Exposing Datapath Elements to Reduce Microprocessor Energy Consumption

by

Mark Jerome Hampton

B.S., Computer and Systems Engineering (1999)
B.S., Computer Science (1999)
Rensselaer Polytechnic Institute

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2001

Certified by . . . .
Krste Asanović
Assistant Professor
Thesis Supervisor

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

Exposing Datapath Elements to Reduce Microprocessor Energy Consumption

by

Mark Jerome Hampton

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In this thesis, we evaluate the feasibility of exposing elements in a processor's datapath to the compiler in order to reduce energy consumption. We focus on eliminating register file traffic by exposing the latches in the bypass network, as our study shows that there is potential for significant benefit by doing this. We present the idea of software restart markers to handle the exception management overhead that results from exposing additional processor state. We then implement our proposed techniques and observe an average energy savings of 7.0% across a range of benchmarks when compared to a low-power MIPS processor. We also explore the implications of changing the pipeline structure to both improve performance and expose more machine state.

Thesis Supervisor: Krste Asanović
Title: Assistant Professor

Acknowledgments

First I want to thank my thesis supervisor, Krste Asanović, for all of his support, advice, and encouragement. I have learned a great deal from working with him and am extremely grateful for the opportunities he has given me.

I also want to thank all of the members of my research group for their support and encouragement. Thanks to my officemate, Jessica Tseng, who showed me how to use and modify the ISA simulator and has been a great friend. My other officemate, Michael Zhang, has also been a great friend, keeping me sane by occasionally joining in when I started singing in the office, discussing sports with me (I forgive him for being a Yankees fan), and laughing at my corny sense of humor. Ronny Krashinsky created the simulator that generated energy statistics for the various benchmarks. He also participated in numerous discussions on Yellow Pekoe and came up with some great ideas, such as using the kernel registers. And I must truly thank Emmett Witchel for coming down from his lofty Java perch and plumbing the depths of the GNU assembler. He completely restructured the instruction scheduler so it was to his liking (and it happened to be to my liking as well) and put in some new and exciting features, and the end result was some nicely written code (if you can call any C code nicely written).

All of my friends in the Black Graduate Students Association have provided me with an invaluable support network for which I will always be grateful. A special thank you to all the members of the Bid Whist crew—you know who you are!

My girlfriend, Emily, gave me support when I needed it most, encouraged me when I felt discouraged, and brought some spontaneity into my life. I can't begin to describe how much she has meant to me. Emily, thank you for everything.

Finally, I want to thank my family for all of their love and encouragement. In particular, I want to thank my parents for always being there for me. If it hadn't been for their advice, I probably would have skipped graduate school, and I would have missed out on one of the best experiences of my life. Mom and Dad, I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Microprocessor performance improves each year at an impressive rate. Technology advances allow for a greater number of transistors to fit onto a chip, enabling faster speeds and more features. Designers continually strive to increase instruction level parallelism (ILP) through methods such as out-of-order execution. The quest to increase ILP has led to a greater focus on levels of the system hierarchy higher than the circuit or microarchitectural levels—more attention is being devoted to architectural and compiler techniques to achieve better performance. This is evidenced by the recent trend of VLIW architectures, which give more control over performance to the compiler. However, in the constant push to make processors faster, the complexity of the hardware has rapidly increased. As a result, energy consumption has become an increasingly critical factor in microprocessor designs, particularly in embedded processors. Maintaining performance while reducing energy consumption is one of the major challenges faced by today's designers.

Many circuit techniques have been developed in efforts to consume less energy. However, there have not been many energy optimizations at higher levels of the system hierarchy, such as compilers and instruction set architectures. These areas have been explored in attempts to increase performance; they can also be approached from an energy perspective. There is an untapped potential to reduce energy consumption by exposing processor energy at these levels.

Current ISAs are designed to enhance performance, with little or no attention paid to the energy implications of design decisions. As a result, many ISA implementations cause several energy-consuming microarchitectural operations to occur when executing a simple instruction. As long as these operations do not affect the throughput or latency of the processor, there is little incentive from a performance perspective to expose them. However, from an energy perspective, exposing these microarchitectural operations to the compiler through the ISA can be very beneficial, as the compiler can make attempts to reduce energy consumption by removing unnecessary operations.

In this study, we examine the effects of exposing microprocessor energy consumption to the compiler. Our goal is to shift some of the hardware complexity in the datapath into software, and thus consume less total energy.

When considering both power and performance in a microprocessor design, simple architectures are preferred [20]. Thus, our research focuses on simple pipelined datapaths. In particular, we use a five-stage pipelined microprocessor that implements a subset of the MIPS II instruction set architecture as our baseline design. MIPS is one of the simpler RISC architectures, making it a good candidate for low-power research. We also use a combination of benchmarks that target both embedded and desktop systems. Our methodology is described in Chapter 2.

Chapter 3 of this paper discusses the motivation for exposing datapath elements to the compiler. We select the register file and the associated bypass network as the focus of our study. In today's microprocessors, register file accesses are typically responsible for a large percentage of the datapath power consumption. In Motorola's M•CORE processor, the register file consumes 16% of the total processor power and 42% of the total datapath power [22]. One way to reduce register file power consumption is to eliminate unnecessary reads and writes. Previous work has shown that many register values are only used once [4], indicating that they may be unnecessarily written back to the register file. Additionally, many values are provided by the bypass network [25], meaning that the corresponding register file reads are unnecessary. We conduct our own register lifetime analysis study for our particular workload to determine a theoretical upper bound on the benefit that we can expect to achieve by trying to eliminate unnecessary register file traffic using compile-time information.

Exposing a microprocessor's organization to the compiler is not a new concept, as VLIW machines, vector processors, and reconfigurable computers all utilize this idea. These techniques add more state that is visible to software, and give the compiler knowledge of the various aspects of a processor's hardware design so that greater performance can be achieved. Traditionally it has been difficult for these types of processors to handle exceptions because of the overhead required to manage the additional exposed machine state. As a result, these systems have not been able to implement features such as virtual memory. This has begun to change in recent years for vector processors [3] and VLIW machines [21], as hardware-based methods have been developed to handle this problem. However, our goal in this study is to shift as much complexity from hardware to software as possible in order to reduce energy consumption. Thus, Chapter 4 presents a software-based solution to the exception management overhead problem in the form of software restart markers, which allow the use of temporary state. This technique provides a simple way to expose machine state without having to introduce additional hardware to preserve the state across exceptions.

Chapter 5 of this paper explores the energy savings that can be achieved in practice. We expose the bypass latches to the compiler and assembler and make various modifications to them in order to save energy. We present results for the percentage of register file accesses that can be eliminated, and also show the energy that is saved by our optimizations.

Additional benefit can be gained by designing the processor's microarchitecture with energy in mind. By splitting the pipeline into two sections, one to handle address computations and one to handle data computations, we create more temporary state that can be exposed and also allow for greater correlation between address and data values, which can theoretically reduce switching energy. This split-pipeline organization can also improve performance [5]. We explore this idea in Chapter 6.

Chapter 7 concludes and discusses future work.

Chapter 2

Methodology

The complexity of microprocessors increases each year as designers attempt to achieve higher levels of performance with techniques such as superscalar datapaths, instruction reordering, and speculative execution. However, the performance gains of these methods typically result in an increase in power consumption. As shown in [20], simple pipelined datapaths are preferred when considering both power and performance in a microprocessor design. Thus, we use a five-stage pipelined MIPS II-compatible microprocessor, code named Vanilla Pekoe, as our baseline design, and we stick with relatively simple pipelines throughout our investigation. The Vanilla Pekoe pipeline is shown in Figure 2-1. It implements a subset of the MIPS II instruction set architecture, not including misaligned load/store instructions, multiprocessor synchronization instructions, or trap instructions. There is a single architected branch delay slot, and all branches are predicted taken with a one cycle branch mispredict penalty. The microprocessor has a single interlocked load-use delay slot, and two interlocked load-use delay slots for 8-bit and 16-bit loads. Integer multiply and divide are non-pipelined and have latencies of 18 cycles and 33 cycles, respectively. Vanilla Pekoe does not have a floating-point unit, and thus all floating-point instructions are trapped and emulated in software.

We gather statistics by obtaining dynamic benchmark instruction traces using a cycle-accurate simulator that models the behavior of Vanilla Pekoe as well as our subsequent pipeline designs. A similar method was used in the development of the

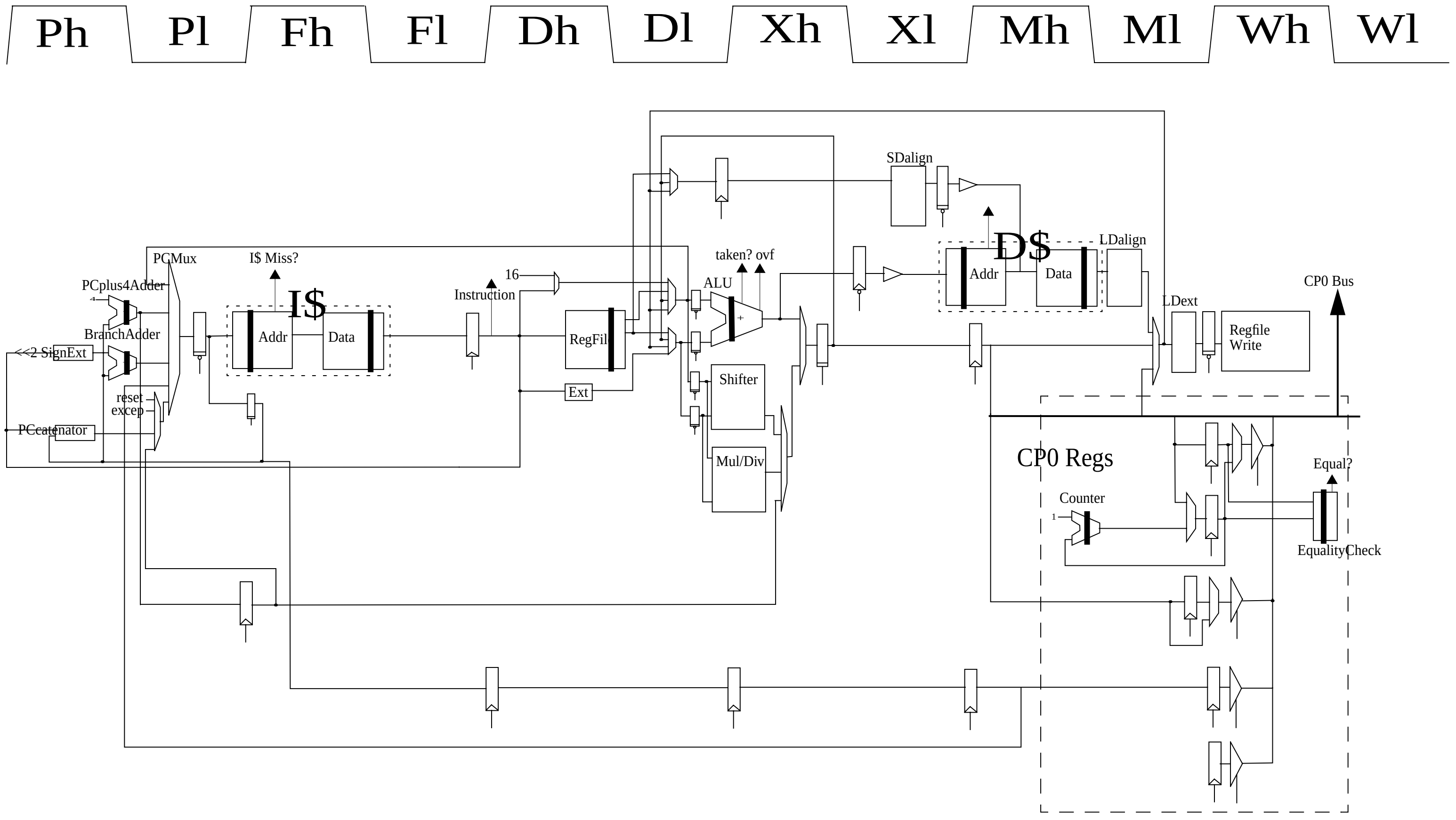


Figure 2-1: Vanilla Pekoe pipeline diagram

M•CORE architecture [6]. Energy statistics are obtained using our cycle-accurate simulator, SyCHOsys [15]. We obtain additional statistics on program behavior by using an ISA simulator that was initially developed for T0 [1], a vector microprocessor with a MIPS core similar to Vanilla Pekoe. We do not include the effects of caches in our simulations as our techniques do not affect cache performance and thus adding stalls due to cache misses would only lessen any performance penalty introduced by our changes.

It is important to use a workload that is representative of programs that will actually be run on our target microprocessors. Thus, we use a combination of embedded and desktop applications from the MediaBench [16] and SPECint2000 [7] benchmark suites. We run all programs to completion. For the SPECint2000 benchmarks, we use small test inputs to keep the simulation time reasonable. A description of the benchmarks used is in Table 2.1, along with user-level instruction counts and cycle counts for Vanilla Pekoe. Table 2.2 gives the distribution of instruction types in each benchmark.

Our C compiler is `egcs-1.0.3a`, a version of GNU `gcc` which generates assembly code that is processed by version 2.8.1 of GNU `gas`. These are widely-used programs which have their source codes freely available, giving us the flexibility to make modifications to achieve our goals. We set up a cross-compiler environment in which we run these tools on both Solaris and Linux workstations, and generate binaries that can be run on our target processors. We statically link the program code with version 1.8.2 of the `newlib` standard C library.

Benchmark {Data Set}	User-Level Instruction Count	User-Level Cycle Count	Description
adpcm(decode) {clinton.adpcm}	5,843,735	6,802,729	An adaptive differential pulse code modulation decoder
adpcm(encode) {clinton.pcm}	6,967,125	8,498,659	An adaptive differential pulse code modulation encoder
bzip2 {test}	9,871,943,186	11,491,425,114	A data compression program
g721(decode) {clinton.g721}	290,462,290	381,244,054	A voice decompression program
g721(encode) {clinton.pcm}	298,023,582	392,065,079	A voice compression program
gcc {test}	1,893,715,242	2,258,779,011	A C Language compiler
gsm(decode) {clinton.pcm.gsm}	90,150,531	140,955,378	A full-rate speech transcoder
gsm(encode) {clinton.pcm}	229,655,663	531,567,516	A full-rate speech transcoder
gzip {test}	3,087,502,624	3,535,713,029	A data compression program
jpeg(decode) {testimg.jpg}	5,084,336	6,552,104	An image decompression program
jpeg(encode) {testimg.ppm}	16,261,685	20,485,752	An image compression program
mcf {test}	216,662,213	252,092,168	A single-depot vehicle scheduling program
parser {test}	3,925,961,662	4,558,530,147	A syntactic parser of English
pegwit(decode) {pegwit.enc, my.sec}	19,237,898	22,979,612	A program that performs public key decryption and authentication
pegwit(encode) {my.pub, pgptest.plain, encryption_junk}	33,855,094	40,913,225	A program that performs public key encryption and authentication
vortex {test}	10,235,479,193	11,796,918,443	A single-user object-oriented database transaction program
Total	30,226,806,059	35,445,522,020	

Table 2.1: Vanilla Pekoe benchmark instruction counts, cycle counts, and descriptions

Benchmark	Arithmetic	Logical	Load	Store	Shift	Lui
adpcm(decode)	29.18	20.20	6.33	2.54	18.94	0.01
adpcm(encode)	37.75	13.43	6.37	1.07	16.95	0.01
bzip2	45.03	6.77	18.47	13.85	2.60	1.47
g721(decode)	39.05	2.51	12.60	4.51	12.93	1.21
g721(encode)	39.54	2.31	12.21	4.10	13.32	1.28
gcc	32.99	3.49	21.91	12.52	4.66	3.38
gsm(decode)	42.03	8.51	6.59	3.66	19.45	0.01
gsm(encode)	40.50	0.54	16.45	3.76	19.69	0.08
gzip	30.99	6.30	19.52	8.52	5.87	9.51
jpeg(decode)	47.27	3.18	17.15	8.96	15.23	0.36
jpeg(encode)	39.81	0.94	20.46	6.50	16.03	0.78
mcf	43.47	0.41	18.37	15.17	2.45	0.87
parser	29.40	4.27	22.66	9.05	3.93	5.49
pegwit(decode)	32.27	15.40	18.44	5.92	12.12	4.44
pegwit(encode)	33.62	15.86	17.58	5.82	12.17	2.86
vortex	28.25	0.87	26.63	18.60	1.97	5.54
Average	36.95	6.56	16.36	7.78	11.14	2.33
Benchmark	Cond. Branch	Uncond. Branch	Mul/ Div	Nop	Other	
adpcm(decode)	22.74	0.03	0.00	0.02	0.01	
adpcm(encode)	21.19	1.08	0.00	2.13	0.00	
bzip2	10.28	0.93	0.00	0.61	0.00	
g721(decode)	17.01	3.01	1.00	6.16	0.00	
g721(encode)	17.22	2.96	0.97	6.10	0.00	
gcc	13.25	3.54	0.24	3.99	0.04	
gsm(decode)	11.05	2.92	5.67	0.09	0.00	
gsm(encode)	4.42	0.29	13.99	0.28	0.00	
gzip	11.75	3.88	0.02	3.66	0.00	
jpeg(decode)	5.48	0.64	1.25	0.48	0.00	
jpeg(encode)	13.01	1.81	0.36	0.31	0.00	
mcf	15.97	0.85	0.04	2.40	0.00	
parser	14.14	3.61	0.20	7.24	0.00	
pegwit(decode)	8.78	1.82	0.00	0.81	0.00	
pegwit(encode)	9.25	1.95	0.00	0.88	0.00	
vortex	10.37	3.89	0.13	3.76	0.00	
Average	12.87	2.08	1.59	2.43	0.00	

Table 2.2: Vanilla Pekoe benchmark instruction type distribution as a percentage

Chapter 3

Exposing Datapath Elements

3.1 Motivation

Energy consumption is becoming increasingly important in today's microprocessors. In the quest to improve performance, microprocessor designers have devised numerous methods to increase instruction level parallelism (ILP), such as superscalar processors, out-of-order execution, and VLIW architectures. However, with each new technique that results in higher performance, the subsequent complexity of the microprocessor typically increases. As a result, the microprocessor's energy consumption increases as well. We want to reduce this energy consumption by exposing datapath elements to the compiler and thus shifting some of the hardware complexity into software complexity.

