The Dissertation Committee for Francis Tseng
certifies that this is the approved version of the following dissertation:

Braids: Out-of-Order Performance with Almost In-Order Complexity

Committee:

Yale N. Patt, Supervisor

Craig M. Chase

Derek Chiou

Aloysius K. Mok

Burton J. Smith

James W. Tunnell

Braids: Out-of-Order Performance with Almost In-Order Complexity

by

Francis Tseng, B.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2007

To my parents.

Acknowledgments

My graduate school experience would not have existed if it were not for the support of my parents. They have always provided for me an example of hard work and perseverance. I would like to thank my wife Teresa and my family for putting up with my irregular work schedule and my seemingly never-ending work.

I am indebted to my advisor, Professor Yale N. Patt, for pushing me to be my best and being there as a friend when I needed to talk. Thank you for your advice and guidance over the years.

I would like to acknowledge past and present members of the HPS research group for providing a stimulating work environment. I would like to thank Robert Chappell for helping me to get started with research and being there to chat about anything. The first half of my graduate life would not have been as exciting without the friendship of Paul Racunas, Mary Brown, and Sangwook Kim. These people provided mentorship and encouragement. I would also like to thank Onur Mutlu, Moinuddin Quereshi, Hyesoon Kim, David Armstrong, and Kameswar Subramaniam for their friendship and also their helpful critiques. The more recent members of the HPS group all have engaged in discussions about research with me. These are Muhammad Aater Suleman, Jose Joao, Chang Joo Lee, Veynu Narasiman, Santhosh Srinath, David Thompson, Danny Lynch, Linda Hastings, Rustam Miftakhutdinov, Eiman Ebrahimi, and Khubaib.

I am thankful to the members of my Ph.D. committee for their commitment to serve on my committee. I would especially like to thank Derek Chiou for always being available to talk and for providing detailed comments on my dissertation.

I would also like to thank Mattan Erez for coming to my defense and providing helpful comments as if he was a committee member.

I would like to thank the mentors of my summer jobs for sharing their expertise. In particular, Burton Smith and Mitch Alsup helped me refine my research topic through many discussions.

Finally, I would like to thank Intel Corporation, Advanced Micro Devices, Cray, and HAL Computer Systems for providing me with summer jobs. Working at these companies has allowed me to gain a better understanding of processor design.

Braids: Out-of-Order Performance with Almost In-Order Complexity

Publication No. ____________________

Francis Tseng, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Yale N. Patt

There is still much performance to be gained by out-of-order processors with wider issue widths. However, traditional methods of increasing issue width do not scale; that is, they drastically increase design complexity and power requirements. This dissertation introduces the braid, a compile-time generated entity that enables the execution core to scale to wider widths by exploiting the small fanout and short lifetime of values produced by the program. A braid captures dataflow and register usage information of the program which are known to the compiler but are not traditionally conveyed to the microarchitecture through the instruction set architecture.

Braid processing requires identification by the compiler, minor augmentations to the instruction set architecture, and support by the microarchitecture. The execution core of the braid microarchitecture consists of a number of braid execution units (BEUs). The BEU is tailored to efficiently carry out the execution of a braid in an in-order fashion. Each BEU consists of a FIFO scheduler, a busy-bit vector, two functional units, and a small internal register file.

The braid microarchitecture provides a number of opportunities for the reduction of design complexity. It reduces the port requirements of the renaming mechanism, it simplifies the steering process, it reduces the area, size, and port requirements of the register file, and it reduces the paths and port requirements of the bypass network. The complexity savings result in a design characterized by a lower power requirement, a shorter pipeline, and a higher clock frequency. On an 8-wide design, the result from executing braids is performance within 9% of a very aggressive conventional out-of-order microarchitecture with the complexity of an in-order implementation.

Three bottlenecks are identified in the braid microarchitecture and a solution is presented to address each. The limitation on braid size is addressed by dynamic merging. The underutilization of braid execution resources caused by long-latency instructions is addressed by context sharing. The poor utilization of braid execution resources caused by single-instruction braids is addressed by heterogeneous execution resources.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The modern high-performance superscalar out-of-order processor is a restricted dataflow machine [53] which differs from a pure dataflow machine [26] in the following way. A restricted dataflow machine has a scope of processing that is limited to a window of instructions dictated by hardware constraints. This window slides along the sequential execution stream. Instructions enter and exit the window in an in-order fashion. Within this window, instructions execute as soon as their sources are ready in an out-of-order fashion. The size of the execution window and the capability of out-of-order processing are governed by the size of structures in the execution core of the processor. As the size of the structures and the number of elements that can be processed every cycle increase, the execution engine can exploit more instruction-level parallelism [7], allowing the restricted dataflow machine to more closely resemble a pure dataflow machine.

Figure 1.1 shows the potential performance that is available at wider issue widths in an aggressive conventional out-of-order processor, assuming the use of a perfect branch predictor and perfect instruction and data caches. It is not unrealistic to assume that future processors will have more accurate branch predictors and also larger caches. Thus, Figure 1.1 provides a measure of the available untapped performance. The baseline for this experiment is a 4-wide conventional out-of-order design. An 8-wide design achieves an average increased performance of 44% over the baseline, and a 16-wide design achieves an average increased performance of 83% over the baseline. Some applications such as crafty, vpr, and mgrid show a performance improvement of 200% when issue width increases from 4 to 16.

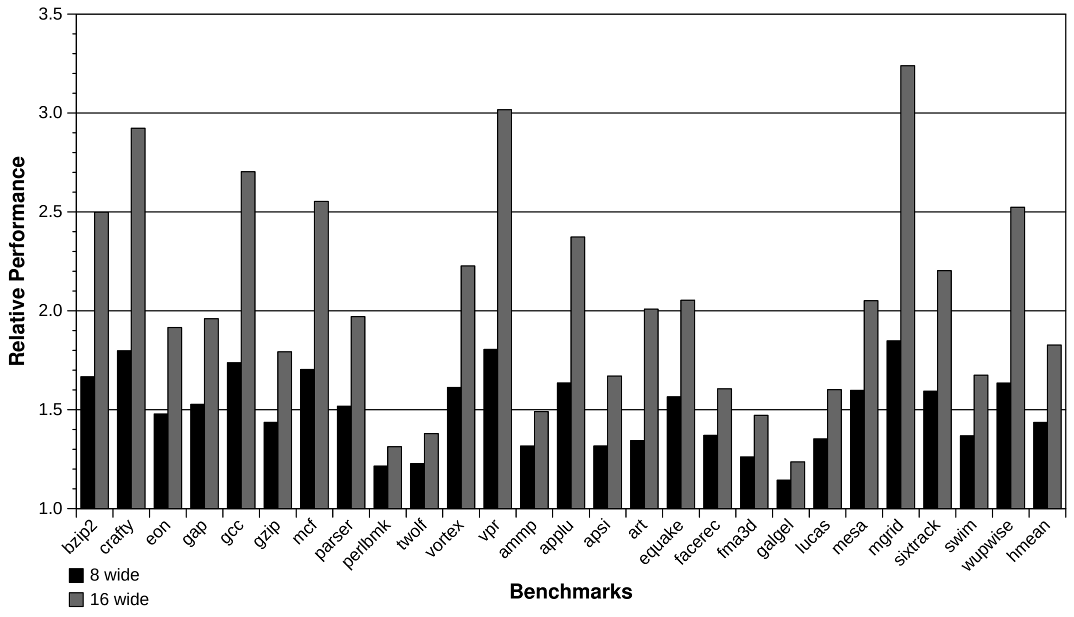


Figure 1.1: Potential Performance of 8-Wide and 16-Wide Designs Over a 4-Wide Conventional Out-of-Order Design Using Perfect Branch Prediction and Perfect Caches

Although Figure 1.1 shows that performance is plentiful at wider issue widths, there are design implications that make such approaches infeasible. The traditional method of increasing issue width does not scale. Increasing issue width beyond that of current processors drastically increases the design complexity [50] and power requirements [79] of the execution core of the processor. A processor with a wider issue width requires structures with more entries in the execution core to support the processing of more in-flight instructions. In particular, these structures include the instruction scheduler, the register file, and the bypass network. All of these structures pose a problem because they do not scale. As the issue width increases, these structures require more entries and ports. In turn, they take up a significantly larger chip area and consume more power. Large structures are pipelined because a signal cannot travel across the length of the structure at the expected frequency. This further increases design complexity.

The instruction scheduler is a structure in a critical loop of execution [50]. The scheduler contains many entries in order to identify instructions that are ready to execute. Pipelining the scheduler creates unavoidable pipeline bubbles in the scheduling process causing significant performance loss [12]. Although pipelining the instruction scheduler without performance loss has been proposed [69], it is not possible to do so without adding additional complexity to the already complex instruction scheduler. The register file is another crucial structure required for high performance. It is the storage for the results of in-flight instructions. Therefore, more entries are needed to support more in-flight instructions. As issue width increases, more ports are added to the register file to support the higher reading and writing bandwidth. Increasing the number of entries and ports of the register file requires more area and energy [78]. Doubling the number of register ports doubles the number of bit-lines and the number of word-lines, causing a quadratic increase in area [29]. A larger register file requires accesses to be pipelined over multiple cycles [34]. Pipelined accesses in turn require a multilevel bypass network to provide values that have been written to but are not yet available from the register file. Each level of the bypass network is a complex point-to-point network. It has been shown that limiting the bypass network can lead to detrimental performance loss [4].

Because of the difficulties associated with scaling the issue width, there have been fewer introductions of commercial aggressive out-of-order superscalar processors at wider issue widths. The issue width of recent high-performance processor introductions has been four. This includes the Intel Core Duo and Core 2

Duo [74]. 8-wide designs have been introduced but at limited frequencies. The AMD Barcelona has a frequency of 2GHz, and the IBM POWER4 and POWER5 [36] have a frequency of 1.9GHz and 2.2GHz, respectively. In the POWER6, IBM implemented an in-order microarchitecture. In order to make use of the greater number of transistors provided by improvements in process technology, designers have looked for alternative forms of parallelism for improving performance. Rather than designing wider processors, designers duplicate cores on the same chip. Today, most general purpose processors introduced are multi-core processors. Although multi-core processors exploit program-level and thread-level parallelism, there is still much to be gained in the performance of single-core designs as shown in Figure 1.1 if designers can get past the complexity issues and power requirements. If designers can improve single-core performance, then multi-core performance can improve as well. A multi-core processor is simply a single-core processor replicated multiple times.

1.1 The Solution

It is clear that the traditional method of increasing issue width is impractical to implement in future high-performance processors. One solution to this problem is to leverage the compiler. If the compiler can provide the information that the microarchitecture needs to execute a program, the microarchitecture does not have to work as hard to achieve the same goals.

Unlike the limited program scope of the hardware, the compiler has an unrestricted view of the entire program. The compiler can easily analyze program dataflow which is composed of an ordered set of instructions and a set of values that are passed among instructions. However, the information known to the compiler is not conveyed to the microarchitecture in an efficient manner due to the syntax limitations of conventional instruction set architectures. Aggressive out-of-order processors reconstruct a limited view of the program dataflow at runtime using structures in the execution core. The more entries and wider the structures are, the more complete the picture of program dataflow is. Figure 1.2a shows the conventional processing paradigm. The thickness of the lines indicates the amount of effort exerted at that level. In this paradigm, the compiler passes limited dataflow information to the microarchitecture. The microarchitecture then extracts parallelism from the limited dataflow information using runtime structures in the execution core.

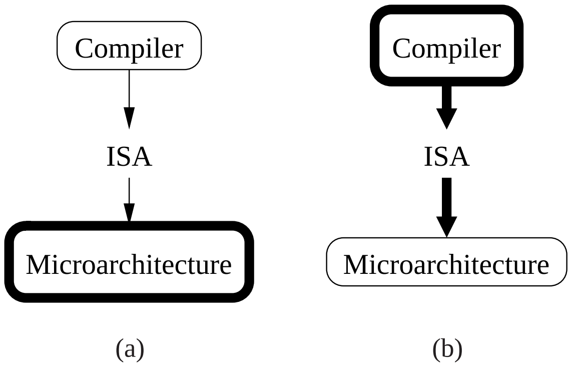


Figure 1.2: Processing Paradigms (a) Conventional: Microarchitecture Does Most of the Work (b) Braid: Compiler Does Most of the Work

This dissertation introduces the braid, a compile-time generated entity which simplifies the design complexity of structures in the execution core of a high-performance processor. A braid is a dataflow subgraph of the program that resides solely within a basic block. It exploits the small fanout and short lifetime of values communicated among instructions. Figure 1.2b shows the braid processing paradigm. In this model, the compiler exerts more effort at compile time and passes more information about the program dataflow to the microarchitecture. This lessens the effort of the microarchitecture to extract program parallelism at runtime and eliminates the need for complex hardware structures in the execution core. Braids enable the use of simple instruction schedulers, reduce the size, ports, and accesses to the register file, and also reduce the ports and values sent to the bypass network. The braid microarchitecture shortens the pipeline reducing the branch misprediction penalty. The reduction in complexity by avoiding large associative structures also enables a higher clock frequency. Traditionally, researchers in the architectural community have avoided making changes to the instruction set architecture. This dissertation shows that with simple augmentations to the instruction set architecture, braids can be easily conveyed to and leveraged by the microarchitecture.

