation.

Dynamically pipelined, RISC architectures were developed in the 1980s at a time when power consumption was not as important as it is today. The general-purpose embedded processors powering today's mobile devices are based on largely this same architecture. We have shown that dynamic pipelining has a number of inefficiencies that can be alleviated by moving the pipelining decisions from being done dynamically in hardware to statically in software. The biggest challenge in doing so is developing a compiler that can perform efficient code generation for such a target. This dissertation shows that a compiler can indeed manage lower level tasks like forwarding, branch prediction and register accesses more efficiently than hardware can. By re-examining the boundary between hardware and software, we can build more efficient processors for the mobile age.

REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[2] J.L. Ayala, M. Lopez-Vallejo, A. Veidenbaum, and C.A. Lopez. Energy aware register file implementation through instruction predecode. 2003.

[3] M.E. Benitez and J.W. Davidson. A Portable Global Optimizer and Linker. *ACM SIGPLAN Notices*, 23(7):329–338, 1988.

[4] R. Canal, A. González, and J.E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, page 190. ACM, 2000.

[5] AC Cheng and GS Tyson. An energy efficient instruction set synthesis framework for low power embedded system designs. *IEEE Transactions on Computers*, 54(6):698–713, 2005.

[6] H. Corporaal and M. Arnold. Using transport triggered architectures for embedded processor design. *Integrated Computer-Aided Engineering*, 5(1):19–38, 1998.

[7] J.W. Davidson and D.B. Whalley. Reducing the cost of branches by using registers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, page 191. ACM, 1990.

[8] J.A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150. ACM, 1983.

[9] C.W. Fraser. A retargetable compiler for ansi c. *ACM Sigplan Notices*, 26(10):29–43, 1991.

[10] L. Goudge and S. Segars. Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications. In *Compcon'96.'Technologies for the Information Superhighway'Digest of Papers*, pages 176–181. IEEE, 2002.

[11] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2002.

[12] S. Hines, J. Green, G. Tyson, and D. Whalley. Improving program efficiency by packing instructions into registers. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 260–271. IEEE Computer Society, 2005.

[13] M. Hiraki, R. Bajwa, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Sasaki, and K. Seki. Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer. In *Proceedings of the 1996 international symposium on Low power electronics and design*, pages 353–358. IEEE Press, 1996.

[14] Atul Kalambur and Mary Jane Irwin. An extended addressing mode for low power. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 208–213, New York, NY, USA, 1997. ACM.

[15] KD Kissell. MIPS16: High-density MIPS for the Embedded Market. In *Salon des solutions informatiques temps réel*, pages 559–571, 1997.

[16] J.G. Koomey. Growth in data center electricity use 2005 to 2010. *Retrieved October*, 9:2011, 2011.

[17] Chingren Lee, Jenq Kuen Lee, Tingting Hwang, and Shi-Chun Tsai. Compiler optimization on vliw instruction scheduling for low power. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):252–268, 2003.

[18] ARM Ltd. Arm thumb-2 core technology. *http://infocenter.arm.com /help/index.jsp?topic= /com.arm.doc.dui0471c /CHDFEDDB.html*, June 2012.

[19] B. Mathew, A. Davis, and M. Parker. A low power architecture for embedded perception. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 46–56. ACM, 2004.

[20] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146. ACM, 2006.

[21] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, page 181. ACM, 2006.

[22] D.A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[23] Mehrdad Reshadi, Bita Gorjiara, and Daniel Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 69–76, Washington, DC, USA, 2005. IEEE Computer Society.

[24] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, R. Zafalon, and D.E. e Informazione. Low-power data forwarding for VLIW embedded architectures. *IEEE transactions on very large scale integration (VLSI) systems*, 10(5):614–622, 2002.

[25] P.G. Sassone, D.S. Wills, and G.H. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 127–136. ACM, 2005.

[26] M.S. Schlansker and B.R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[27] J. Scott, L.H. Lee, J. Arends, and B. Moyer. Designing the Low-Power MâĂćCORE TM Architecture. In *Power Driven Microarchitecture Workshop*, pages 145–150. Citeseer, 1998.

[28] A.A.R. Sethi and J. Ullman. Compilers: Principles Techniques and Tools. *Addision Wesley Longman*, 2000.

[29] H. Shimada, H. Ando, and T. Shimada. Pipeline stage unification: a low-energy consumption technique for future mobile processors. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 326–329. ACM, 2003.

[30] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[31] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. Flexcore: Utilizing exposed datapath control for efficient computing. *Journal of Signal Processing Systems*, 57(1):5–19, 2009.

[32] J. H. Tseng and K. Asanovic. Energy-efficient register access. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 377, Washington, DC, USA, 2000. IEEE Computer Society.

[33] S. Woo, J. Yoon, and J. Kim. Low-power instruction encoding techniques. In *SOC Design Conference*. Citeseer, 2001.

[34] Chengmo Yang and Alex Orailoglu. Power efficient branch prediction through early identification of branch addresses. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 169–178, New York, NY, USA, 2006. ACM.

[35] V. Zyuban and P. Kogge. Split register file architectures for inherently lower power microprocessors. In *Proc. Power-Driven Microarchitecture Workshop*, pages 32–37.

BIOGRAPHICAL SKETCH

Ian Finlayson

Ian Finlayson was born October 26, 1984 in St. Petersburg, Florida. He attended Winthrop University, graduating in the Spring of 2003 with a Bachelor of Science Degree in Computer Science. He then attended Florida State University where he worked under the advisement of Professors David Whalley and Gary Tyson. Ian completed his Doctor of Philosophy degree in Computer Science in the Summer of 2012. He is currently employed by the University of Mary Washington in Fredericksburg, Virgina where he resides with his wife, Caitie and his son, Ryan. His main areas of interest are compilers, computer architecture and programming languages.