3.2 Choosing Datapath Elements to Expose

We must decide which elements in the datapath should be exposed to the compiler. We want to select hardware components that are easily made software-visible and that are responsible for a significant percentage of the microprocessor's energy consumption. If one or both of these criteria are not met, then it is unlikely that the energy savings we can achieve by exposing the selected elements will offset the additional software (and possibly hardware) complexity required.

One datapath component that is already visible to the compiler is the register file. In today's microprocessors, most operations get their source values from the register file and write any destination values to the register file. The compiler is responsible for determining which architectural registers are used. However, the compiler does not have access to all of the components related to the register file, namely the bypass network. It should be relatively simple to expose the latches within the bypass network, as they are essentially extensions of the register file. Thus, the register file and its associated components satisfy the first criterion mentioned above.

Additionally, register file accesses typically consume a significant percentage of the processor datapath power consumption. For example, in Motorola's M•CORE processor, the register file consumes 16% of the total processor power and 42% of the total datapath power [22]. This means that the second criterion is satisfied as well. We therefore choose to focus on register file accesses in our study. If we can demonstrate that our approach is a feasible method of reducing microprocessor energy consumption, then we can extend it to other elements of the datapath.

3.3 Register Lifetime Analysis

It has been shown previously that a large fraction of register values have a short lifetime, frequently being used only once [4]. For example, consider the following code sequence which is used to increment a memory variable:

```
lw r1, (r3)             # Load value.
add r1, r1, 1           # Increment.
sw r1, (r3)             # Update memory.
```

The values generated by the load and add instructions are each only used once by the subsequent instruction and are normally read from the bypass network rather than the register file. A conventional processor does not have access to this lifetime information and thus must always write values back to the register file, as it must assume that a value can be used at any point in the future or that an interrupt can occur at any time. By communicating lifetime information to the processor, we can enable optimizations that will reduce register file traffic. We intend to do this by exposing latches in the bypass network to the compiler and explicitly targeting them when a value has a short lifetime. When the processor sees an explicit use of a bypass latch, it can disable the corresponding register file access. To determine an upper bound on the benefit that we can achieve in our workload, we conducted a register lifetime analysis study by running programs through our ISA simulator.

3.3.1 Terminology

Instruction Count

All of our statistics are based on user-level instructions; kernel code is ignored. We define the "instruction count" at a particular point in a program to be the number of user-level instructions executed up to that point, and measure lifetime lengths and other corresponding statistics in terms of instruction counts.

Register Value Lifetime

The lifetime of a register value consists of the time between the writing of that value and the last read of that value. We define a value's lifetime length using the following formula:

lifetime length = (instruction count when the last read of a register value occurs) −
(instruction count when the value was written into the register)

In other words, the lifetime indicates the span of instructions over which a particular register value is active. To illustrate this concept, consider the following code segment, in which the instruction count is listed to the left of the instruction.

```
100 add r1, r2, r3            # r1 = r2 + r3
101 and r6, r1, r7            # r6 = r1 & r7
102 add r7, r2, r4            # r7 = r2 + r4
103 sub r5, r1, r4            # r5 = r1 - r4
```

```
104 add r8, r8, 1              # r8 = r8 + 1
105 sub r1, r7, r8             # r1 = r7 - r8
```

Register `r1` is written with a new value when the instruction count is 100. It is then read in two of the next three instructions, and the last read of the value occurs when the instruction count is 103. A new value is then written to `r1` when the instruction count is 105. Thus, the lifetime length of the value in `r1` for this particular code segment is:

lifetime length = 103 − 100 = 3

It is possible for a register value to have a zero-length lifetime if the value is written and never read. This can occur if the value is written in one basic block and only used in another basic block. A conditional branch at the end of the first block can cause the program to never enter the second block, and thus the value will never be read. A high-level example of this is shown by the following C code segment, which tries to find the first non-zero bit in an integer value stored in memory.

```
temp = a[0];
for (i = 0; i < 8*sizeof(int); i++) {
  if (temp & 1 == 1)
    break;
  temp = temp >> 1;
}
```

If the value stored in `a[0]` is 0, then during the last iteration of the loop the value in `temp` will be shifted to the right by 1 but will never be read, as the loop will end. This illustrates how a value can have a lifetime length of 0.

Bypass Latches

The bypass latches are used to store values that either come from the register file or are forwarded through the bypass network from the result of a previous computation.

Values that are to be written to the register file are also stored in these latches. Figure 3-1 shows the part of the Vanilla Pekoe pipeline that contains the bypass network. We intend to target the RS, RT, SD, and X latches in our study.

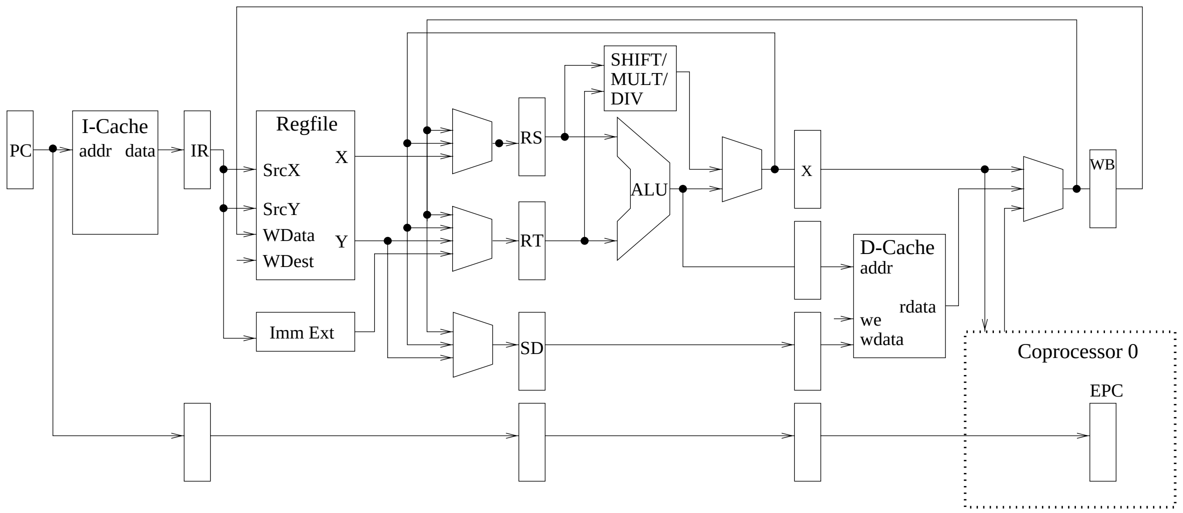


Figure 3-1: Vanilla Pekoe bypass network

3.3.2 Results

Register Lifetime Length Distribution

We gathered statistics on the distribution of lifetime lengths of register values in the programs in our workload. Figure 3-2 shows a distribution of the lifetime lengths for selected benchmarks. As can be seen from the figure, most values have a short lifetime. We do not base our analysis on statistics for average lifetime length because a few values have extremely long lifetime lengths, and this distorts the average. For example, MIPS convention categorizes register 28 as the global pointer (gp) register. This register points to the middle of the region that contains any global or static data in the program. The gp register is loaded at the beginning of the program with the correct address, and then it is not written again while the program executes. Since the gp register is typically referenced throughout the program, the value it holds is live. Thus, the value's lifetime length is on the same order of magnitude as the number of instructions in the program, as shown in Figure 3-3, which displays the average lifetime length for each register in the gcc benchmark. Across all registers, the global average lifetime length in gcc is about 15, in spite of the fact that the majority of register values in the benchmark have a lifetime length of less than 15, as shown in Figure 3-2. This illustrates that the average lifetime length does not provide much insight into the behavior of the program. Note that accesses of register 0 are not included in any of our statistics because it always has a constant value of 0. Also, MIPS convention categorizes registers 26 and 27 as kernel registers, only to be used when executing kernel-level code. Since we only take statistics for user-level code, those registers have average lifetime lengths of 0 in Figure 3-3.

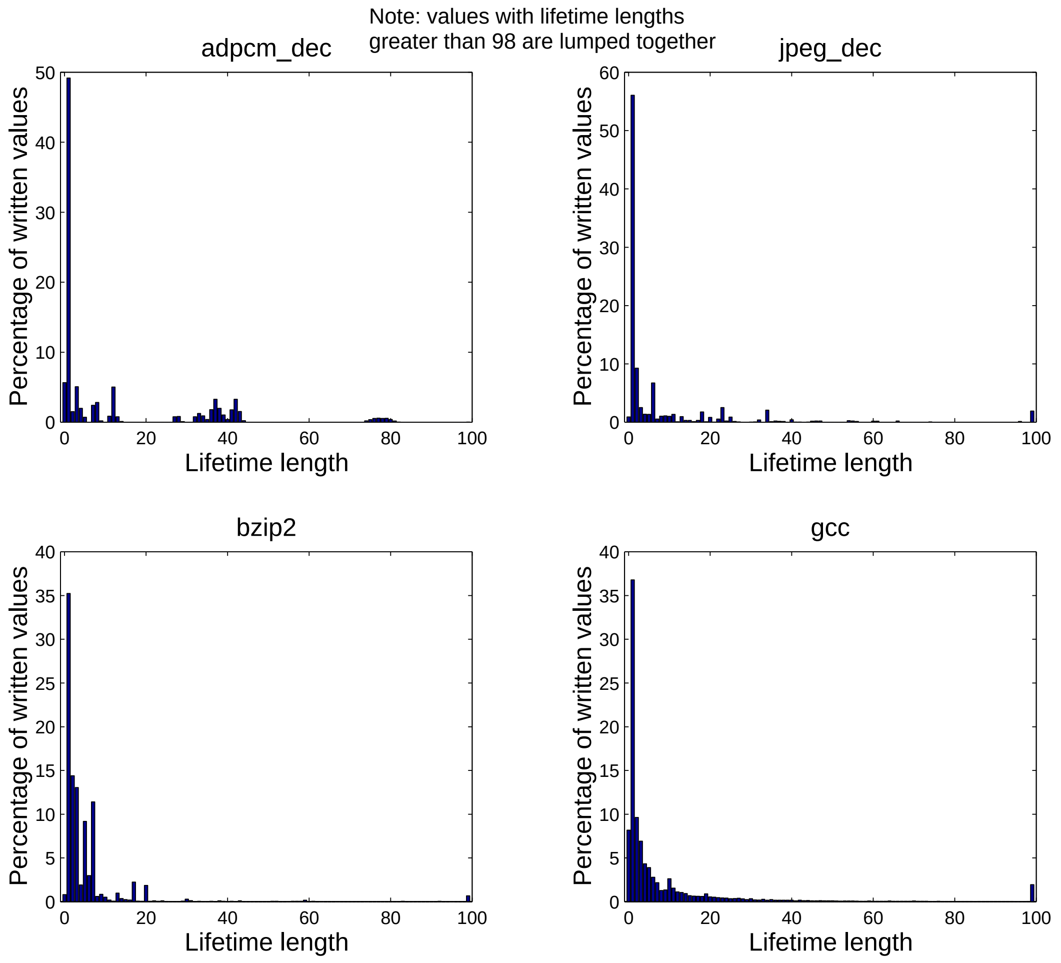


Figure 3-2: Register value lifetime length distribution for selected benchmarks

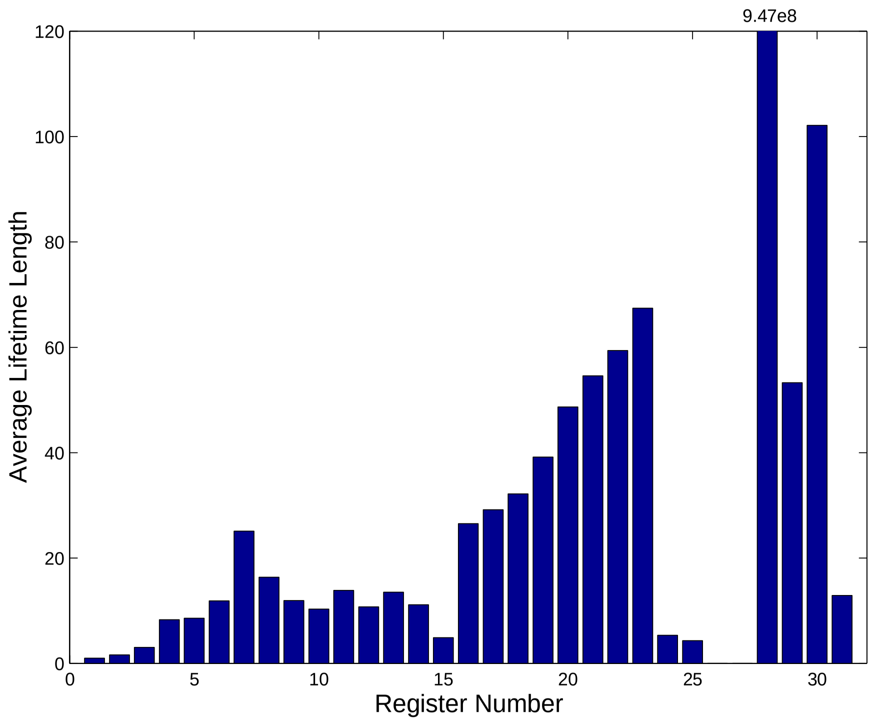


Figure 3-3: Average lifetime length of a register value in gcc benchmark

Register File Writes That Can Be Avoided

Within our five-stage pipeline, a value that has a lifetime length of 2 or less probably does not need to be written back to the register file. This is because in most cases all of the reads of the value will occur before the value reaches the Write Back stage of the pipeline, and thus it will no longer be live at this point. There are some cases when this will not be true; for example, in the Vanilla Pekoe pipeline, the `lhu` instruction has two interlocked load-use delay slots. So, in the following code segment:

```
lhu r1, 0(r2)
add r3, r1, r4
sub r5, r1, r6
```

The value to be loaded into `r1` will be delayed for two cycles, and thus the `add` instruction will be stalled in the decode stage of the pipeline, while the `sub` instruction will be stalled in the instruction fetch stage of the pipeline. However, these delays can frequently be avoided, either by having the compiler schedule instructions to avoid interlocks, or by changing the microarchitecture to remove these delay slots completely. Figure 3-4 shows the percentage of values that have lifetime lengths of 2 or less.