1.2 Thesis Statement

Compiler-identified braids can be efficiently conveyed through the instruction set architecture to the microarchitecture, simplifying the design complexity of elements on the critical path in the execution core of the microarchitecture while approximating the performance of an aggressive out-of-order design.

1.3 Contributions

This dissertation makes the following contributions.

- This dissertation introduces the braid which enables the compiler to convey useful program dataflow and value information to the microarchitecture allowing the compiler to play a greater role in processing the program. The microarchitecture that implements braids is able to achieve performance close to that of an aggressive out-of-order design while maintaining the design complexity closer to that of an in-order design.

- This dissertation discusses the compiler, the instruction set architecture, and the microarchitecture modifications required to implement braids. First, this dissertation shows how braids can be identified by the compiler, or by performing profiling analysis and binary translation. Second, this dissertation shows the instruction set architecture augmentations needed to convey braids to the microarchitecture. Third, this dissertation proposes a microarchitecture to efficiently carry out the processing of braids. The design parameters needed for a balanced design are analyzed.

- Three bottlenecks are identified in the braid microarchitecture and a solution is presented to address each. First, the limitation on braid size is addressed by dynamic merging. Second, the underutilization of braid execution resources caused by long-latency instructions is addressed by context sharing. Third, the poor utilization of braid execution resources caused by single-instruction braids is addressed by heterogeneous execution resources.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 characterizes the use of program values, introduces the concept of braids, and analyzes braid characteristics. Chapter 3 presents the compiler, instruction set architecture, and microarchitecture requirements for processing braids. Chapter 4 describes the simulation infrastructure used in this dissertation and analyzes the performance results of processing braids. Chapter 5 suggests three optimizations to address three bottlenecks of the braid microarchitecture. Chapter 6 analyzes the hardware and software complexity of the braid microarchitecture. Chapter 7 compares and contrasts the braid microarchitecture with other processing paradigms and proposals. Chapter 8 concludes and discusses the future research directions of processing braids.

Chapter 2

The Braid

The compiler has an unrestricted view of the entire program dataflow graph at compile time. It also has precise knowledge about the usages and lifetime of each value in the program. Rather than coming up with a pure hardware solution to simplify an execution core that can support wider issue widths, the compiler is used to expose program dataflow and value information to the microarchitecture. To understand how the characteristics of program values can be exploited for more efficient processing by the microarchitecture, this chapter characterizes the usage of values of programs and provides insight into how the program dataflow graph can be broken into dataflow subgraphs. A braid is defined, and an example is provided. Braids found in programs are then characterized.

2.1 Motivation

A program is an implementation of an algorithm in a programming language. It can be described as a dataflow graph consisting of nodes representing instructions and arcs representing values that are passed among instructions. The entire dataflow graph of a program is very irregular, consisting of arbitrary arcs that connect nodes together. However, it will be shown that when the dataflow graph is divided into smaller dataflow subgraphs, some structure and patterns are revealed.

2.1.1 Distribution of Value Fanout

An experiment was performed to analyze the characteristics of values for all the programs in the SPEC CPU2000 benchmark suite compiled for the Alpha ISA. Figure 2.1 plots the distribution of the dynamic fanout of values produced in the benchmark programs. The fanout of a value is defined as the number of times the value is read. Most values are read infrequently. On average, over 70% of the values are used only once, and about 90% of the values are used at most twice. Values that are used more than three times represent only about 5% of all values.

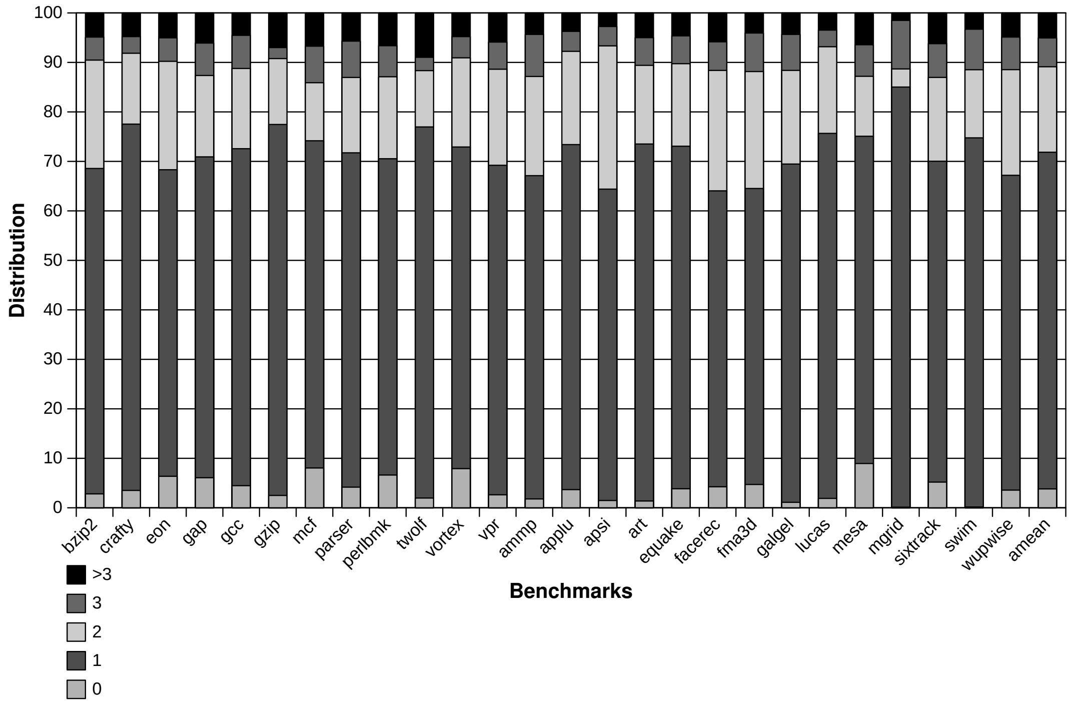


Figure 2.1: Distribution of Value Fanout

About 4% of the values are produced but not used. These instances are caused by the conservative nature of the compiler. The compiler produces a value to be used along a specific control-flow path. However, that control-flow path was not traversed at runtime causing the value not to be read.

Infrequently-used values exist for a number of reasons. First, the process of moving values to and from the stack at function call boundaries creates infrequently used values. Moving values is required since the compiler defines a standard interface for supporting parameter passing for function calls. Second, certain compiler optimizations and transformations can increase the number of infrequently-used values. One such example is common sub-expression elimination. Figure 2.2a shows an unmodified code sequence. In this example, the expression $r1 + r2$ is a common subexpression. Instances of this expression are replaced with X shown in Figure 2.2b. This transformation converts registers $r1$ and $r2$ into single-use values. Third, and perhaps the most common reason why many infrequently-used values exist has to do with the specification of instructions in a general-purpose ISA. The compute form of an instruction generally consists of two inputs and one output. Since the instruction encoding space is limited, it would be impossible to assign an instruction for every possible operation. Complex operations must be constructed from a sequence of instructions as shown in Figure 2.3. The output from one instruction is read as the input of another. Passing values between instructions of complex operations makes them temporary.

2.1.2 Distribution of Value Lifetimes

Another characteristic of a value is its lifetime. This is defined as the number of instructions between the producer and consumer of a value. Figure 2.4 shows the distribution of the dynamic lifetime of values produced in the benchmark programs. On average, about 80% of values have a lifetime of 32 instructions or fewer. This translates to four or fewer processor cycles on a processor that fetches eight instructions per cycle.

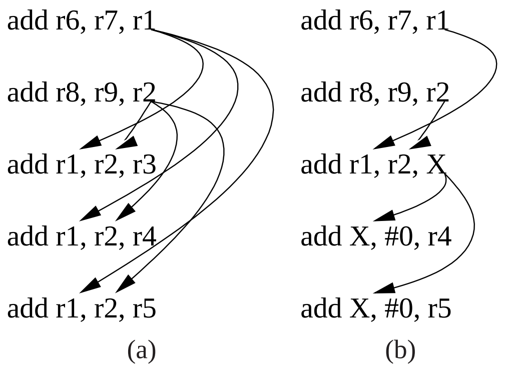


Figure 2.2: Example of Common Sub-Expression Elimination (a) Original Code Sequence (b) Optimized Code Sequence

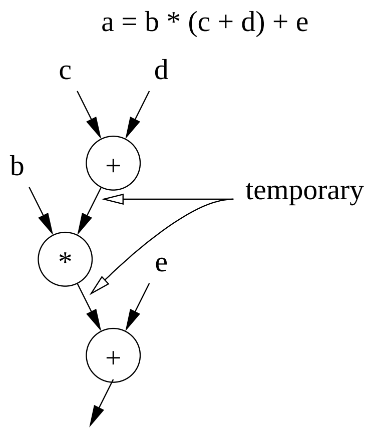


Figure 2.3: Composition of a Complex Operation

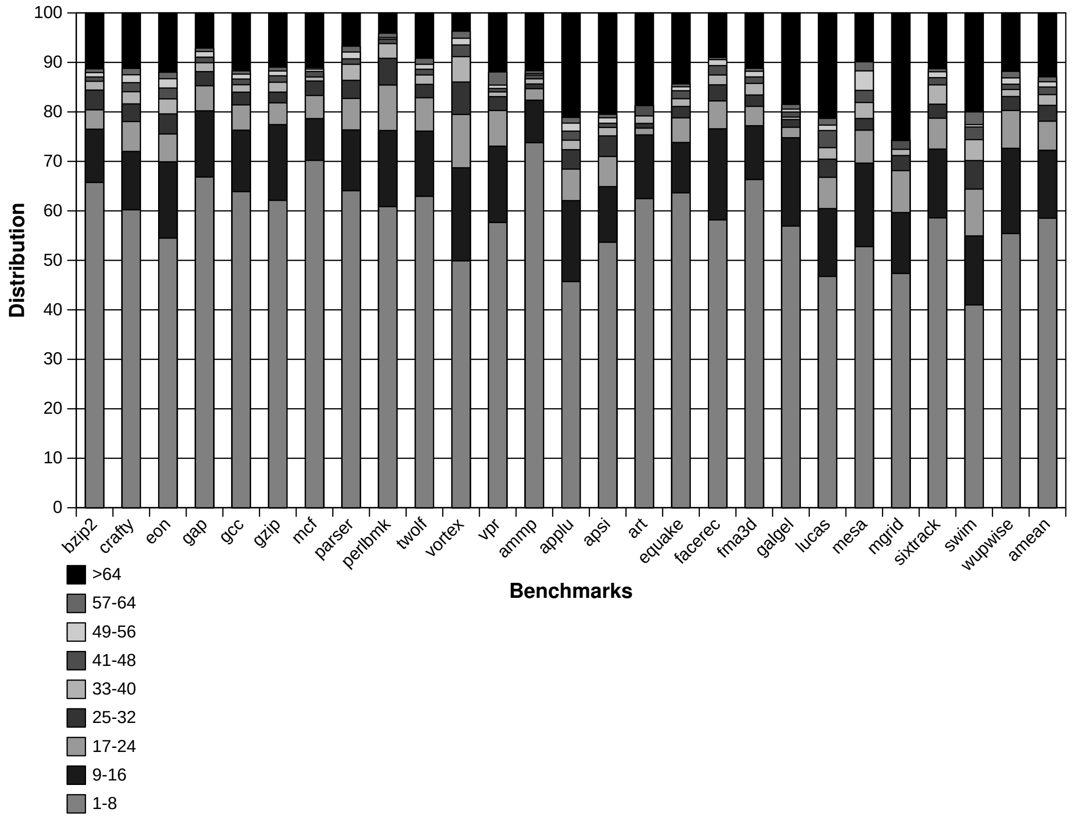


Figure 2.4: Distribution of Value Lifetime

Figures 2.1 and 2.4 show that most values have a small fanout and a short lifetime. This suggests the entire irregular dataflow graph of the program has some regularity to it. To exploit the regular dataflow subgraphs of the program, the compiler partitions the dataflow graph of the program into entities that can be more easily processed by the microarchitecture. Although some researchers have characterized the behavior of values [31] [17], they have not partitioned the program dataflow graph based on these realizations.

2.2 The Braid

A braid is a dataflow subgraph of a program contained wholly within a basic block. A braid has instructions with external inputs and external outputs. These values are communicated with instructions from other braids. A braid also has internal values. These values are communicated among instructions within the braid. Since a braid is a dataflow subgraph, it has an instruction-level parallelism that is generally greater than one.

Figure 2.5a shows a snippet of C source code from the `life_analysis` function in the gcc benchmark program from the SPEC CPU2000 benchmark suite. Figure 2.5b shows the assembly code of the basic block corresponding to lines 1 through 8 of the source code. The three different color shades identify three disjoint dataflow subgraphs within the basic block. A dataflow graph of the corresponding assembly code is shown in Figure 2.5c. Again, the three different color shades identify the same three disjoint dataflow subgraphs. The arrows indicate data dependencies where solid lines represent values communicated internally within the dataflow subgraph, and dashed lines represent values communicated externally to and from the dataflow subgraph. Each of these dataflow subgraphs represents a different braid. Thus, the basic block in this example is partitioned into three braids.

Each braid corresponds to one operation being performed in the high-level source code. The assembly code of braid 1 corresponds to the work being done by lines 4 through 8 of the source code. This braid has six external inputs, one external output, and eleven internal values. The assembly code of braid 2 corresponds to the increment of the induction variable in the `for` statement. This braid has one external input, one external output, and one internal value. Braid 3 consists of a single lda instruction with one external input, one external output, and no internal values. It is a single-instruction braid.

2.3 Braid Characteristics

Figure 2.5b shows that a basic block contains instructions for specifying one or more high-level operations in the source code. Since each one of these high-level operations corresponds to a braid, one or more braids exist within a basic block. The first characteristic of braids measured is the number of braids per basic block for all the SPEC CPU2000 benchmark programs. This is shown in Table 2.1. On average, the basic block of the integer benchmark programs consists of 2.8 braids, and the basic block of the floating point benchmark programs consists of 4.2 braids. These numbers are skewed by the presence of single-instruction braids. When single-instruction braids are factored out, the average number of braids per basic block falls to 1.15 and 1.89 for the integer and floating point benchmark programs respectively. This is shown in the column labeled with an asterisk. Single-instruction braids account for 14% of all instructions. 12% of the single-instruction braids are branch and NOP instructions. These single-instruction braids are present due to the experimental analysis with preexisting program binaries generated by a non-braid-aware compiler.

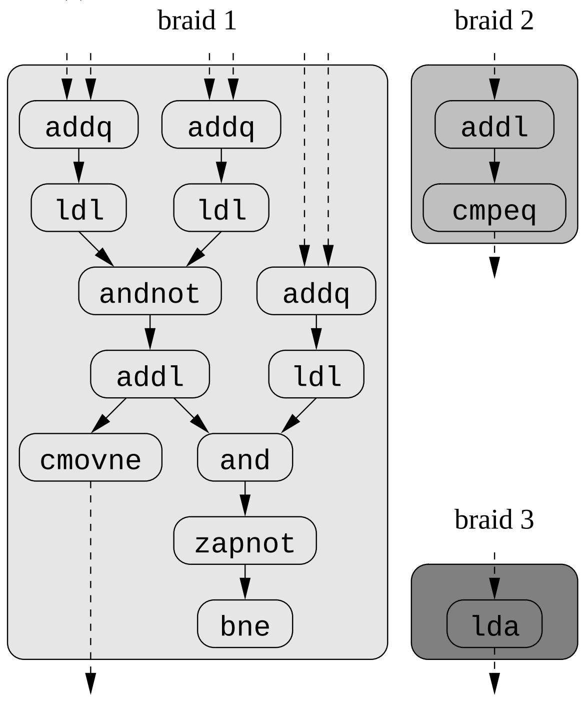
To put the number of braids per basic block in perspective, Table 2.2 shows

```
for (j = 0; j < regset_size; j++)
  {
    register REGSET_ELT_TYPE x
      = (basic_block_new_live_at_end[i][j]
         & ~basic_block_live_at_end[i][j]);
    if (x)
      consider = 1;
    if (x & basic_block_significant[i][j])
      {
        must_rescan = 1;
        consider = 1;
        break;
      }
  }
```

(a)

```
0x10  addq a1, t4, t0
0x14  addq a0, t4, t1
0x18  addq t8, t4, t2
0x1c  ldl t3, 0(t0)
0x20  addl t5, #1, t5
0x24  ldl t0, 0(t1)
0x28  cmpeq t9, t5, t7
0x2c  ldl t1, 0(t2)
0x30  lda t4, 4(t4)
0x34  andnot t3, t0, t0
0x38  addl zero, t0, t0
0x3c  and t0, t1, t1
0x40  zapnot t1, #15, t1
0x44  cmovne t0, #1, t6
0x48  bne t1, target
```

(b)



(c)

Figure 2.5: Example Braids (a) Snippet of C Source Code from the `life_analysis` Function in gcc (b) Assembly Code of Basic Block (c) Dataflow Graph of Basic Block

Integer		
benchmarks	braids	braids*
bzip2	2.6	1.31
crafty	2.6	1.13
eon	4.2	1.44
gap	2.4	1.16
gcc	2.4	0.95
gzip	2.6	1.23
mcf	2.0	0.62
parser	2.7	1.00
perlbmk	2.8	1.25
twolf	3.0	1.31
vortex	3.5	1.40
vpr	2.8	1.03
amean	**2.8**	**1.15**

Floating Point		
benchmarks	braids	braids*
ammp	2.0	1.25
applu	6.2	2.08
apsi	5.0	1.76
art	2.9	1.30
equake	2.5	1.22
facerec	2.7	1.11
fma3d	2.8	1.18
galgel	5.8	2.10
lucas	4.0	1.95
mesa	2.8	1.17
mgrid	7.6	5.25
sixtrack	3.1	1.20
swim	8.2	3.45
wupwise	3.7	1.40
amean	**4.2**	**1.89**

Table 2.1: Braids per Basic Block, *Braids per Basic Block Ignoring Single-Instruction Braids

the average number of instructions in a basic block for all the benchmark programs. On average, the integer programs have basic blocks consisting of 7.0 instructions, and the floating point programs have basic blocks consisting of 14.3 instructions. The streaming nature of the floating point benchmark programs causes their basic block to be larger.

Integer	
benchmarks	instructions
bzip2	8.3
crafty	8.0
eon	8.4
gap	6.0
gcc	5.5
gzip	8.7
mcf	3.9
parser	5.8
perlbmk	6.5
twolf	8.7
vortex	7.2
vpr	7.1
amean	**7.0**

Floating Point	
benchmarks	instructions
ammp	5.5
applu	16.9
apsi	13.1
art	7.6
equake	6.2
facerec	5.8
fma3d	7.7
galgel	11.7
lucas	17.2
mesa	5.9
mgrid	53.3
sixtrack	7.1
swim	31.8
wupwise	10.2
amean	**14.3**

Table 2.2: Instructions per Basic Block

The next characteristic of braids measured is size and width. The size of a braid is computed by counting the number of instructions in the braid. The width of a braid measures the instruction-level parallelism of a braid. It is computed by dividing the size of a braid by the number of instructions along the longest dependency chain, also known as the critical path of a braid. Table 2.3 shows the size and width of braids for all the benchmark programs. On average, the braids in the integer benchmark programs have a size of 4.7 instructions and a width of 1.1.

The braids in the floating point benchmark programs have a size of 5.4 instructions and a width of 1.4 instructions. Since the floating point benchmark programs have larger basic blocks, it is not surprising to find larger braid sizes. It is encouraging to see the average width of the braids in the floating point benchmark programs remains similar to that of the braids in the integer benchmark programs.

Integer		
benchmarks	size	width
bzip2	5.4	1.1
crafty	5.8	1.2
eon	4.0	1.2
gap	4.1	1.1
gcc	4.3	1.1
gzip	6.0	1.1
mcf	4.1	1.0
parser	4.2	1.1
perlbmk	3.9	1.2
twolf	5.3	1.1
vortex	3.7	1.2
vpr	5.1	1.2
amean	**4.7**	**1.1**

Floating Point		
benchmarks	size	width
ammp	3.8	1.1
applu	6.2	1.5
apsi	5.6	1.4
art	4.6	1.0
equake	4.0	1.1
facerec	3.8	1.2
fma3d	5.1	1.2
galgel	3.8	1.1
lucas	7.7	2.0
mesa	3.7	1.1
mgrid	9.7	3.5
sixtrack	4.3	1.2
swim	7.8	1.6
wupwise	5.6	1.3
amean	**5.4**	**1.4**

Table 2.3: Braid Size and Width

Another characteristic of braids measures the number of braid dependencies. Table 2.4 shows the number of internal values, external inputs, and external outputs of braids. This result provides a sense of the potential communication savings that can be provided by the braid microarchitecture. On average, the integer benchmark programs have 4.0 internal values, 2.3 external inputs, and 1.0 external output. The number of inputs and outputs is not unlike the operand specifications of a two-source compute instruction. The floating point benchmark programs have

7.6 internal values, 4.0 external inputs, and 1.1 external outputs. The benchmark programs lucas, mgrid, and swim have a significantly larger number of external inputs than the rest of the benchmark programs. If these programs are removed from the calculation of the arithmetic mean, floating point benchmark programs have an average of 4.3 internal values, 2.1 external inputs, and 0.9 external outputs. This is shown in the last row indicated by the asterisk. The number of external inputs and outputs is very similar to that of the integer benchmark programs.

Integer			
benchmarks	internal values	external inputs	external outputs
bzip2	5.5	2.7	1.1
crafty	5.6	2.6	1.0
eon	3.3	2.1	0.9
gap	3.3	1.9	0.9
gcc	3.4	2.0	0.8
gzip	5.5	3.3	1.3
mcf	3.1	2.4	0.9
parser	3.3	2.2	0.9
perlbmk	3.2	1.8	1.1
twolf	5.0	2.6	0.9
vortex	2.7	1.9	0.8
vpr	4.3	2.5	1.3
amean	**4.0**	**2.3**	**1.0**

Floating Point			
benchmarks	internal values	external inputs	external outputs
ammp	3.1	2.2	0.8
applu	7.2	3.9	0.8
apsi	6.5	3.8	1.1
art	3.7	3.0	0.7
equake	3.2	2.1	0.8
facerec	3.4	2.3	1.0
fma3d	5.0	2.6	1.0
galgel	3.0	3.2	0.8
lucas	10.7	4.8	1.4
mesa	2.8	2.1	0.7
mgrid	34.0	12.8	3.4
sixtrack	3.6	2.5	1.0
swim	14.8	7.5	0.9
wupwise	6.0	2.6	1.0
amean	**7.6**	**4.0**	**1.1**
amean*	**4.3**	**2.1**	**0.9**

Table 2.4: Internal and External Braid Inputs and Outputs, *Arithmetic Mean without lucas, mgrid, and swim

Chapter 3

Implementation

Braid processing requires the compiler to play a key role in orchestrating the execution of a program. This chapter discusses the changes required at three levels of transformation to efficiently implement braid processing. First, the compiler must identify braids from the program dataflow. Second, the instruction set architecture must convey braid information from the compiler to the microarchitecture. Third, the microarchitecture must support braids to exploit its characteristics.

3.1 Compiler Requirements

A braid is an entity that is identified at compile time. Braids do not span basic block boundaries. This restriction maintains implementation simplicity and avoids unnecessary code duplication. Allowing braids to span control-flow boundaries introduces a set of problems due to the existence of control flow merge and fork points. Suppose at compile time, a braid was formed across two consecutive basic blocks. At runtime, if the second basic block did not follow the first basic block, then the operands specified as external and internal by the compiler would most likely no longer be valid. By restricting braids to reside completely within a single basic block, problems associated with the changing runtime control-flow path can be avoided.

3.1.1 Profiling and Translation

This dissertation used binary profiling analysis and binary translation to identify and specify braids. This approach can be used if source code is not readily available or if recompilation is not feasible. The end result of performing binary profiling and translation is a braid-annotated binary comparable to what is produced by a braid-aware compiler. Figure 3.1 shows a diagram of the profiling and translation workflow. In this diagram, black arrows indicate the ordering of steps for the profiling and translation process, and white arrows indicate information that is passed between steps. Four steps are carried out to generate a braid-annotated binary that will be used by the microarchitecture simulation: basic block analysis, register usage analysis, braid identification, and binary translation. The first three steps require profiling, and the fourth step requires translating.

The first step of binary profiling is basic block analysis. This step identifies the basic blocks generated by the compiler by identifying basic block boundaries. Basic block identification requires two passes. In the first pass, the program binary is scanned instruction by instruction to determine the static targets of control-flow instructions. In the second pass, the program is profiled to determine the dynamic targets of control-flow instructions. Both passes are necessary to generate a more complete breakdown of the program into basic blocks. The first pass is necessary because a branch may redirect control into the middle of a block of code. Without the first pass, a block of code will look like one large basic block instead of two smaller ones. The second pass is necessary because static analysis cannot identify the targets of indirect branches. Basic block analysis is necessary to enable the formation of braids since a braid must reside completely within a basic block. The information gathered during basic block analysis is written to the block database.