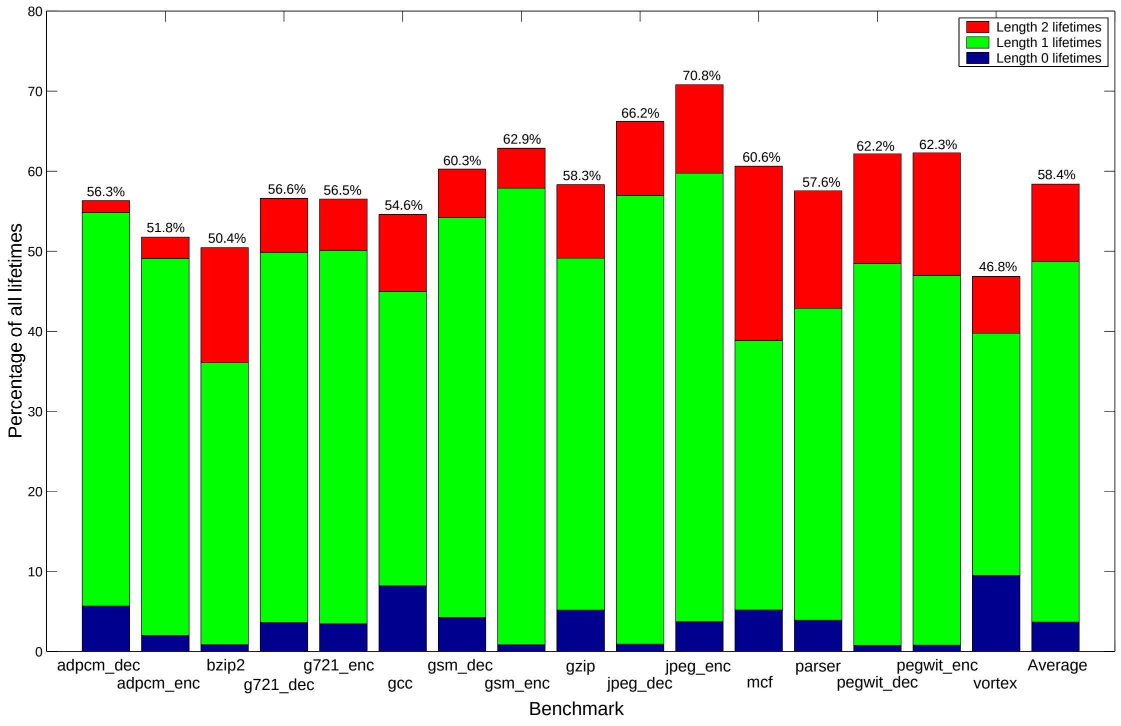


Figure 3-4: Percentage of values that have lifetime lengths of 2 or less

Dynamic Read Bypassing

When an instruction needs a value that has not yet been written back to the register file, it is forwarded by the bypass network. Control logic is used to detect when a value should be forwarded. This improves performance, but also increases the datapath's complexity. In addition, in many processors, the register file will still be read even though the value it provides will not be used. This can be avoided through a technique called *bypass skip* [25], but this requires extra control logic—not only does this add hardware complexity, but if this logic is inserted into the processor's critical path, the cycle time can increase. Ideally, we would like to eliminate these unnecessary reads and shift the complexity of the bypass network into software, but first we must determine an upper bound on how much benefit we can expect to gain. Figure 3-5 shows the percentage of read values that are provided from the bypass network instead of the register file.

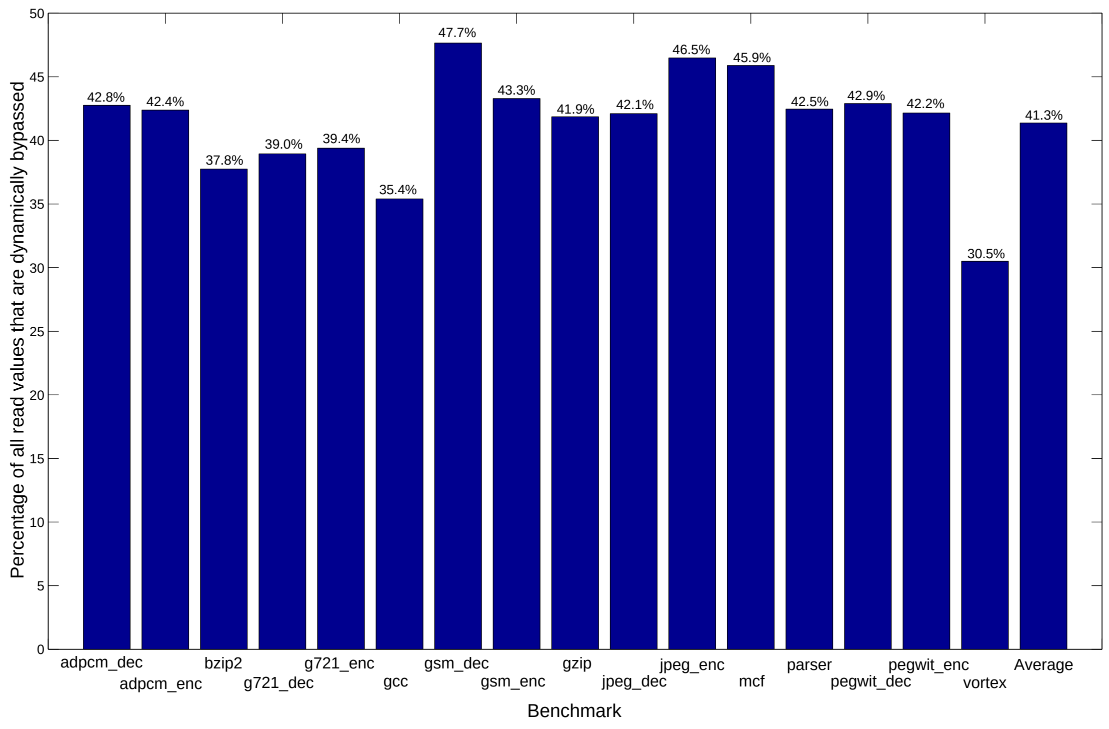


Figure 3-5: Percentage of read values that are provided from the bypass network

Dynamic Read Caching

Another technique that can eliminate register file reads is *read caching* [25]. For example, when a procedure is called, it may save registers on the stack before using them, as shown in the following code segment:

```
sw r3, 8(sp)
sw r2, 4(sp)
sw r1, 0(sp)
```

The value in the stack pointer register does not change in the above sequence, yet it is read from the register file and clocked into the RS bypass latch for each instruction. As with dynamic read bypassing, avoiding the read of the register file requires additional hardware complexity. Figure 3-6 shows the percentage of reads that can be dynamically cached.

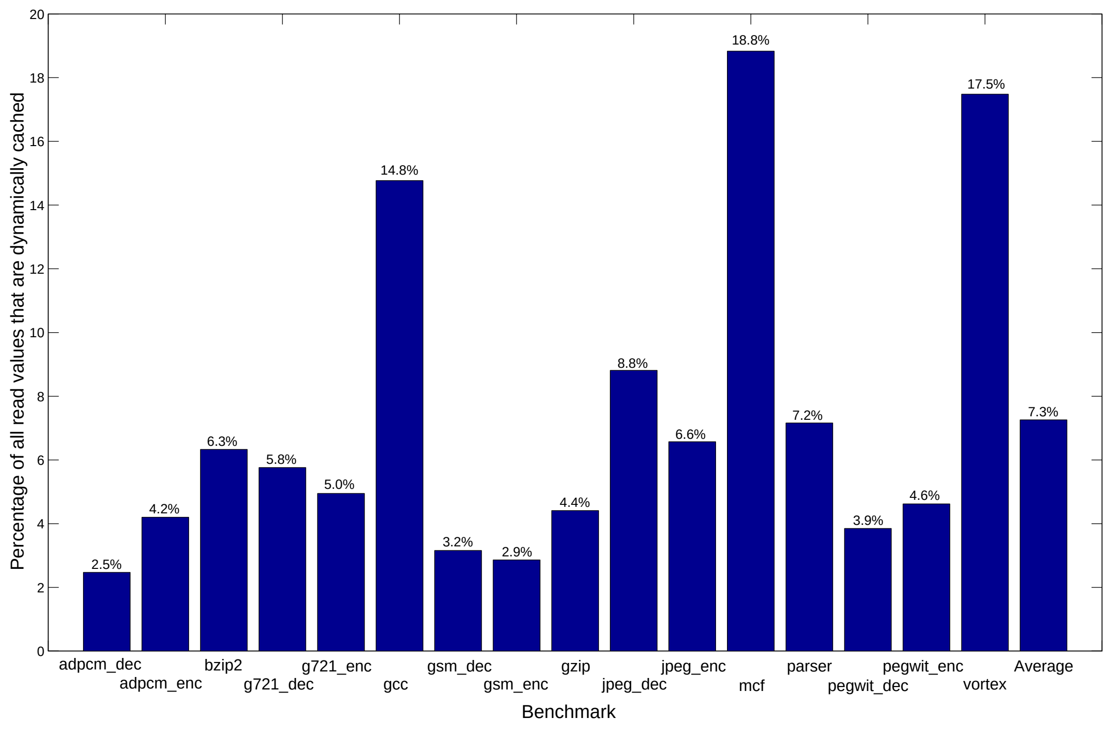


Figure 3-6: Percentage of read values that can be cached

Summary

The register lifetime analysis indicates that there are multiple opportunities to reduce register file traffic. By exposing the bypass latches to the compiler, the potential exists to achieve a significant benefit.

3.4 Related Work

The idea of exposing machine organization to software and thus trading hardware complexity for software complexity is not new. VLIW microprocessors are built around this concept, as the compiler is responsible for extracting ILP from programs. Vector machines and reconfigurable computers are other examples of systems that shift some of the complexity into software. But traditionally, these types of processors have had difficulties handling exceptions. To successfully expose datapath elements and also reduce energy consumption in a way that is applicable to today's microprocessors, this problem must be dealt with. We present a solution in the next chapter.

An interesting extension of the VLIW paradigm can be found in *transport-triggered architectures* (TTAs) [2]. These architectures attempt to reduce the complexity of high-performance VLIW machines by making data transports visible to software. In this type of architecture, an instruction specifies a data move to a functional unit, which can implicitly trigger an operation. TTAs implement the register file optimizations discussed in this chapter: elimination of unnecessary writes, software bypassing, and read caching (which is referred to as operand sharing). We differ from the work on TTAs in three ways. First, we intend to show the feasibility of these techniques for traditional *operation-triggered architectures*, which still dominate the microprocessor market. Second, we draw comparisons between the dynamic and static versions of these techniques where possible. Third, we present a method for handling the exception management overhead that comes from exposing additional machine state.

There has also been work specifically aimed at reducing register file accesses by exploiting register values with short lifetimes. Yung and Wilhelm [27] present a scheme for replacing the register file with a smaller register cache, with the idea of taking advantage of the frequent use of the bypass network. However, this method is designed to improve performance in a processor with a large number of registers, not to reduce energy, and thus it does not consider the possibility of avoiding writes to the register cache. Martin et al. [19] propose compiler support to take advantage of dead register values, but their scheme requires additional instructions to be inserted into the program executable. Lozano and Gao [17] examine the feasibility of exposing hardware to the compiler to eliminate register file commits of short-lived variables, and propose a compiler register allocation method for these variables. However, their proposed schemes for committing an instruction either require additional hardware complexity in a superscalar processor's reorder buffer or impose the restriction that a minimum number of instructions must be kept in the buffer. In addition, they focus on how to improve performance through their register allocation scheme, not on how to reduce energy consumption. Hu and Martonosi [9] explore the reduction in register file power consumption made possible by values with short lifetimes, but their method requires more hardware complexity in the datapath.

We propose to both reduce register file accesses and hardware complexity simultaneously, and thus lower energy consumption. But before that is possible, the issue of exception management overhead must be addressed.

Chapter 4

Software Restart Markers

4.1 Motivation

Our register lifetime analysis in Chapter 3 indicates the potential for significant energy savings if we can reduce register file accesses by exposing the bypass latches in the datapath. We can use the lifetime information stored by the compiler to explicitly target latches instead of the register file when processing values with short lifetimes. However, exposing more machine state to the compiler introduces the problem of exception management overhead. Many ISA implementations are required to provide precise exceptions, which means that all earlier instructions must have committed state updates, no later instructions have affected architectural state, and the saved program counter points to the faulting instruction [24]. Each time an exception occurs, the process state must be saved before handling the exception. Exposing the bypass latches—and more generally, any datapath elements—means that they must be preserved across exceptions by saving and restoring them. Not only does this impact performance, it can also negate much of the energy savings we hope to achieve, as the circuitry required to support saving and restoring the additional values during each exception will cost energy on each instruction, not just during exception events.

We would also like our techniques to be applicable to various types of microprocessors, not just simple pipelines. However, the problem becomes worse when considering more advanced processors such as out-of-order machines, as the hardware required to support precise exceptions can become quite complex. Various techniques such as reorder buffers, history buffers, future files, shadow registers, and checkpointing hardware have been devised to implement precise exceptions [26], but they all entail additional hardware overhead, which is magnified by the fact that we intend to expose much of the processor state.

One way to handle the problem is to avoid supporting precise exceptions. This is the approach traditionally used for vector processors, reconfigurable computers, and classical VLIW machines. However, the problem with this approach is that certain features expected on today's microprocessors can not be implemented in the absence of precise exception support. For example, debugging and IEEE floating-point arithmetic support require precise exceptions. Even features that only require restartability and not fully precise exceptions, such as demand-paged virtual memory, have not been implemented on traditional state-exposed processors. This has begun to change in recent years, as techniques have been developed to handle precise exceptions in systems such as vector processors [3] and VLIW machines [21]. However, these techniques do entail some additional hardware complexity. We propose to handle the exception management problem entirely in software.

4.2 Software Exception Management

In the precise exception model, each instruction is implicitly marked as a trap barrier; if an instruction causes an exception, the process will restart from that instruction. This constraint can be relaxed by allowing software to explicitly mark points in the instruction stream where restart is required. The last instruction in a restart region will be marked as a barrier instruction—it acts as a trap barrier that will commit and update the machine state only if it does not raise an exception and no preceding instructions raise an exception. Also, if an exception occurs before the barrier instruction commits, no state updates or exceptions from subsequent instructions will be visible. When the barrier instruction commits, it updates a kernel visible register, the restart program counter (PC), to point to the next instruction to be executed.

The code in the restart region must be such that the kernel can restart a process after an exception by simply jumping to the restart PC. This imposes the requirement that each region must contain idempotent code (except for the barrier instruction)—i.e. the code can be re-executed multiple times without changing the result. This restriction still allows for very large restart regions. For example, several functions in the standard C library can each be entirely contained within one restart region. The prototypes for some of those functions are given below.

```
int sprintf(char *s, const char *format, ...);
int sscanf(char *s, const char *format, ...);
char *strcpy(char *s1, const char *s2);
```

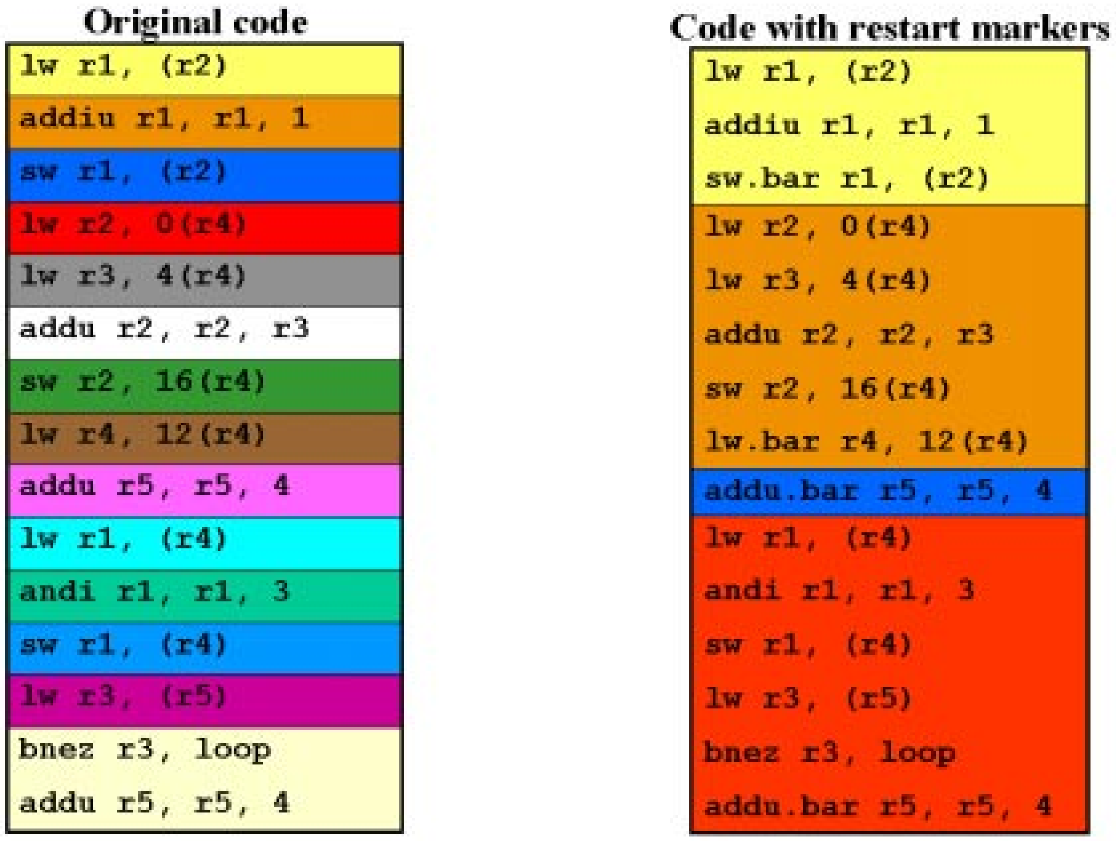
To illustrate how a function can be placed into a single restart region, consider the `sprintf()` function. This function uses the `format` string as an input argument to write to the string pointed to by `s`. The number of bytes written into `s` is returned. The `sprintf()` prototype shows that the data pointed to by `format` is `const` and thus is not modified by the function. As long as the input arguments to the function are passed in stack memory and are not altered by the routine (meaning that the memory space containing the input arguments can not overlap the memory space for the output arguments), the function can be restarted multiple times and still produce the same result. This is true of any function that does not modify its arguments: it can use an arbitrary amount of local read/write workspace and still be idempotent.