The second step of binary profiling is register usage analysis. This step

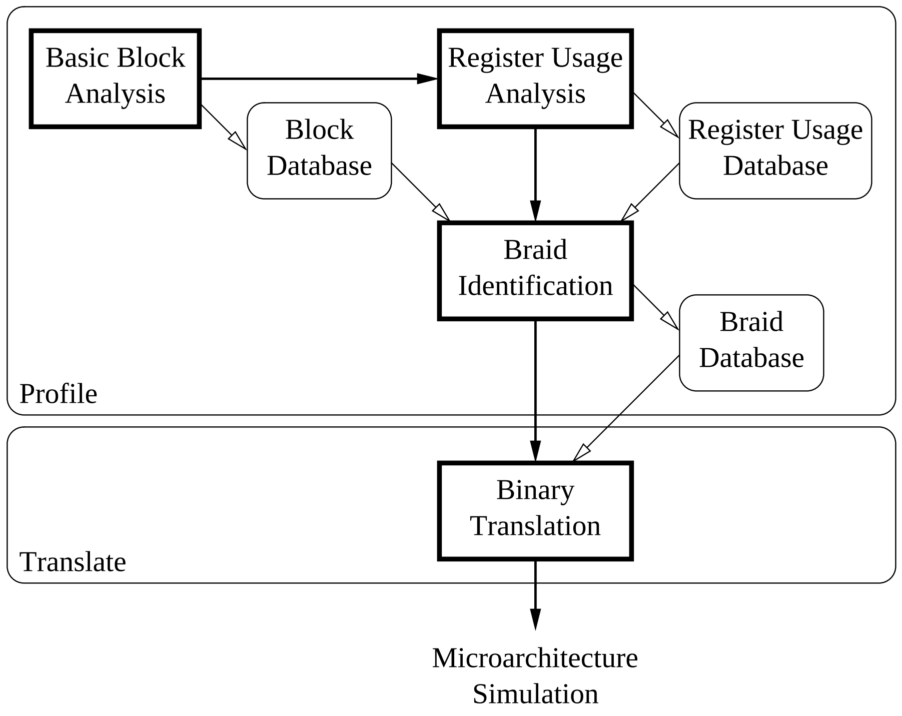


Figure 3.1: Profiling and Translation Workflow

reconstructs the dataflow graph of the program, mimicking the dataflow analysis phase of the compiler. As each instruction is encountered in the profiling process, a data structure associated with each operand of the instruction tracks its production or use. For each destination operand, all of its consumers are logged. For each source operand, all of its producers are logged. The information gathered during register usage analysis is written to the register usage database.

The third step of binary profiling is braid identification. Using the information contained in the block database from the first step and the register usage database from the second step, this step partitions the dataflow graph of a basic block into dataflow subgraphs. A graph coloring algorithm is applied to the instructions in the basic block. All instructions within the basic block do not have a color associated with them at the start of the algorithm. The first instruction without a color in the basic block is located and a color is associated with it. Next, the instruction with a color propagates its color to all of its children and parent instructions in the basic block. The children and parent instructions are identified using the information gathered during the register usage analysis. The propagation of color does not propagate to instructions beyond the basic block boundaries identified during basic block analysis. Continuing the algorithm, each colored instruction propagates its color to its children and parents until the entire dataflow subgraph rooted from the original instruction in the basic block is colored. The set of instructions so colored identifies exactly one braid in the basic block. Using a new color, this algorithm repeats with the next instruction without a color. The end result is the identification of another braid inside the basic block. The algorithm terminates when all the instructions within the basic block are associated with a color. After the set of braids is identified in the basic block, the braid information is to the braid database. This step concludes the last profiling step.

The next step of the workflow is binary translation. This step does not require profiling. This step sorts the braids within each basic block and annotates them to encode braid information using the information from the braid database generated in the last profiling step. First, braids are sorted. This involves rearranging instructions within the basic block such that instructions belonging to the same braid are laid out as a consecutive sequence of instructions within the basic block. Having a set of sorted braids greatly simplifies various pipeline operations. Figure 3.2a show the original code, and Figure 3.2b show the same code sorted by braid. Second, each instruction is annotated with a bit indicating whether the instruction begins a new braid. Third, each operand in the instruction is annotated with the proper bits indicating whether that operand sources the internal register file or the external register file. Next, register name rewriting is performed separately for the external and internal registers. Register name rewriting is performed for the external registers across the entire program. It is also performed for the internal registers of a braid for each braid in the program. After the binary is modified, the result is a braid-enabled program binary capable of being processed by the braid microarchitecture.

The last instruction slot of a basic block is always reserved for the control-flow instruction of a basic block if there is one. This is accomplished by rearranging braids such that the braid containing the control-flow instruction is ordered last in the basic block. This requirement preserves the block as one basic block. Furthermore, this eliminates the requirement to modify branch offsets.

3.1.2 Issues with Profiling Analysis

The size of braids can be restricted by two conditions. First, the braid microarchitecture supports a fixed number of internal registers for each braid. There-

```
0x10  addq a1, t4, t0
0x14  addq a0, t4, t1
0x18  addq t8, t4, t2
0x1c  ldl t3, 0(t0)
0x20  addl t5, #1, t5
0x24  ldl t0, 0(t1)
0x28  cmpeq t9, t5, t7
0x2c  ldl t1, 0(t2)
0x30  lda t4, 4(t4)
0x34  andnot t3, t0, t0
0x38  addl zero, t0, t0
0x3c  and t0, t1, t1
0x40  zapnot t1, #15, t1
0x44  cmovne t0, #1, t6
0x48  bne t1, target
```

(a)

```
0x10  addl t5, #1, t5
0x14  cmpeq t9, t5, t7
0x18  lda t4, 4(t4)
0x1c  addq a1, t4, t0
0x20  addq a0, t4, t1
0x24  addq t8, t4, t2
0x28  ldl t3, 0(t0)
0x2c  ldl t0, 0(t1)
0x30  ldl t1, 0(t2)
0x34  andnot t3, t0, t0
0x38  addl zero, t0, t0
0x3c  and t0, t1, t1
0x40  zapnot t1, #15, t1
0x44  cmovne t0, #1, t6
0x48  bne t1, target
```

(b)

Figure 3.2: Code Scheduling (a) Original Code Schedule (b) Code Schedule Sorted by Braid

fore, the number of active internal operands of a braid must not exceed the number of supported internal registers. As instructions are incorporated into a braid, the working set size of the internal operands increases. Since register usage analysis is performed, the profiling tool knows whether an instruction is the last consumer of an operand. Knowing this allows the profiling tool to release an internal register by allowing it to be written by another instruction. When the number of active internal operands exceeds the number of internal registers, the braid is artificially split into two braids. A split braid caused by this condition accounts for about 2% of the braids analyzed. This situation is an artifact of performing profiling analysis and binary translation on preexisting program binaries. A braid-aware compiler can solve this problem by morphing the dataflow graph via software transformations.

Second, since the sorting of braids within the basic block rearranges instructions, memory instructions can be reordered. This can lead to memory dependency violations because memory order may be violated. Most of the memory instructions access the stack. Identifying the aliasing of these operations is easy because these memory instructions use the stack pointer as a base register. For example, Figure 3.3 shows a store-load pair that the profiling tool can identify as a memory dependency. In this simple example, it is assumed that the stack pointer does not change between the two instructions. For the rest of the store-load pairs where the compiler cannot make such a guarantee, braids must be ordered such that the original partial ordering of memory instructions is maintained. If this ordering cannot be maintained while sorting braids within the basic block, the braid is split into two braids at the location of the memory ordering violation to enable the partial ordering. A split braid caused by this condition accounts for less than 1% of the braids analyzed. Like the previous situation, this situation is an artifact of performing profiling analysis and binary translation on preexisting program binaries, and a braid-aware compiler can easily cope with this problem.

```
st r1, sp, #1
  ⋮
ld r2, sp, #1
```

Figure 3.3: Identifying Memory Ordering Violations

Since profiling analysis operates on preexisting program binaries, it is limited to using the dataflow that has already been constructed by a compiler. Most of the braids have more than one instruction. Table 3.1 shows the percent of instructions that belong to a braid with a size of two or greater using profiling analysis. However, there will also be single-instruction braids. The percent of instructions that belong to single-instruction braids is the converse of the data in Table 3.1. Single-instruction braids do not provide any benefit for the braid microarchitecture since there are no internal registers. A braid-aware compiler can reduce or completely remove the number of single-instruction braids to maximize the benefit of braid processing.

3.2 Instruction Set Architecture Requirements

Minor augmentations are made to the ISA to allow the compiler to effectively convey braids to the microarchitecture. Figure 3.4 shows the specification of a zero-destination, one-source register, and two-source register braid ISA instructions. The shaded bits represent differences from a conventional ISA instruction. These bits have special meanings in a braid ISA instruction. The *braid start bit*, `S`, associated with an instruction specifies whether the instruction is the first instruction of a braid. The *temporary operand bit*, `T`, associated with each source operand specifies whether the operand obtains its value from the external register file or the

Integer	
benchmarks	percent
bzip2	84.9
crafty	81.9
eon	67.5
gap	79.2
gcc	73.6
gzip	84.6
mcf	65.2
parser	71.3
perlbmk	75.5
twolf	80.5
vortex	70.9
vpr	74.8
amean	**75.8**

Floating Point	
benchmarks	percent
ammp	86.6
applu	75.9
apsi	75.7
art	78.6
equake	78.6
facerec	71.9
fma3d	78.5
galgel	68.6
lucas	87.8
mesa	73.1
mgrid	95.6
sixtrack	72.5
swim	84.9
wupwise	77.8
amean	**79.0**

Table 3.1: Percent of Instructions Belonging to a Braid of Size Two or Greater

internal register file. The *external destination bit*, `E`, and the *internal destination bit*, `I`, associated with each destination operand specify whether the instruction writes its result to the external register file, the internal register file, or both register files. The augmentations made to support braids in the ISA do not require increasing the number of bits in the instructions. This is done by reinterpreting the fields from the existing ISA instructions.

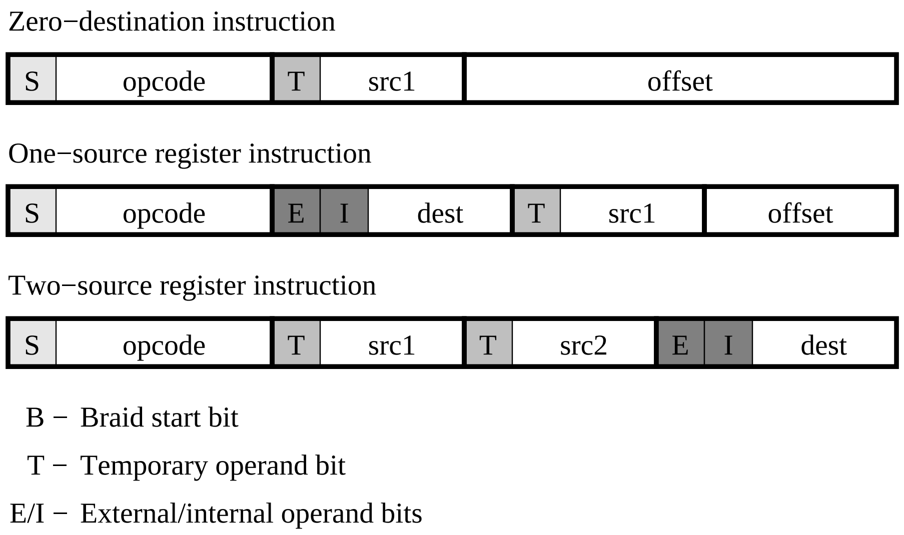

Zero−destination instruction
S
opcode
T
src1
offset
One−source register instruction
S
opcode
E
I
dest
T
src1
offset
Two−source register instruction
S
opcode
T
src1
T
src2
E
I
dest
B − Braid start bit
T − Temporary operand bit
E/I − External/internal operand bits

Figure 3.4: Braid Instruction Encoding

3.3 Microarchitecture Requirements

To implement braid processing, the microarchitecture requires some changes. The braid microarchitecture must leverage the dataflow and value information conveyed by the compiler. The execution core of the braid microarchitecture shares similarities with a conventional in-order microarchitecture.

3.3.1 Pipeline Overview

Figure 3.5 shows the block diagram of the pipeline of the braid microarchitecture. The shaded regions highlight the differences from a conventional out-of-order microarchitecture. These differences include a simpler allocator, a simpler renaming mechanism, a distribute mechanism, a set of braid execution units, a simpler bypass network, and a simpler external register file.

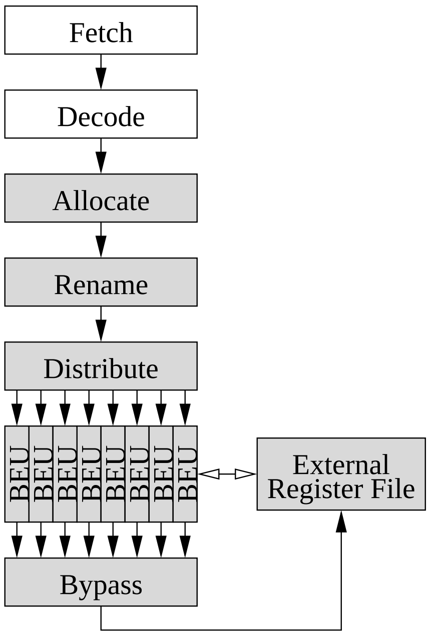


Figure 3.5: Block Diagram of the Braid Microarchitecture

The front-end of the pipeline is the same as the pipeline in a conventional out-of-order microarchitecture. A cache line is first fetched from the instruction cache. The set of instructions fetched is known as the fetch packet. The fetch packet contains a sequence of instructions in program order. Since the compiler has grouped the instructions of a braid together in the binary, a braid always enter the pipeline in its entirety before a subsequent braid enters the pipeline. That is, braids enter the pipeline in program order. This is useful and is leveraged by subsequent stages of the pipeline.

All instructions in the fetch packet are decoded as they enter the decode stage. Decoding an instruction is performed no differently from a conventional microarchitecture.

The allocate stage of the pipeline is responsible for assigning sequence numbers and allocating physical resources of various structures in the pipeline for an instruction. If an instruction requires a resource that cannot be allocated due to a lack of entries, this stage stalls until a free entry becomes available. First, each instruction in the fetch packet is assigned a sequence number. A sequence number is a unique number identifying the ordering of the instruction in the sequential instruction stream. This enables certain microarchitectural functions like memory disambiguation. Second, each instruction in the fetch packet is allocated an entry in the reorder buffer. The reorder buffer maintains the semantics of in-order execution regardless of how instructions are executed in the execution core. Third, each instruction in the fetch packet requiring an external destination operand is allocated an entry in the external register file. An instruction requiring only an internal destination operand does not require any entries to be allocated. The allocator identifies the need to allocate registers by examining the external/internal operand bits in the instruction. This is different from a conventional microarchitecture where all instructions with a destination operand require a physical register entry to be allocated. Fourth, each memory instruction in the fetch packet is allocated an entry in the load-store queue.

The next stage of the pipeline is the operand rename stage. Operand renaming removes anti and output dependencies in the code due to the use of a limited architectural register space in an out-of-order design. It is performed using the register alias table (RAT) which maps architectural register names to physical register names. The RAT contains one entry for each architectural register. In the braid microarchitecture, only the external operands of an instruction need to be renamed. This is because braids can execute out of order with respect to one another. Since the instructions inside a braid are executed in order, the internal operands do not need to be renamed. Like the allocate stage, the rename stage determines external operands by examining the temporary operand and external/internal operand bits in the instruction. Since not all of the operands of instructions in a fetch packet need to be renamed, the renaming mechanism does not need to support the entire fetch bandwidth as compared to a conventional microarchitecture. If there are more external operands to be renamed than the rename bandwidth, the renaming mechanism takes multiple cycles to process the fetch packet stalling the stages up to this point. A braid-aware compiler can enforce this requirement.

After operand renaming, the fetch packet enters the distribute stage where instructions in a braid are distributed to one of the braid execution units (BEU). In order to receive a braid, a BEU must be ready. Being ready is not the same as being empty. The BEU is ready if it is both empty and has the available context to process a new braid. That is, a BEU is ready if it does not have an in-flight braid. The use of the braid start bit in the instruction greatly simplifies the identification of braid boundaries. When a braid start bit is encountered, the distribute mechanism directs the instructions of the new braid to a ready BEU. If no BEUs are ready, this stage stalls until a BEU becomes available. The end of a braid is identified when a new braid is encountered. At this point, the BEU of the braid that was last distributed is notified that it has received the last instruction of its braid.

The BEUs contain the scheduling and execution stages of the pipeline. The internals of a BEU will be discussed in detail.

The external register file contains the values that are passed between different braids. These values are global in the view of the program. Compared to the register file in a conventional design, there are fewer entries in the external register file due to more efficient partitioning of the register space. Fewer entries mean fewer cycles needed for access. Furthermore, the smaller set of external operands puts a lighter demand on the ports of the external register file. Thus, the external register file requires fewer read and write ports.

The bypass stage of the pipeline corresponds to the bypass network. This network plays an important role in high performance designs. Writing to the register file may be pipelined, taking multiple cycles. A reader of an operand being written cannot access the value until the pipelined write completes. The bypass network provides values to readers before the pipelined write completes. The number of levels in the bypass network corresponds to the number of cycles needed to complete a write into the register file. By reducing the number of register file access cycles, the number of levels in the bypass network is also reduced. In a conventional design, each level of the bypass network supports the capability of bypassing n values per cycle where n is the issue width. In the braid microarchitecture, fewer values external are generated per cycle. Thus, a bypass level needs to support only a limited bandwidth.

Aside from the execution core, the rest of the pipeline is very similar to that of a conventional design. A conventional memory disambiguation structure such as the load-store queue is used to enforce memory ordering at runtime.

When an instruction becomes the oldest instruction in the reorder buffer, it is considered for retirement. An instruction is eligible for retirement if it is on the correct path, is the oldest instruction in the machine, completed execution, and did not generate an exception. When an instruction retires, it frees the resources allocated to it during the allocate stage including the reorder buffer entry, an external register if one was allocated, and an entry from the load-store queue if one was allocated. These freed entries go back into the free pool making them available for new instructions to use.

3.3.2 Execution Core Overview

Figure 3.6 shows a more detailed view of a BEU. The shaded regions highlight the differences from a conventional out-of-order design. These differences include a FIFO scheduler, a busy-bit vector, and a simpler internal register file. There are also two functional units within a BEU.

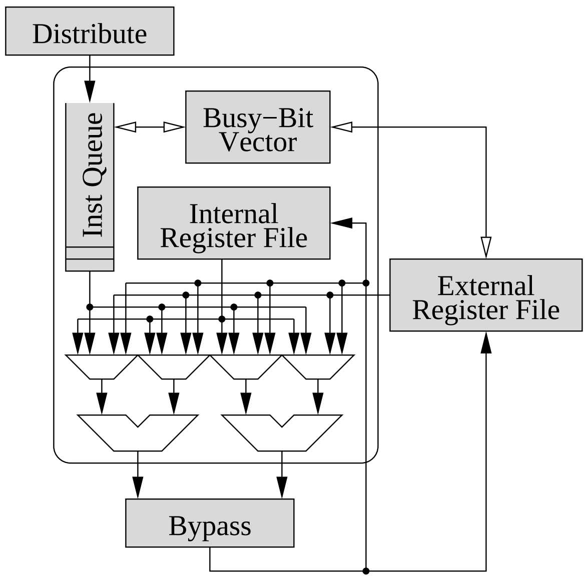


Figure 3.6: The Braid Execution Unit

When the instructions of a braid are distributed to a BEU, they first enter an instruction queue. This queue serves as a waiting area for newly issued[1] instruc-

[1]In this dissertation, issue refers to the process of inserting an instruction into the scheduling

tions. It is necessary for this queue to be large enough to accommodate most of the braids. An insufficient queue size can unnecessarily stall the front-end of the pipeline.

Two instructions located at the head of the instruction queue are examined for readiness and considered for in-order schedule. This 2-entry window is known as the scheduling window. Since the ready instructions of a braid are likely located at the head of the FIFO, instructions in the non-leaf nodes of the subgraph need not be examined for execution. The structure of a FIFO queue supports the characteristics of a braid dataflow which is long and narrow. Because of this characteristic, a small scheduling window is sufficient to efficiently process a braid.

The busy-bit vector maintains the availability of values in the external register file. It is similar to the scoreboard used in the CDC 6600 [70] and the busy-bit table used in the MIPS R10000 [75]. This vector has a bit for each external register. Each cycle, the instructions in the scheduling window consult the busy-bit vector for the availability of their external operands. Internal operands are guaranteed to be ready due to the in-order execution of the braid. External operands are produced by other braids and need to be checked for their availability. When all the operands of an instruction are available, the instruction is scheduled to one of the functional units for execution.

Each operand of an instruction is read from one of four locations: the instruction queue in the case of immediate operands, the internal register file, the external register file, or the bypass network. An instruction can write its result in the internal register file, the external register file, or both locations.

window. Schedule refers to the process of sending an instruction from the scheduling window to an execution unit.

The internal register file stores the internal operands of a braid. It is a very small structure containing only a few entries. The internal register file is designed with enough ports to support two instructions that can execute every cycle. Thus, four read and two write ports are needed. Because the values in the internal register file are not required outside the braid, they are safely discarded once the last instruction of a braid executes. These values do not need to be written back to the external register file.

3.3.3 Recoveries and Exceptions

Braids do not span control-flow boundaries. Therefore, recovering from a branch misprediction is a simple matter. This assumes the microarchitecture supports checkpoint recovery [35] like the MIPS R1000 [75] and the Compaq Alpha 21264 [33]. Previous researchers have shown that checkpoint recovery can be easily implemented [16] and is a technique that continues to be used in research [67] [66] [5] [24]. Since the processor already creates checkpoints for branch instructions, no additional structures or storage is required to support braids. In fact, checkpoints require less state in the braid microarchitecture because internal values of a basic block are not needed in the subsequent basic block. Therefore, internal register values do not need to be stored in the checkpoint. When a recovery initiates, the processor restores the checkpoint taken prior to the branch misprediction and begins execution along the correct path.

Handling exceptions is also a simple procedure but requires slightly more effort. When an exception is encountered, state is rolled back to the most recent checkpoint prior to the exception. The processor enters a special exception processing mode. In this mode, all BEUs are disabled except for one. All instructions are sent to the predetermined BEU as shown in Figure 3.7. Since a BEU contains an in-order scheduler, forcing instructions to one BEU turns the processor into an in-order processor. Internal register operands access the internal register file of the BEU, and external register operands access the external register file. When the excepting instruction is encountered, the exception handler is invoked. To access the internal register state, the exception handler does not require any changes. It has access to the internal register file through normal operand addressing. When the exception handler routine returns, the processor resumes normal execution mode from the same restored checkpoint. Simplicity was chosen over speed for handling exceptions due to the rarity of their occurrences.

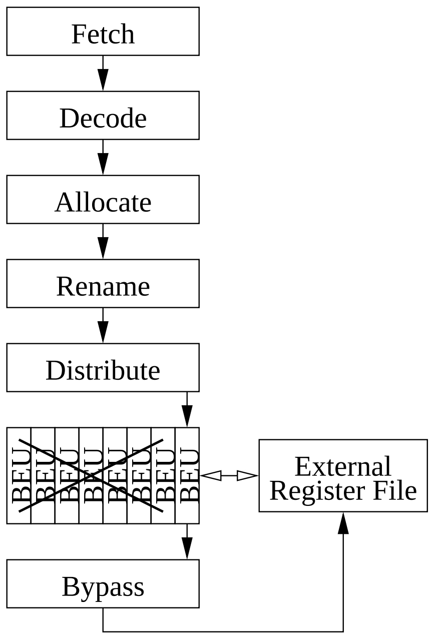


Figure 3.7: Block Diagram of Instruction Flow During Exception Processing Mode

3.4 Alternative Considerations

3.4.1 Using a Compiler

If the source code is readily available and recompilation is feasible, the compiler can offer the most flexible way to identify the most beneficial braids. A compiler that is knowledgeable of the underlying braid microarchitecture will produce braids that can be more efficiently processed by the microarchitecture. Loop transformations, inter-procedural analysis, and code optimizations must all take into account the physical makeup of braids.

Forming a braid builds upon the information gathered from two commonly-used compiler dataflow analysis techniques implemented in all compilers. These are reaching definitions and liveness analyses. These two techniques allow the compiler to identify the usage information of values in the program. The compiler uses the same dataflow graph coloring algorithm presented in the profiling analysis to partition the program dataflow into braids. As in the profiling analysis, a limit on the number of active internal operands within a braid is enforced.

Once braids are identified for a given basic block, the compiler performs register allocation. Since the register set is partitioned into two disjoint sets, register allocation is performed for each set separately. Register allocation for the external operands is identical to traditional register allocation. It is performed for the entire procedure. However, there are fewer operands which require external register names. Spill and fill code are inserted when the working set size of external values do not fit within the set of external registers. Register allocation for the internal registers is performed within a braid for all the braids in the program. The compiler has already guaranteed that the maximum number of active internal operands will not exceed the number of internal registers.

Braids are sorted and scheduled in the basic block. Like profiling and binary translation, the compiler rearranges instructions such that instructions from the same braid are laid out as a sequence of consecutive instructions within the basic block. Braids are ordered to avoid memory ordering violations.

3.4.2 Compiling Versus Profiling

Braids can be identified by the compiler if source code is available. Otherwise, braids must be identified by a binary profiling tool. The compiler requires compiling of code whereas the binary profiling tool requires profiling. There are tradeoffs with either approach.