Figure 4-1 shows an example of restart regions. On the left is the original code with the exception model implemented on our baseline MIPS processor—each instruction is within a separate restart region. Note that the MIPS instruction set [12] already incorporates a limited form of expanded restart regions in the form of its architected branch delay slot. Branches are never marked as barriers so that if the instruction in the delay slot causes an exception, the process will restart at the branch itself to recreate the next PC, which is not saved by the kernel. On the right of

Figure 4-1 is the same code divided into larger restart regions. If an exception occurs within a region, the kernel will restart the process from the beginning of the region. In the first region, the store is marked as a barrier because it updates the memory location read by the load in a non-idempotent manner. The second region contains a store that is not marked as a barrier, but the region is still idempotent because the store writes to a different memory address than those accessed previously by the load instructions. The third region contains a single instruction and shows that the compiler can revert to marking each instruction as a barrier as in the baseline model, and thus can implement precise exceptions in this manner.

A sufficient, but not necessary, condition for idempotency is that the set of all external sources (registers and memory) read by the region is disjoint from the set of destinations written by the region (however, a value that is produced within the region can be overwritten). This is not a necessary condition as shown by the last region in Figure 4-1. The store to memory changes input source data, but the operation is still idempotent—the bottom two bits are masked out. Note that the above condition requires one of two methods to handle exception-causing instructions such as breakpoints or system calls. Either these instructions must be marked as trap barriers or there must be a guarantee that the exception handler will preserve the idempotency of the restart region. Because the latter approach is extremely difficult, we adopt the former method of imposing the constraint that all exception-causing instructions are trap barriers.

The use of software restart markers introduces the notion of three different types of machine state: checkpointed, stable, and temporary. Checkpointed state is copied into checkpoint registers when a barrier instruction commits, and is recoverable if an exception occurs. In our scheme, the only checkpointed state is the PC. Stable state is preserved across an exception by the kernel: registers and memory fall into this category. Finally, temporary state is only valid within a restart region, and is not preserved across an exception. Exposed bypass latches are an example of temporary state. This shows the power of software restart markers: we can expose a great deal of machine state and not impose additional exception management overhead in hardware if the state is temporary. Instead, the complexity is shifted into software.



```
lw r1, (r2)
addiu r1, r1, 1
sw r1, (r2)
lw r2, 0(r4)
lw r3, 4(r4)
addu r2, r2, r3
sw r2, 16(r4)
lw r4, 12(r4)
addu r5, r5, 4
lw r1, (r4)
andi r1, r1, 3
sw r1, (r4)
lw r3, (r5)
bnez r3, loop
addu r5, r5, 4
```

```
lw r1, (r2)
addiu r1, r1, 1
sw.bar r1, (r2)
lw r2, 0(r4)
lw r3, 4(r4)
addu r2, r2, r3
sw r2, 16(r4)
lw.bar r4, 12(r4)
addu.bar r5, r5, 4
lw r1, (r4)
andi r1, r1, 3
sw r1, (r4)
lw r3, (r5)
bnez r3, loop
addu.bar r5, r5, 4
```

Figure 4-1: Restart region example

4.3 Related Work

The Transmeta Crusoe processor has software-controlled exception barriers at the borders of x86 code that have been translated into the native VLIW format [13]. It maintains shadow registers and a speculative store buffer so that if an exception occurs, a rollback operation undoes any state updates. When a barrier is reached and no exceptions have occurred, state updates are committed to the architectural state. Our scheme differs from theirs in that it does not require any state buffers, because it allows irrevocable state changes in the middle of a restart region. Also, our scheme allows for the inclusion of temporary state.

The Alpha floating-point architecture has imprecise floating-point traps that require the user to insert trap barrier instructions to delimit safe regions to allow code to resume after the trap [23], but this scheme does not consider other types of exceptions or allow irrevocable state updates.

The Inmos transputer family has a three-register temporary evaluation stack that is not saved on hardware context switches [10]. However, the transputer does not provide demand-paged virtual memory and delays context switches until certain software-defined descheduling points, whereas our scheme allows interrupts at any point.

There are some similarities with our idempotent region analysis and sentinel scheduling for a speculative processor [18]. The instruction sequence between a speculative instruction and its sentinel, which flags an exception, must be idempotent. To ensure idempotency, sentinel scheduling imposes the constraint that instructions such as system calls must end an idempotent sequence. Additionally, the constraint is imposed that an instruction's inputs are never overwritten, either by itself or by a subsequent instruction in the same sequence. However, as we showed earlier, this latter constraint is not necessary. Also, the sentinel scheduling technique still requires significant hardware complexity and does not allow for the exposing of temporary state.

4.4 Software Restart Marker Implementation

We made modifications to our assembler to add support for software restart markers, so that we could subsequently enable the bypass latch optimizations. The restart region analysis operates at a basic block level. For our initial implementation, we adopt the approach proposed earlier of marking all exception-causing instructions as trap barriers and keeping a read set and write set for all of the registers, with a single bit being used in each set to represent memory. If an instruction causes a conflict between the two sets, that instruction is marked as a trap barrier. Since we operate at the basic block level, all branch delay slot instructions are marked as trap barriers as well. We make a slight extension to this model to incorporate a limited form of memory analysis. Although we represent memory with a single bit, we also separately track the base register and offset for each memory instruction. When the first memory instruction in a restart region is encountered, we store the base register, and we create a linked list to hold the offset. For each subsequent memory instruction, if the same base register is used, we can look at the offset to determine whether the access is to a distinct memory location from all previous memory operations. If the base register is modified, or a different base register is used, we revert to treating all of memory as a single location.

Note that just because the read and write sets overlap does not mean that a conflict has occurred. For example, if a register is written and then the value is read, the read and write sets overlap—however, a read is harmless and thus there is no conflict. Also, we only mark conflicts as occurring between external sources read and destinations written within the region. This means that if a value is written into a register and subsequently read, and then the same register is overwritten with a new value, there is no conflict, because the first value is not external to the region—it can be recreated by executing the region from the beginning.

Table 4.1 shows an example of how our restart marker implementation operates. Initially, our read set, write set, and memory information (base register and offsets) are all clear. When the first instruction—`lw r1,(r4)`—is processed, we determine which registers it reads and which register it writes. If the register it writes conflicts with either one of the registers it reads or the current read set, and that same register has not been previously written, we mark the instruction as a barrier. Thus, the first instruction is not a barrier instruction. The second instruction—`sw r1,4(r4)`—writes memory, but it writes to a different location than the one read by the previous `lw` instruction. Thus, it is not marked as a barrier either. The third instruction—`sw r1,(r5)`—also writes memory, but it uses a different base register than the previous two instructions, so we revert to treating all of memory as a single location, and the instruction is marked as a barrier. We clear all of our state information and process the fourth instruction—`addu r1,r2,r3`—which is not a barrier instruction. The fifth instruction—`srl r1,r1,2`—writes a register (`r1`) that is read by the same instruction, but because this register was written by the previous instruction, the

Read Set	Write Set	Base Reg	Read Offsets	Write Offsets	Code	Read Regs	Write Reg	Barrier?
{}	{}	-	{}	{}	lw r1,(r4)	r4,mem	r1	No
{r4,mem}	{r1}	r4	{0}	{}	sw r1,4(r4)	r1,r4	mem	No
{r4,mem,r1}	{r1,mem}	r4	{0}	{4}	sw r1,(r5)	r1,r5	mem	Yes
{}	{}	-	{}	{}	addu r1,r2,r3	r2,r3	r1	No
{r2,r3}	{r1}	-	{}	{}	srl r1,r1,2	r1	r1	No
{r1,r2,r3}	{r1}	-	{}	{}	addiu r2,r2,1	r2	r2	Yes

Table 4.1: Software restart marker implementation example

value can be recreated by restarting at the `addu` instruction, and thus the `srl` is not marked as a barrier. However, the last instruction—`addiu r2,r2,1`—writes a value that has been previously read but not written, so it must be marked as a barrier. This implementation is somewhat restricted because of the very coarse memory analysis as well as the fact that we operate at the basic-block level. A more sophisticated analysis would allow us to place an entire function into a single restart region.

4.5 Results

We present the results of our restart region analysis in this section for an energy-exposed processor, codenamed Yellow Pekoe, that has software restart markers enabled, and for a baseline processor that has restart markers disabled. Our programs are compiled using a version of `gcc` that has branch delay slot filling disabled—this was necessary to avoid a problem with our exposed bypass latch implementation described in the following chapter. Figure 4-2 shows the percentage of instructions that are marked as barriers in both our baseline and energy-exposed processors. As noted previously, branches and jumps are the only instructions not marked as barriers in our baseline processor, and thus on average 86% of all instructions are barriers. For our energy-exposed processor, only 34% of all instructions are barriers on average, corresponding to about 3 instructions in each restart region. With more aggressive compiler analysis, we expect to generate even larger regions.

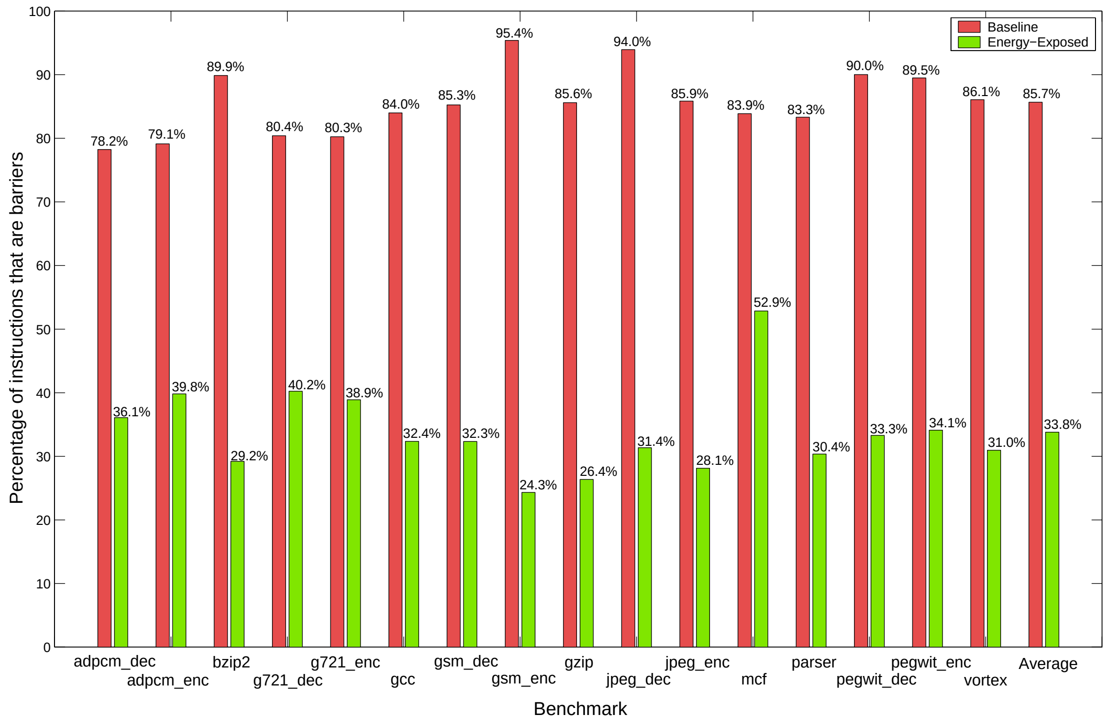


Figure 4-2: Percentage of instructions that are barriers for baseline and energy-exposed Yellow Pekoe processors

Chapter 5

Compiler-Visible Bypass Latches

Software restart markers make it possible for us to expose datapath elements without introducing additional exception management overhead. This enables us to pursue our original goal of making the bypass latches in the datapath visible to the compiler. We implement the exposed bypass latches in a modified version of the Vanilla Pekoe pipeline that is codenamed Yellow Pekoe. Figure 5-1 shows the Yellow Pekoe bypass network. The RS, RT, SD, and X latches are explicitly targeted at compile-time. We map the latches to temporary state so that they do not need to be preserved across exceptions. Thus, in the following code sequence to increment a memory variable,

```
lw r1, (r3)             # Load value.
add r1, r1, 1           # Increment.
sw r1, (r3)             # Update memory.
```

the load and add instructions can be changed to target the RS and SD bypass latches, as the values being written are each only used once by the subsequent instructions. This leads to the following transformed code sequence:

```
lw RS, (r3)             # Load RS latch with memory value.
add SD, RS, 1           # Increment and place result into SD latch.
sw.bar SD, (r3)         # Update memory with barrier instruction.
```

The explicit use of the bypass latches in the above code allows us to eliminate two register file writes and two register file reads.

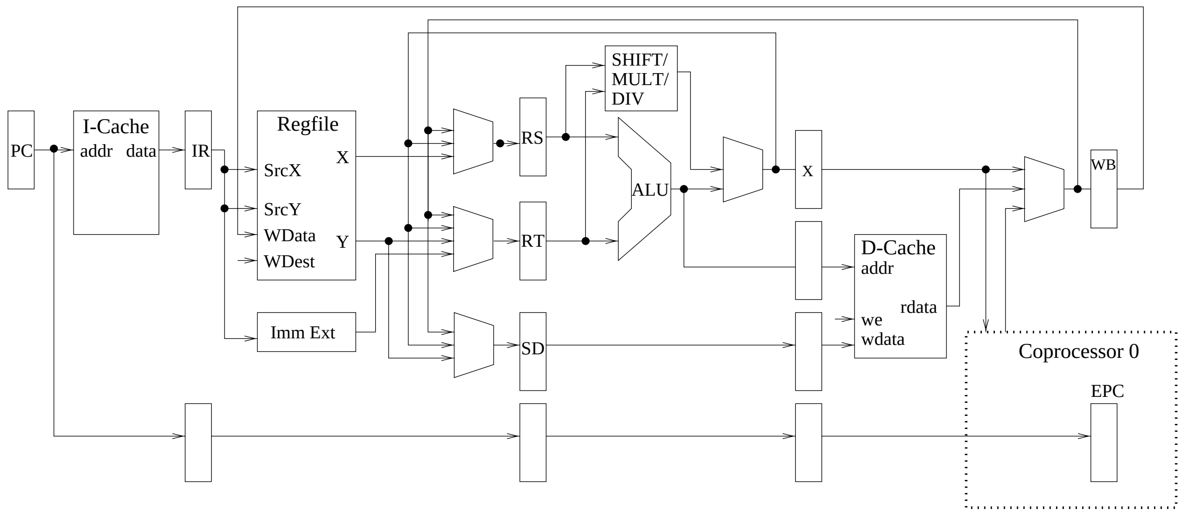


Figure 5-1: Yellow Pekoe bypass network

5.1 Compiler and Assembler Modifications

We make modifications to both the GNU compiler and assembler to implement the exposed bypass latches. Our compiler changes take advantage of the static liveness information that is already maintained by `gcc`. When the compiler determines that a value read by an instruction is being referenced for the last time—i.e. the value will be dead after the instruction executes—we have it append a "`.l`" suffix to the assembly opcode with a corresponding operand number to indicate the last use of the value. For example, consider the following code sequence:

```
add r3, r1, r2
sub r4, r3, r5
or r1, r3, r6
```

If the value in `r3` will not be referenced again after the `or` instruction, the compiler will output `or.l1`. If the value in `r6` will not be referenced again after the `or` instruction, the compiler will output `or.l2`. If neither value will be referenced again, the compiler will output `or.l12`.

Our use of `gcc`'s liveness information does pose one problem. The last compiler pass before generating an assembly file fills branch delay slots; unfortunately, `gcc` does not fully update its liveness information during this pass. As a result, to enable correct operation of all programs, we disable branch delay slot filling in the compiler. This decreases performance, as fewer branch delay slots are filled. However, our results should be similar to those that would be obtained using a compiler that does not have this problem as it should be frequently possible to fill delay slots with instructions that could not use a bypass latch.

The liveness information generated for each instruction is used by an instruction scheduler that we added to the assembler. The scheduler reorders instructions within a basic block. It performs several passes on the code. First, it attempts to maximize performance by reordering instructions to mask latencies that can cause pipeline stalls—in particular, it tries to fill load-use delay slots with independent instructions. It also attempts to fill the branch delay slot. Next, the scheduler uses the lifetime information generated by the compiler to determine if bypass latches can be used in place of general-purpose registers to statically bypass a value. The scheduler then looks for static read caching opportunities in its next pass, after which it tries to perform additional static bypassing from the memory stage of the pipeline, and then creates the restart regions discussed in the previous chapter.

Note that static bypassing from the memory stage raises additional constraints not required for bypassing from one instruction to a subsequent instruction. Consider the following example:

```
add r1, r2, r3
sub r4, r5, r6
and.l1 r7, r1, r5
```

In the above code segment, `r1` is read for the last time by the `and.l1` instruction. This would appear to provide an opportunity for static bypassing by having the scheduler write to the X latch. However, in this scenario, if there is an instruction cache miss for the `and.l1` instruction, the `sub` instruction will overwrite the value in the

X latch. To avoid this problem, we must either require strict pipeline sequencing, so that instructions go down the pipeline together, or not permit the intermediate instruction (in the above example, the `sub` instruction) to overwrite the X latch. We choose the latter option, as this places no additional constraints on the hardware implementation.