Generating braids using the compiler has several advantages over profiling preexisting binaries. First, the compiler has the ability to transform the dataflow graph of the algorithm through compiler optimizations and transformations. This is useful because the compiler is aware of the underlying microarchitecture and thus can produce longer and narrower braids. The compiler can also transform the dataflow graph to eliminate single-instruction braids. The profiling analysis method cannot transform the dataflow graph and is limited to producing braids from the dataflow graph in the preexisting program binary.

Second, since the compiler has the ability to perform register allocation, it can make better use of both the external and internal register sets. By performing register allocation for the external registers separately from the internal registers, the compiler can minimize the amount of spill and fill code which reduces memory accesses.

Third, since the compiler has more knowledge of the program dataflow than the microarchitecture, it has better knowledge of the memory operations of a program. The better knowledge allows the compiler to have more flexibility in reordering memory instructions forming more meaningful braids.

Generating braids using the profiling method has its advantages. There are many preexisting program binaries and libraries that make recompilation infeasible. One reason is due to the unavailability of source code. Profiling analysis can be an effective method of transforming these binaries into a suitable form for execution on the braid microarchitecture.

Chapter 4

Methodology and Performance

This chapter discusses the simulation infrastructure used to model the braid microarchitecture and presents a performance analysis of the braid microarchitecture. First, details of the simulator and input sets are presented as well as the parameters chosen to represent an aggressive high-performance future processor. Second, the performance sensitivity of various design parameters in the execution core is analyzed. Third, the braid microarchitecture is compared with other microarchitectural paradigms.

4.1 Machine Model

To show how braids can be useful for the design of future aggressive processors, the experiments were done on 8-wide configurations. For comparison, the results for 4-wide and 16-wide configurations are also presented. Table 4.1 shows the detailed baseline configuration of an aggressive conventional out-of-order microarchitecture and the braid microarchitecture studied in this dissertation.

4.1.1 Shared Front and Back-Ends

Both the baselines share a similar front and back-end. The front-end is capable of fetching up to eight instructions and predicting up to three branches per cycle. This aggressive front-end is intended to mimic the advanced microarchitec-

Common Baseline Parameters	
Instruction Cache	64KB, 4-way associative, 3-cycle latency
Branch Predictor	perceptron with 64-bit history and 512-entry weight table
Fetch Width	8 instructions, capable of processing 3 branches per cycle
Issue Width	8 instructions
Instruction Window	256-entry ROB
L1 Data Caches	64KB, 2-way associative L1 data cache with 3-cycle latency
L2 Cache	1MB, 8-way associative unified L2 data cache with 6-cycle latency
Main Memory	400-cycle latency
Out-of-Order Baseline Parameters	
Misprediction Penalty	minimum 23 cycles
Allocate	8 operands
Rename	16 source operands and 8 destination operands
Scheduler	8 distributed 16-entry schedulers
Functional Unit	8 general purpose
Register File	256 entries with 16 read ports and 8 write ports
Bypass Network	3 levels, each with full paths
Braid Baseline Parameters	
Misprediction Penalty	minimum 19 cycles
Allocate	4 operands
Rename	8 source operands and 4 destination operands
BEU	8
FIFO	16-entry instruction queue per BEU
Scheduling Window	2-entry in-order scheduler per BEU
Busy-Bit Vector	8 bits per BEU
Functional Unit	2 general-purpose units per BEU
Internal Register File	8 entries with 4 read ports and 2 write ports per BEU
External Register File	8 entries with 6 read ports and 3 write ports
Bypass Network	1 level with limited paths

Table 4.1: Baseline Processor Configurations

ture capabilities of future processors which should provide higher fetch bandwidth. Mechanisms such as the trace cache can already be found in the Pentium 4 processor [34] to provide higher fetch bandwidth than is possible with an instruction cache alone.

There is another reason why an aggressive front-end is needed. Since this dissertation focuses on the design of the execution core, the execution core must be stressed. An aggressive front-end accomplishes this by not constraining the number of instructions delivered to the execution core (see for example, Salverda and Zilles [60]).

Both baselines also share a similar retirement mechanism. Each supports 256 in-flight instructions via a 256-entry ROB.

4.1.2 Misprediction Pipeline

Figure 4.1a shows the breakdown of the 23-cycle misprediction pipeline of the conventional out-of-order baseline. A branch instruction is fetched, decoded, allocated, and renamed in the front-end of the pipeline. Then it is distributed and queued in the execution core. Next, the instruction schedules for execution, reads the register file, and executes. The branch condition is verified, and upon a misprediction, the front-end is notified to redirect fetch.

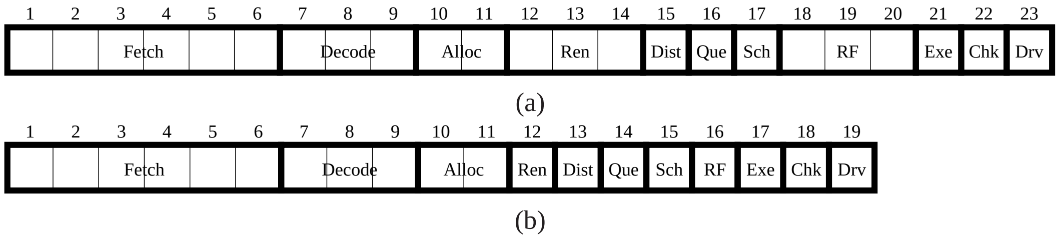


Figure 4.1: Misprediction Pipeline (a) Baseline (b) Braid

Figure 4.1b shows the breakdown of the 19-cycle misprediction pipeline of the braid microarchitecture. The 4 cycle difference between the two misprediction pipelines comes from a reduction of 2 stages in operand rename and a reduction of 2 stages in register file access. These savings are attributed to the smaller area and lower bandwidth designs of the structures in these stages.

4.1.3 Out-of-Order Execution Core

The out-of-order baseline has a minimum branch misprediction penalty of 23 cycles. The allocator is capable of processing eight instructions per cycle. The renaming mechanism is capable of processing 16 source operands and eight destination operands per cycle. There are eight distributed 16-entry out-of-order schedulers and eight general-purpose functional units. There is a 256-entry monolithic register file with 16 read ports and eight write ports. The bypass network consists of three levels with a full set of paths at each level.

4.1.4 Braid Execution Core

Because of the design simplification of the braid microarchitecture, it has a pipeline that is shorter by four stages than that of a comparable conventional design. The savings come from a shorter operand rename stage and a shorter register access stage. The braid microarchitecture baseline has a minimum branch misprediction penalty of 19 cycles. The allocator is capable of processing four instructions per cycle. The renaming mechanism is capable of processing eight source operands and four destination operands per cycle. There are eight BEUs. Each BEU contains a 16-entry FIFO instruction queue with a 2-entry instruction scheduling window. Each BEU also has an 8-bit busy-bit vector, two functional units, and a small 8-entry internal register file with four read ports and two write ports. There is an 8-

entry global external register file with six read ports and two write ports. The bypass network consists of one level with a limited set of paths. For the experiments, clock frequency was not varied.

4.1.5 Shared Memory System

The memory hierarchy consists of split L1 instruction and data caches, a unified L2 instruction and data cache, and main memory. The L1 instruction cache is a 4-way set associative 64KB cache with a 64-byte line size. The L1 data cache is a direct mapped 64KB cache with a 64-byte line size. The L1 instruction cache has a 1-cycle access latency while the L1 data cache has a 3-cycle access latency. The access time does not include the extra cycle of address generation required for load and store instructions. The unified L2 cache is an 8-way set associative 1MB cache. It has a 64-byte line size and a 6-cycle access latency. The L2 cache is modeled as having eight banks interleaved on 64-byte boundaries. Main memory has a 400-cycle minimum access latency and is modeled as having 32 banks interleaved on 64-byte boundaries.

A memory request that misses in either L1 cache is allocated a miss request buffer entry. There are 32 buffers available for handling outstanding misses. A new miss that maps to the same line as an outstanding miss can piggyback on the outstanding miss. A memory request that misses in the L2 cache is allocated a memory request buffer entry. There are 32 buffers for handling L2 cache misses. Piggybacking is also allowed at this level to reduce the number of memory requests.

Memory paging is not modeled. Therefore, there are neither translation lookaside buffer accesses nor page faults. A conventional memory disambiguation structure enforces memory ordering at runtime. The load-store queue supports 32 entries.

4.2 Simulator

The experiments in this dissertation were carried out on the second version of an in-house, cycle-accurate, execution-driven simulator called SCARAB [20]. This new version has been rewritten in C++ to take advantage of some of the benefits provided by the C++ language. SCARAB allows various simulation models to coexist within the same simulator. Student A can work with one model that does not interfere with student B who is working with another model. The SCARAB simulation infrastructure is also modularized. Components of the pipeline can be easily added, removed, replaced, and shared among different models. The simulation speed of SCARAB is optimized by utilizing tuned data structures. It achieves simulation speeds faster than the SimpleScalar simulator [15].

SCARAB processes elf64-alpha binaries produced for the Linux operating system. System calls in the program are emulated by the simulator on the host machine following the POSIX standard. Operating system code is not simulated. Exceptions are not handled due to their rarity.

The simulator is fully capable of executing the wrong path and producing wrong-path values. These values are correctly generated but will not commit into the architectural state. Various researchers have found that processing wrong-path instructions have a non-trivial impact on IPC by prefetching useful data [6] [55].

Table 4.2 shows the latencies of various classes of instructions. The functional units are fully pipelined for every operation except for floating point divide.

4.3 Benchmarks

The SPEC CPU2000 benchmark suite [2] was chosen for the experiments. This suite consists of 26 benchmarks programs of which 12 are integer and 14 are

Instruction Class	Latency (in cycles)
Integer arithmetic	1
Integer multiply	8, pipelined
Floating point arithmetic	4, pipelined
Floating point divide	16
Logical	1
Memory	3 minimum
Memory forwarding	3 minimum
All others	1

Table 4.2: Instruction Latencies

floating point. The programs were compiled with gcc 4.0.1 [1] on Linux for the Alpha EV6 [64] ISA with the `-O2` optimization flag and feedback profiling enabled. All benchmark programs were run for 500 million instructions using the MinneSPEC reduced input sets [42]. The reduced input sets approximate the program behavior when running with the SPEC reference input sets but allow the programs to complete within a reasonable amount of time. Table 4.3 lists for each benchmark program its name, description, and the input set used.

4.4 Sensitivity Studies Varying Braid Execution Unit Parameters

Various design parameters were considered for the execution core of the braid microarchitecture. These include the number of BEUs, the size of the FIFO queue, the size of the FIFO scheduling window, and the number of functional units per BEU. The following experiments are sensitivity studies showing the effects of each design parameter. The control configuration is the braid microarchitecture with eight BEUs. Each BEU contains a 16-entry FIFO instruction queue with a 2-entry in-order scheduling window and two functional units. In each experiment,

	Benchmark Name	Description	Input Set
Integer	bzip2	Compression	large reduced source
	crafty	Game Playing: Chess	large reduced
	eon	Computer Visualization	large reduced
	gap	Group Theory, Interpreter	large reduced
	gcc	C Programming Language Compiler	large reduced cp-decl.i
	gzip	Compression	large reduced
	mcf	Combinatorial Optimization	large reduced
	parser	Word Processing	large reduced
	perlbmk	PERL Programming Language	large reduced
	twolf	Place and Route Simulator	large reduced
	vortex	Object-oriented Database	large reduced
	vpr	FPGA Circuit Placement and Routing	large reduced place
Floating Point	wupwise	Physics / Quantum Chromodynamics	large reduced
	swim	Shallow Water Modeling	large reduced
	mgrid	Multi-grid Solver: 3D Potential Field	large reduced
	applu	Parabolic / Elliptic Partial Differential Equations	large reduced
	mesa	3-D Graphics Library	large reduced
	galgel	Computational Fluid Dynamics	large reduced
	art	Image Recognition / Neural Networks	large reduced
	equake	Seismic Wave Propagation Simulation	large reduced
	facerec	Image Processing: Face Recognition	large reduced
	ammp	Computational Chemistry	large reduced
	lucas	Number Theory / Primality Testing	large reduced
	fma3d	Finite-element Crash Simulation	large reduced
	sixtrack	High Energy Nuclear Physics Accelerator Design	large reduced
	apsi	Meteorology: Pollutant Distribution	large reduced

Table 4.3: SPEC CPU2000 Benchmark Descriptions and Input Sets

one of the parameters was varied while the other parameters were held constant. All the results were normalized to the performance of the 8-wide baseline conventional out-of-order configuration. This is indicated by the thick line on the 1.0 mark on the y-axis.

Figure 4.2 plots the performance as a function of the number of BEUs. This result confirms there are more braids ready to execute than the number of BEUs in the microarchitecture. Increasing the number of BEUs improves performance in two ways. First, adding more BEUs increases the number of execution resources. More execution resources allow more braids to execute in parallel. Second, a long-latency instruction stalls the BEU, causing the BEU to be idle. For example, if an instruction waiting to be scheduled is dependent on an instruction that misses in the cache, the waiting instruction cannot execute which in turn causes the functional units in the BEU to be idle. Having more BEUs allows younger braids with ready external operands to execute ahead of older braids that are stalled. Using eight BEUs, there is a 8.3% performance drop from the baseline out-of-order microarchitecture.