For our simulations, we model the bypass latches by reserving four general-purpose registers in the compiler and using their specifiers in the scheduler when modifying an instruction to target a bypass latch. The loss of these registers in the compiler's register allocator does not have an adverse effect on performance. However, in the ideal case, the use of the bypass latches would be encoded in the instruction set.

Figure 5-2 shows how a program flows through our toolchain.

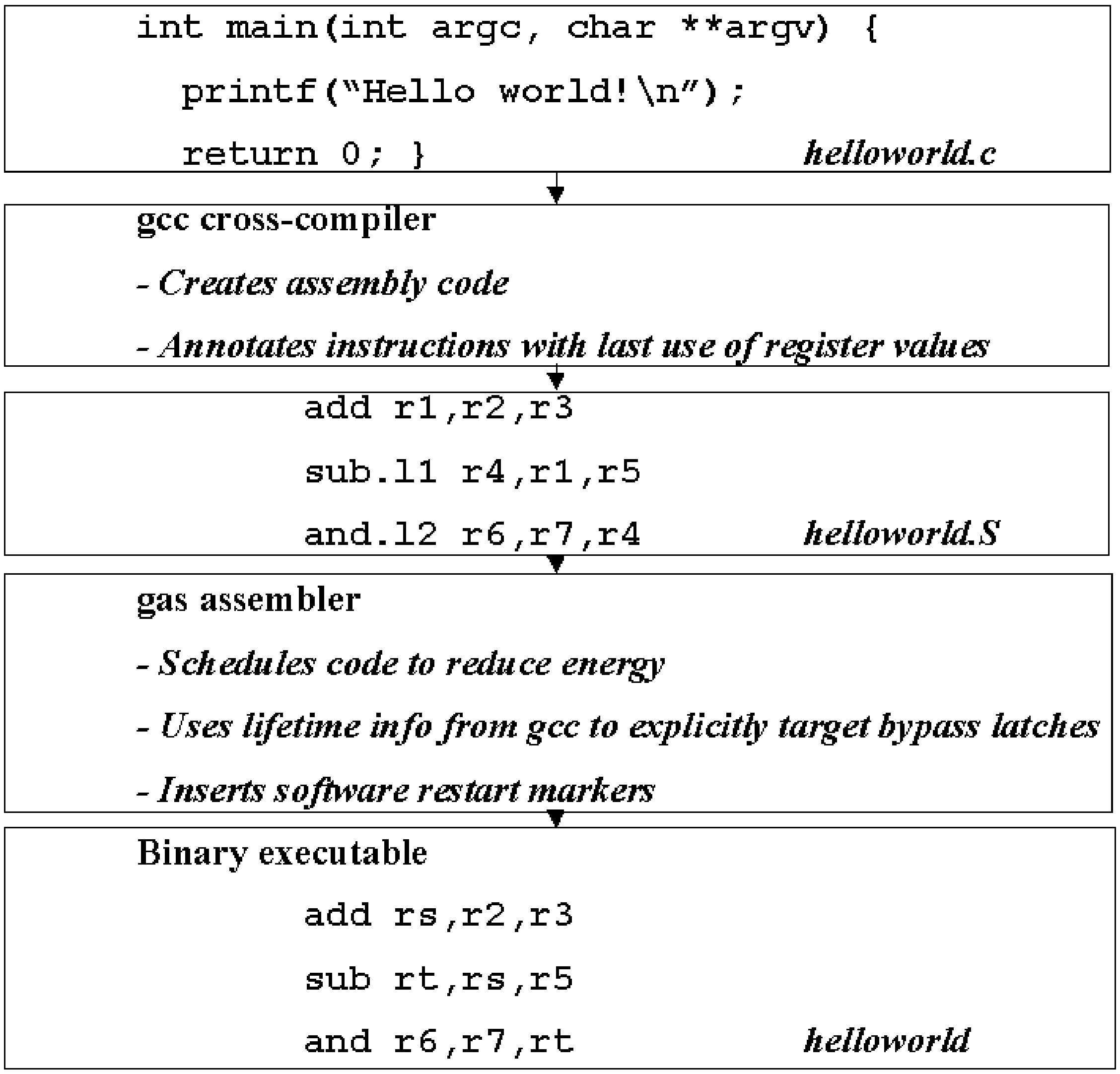
5.2 Results

We present statistics for programs run on our energy-exposed Yellow Pekoe processor in this section. We also draw some comparisons between these statistics and those derived from a baseline case of running modified versions of these programs with the use of software restart markers and bypass latches disabled in the instruction scheduler (but with branch delay slot filling still disabled in the compiler so that performance will be comparable).

5.2.1 Bypass Latches

Bypass Latch Reads

We show the percentage of values that are statically bypassed in Figure 5-3 and compare against the percentage of values that could be dynamically bypassed. Note that dynamic bypassing will generally be more effective than static bypassing because compilers can not always statically guarantee whether a value is live across basic blocks. As shown in [25], however, turning off unnecessary reads of the register file using the bypass skip technique may increase processor cycle time. With static



```
int main(int argc, char **argv) {
  printf("Hello world!\n");
  return 0; }
```

gcc cross-compiler

- Creates assembly code

- Annotates instructions with last use of register values

```
add r1,r2,r3
sub.l1 r4,r1,r5
and.l2 r6,r7,r4
```

gas assembler

- Schedules code to reduce energy

- Uses lifetime info from gcc to explicitly target bypass latches

- Inserts software restart markers

Binary executable

```
add rs,r2,r3
sub rt,rs,r5
and r6,r7,rt
```

Figure 5-2: Sample compilation of a program

bypassing, this does not occur.

The percentage of values that are statically read cached is shown in Figure 5-4, and this is compared against the percentage of values that could be dynamically cached. Once again, dynamic caching can eliminate a greater number of register file reads, because the scheduler can not read cache across a barrier instruction. However, the additional hardware required has been shown to remove most of the potential energy savings [25].

Finally, we compare the percentage of reads that can be statically eliminated through both bypassing and read caching to the percentage of reads that can be dynamically eliminated in Figure 5-5. Note that bypassing and read caching are not completely orthogonal. In the following instruction sequence,

```
add r3, r1, r2
sub r4, r3, r5
or r1, r3, r6
```

the value in `r3` for the last instruction can be both bypassed and read cached. In our statistics, we count this scenario as a case of read caching, as that avoids the clocking of the bypass latch in addition to eliminating the register file access.

Bypass Latch Writes

Figure 5-6 shows the percentage of register file writes that we eliminate in our energy-exposed processor. Note that these can not be easily eliminated with a purely hardware scheme as this requires knowledge of register value lifetime.

5.2.2 Instruction Chains

To understand how the compiler uses bypass latches, we track the most common *instruction chains* for each program. An instruction chain is started when a bypass latch is written, and each subsequent instruction in the chain reads the previously

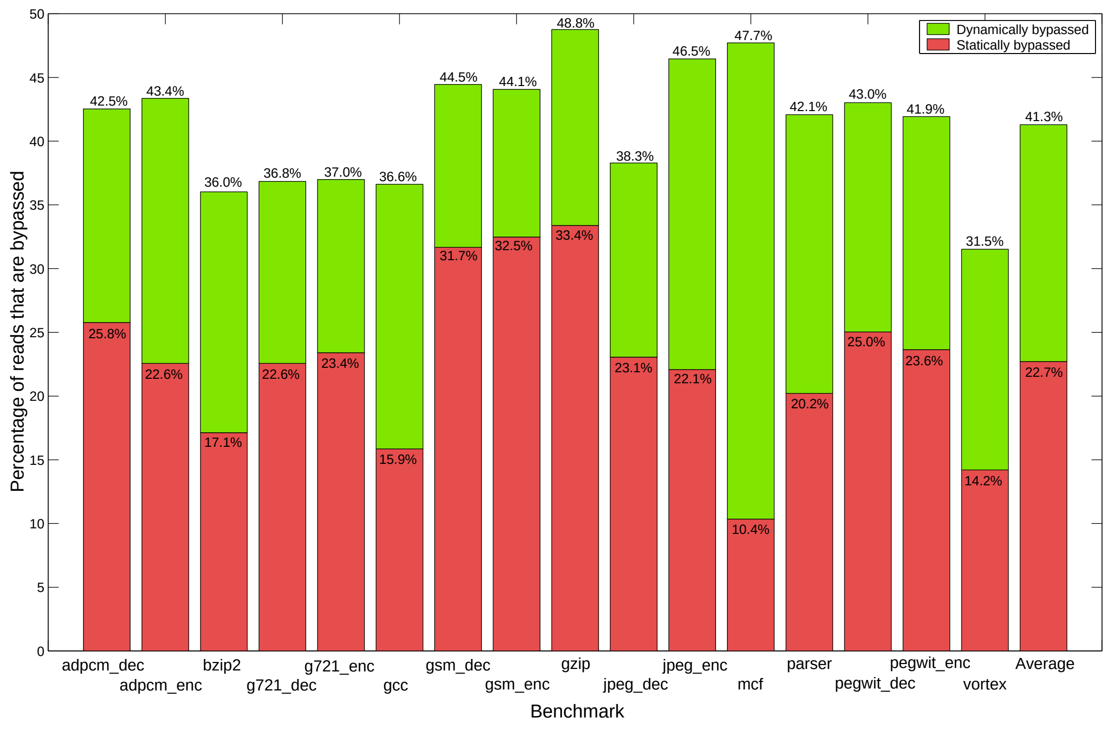


Figure 5-3: Percentage of reads that are bypassed for baseline and energy-exposed Yellow Pekoe processors

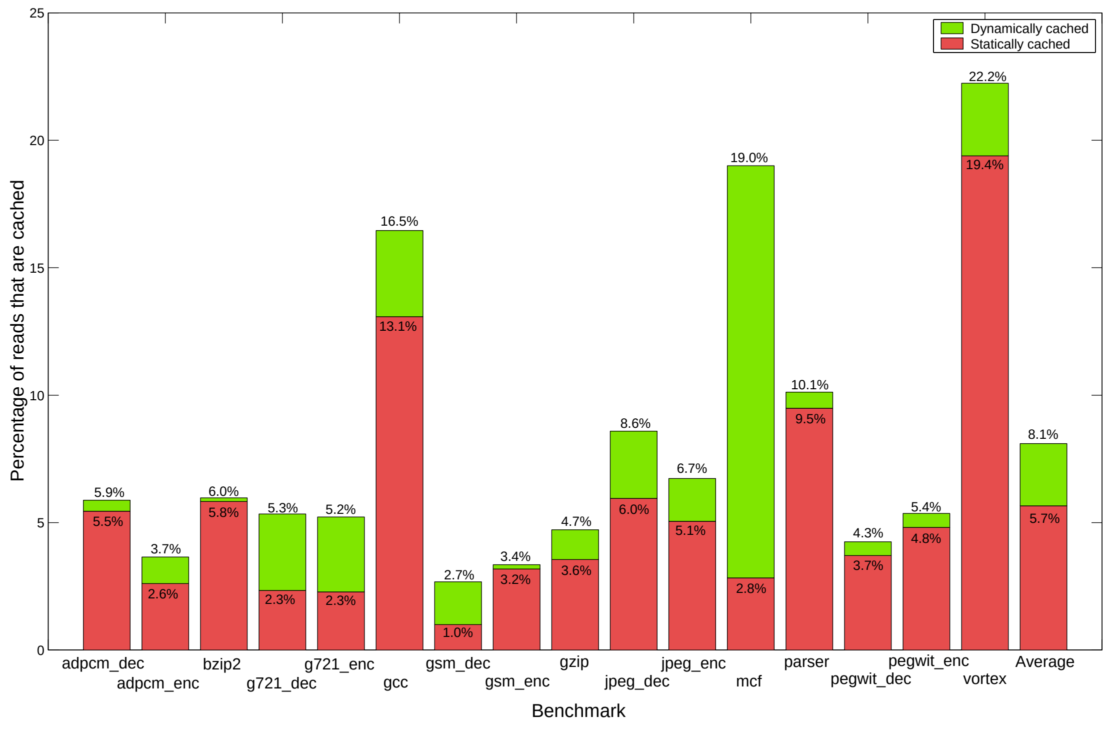


Figure 5-4: Percentage of reads that are cached for baseline and energy-exposed Yellow Pekoe processors

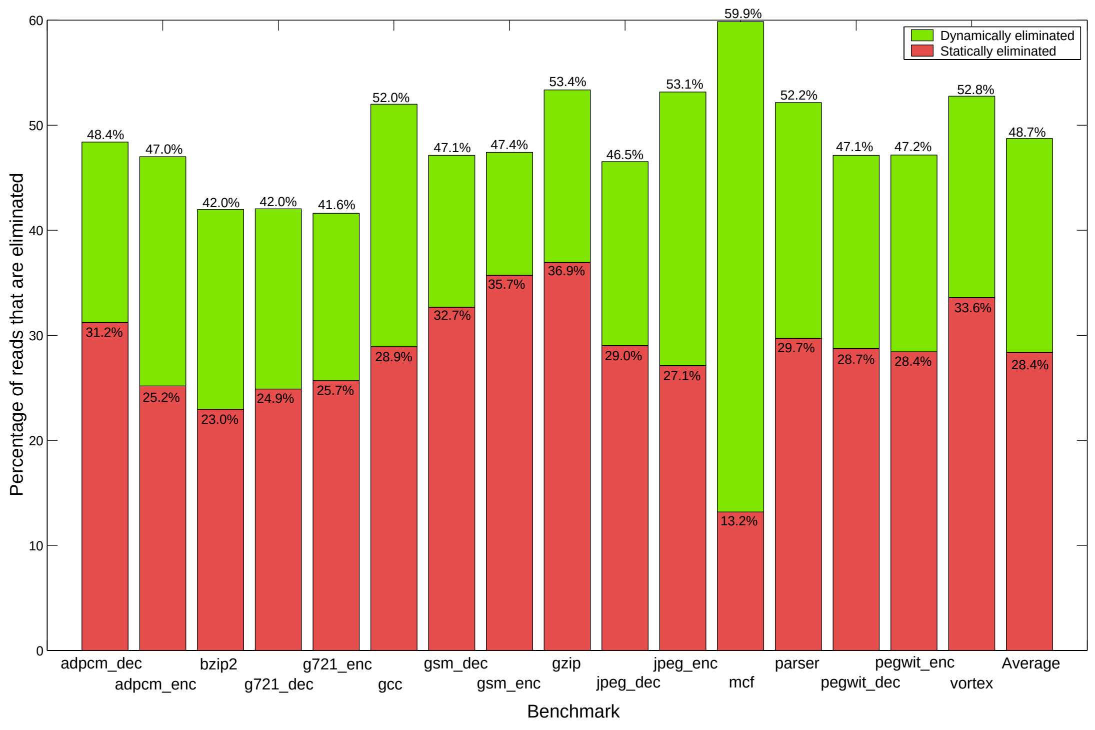


Figure 5-5: Percentage of reads that are eliminated for baseline and energy-exposed Yellow Pekoe processors

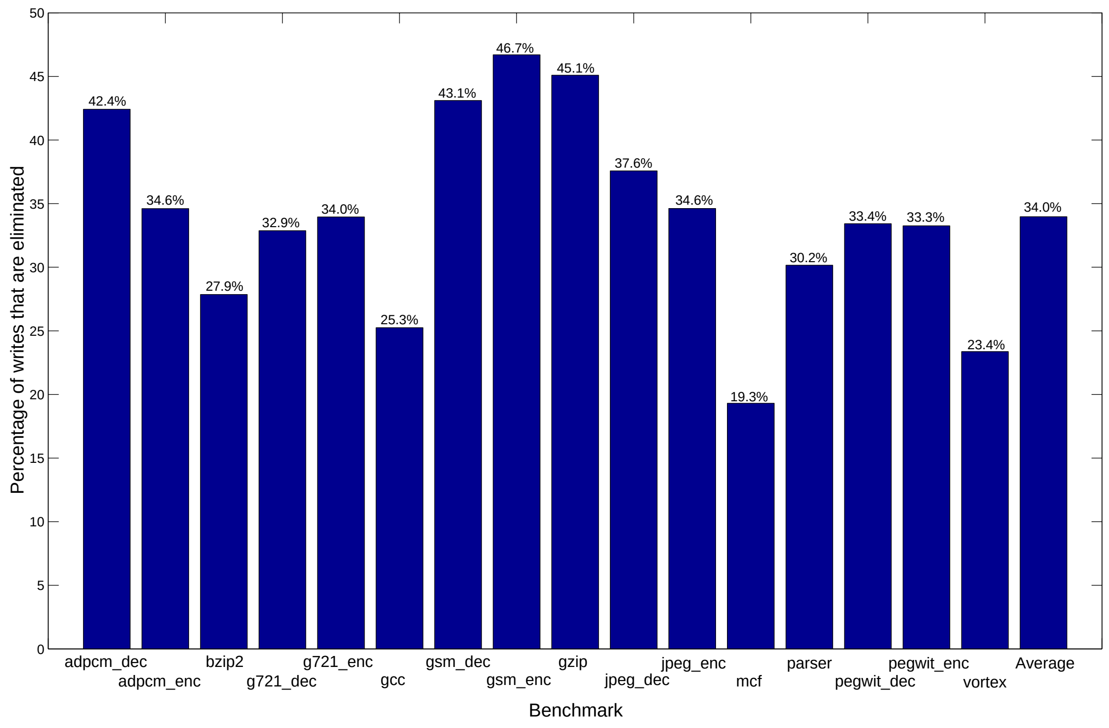


Figure 5-6: Percentage of writes that are eliminated for energy-exposed Yellow Pekoe processor

computed value and writes a new value to a latch. The chain is terminated when an instruction reads a value from a latch, but does not write a new value into a latch. We present the most common instruction chains (those that make up at least 1% of all chains) for selected benchmarks in Tables 5.1, 5.2, and 5.3. Figure 5-7 shows the percentage of instructions which are in a chain across all benchmarks. Our results show that there is considerable variety in the instruction chains that are generated for each program. This indicates that a small number of instruction patterns will not suffice when designing a new instruction set—generality is needed. RISC architectures typically have fixed-length instructions with a limited number of instruction formats. By comparison, CISC architectures typically support variable-length instructions, and a single instruction may represent several operations. For the purposes of encoding instruction chains, which can vary widely in length and trigger multiple operations, a typical RISC architecture is too inflexible. A CISC-style variable-length encoding would be better able to represent instruction chains. A possible area for future work is to explore more compact instruction set encodings which should enable energy savings in instruction fetch. Additionally, more portable definitions of bypass latches that support a wider variety of processor implementations can be explored.

Another interesting observation is that several of the instruction chains contain a bypass of a value from a `lw` instruction to the following instruction. For our pipeline, this results in a load interlock. There are some situations in which this is unavoidable due to dependencies that prevent reordering of instructions. Removing the load-use delay slot from the pipeline could result in both improved performance and more opportunities for chaining. One way to achieve this is by splitting the pipeline into two parts: one to handle memory operations and one to process data operations [5]. This has advantages and disadvantages, which will be considered in the next chapter.

5.2.3 Energy Savings

We ran the Yellow Pekoe benchmarks on our energy simulator and compared the results to those generated by running the same benchmarks with the software restart marker and bypass latch optimizations turned off on our low-power baseline processor.

Instruction Chain	Number of Occurrences	Percentage of All Chains	Percentage of All User-Level Instructions
andi,beq	442560	27.66	13.06
sll,addu	295040	18.44	8.71
slti,beq	147520	9.22	4.35
nor,sra,and	147520	9.22	6.53
slti,bne	147520	9.22	4.35
lw,addu	147520	9.22	4.35
slt,beq	147520	9.22	4.35
sra,addu	124162	7.76	3.66

Table 5.1: Most common instruction chains for adpcm_dec benchmark

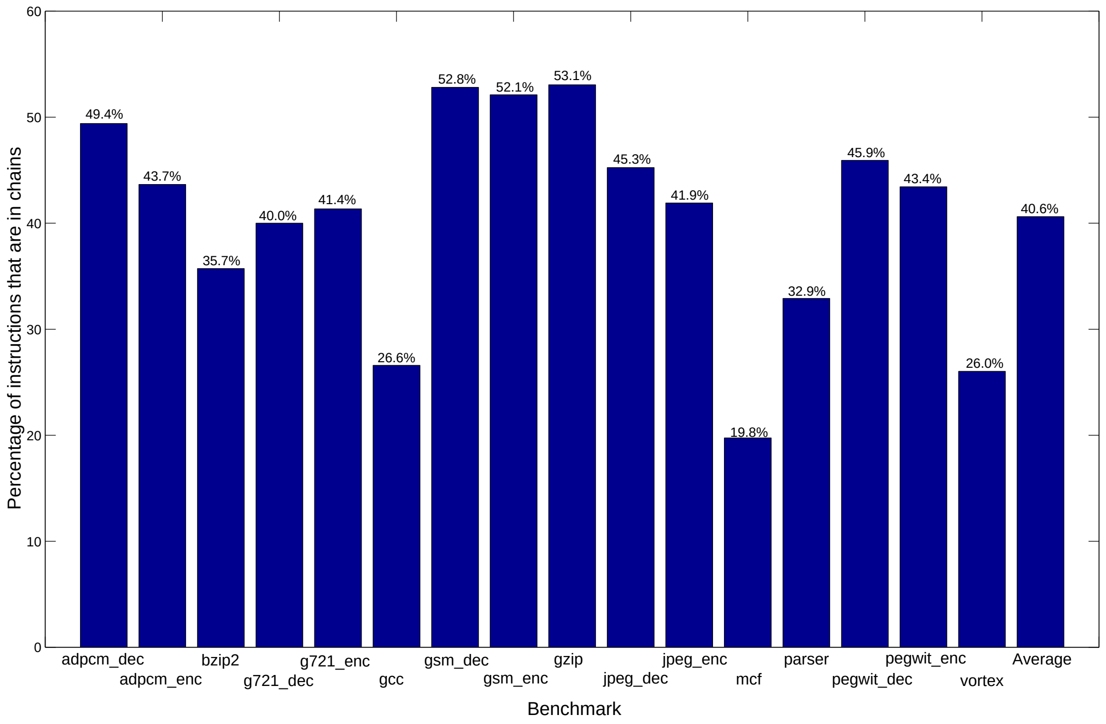


Figure 5-7: Percentage of instructions that are in chains

Instruction Chain	Number of Occurrences	Percentage of All Chains	Percentage of All User-Level Instructions
sll,addu	122611	16.76	4.77
addu,addiu,sra	67200	9.18	3.92
addu,addu	66604	9.10	2.59
sltu,bne	34438	4.71	1.34
lw,or	34040	4.65	1.32
lw,addu,addu	33823	4.62	1.97
addu,sra,addu,addu	33823	4.62	2.63
sll,addu,sll,subu	24531	3.35	1.91
addu,addu,sra,andi,addu	23996	3.28	2.33
subu,addu,sra,andi,addu	23996	3.28	2.33
slti,bne	20072	2.74	0.78
sll,subu,sll,subu, sll,addu,sll,addu,subu	10355	1.42	1.81
slti,beq	9841	1.34	0.38
lw,mult	9472	1.29	0.37
addu,addiu,sra,sw	8712	1.19	0.68
subu,addiu,sra,sw	8712	1.19	0.68
sll,subu	8689	1.19	0.34
addu,sll	8201	1.12	0.32
sll,addu,sll,addu,sll, subu,sll,addu	8177	1.12	1.27
sll,addu,sll,subu,sll	8177	1.12	0.79
sll,subu,sll,addu,sll,addu	8177	1.12	0.95
sll,addu,sll,subu,sll, subu,sll,addu,sll,subu	8177	1.12	1.59
sll,addu,sll,addu,sll, addu,sll,subu	8177	1.12	1.27
sll,addu,sll,addu,sll, subu,sll,subu	8177	1.12	1.27
sll,subu,sll	8177	1.12	0.48
addiu,mult	8177	1.12	0.32
addiu,srav,andi	8061	1.10	0.47
lw,addu	8009	1.09	0.31
slt,beq	7783	1.06	0.30

Table 5.2: Most common instruction chains for jpeg_dec benchmark

Instruction Chain	Number of Occurrences	Percentage of All Chains	Percentage of All User-Level Instructions
lw,slt,beq	2694018	14.62	3.58
subu,addu	2602817	14.12	2.30
lw,slt,bne	2391340	12.97	3.17
slt,bne	1840607	9.99	1.63
slt,beq	1686240	9.15	1.49
sll,addu	1353098	7.34	1.20
lw,beq	784234	4.25	0.69
lw,bne	685775	3.72	0.61
slti,bne	480697	2.61	0.43
lui,lw	452129	2.45	0.40
lw,addu,slt,bne	416670	2.26	0.74
lui,addiu	403404	2.19	0.36
slt,and,beq	388318	2.11	0.52
lui,addu,lw	388318	2.11	0.52
srl,addu,sra,sll	388318	2.11	0.69
sltu,bne	359946	1.95	0.32
lw,sw	232696	1.26	0.21
lw,subu	210731	1.14	0.19
lw,addu	209129	1.13	0.19

Table 5.3: Most common instruction chains for mcf benchmark

Our baseline design still implements the bypass skip technique of turning off the register file read when a value is provided by the bypass network. Figures 5-8, 5-9, 5-10, and 5-11 show the energy breakdowns in selected benchmarks for the pipeline stages in the baseline and energy-exposed processors. The energy-exposed processor typically saves energy in the PC generation and coprocessor 0 stages (since it does not have to propagate the exception PC through the pipeline as often), the decode stage (through the avoidance of register file reads), the execute stage (by avoiding the clocking of bypass latches when a value is read cached), and the writeback stage (by avoiding register file writes). Table 5.4 shows the energy savings across most of our benchmarks. Our energy simulator could only execute a limited number of processor cycles for a given program, so we do not have the results for the bzip2, parser, or vortex benchmarks. We observe an average energy savings of 7.0% across the remaining benchmarks. Our savings would likely be even greater if the same optimizations are implemented on a processor that is not already targeted for low-power uses.

5.2.4 Summary

By exposing the bypass latches to the compiler and implementing software restart markers, we reduce the energy consumption of a low-power processor by 7.0% on average. These techniques are easily implemented and require few hardware modifications. Our results show the feasibility of making datapath elements visible to the compiler for the purpose of lowering energy consumption, and indicate that further experiments are warranted.