There is a constraint on the number of BEUs that can be supported in the braid microarchitecture. Too many BEUs in the braid microarchitecture increase communication latency for communicating operands and tags between BEUs. This increases the complexity for synchronizing the busy-bit vectors. Since 8-wide designs are slowly becoming available in the processor market, a configuration with eight BEUs was chosen as a design point that provides a good tradeoff between performance and what can be implemented. It will be shown that the braid microarchitecture passes far fewer values between each way in the pipeline compared to a conventional out-of-order design of the same issue width. Thus, it will be possible to further increase the number of BEUs in the braid microarchitecture.

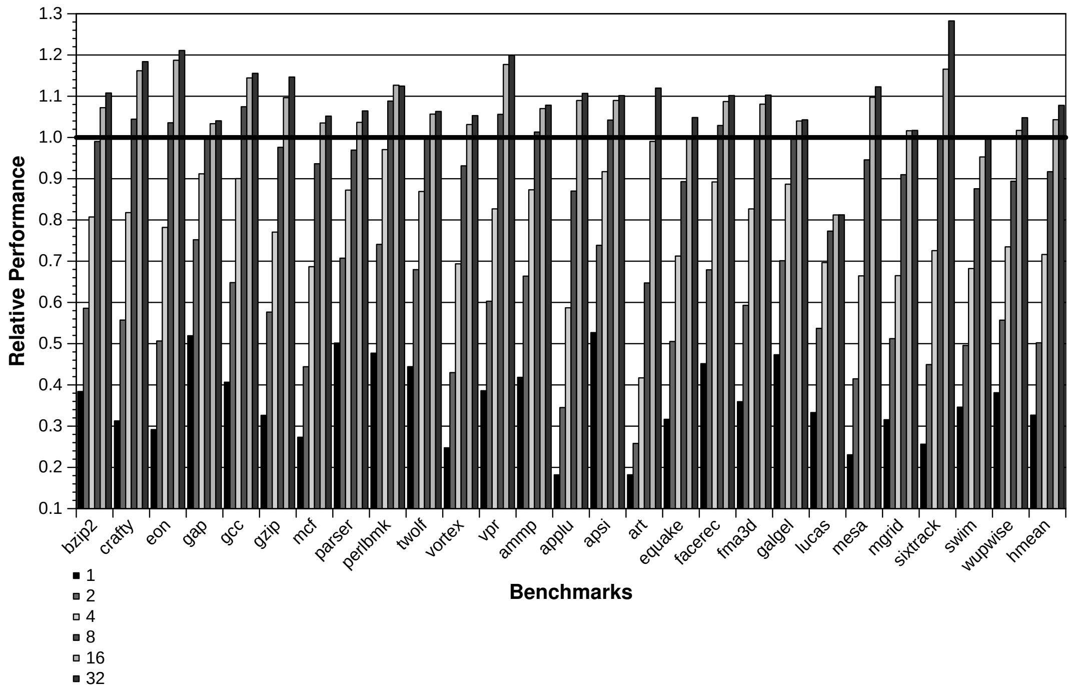


Figure 4.2: Performance Sensitivity to the Number of BEUs

The FIFO queue in each BEU is an instruction waiting buffer. An instruction waits in the queue until it reaches the queue entries corresponding to the instruction scheduling window. The queue should be large enough to buffer all of the instructions of a braid. Figure 4.3 plots the performance as a function of the number of entries in the FIFO queue. On average, as few as 16 entries are enough to support most of the braids for the benchmark programs. This is because 97% of braids consist of 16 instructions or fewer. Without the proper instruction buffering, all the instructions of a braid cannot be queued in the BEU. This situation stalls the distribution mechanism, subsequent braids, and eventually the front-end.

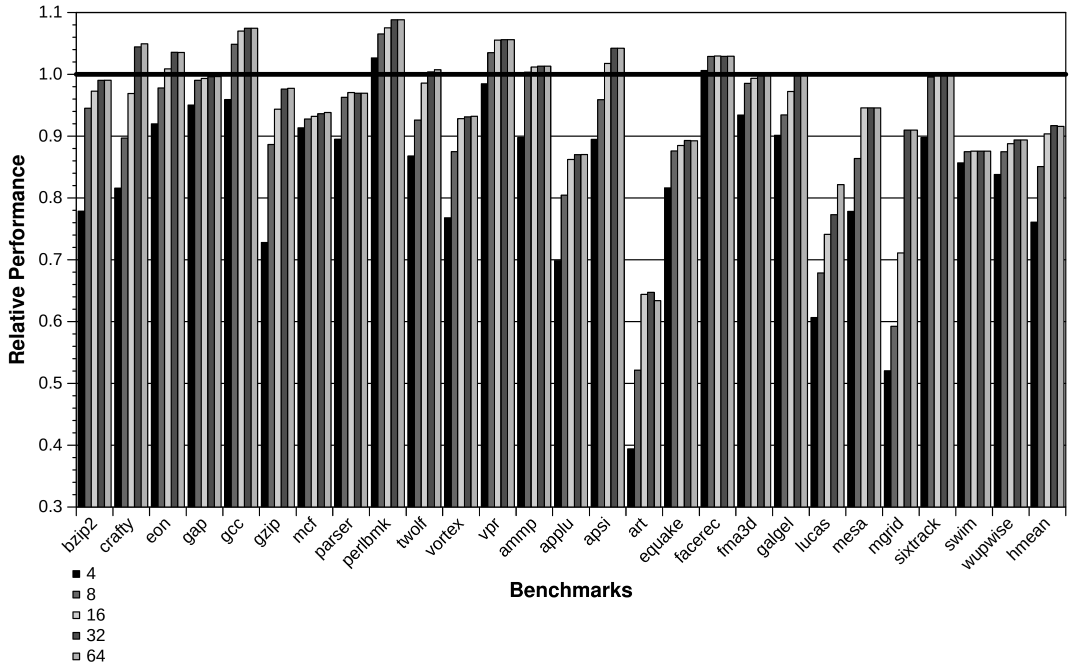


Figure 4.3: Performance Sensitivity to the Number of FIFO Queue Entries

The following experiment examines the likelihood that ready instructions in a braid are located at the head of the FIFO by using different scheduling window sizes. The scheduling window size denotes the number of entries in the FIFO queue. The instructions within this window are examined for readiness each cycle. Figure 4.4 plots the performance as a function of the scheduling window size. It is encouraging to see the steep rise going from one to two entries, and then the plateau from two to 16 entries. On average, a window of two entries is sufficient for the benchmark programs. This result is consistent with the measurement of braid width. Since most braids have narrow widths, using a larger scheduling window is not beneficial to performance. The floating point benchmark programs `applu`, `apsi`, `lucas`, and `mgrid` have slightly wider widths on average and benefit more from a larger scheduling window.

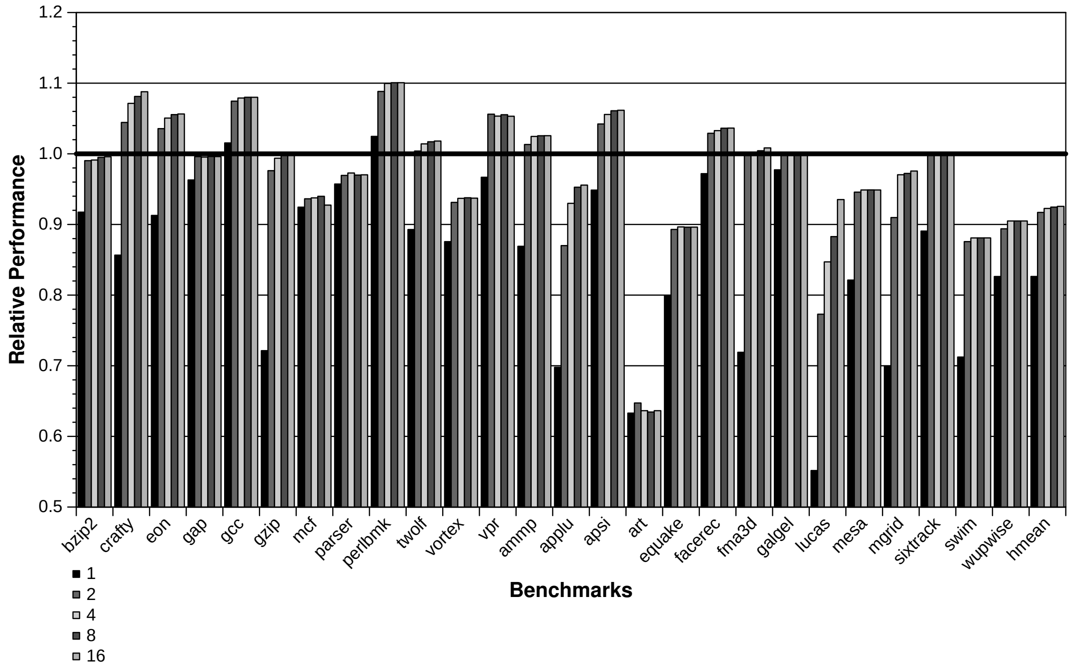


Figure 4.4: Performance Sensitivity to the Scheduling Window Size

In Figure 4.4, the number of functional units was fixed as the scheduling window size changes. The following experiment examines whether the number of functional units is a performance bottleneck as the scheduling window size changes.

Figure 4.5 plots the performance as a function of both the number of functional units and the scheduling window size. This graph shows a similar trend as the graph for scaling the scheduling window alone. This result confirms that performance is not limited by the number of functional units and reaffirms that two functional units is enough to process a braid.

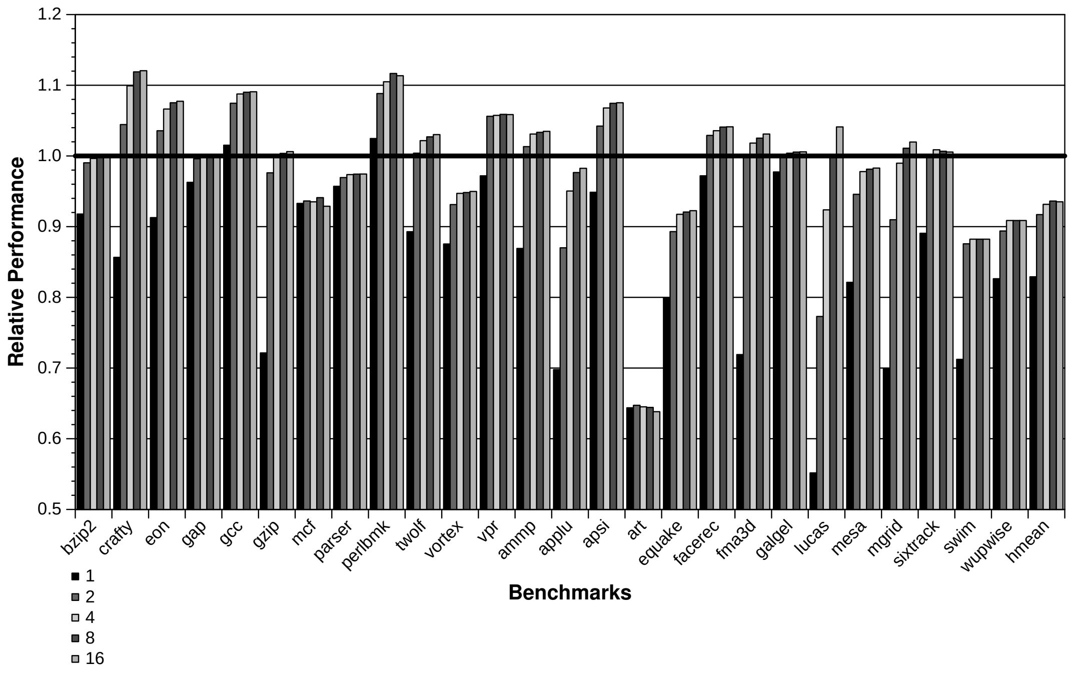


Figure 4.5: Performance Sensitivity to the Number of Functional Units per BEU

4.5 Comparison to Other Processing Paradigms

Figure 4.6 plots the performance comparison of four microarchitectural paradigms at three issue widths. Each stacked bar plots the performance of four different microarchitectures. From bottom to top, they are in-order, FIFO-based dependence steering, the braid, and out-of-order microarchitectures. The set of bars for each benchmark program from left to right represent the performance of 4-wide, 8-wide, and 16-wide designs. The result of using a dependence-based steering algorithm [50] is presented to illustrate one simple and implementable algorithm with a design complexity that is comparable to braids.

At least three observations can be made from this graph. First, significant gains are still available at wider widths. Second, the braid microarchitecture achieves performance that is within 9% of a very aggressive conventional 8-wide out-of-order design. Third, the performance gap between the braid and out-of-order configurations gets smaller as the issue width increases.