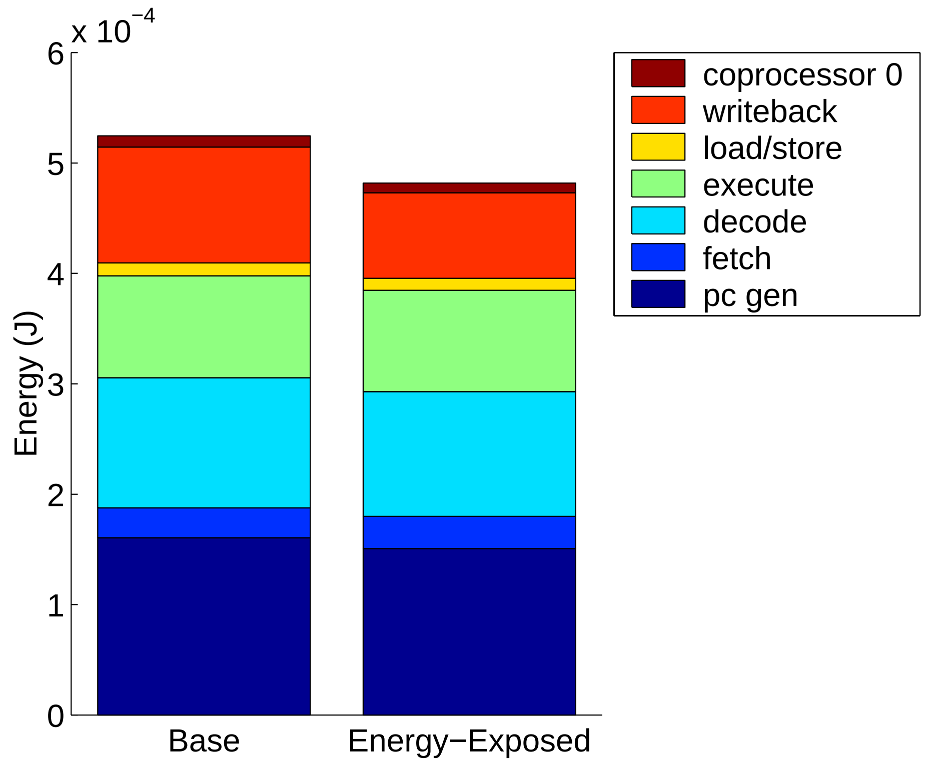


Figure 5-8: Energy breakdown for adpcm_dec benchmark

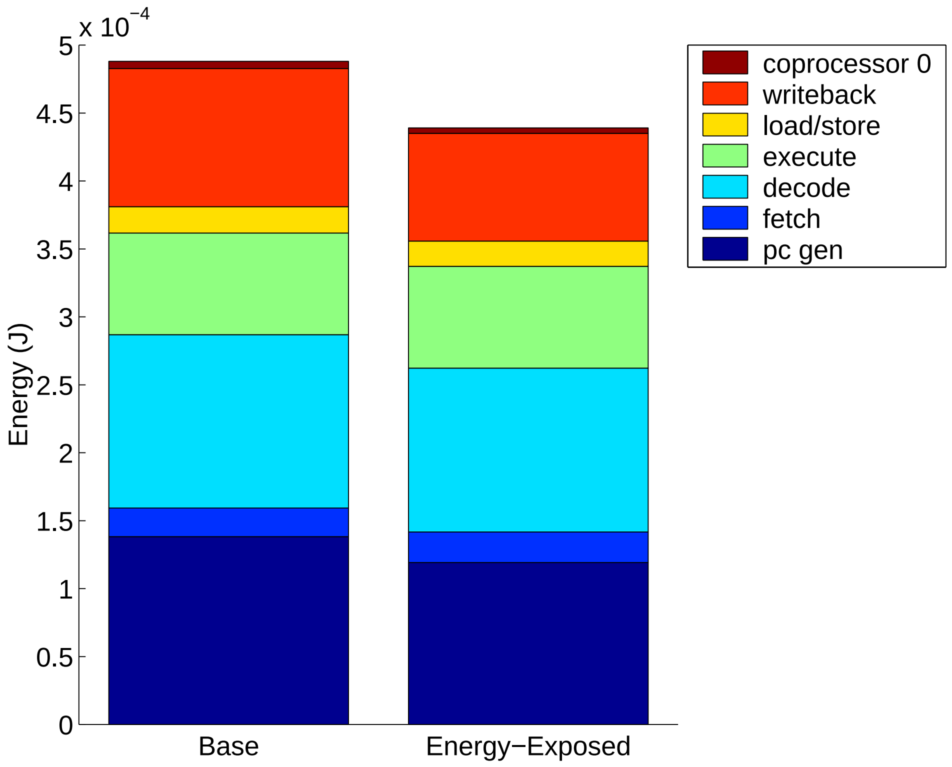


Figure 5-9: Energy breakdown for jpeg_dec benchmark

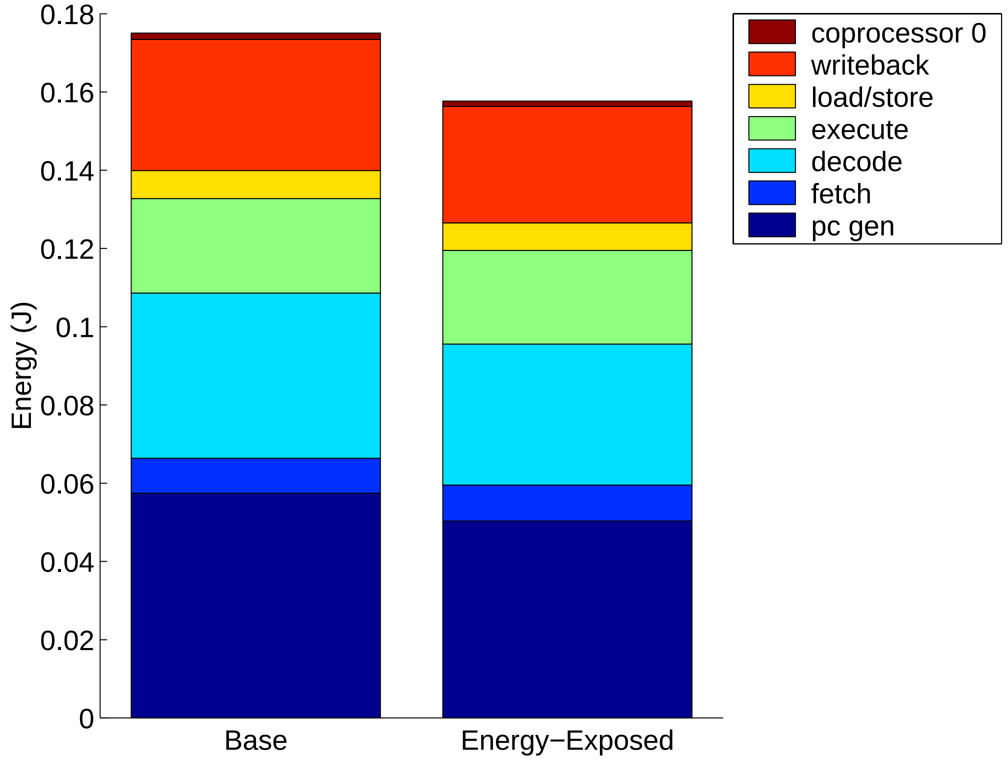


Figure 5-10: Energy breakdown for gcc benchmark

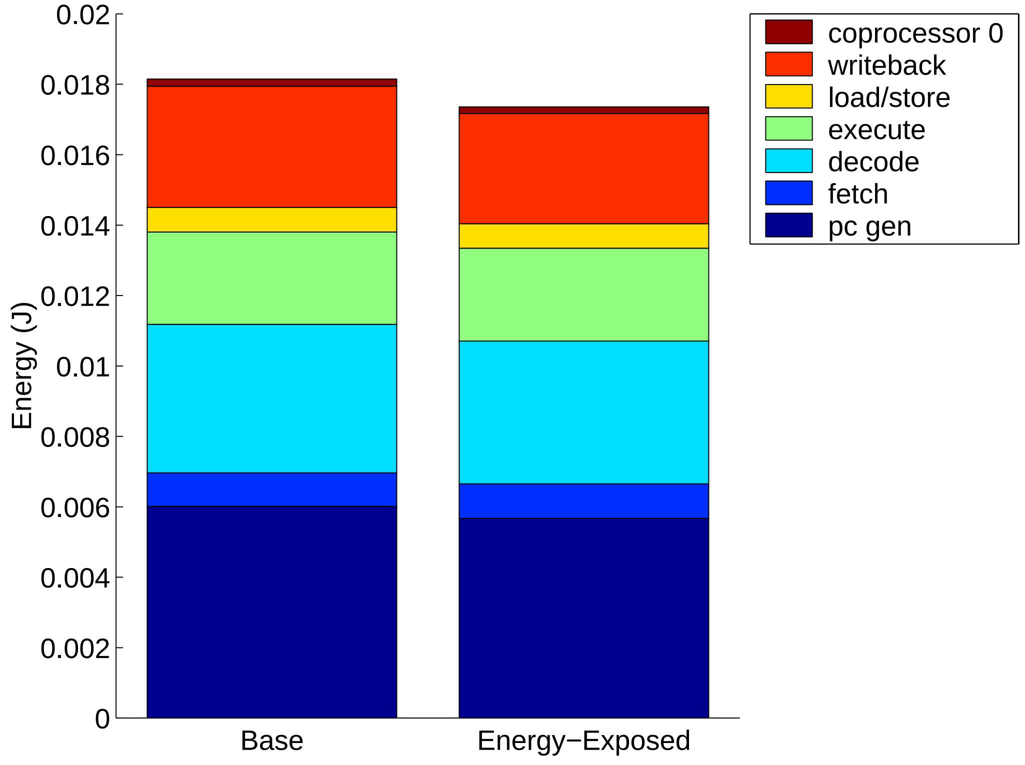


Figure 5-11: Energy breakdown for mcf benchmark

Benchmark	Baseline Processor Energy (mJ)	Energy-Exposed Processor Energy (mJ)	Energy Savings (Percentage)
adpcm(decode)	0.5247	0.4818	8.2
adpcm(encode)	0.6411	0.6055	5.6
g721(decode)	25.83	24.88	3.7
g721(encode)	26.58	25.17	5.3
gcc	175.1	157.7	9.9
gsm(decode)	10.01	9.606	4.0
gsm(encode)	29.10	26.65	8.4
gzip	294.43	266.66	9.4
jpeg(decode)	0.4880	0.4391	10.0
jpeg(encode)	1.509	1.398	7.4
mcf	18.15	17.36	4.4
pegwit(decode)	1.946	1.812	6.9
pegwit(encode)	3.478	3.221	7.4
Average			7.0

Table 5.4: Energy comparison between baseline and energy-exposed processor

Chapter 6

Split-Pipeline Processor

6.1 Motivation

One of the factors that limits performance in the Vanilla Pekoe and Yellow Pekoe pipelines is the load-use delay slot. Table 6.1 shows the percentage of all load-use delay slots in our Vanilla Pekoe pipeline that are unfilled, and the percentage of user-level processor cycles that are load interlocks as a result. On average, about 9% of all user-level processor cycles are load interlocks. Table 6.2 shows the same data for our Yellow Pekoe pipeline. Even though the Yellow Pekoe programs are passed through an instruction scheduler that tries to place independent instructions in load-use delay slots, frequently the percentage of load-use delay slots that are unfilled is basically unchanged (and in some cases is slightly worse). (Note that the percentages of user-level cycles listed in Table 6.2 are lower than if branch delay slot filling was enabled in the compiler, as this would reduce the total number of processor cycles.)

The prevalence of load interlock cycles has caused some processor designers to restructure the pipeline so that the load-use delay slot is removed. This is accomplished by splitting the pipeline into two parts: one to handle memory address computations and memory accesses, and the other to handle data computations. This approach is adopted in the TFP microprocessor [8], and was considered in the design of the MultiTitan CPU [11]. A split pipeline has the advantage of removing the load-use delay slot, but introduces an address-generation delay slot and causes branches to be resolved one cycle later than they would be in the single pipeline organization. However, as shown in [5], the performance of the split pipeline should be equivalent to or better than the performance of the standard single pipeline organization, as long as a static branch predictor is used that is at least 80% accurate. The split-pipeline organization is also potentially appealing from an energy perspective. By separating address and data computations, there might be greater correlation between the values that are processed by the data ALU and the adder for memory addresses. This can result in less bitline switching energy. Additionally, splitting the pipelines introduces more latches into the bypass network, resulting in more temporary state to expose. This can also be viewed as a disadvantage as the extra latches and adder will consume energy. However, if we can show that our bypass latch use significantly increases, that will indicate that splitting the pipelines is probably worth the energy cost.

Figure 6-1 shows the bypass network for our split-pipeline processor, which we codename Tangerine Pekoe. We target the AX1, AY1, AX2, AY2, DX, DY, SD1, and SD2 bypass latches. A similar methodology is used for the Tangerine Pekoe toolset as for the Yellow Pekoe toolset. Eight general-purpose registers are removed from `gcc`'s register allocation pool to represent the latches. Additionally, branch delay slot filling is disabled in the compiler. The assembly-level instruction scheduler tries to fill the address-generation delay slot whenever possible. The Tangerine Pekoe processor also implements static bypassing, static read caching, and the elimination of register file writes. Clock gating is used to conserve energy and to avoid clobbering the values in the latches when they are not being used. For example, if a load instruction is proceeding down the pipeline, the AX2, AY2, DX, and DY latches do not have to be clocked, as they are not used for this instruction. This split-pipeline organization allows static bypassing possibilities that can not be achieved dynamically, as illustrated by the following code sequence:

```
lui r1, 0x8003
addu r2, r3, r4
srl r3, r5, 2
```

```
lw r1, 16(r1)
```

The `lw` instruction uses `r1` as a base register and then overwrites the value in `r1`. If the `lw` instruction was swapped with the `srl` instruction, the value placed into `r1` by the `lui` instruction could be dynamically bypassed to the `lw` instruction. In the sequence as written, this bypassing is not possible. However, the value can be statically bypassed, as the instructions between the `lui` and `lw` instruction are processed by the data section of the pipeline, and thus do not clobber the latches in the memory section of the pipeline. We can therefore rewrite the code sequence to use a bypass latch.

```
lui AX1, 0x8003
addu r2, r3, r4
srl r3, r5, 2
lw r1, 16(AX1)
```

This example shows that statically bypassed read values are not necessarily a subset of dynamically bypassed read values in the Tangerine Pekoe processor.

6.2 Results

We executed programs on the Tangerine Pekoe processor so that we could compare it to the Yellow Pekoe processor in terms of performance as well as the usage of bypass latches. We make the performance comparison because although the focus of this study is energy consumption, we do not want to adversely affect performance when implementing our techniques. The bypass latch usage analysis should give an indication as to whether the Tangerine Pekoe processor is more appealing from an energy perspective than the Yellow Pekoe processor. Since the same technique is used to create restart regions for both Yellow Pekoe and Tangerine Pekoe programs, we do not compare the two processors in terms of percentages of instructions that are barriers.

Benchmark	Percentage of Load-Use Delay Slots That Are Unfilled	Percentage of User-Level Cycles That Are Interlocks
adpcm(decode)	66.62	4.35
adpcm(encode)	74.92	5.22
bzip2	48.12	13.12
g721(decode)	65.10	10.64
g721(encode)	67.87	10.92
gcc	39.58	8.54
gsm(decode)	18.11	1.47
gsm(encode)	38.15	4.93
gzip	33.45	7.84
jpeg(decode)	63.83	13.23
jpeg(encode)	73.01	14.06
mcf	45.22	7.20
parser	30.77	7.33
pegwit(decode)	45.82	11.61
pegwit(encode)	50.24	12.27
vortex	31.61	7.60
Average	49.53	8.77

Table 6.1: Vanilla Pekoe load interlocks

Benchmark	Percentage of Load-Use Delay Slots That Are Unfilled	Percentage of User-Level Cycles That Are Interlocks
adpcm(decode)	66.54	3.93
adpcm(encode)	74.86	5.07
bzip2	47.39	11.70
g721(decode)	66.02	10.78
g721(encode)	68.77	11.06
gcc	37.71	7.55
gsm(decode)	21.12	1.67
gsm(encode)	15.58	2.45
gzip	32.82	6.95
jpeg(decode)	45.81	9.76
jpeg(encode)	43.12	7.77
mcf	44.54	6.38
parser	30.40	6.46
pegwit(decode)	46.66	11.31
pegwit(encode)	50.97	11.88
vortex	29.21	6.39
Average	45.10	7.57

Table 6.2: Yellow Pekoe load interlocks

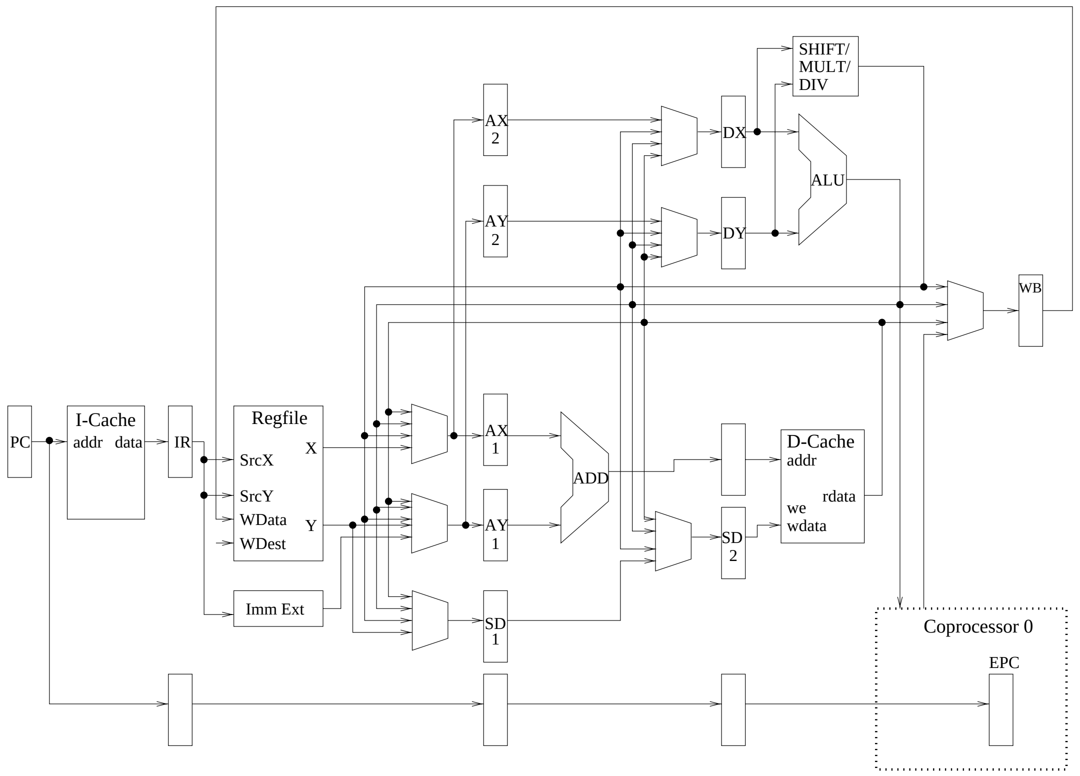


Figure 6-1: Tangerine Pekoe bypass network

6.2.1 Performance

Tables 6.3 and 6.4 show the number of user-level instructions and processor cycles for programs run on both the Yellow Pekoe and Tangerine Pekoe processors. We present the user-level instruction counts because the assembler code generated by `gcc` for each processor (before the software restart marker and bypass latch optimizations are performed) will not necessarily be identical due to the different number of registers available to the compiler's register allocator, and this may have an effect on the number of program cycles. Table 6.3 shows that the Tangerine Pekoe programs have slightly fewer instructions. The cycle count difference is more noticeable, as there is about a 2% total reduction in processor cycles for Tangerine Pekoe. Note that we are still using a simple branch prediction strategy—all conditional branches are predicted as taken. This results in relatively high branch misprediction rates, as illustrated by Figure 6-2. With a more accurate branch predictor, the performance improvement for Tangerine Pekoe would be even greater.

A similar study comparing the single pipeline organization of Vanilla and Yellow Pekoe to the split-pipeline organization of Tangerine Pekoe in terms of performance was conducted in [14]. That study concluded that splitting the pipelines actually resulted in slightly worse performance. We account for the differences in our conclusions in the following ways. First, although our methodology is similar to the one used in [14]—we both use the same type of instruction scheduler in the assembler that tries to maximize performance—the scheduler used for this study is more recent than the one used in [14], and does a better job of removing address-generation interlocks. Second, as shown in Table 6.3, reserving additional general-purpose registers for the Tangerine Pekoe programs actually results in a slight decrease in the number of instructions, which probably contributes to the performance improvement. Finally, some of the benchmarks we use in this study differ than those used in [14]. In particular, we use SPECint2000 benchmarks which the other study does not use. Table 6.4 shows that many of the biggest performance gains on Tangerine Pekoe occur for the SPECint2000 benchmarks. Thus, these factors all account for the difference in our conclusions.

6.2.2 Bypass Latches

Figure 6-3 shows the percentage of register file reads that are statically eliminated in Tangerine Pekoe because the corresponding values are taken from the bypass latches. Figure 6-4 shows the percentage of register file writes that are eliminated. We observe that on average, the percentage of eliminated register file reads is slightly less than the corresponding percentage for Yellow Pekoe—27% of reads are eliminated for Tangerine Pekoe as opposed to 28% for Yellow Pekoe. The percentage of eliminated register file writes is the same on average for both processors at 34%. It appears that the additional bypass latches and the split pipeline found in Tangerine Pekoe do not help eliminate register file accesses; in fact, they slightly degrade our results. The slight degradation in the statistics on register file reads can probably be attributed to our scheduling algorithm—more fine-tuning should allow us to achieve at least the same results that were seen for the Yellow Pekoe processor. The more pressing issue

Benchmark	Yellow Pekoe Instruction Count	Tangerine Pekoe Instruction Count	% Difference From Yellow Pekoe
adpcm(decode)	6,777,414	6,631,374	-2.15
adpcm(encode)	7,776,642	7,778,270	+0.02
bzip2	10,929,985,995	10,955,264,579	+0.23
g721(decode)	299,896,064	300,486,172	+0.20
g721(encode)	307,477,742	308,067,834	+0.19
gcc	2,008,399,517	1,992,580,495	-0.79
gsm(decode)	93,565,719	93,505,772	-0.06
gsm(encode)	243,171,726	243,606,473	+0.18
gzip	3,357,103,233	3,379,505,713	+0.67
jpeg(decode)	5,145,100	5,190,717	+0.89
jpeg(encode)	17,721,228	17,682,135	-0.22
mcf	226,051,602	226,042,537	0.00
parser	4,207,856,366	4,122,900,228	-2.02
pegwit(decode)	20,410,353	20,416,696	+0.03
pegwit(encode)	36,067,836	36,114,430	+0.13
vortex	10,500,922,643	10,434,546,033	-0.63
Average			-0.21

Table 6.3: User-level instruction count comparison for Yellow Pekoe and Tangerine Pekoe processors

Benchmark	Yellow Pekoe Cycle Count	Tangerine Pekoe Cycle Count	% Difference From Yellow Pekoe
adpcm(decode)	7,518,302	7,447,906	-0.94
adpcm(encode)	8,735,487	8,809,436	+0.85
bzip2	12,703,422,435	11,865,898,973	-6.59
g721(decode)	383,565,226	385,314,390	+0.46
g721(encode)	394,406,640	396,064,495	+0.42
gcc	2,468,749,329	2,436,496,890	-1.31
gsm(decode)	144,690,193	148,325,159	+2.51
gsm(encode)	444,967,285	439,616,850	-1.20
gzip	3,985,268,606	3,965,489,064	-0.50
jpeg(decode)	6,380,858	6,065,456	-4.94
jpeg(encode)	21,886,650	21,151,311	-3.36
mcf	280,385,809	271,438,698	-3.19
parser	5,126,436,845	4,972,638,717	-3.00
pegwit(decode)	24,037,245	23,905,095	-0.55
pegwit(encode)	42,970,788	42,458,130	-1.19
vortex	12,972,417,102	12,380,997,088	-4.56
Average			-1.69

Table 6.4: User-level cycle count comparison for Yellow Pekoe and Tangerine Pekoe processors

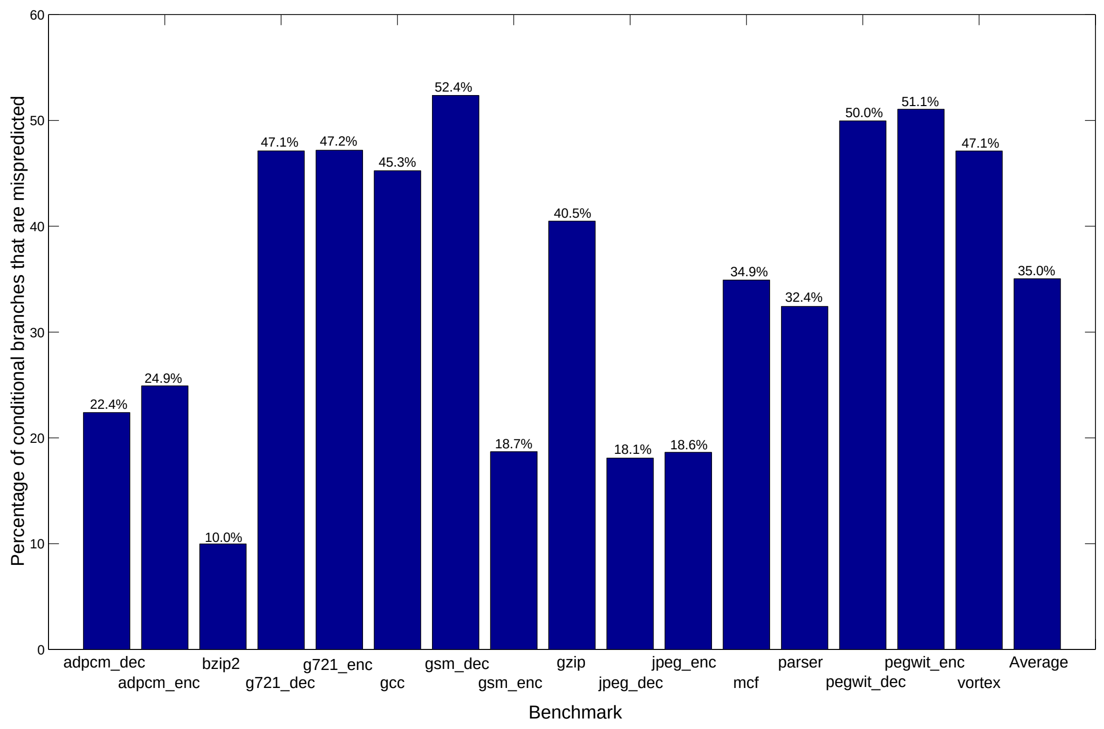


Figure 6-2: Percentage of conditional branches that are mispredicted for Tangerine Pekoe processor

is why the reorganization of the pipeline had little effect overall on bypass latch usage. This is probably due to the fact that we are scheduling instructions within a basic block. For our set of Tangerine Pekoe benchmarks, about 14% of all instructions on average are branches or jumps. This means that a basic block consists of at most 7 instructions on average. That leaves little freedom for our assembly-level scheduler to reorder instructions and take advantage of the bypass latches, especially since dependencies between instructions frequently exist which impose further restrictions. As a result, we can not take full advantage of the additional exposed state in the Tangerine Pekoe pipeline without scheduling across basic block boundaries. This indicates the need for operating at a higher level than the assembler in order to achieve the maximum benefit.

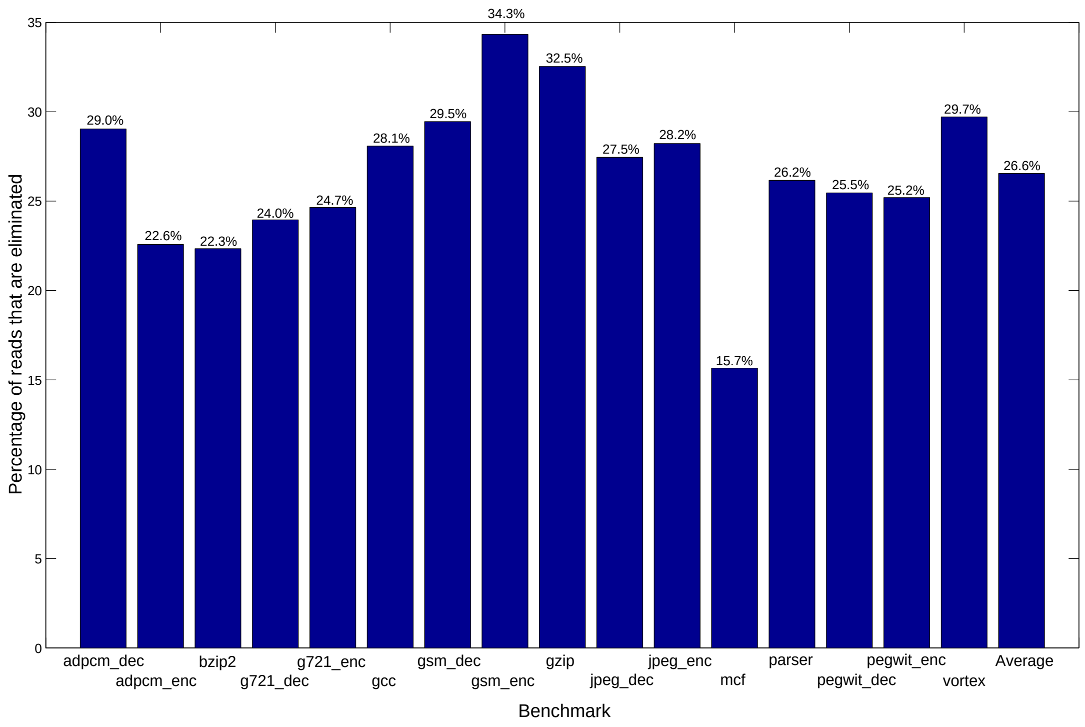


Figure 6-3: Percentage of reads that are eliminated for Tangerine Pekoe processor

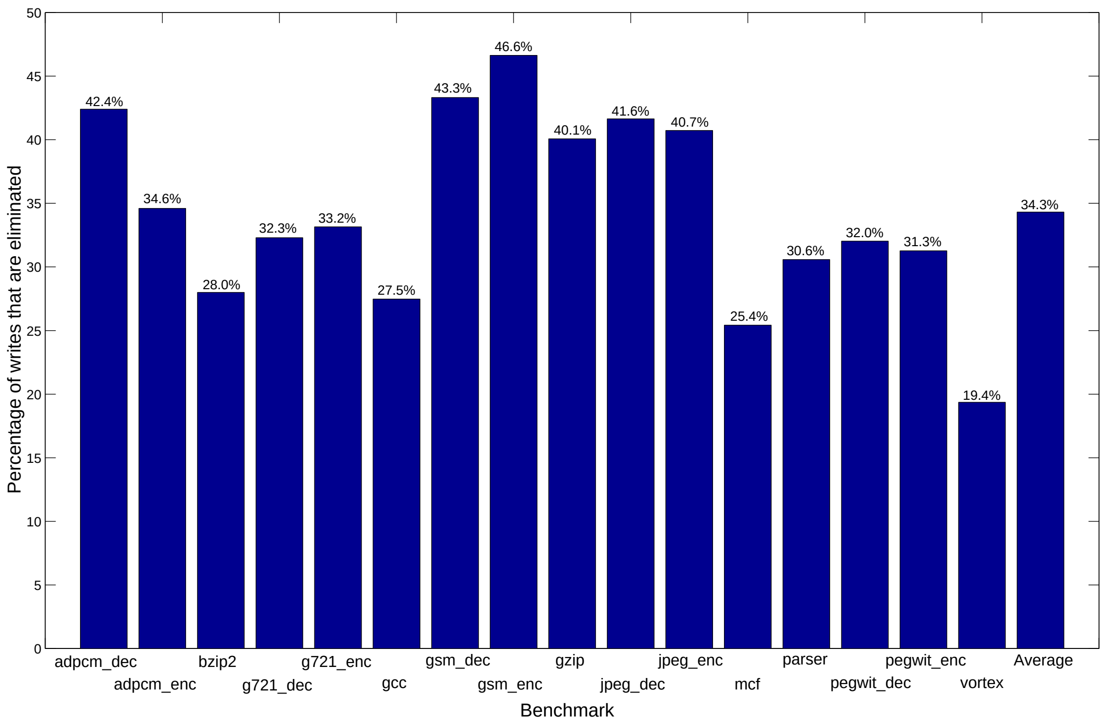


Figure 6-4: Percentage of writes that are eliminated for Tangerine Pekoe processor

6.2.3 Summary

Although the Tangerine Pekoe processor appears somewhat promising in terms of performance, the additional bypass latches do not lead to better results in terms of eliminating register file accesses. A scheduler that can operate across basic blocks might be able to achieve greater benefits. However, based on our current results, the additional hardware required for Tangerine Pekoe does not seem to be worth the advantages it provides. A definite conclusion can not be drawn until a full energy analysis is conducted.

Chapter 7

Conclusion

In this thesis we evaluated the feasibility of making datapath elements visible at the compiler level in order to reduce microprocessor energy consumption. For our initial experiments, we selected the register file and associated bypass network as targets for our study, as these elements are easily exposed to software and are typically responsible for a significant fraction of datapath power in today's microprocessors. We conducted a register lifetime analysis which indicated that a large percentage of register file reads and writes are unnecessary and can be avoided using information available at compile-time. The results of our analysis showed that the techniques of static read bypassing, static read caching, and elimination of register file writes in our baseline Vanilla Pekoe processor warranted further investigation.

Exposing additional machine state to the compiler introduced a new problem in the form of exception management overhead. Traditional hardware-based implementations of precise exception support are costly in terms of energy. We needed a method that would allow us to handle exceptions in a simple, energy-efficient manner. Our proposed solution to this problem was the use of software restart markers. This technique makes the compiler responsible for dividing code segments into idempotent restart regions, with a barrier instruction at the end of each region. If an exception such as a page fault occurs anywhere within the region, the kernel can restart the process at the beginning of the region. This allows for the use of temporary state which does not have to be preserved across exceptions. In addition, the processor can make permanent updates to stable state without consequences, as the idempotent nature of restart regions guarantees correct execution. Software restart markers solved our exception management overhead problem and made the exposing of bypass latches possible. A simple assembly-level implementation of restart markers in our energy-exposed Yellow Pekoe processor allowed us to create restart regions which contained about 3 instructions on average. We expect a higher-level analysis to allow us to place entire functions within a single restart region.

Software restart markers allowed us to make bypass latches visible at the compiler level by mapping them to temporary state. We made modifications to the compiler and assembler to support static read bypassing, static read caching, and elimination of register file writes. On average, we statically eliminated 28% of register file reads. This fell short of the 49% of register file reads that could be dynamically eliminated, but was still a significant figure, especially since many processors do not implement the techniques of bypass skip or dynamic read caching required to avoid reads using a hardware-based method. Even those processors that already implement dynamic elimination of reads could benefit from our techniques. For example, if the control logic necessary to implement bypass skip is on a processor's critical path, the cycle time could be decreased by switching to a static method of eliminating register file reads. Also, the hardware necessary to support bypass skip and dynamic read caching consumes additional energy, while our methods are software-based and therefore the only complexity added to the processor is the control logic necessary to recognize the use of a bypass latch. Thus, our static elimination of register file reads could be useful in many processor designs. We were also able to eliminate 34% of register file writes on average. When compared against a low-power processor that implemented bypass skip, our energy-exposed processor reduced energy consumption by an average of 7.0%.

A significant percentage of cycles in both our baseline and energy-exposed processors were due to interlocks caused by unfilled load-use delay slots. One way to remove these interlocks is to split the pipeline into two parts—one to handle memory operations and one to handle data computations. However, this introduces an address-generation delay slot and causes conditional branches to be resolved a cycle later. Overall, this new pipeline organization has the potential to improve performance. We surmised that it could also be attractive from an energy perspective, as the separation of address and data computations could reduce bitline switching energy. Additionally, the split pipeline would introduce more temporary state because of the increased number of bypass latches. We explored the implications of this latter point in our Tangerine Pekoe processor. Our results showed that although the performance of Tangerine Pekoe improved over that of Yellow Pekoe, the bypass latch usage actually slightly decreased. It appeared that our instruction scheduler needed to operate at a higher level than the assembler to take full advantage of the additional exposed state.

7.1 Future Work

Most of our modifications were made to the assembler and we operated at the granularity of a basic block. This conservative implementation achieved significant benefits; however, we expect that better results can be attained by operating at the compiler level. For example, compiler modifications could allow an entire function to be placed in a restart region. The usage of temporary state such as bypass latches could span basic blocks, possibly eliminating a greater number of register file accesses.

Another area for future work is to determine the effect of eliminating the restrictions that we placed on `gcc`. We modeled the bypass latches by reserving general-purpose registers. The implications of removing registers from `gcc`'s register allocation pool could be further explored. For example, if reducing the number of registers still results in comparable performance across a variety of benchmarks, future microprocessor designs could incorporate smaller register files, which could lead to faster performance and lower energy consumption. We also disabled branch delay slot filling in `gcc` because the register lifetime information was not correctly updated after this pass. If this problem was fixed, a direct comparison could be made between the Vanilla Pekoe and Yellow Pekoe programs.

We selected bypass latches as datapath elements to expose in this study, but there are other components that can be explored as well, such as instruction and data caches or other latches in the datapath. These elements could all conceivably be mapped to temporary state, and compile-time knowledge could be used to eliminate unnecessary microarchitectural operations.

Our examination of the split-pipeline Tangerine Pekoe processor was limited to measuring performance and the frequency of bypass latch usage. Conducting an energy analysis of the processor is warranted before determining whether the design should be used.

Although we focused on simple pipelines in our investigation, our techniques could be extended to apply to more complex processors, such as superscalar or out-of-order machines. The complexity of these processors tends to result in higher energy consumption, so exposing datapath elements to the compiler could have a greater impact.

We took an existing architecture and implementation and made various changes in our efforts to reduce energy consumption. Better results could be achieved by designing a new architecture with energy in mind as well as performance. An energy-exposed architecture would have low-power features built into it, and could be ideal for use in future embedded processors. One statistic of interest is the fact that 41% of instructions were in chains in our Yellow Pekoe processor. A new architecture could employ variable-length encodings to represent instruction chains as well as regular instructions. Ironically, computer architects have migrated from CISC architectures to RISC architectures in order to achieve higher performance; now, as we try to lower energy consumption, we advocate a shift back towards the direction of CISC architectures. To achieve a high-performance, low-power microprocessor, a hybrid CISC-RISC architecture may be ideal.

Bibliography

[1] K. Asanović, J. Beck, B. Irissou, B. E. D. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 vector microprocessor. In *Proceedings of Hot Chips VII*, August 1995.

[2] H. Corporaal. ILP architectures: trading hardware for software complexity. In *Proceedings of the 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 141–154, 1997.

[3] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 160–170, December 1997.

[4] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, December 1992.

[5] M. Golden and T. Mudge. A comparison of two pipeline organizations. In *Proceedings of the 27th annual international symposium on microarchitecture*, pages 153–161, November 1994.

[6] D. R. Gonzales. Micro-RISC architecture for the wireless market. *IEEE Micro*, 19(4):30–37, July/August 1999.

[7] J. L. Henning. Spec CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[8] P. Y. Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[9] Z. Hu and M. Martonosi. Reducing register file power consumption by exploiting value lifetime characteristics. In *Workshop on Complexity-Effective Design, 27th ISCA*, Vancouver, Canada, June 2000.

[10] Inmos. *The Transputer Data Book*, first edition, 1988.

[11] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *Proceedings of the 16th annual international symposium on computer architecture*, pages 281–289, May 1989.

[12] G. Kane. *MIPS RISC Architecture (R2000/R3000).* Prentice Hall, 1989.

[13] A. Klaiber. The technology behind Crusoe processors. White paper, Transmeta Corporation, January 2000.

[14] R. Krashinsky. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.

[15] R. Krashinsky, S. Heo, M. Zhang, and K. Asanović. SyCHOSys: Compiled energy-performance cycle simulation. In *Workshop on Complexity-Effective Design, 27th ISCA*, Vancouver, Canada, June 2000.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[17] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, December 1995.

[18] S. A. Mahlke, W. Y. Chen, W. m. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, Boston, Massachusetts, October 1992.

[19] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 125–135, December 1997.

[20] V. Oklobdzija. Architectural tradeoffs for low power. In *Power Driven Microarchitecture Workshop at ISCA98*, Barcelona, Spain, June 1998.

[21] E. Özer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings, and T. M. Conte. A fast interrupt handling scheme for VLIW processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 136–141, 1998.

[22] J. Scott. Designing the low-power M•CORE architecture. In *Power Driven Microarchitecture Workshop at ISCA98*, Barcelona, Spain, June 1998.

[23] R. L. Sites. *Alpha Architecture Reference Manual.* Digital Press, October 1992.

[24] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 1985.

[25] J. Tseng and K. Asanović. Energy-efficient register access. In *Proceedings of the 13th Symposium on Integrated Circuits and System Design*, pages 377–382, Manaus, Amazonas, Brazil, September 2000.

[26] W. Walker and H. G. Cragon. Interrupt processing in concurrent processors. *IEEE Computer*, 28(6):36–46, June 1995.

[27] R. Yung and N. Wilhelm. Caching processor general registers. In *Proceedings, 1995 International Conference on Computer Design*, pages 307–312, October 1995.