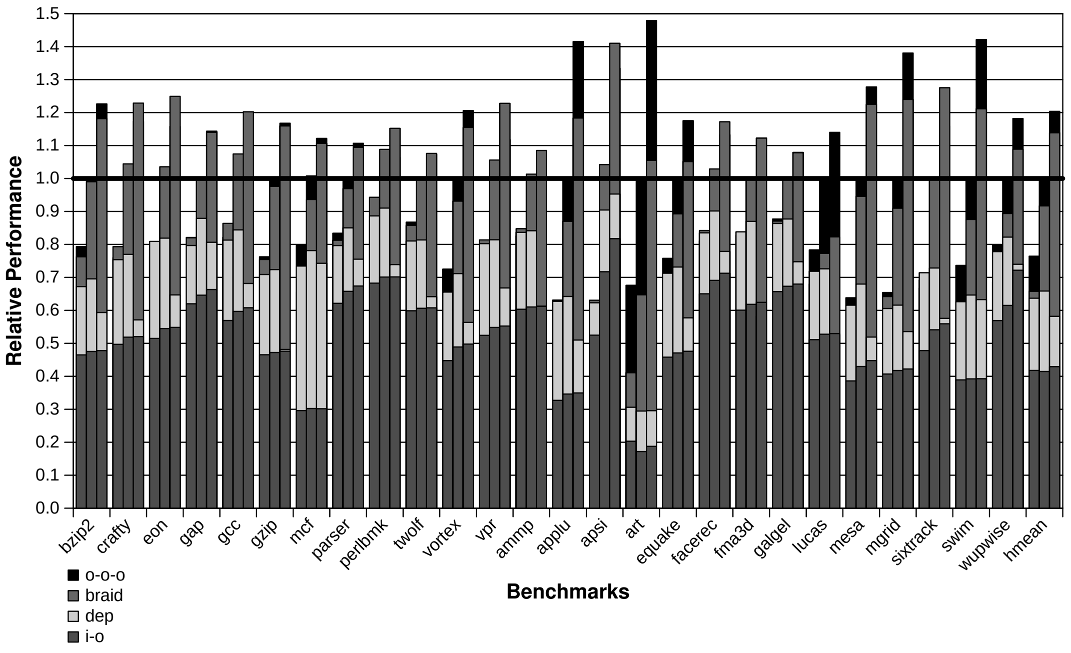


Figure 4.6: Performance of In-Order, Dependence-Based Steering, Braid, and Out-of-Order Designs at Various Issue Widths

Since the braid microarchitecture uses braids as a unit of processing, the braid microarchitecture can more efficiently manage the instructions in a large window than an out-of-order design with a distributed scheduling window. It is interesting to note that at the 4-wide configuration, dependence-based steering almost achieves the same performance as the braid microarchitecture. This is due to the fact that the braid microarchitecture suffers from an unbalanced design with only four BEUs. Braids provide greater benefit when there are more BEUs because many braids can execute in parallel. Limiting the number of BEUs to four greatly constrains execution core resources. The braid microarchitecture requires more BEUs to be effective.

Chapter 5

Braid Optimizations

Although the braid microarchitecture implements an execution core with a simplistic design, the simplification incurs inefficiencies which limit its performance. These inefficiencies do not exist in an out-of-order design. This chapter discusses three bottlenecks of the braid microarchitecture and suggests techniques to reduce their impact on performance. The first technique addresses the limitation on braid size caused by control flow instructions. The second addresses the underutilization of braid execution resources caused by long-latency instructions. The third addresses the poor utilization of execution resources caused by single-instruction braids.

5.1 Dynamic Merging

The first problem is the lost opportunity by not being able to build longer braids. To avoid problems associated with changing control-flow, a braid is defined to reside completely within a basic block. Thus, the average size of a braid is always less than or equal to the average size of a basic block. Limiting the size of a braid limits the amount of internal values that are communicated. Although braids are identified at compile time, they can effectively become larger at runtime through a technique called dynamic merging. Dynamic merging allows a braid from a basic block to join a braid from another basic block at runtime to form a larger braid.

Figure 5.1a show a control-flow merge point, and Figure 5.1b show a control-flow join point. These figures will be used to illustrate why control-flow changes make it difficult for the compiler to span a braid across a control-flow boundary. Numbers identify basic blocks. Lowercase letters identify braids within a basic block. Uppercase letters identify the control-flow paths. Register identifiers identify the external operands of a braid. Each basic block contains two braids designated by lower case letters. At a control-flow fork point, child blocks `2` and `3` are reached from parent block `1`. At a control-flow join point, parent blocks `1` and `2` precede child block `3`. In Figure 5.1a, suppose the compiler builds braid `ac` that spans blocks `1` and `2` as shown by the thick line. In this case the external operands of braid `a` should be treated as internal operands of braid `ac`. However, if path `AB` is traversed at runtime, the assumption made for the operands `r1`, `r2`, and `r3` no longer hold true. In this case, `r2` and `r3` should be treated as internal operands and `r1` should be treated as an external operand. The same argument can be made for the control-flow join point in Figure 5.1b.

Control-flow fork and join points complicate the formation of larger braids. However, at runtime, only one path is traversed at a time, and two consecutive basic blocks become logically contiguous. A runtime approach is used to merge braids across control-flow boundaries.

The concept of dynamic merging is a simple one. Two braids can merge if they share values. That is, the external inputs of the child braid are a subset of the external outputs of the parent braid. When a match is confirmed, the two braids merge by distributing the second braid to the BEU holding the first braid such that the second braid immediately follows the first braid. The shared external register communication between the two braids are identified and communicated through the internal register file potentially eliminating external register file accesses.

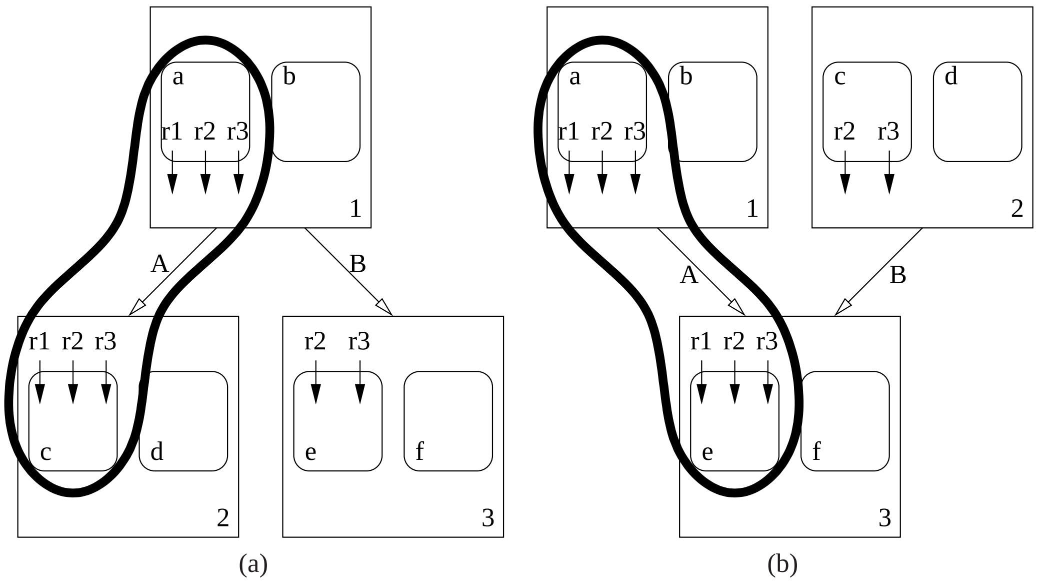


Figure 5.1: Control-Flow Points (a) Fork (b) Join

Implementing dynamic merging requires the use of a braid merging table shown in Figure 5.2a. The braid merging table maintains the external outputs of active braids being processed by the BEUs. Figure 5.2b shows one entry of the table. Each entry tracks one braid and contains a valid bit, a 8-bit external output vector, and a 3-bit BEU Id vector. The valid bit indicates whether the braid is still being processed in the BEU. The 8-bit vector encodes the external outputs of the braid. Each bit of the vector maps to an external register. The 3-bit vector identifies the BEU to which the braid was distributed.

Up to four braids per basic block are tracked and up to two most recent basic blocks are tracked by the braid merging table. This is shown at the top of Figure 5.2a. Tracking up to four braids per basic block was chosen because the average number of braids in a basic block is 3.6 for all the benchmark programs. As a braid is distributed, the external output vector of the braid is inserted into one of the four

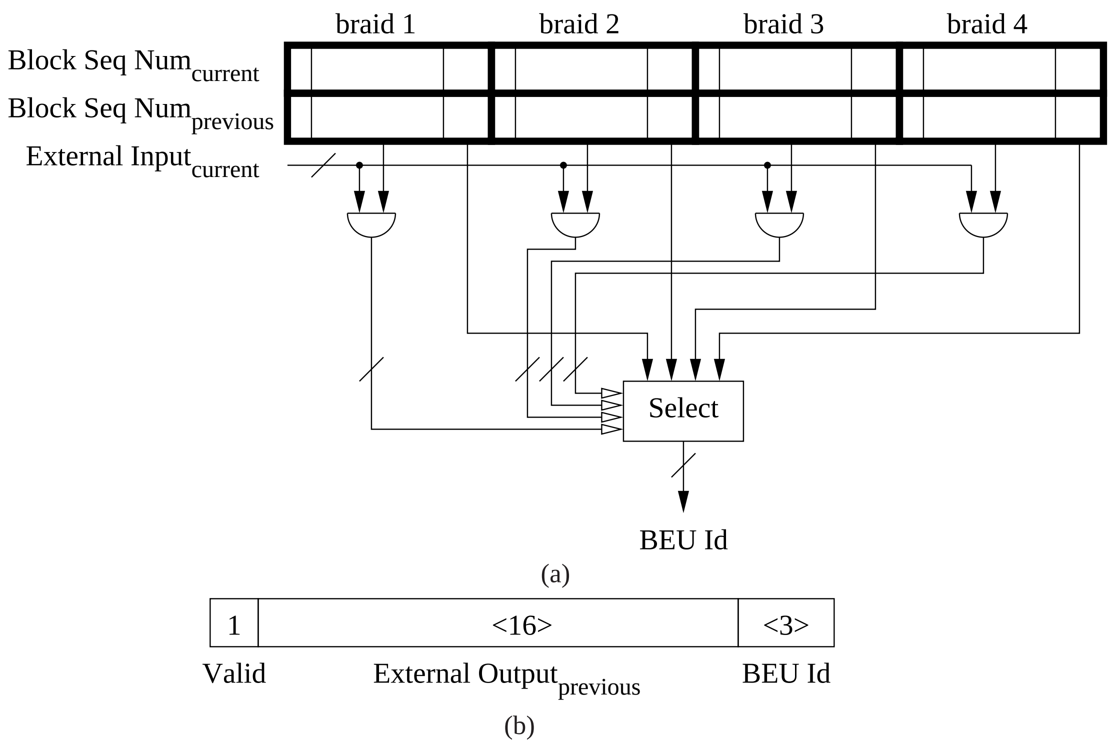

braid 1
braid 2
braid 3
braid 4
Block Seq Num$_{current}$
Block Seq Num$_{previous}$
External Input$_{current}$
Select
BEU Id
(a)
1
<16>
<3>
Valid
External Output$_{previous}$
BEU Id
(b)

Figure 5.2: Mechanism to Support Dynamic Merging (a) Merging Table Entry (b) Merging Identification Logic

entries associated with the current block sequence number. The block sequence number is a unique number assigned to basic blocks in program order. When a new basic block is encountered, the entries associated with the current block sequence number are shifted into the entries associated with the previous block sequence number. When a braid is distributed, the external output fields from the entries in the previous block sequence number are probed. The braid matches its external input vector which identifies its external inputs against the external output vectors from the table. The logic needed to implement this is shown at the bottom of Figure 5.2a. A match indicates an opportunity to merge and the BEU Id of the matched entry is used to distribute the braid to the BEU of its parent.

The external input and output vectors of a braid encode its external operands. These vectors can either be generated by the compiler or by the hardware. If the compiler generates this vector, it must encode the external input and output vectors of a braid in special instructions. Since each vector requires eight bits, 16 total bits are needed. This can be encoded in the unused field of a NOP instruction. The hardware can also generate the two vectors at runtime. Since the renaming mechanism already identifies the external and internal source and destination registers of the instructions of a braid, it can also produce the external input and output vectors and insert them into the pipeline along with the braid.

For dynamic merging to be useful, another vector called the single-use vector is required. This vector tracks if there is a single consumer for each external output register. Since this information is known to the compiler, this vector is generated by the compiler. One single-use vector is associated with each braid and encoded in the unused field of a NOP instruction.

When an opportunity to merge is identified, the BEU processing the parent braid is first notified that a child braid will merge with it. The external register file writes present in the intersection of the two external vectors are the set of values communicated between the two merging braids. These external register file operands are redirected to write into the internal register file. Depending on the contents of the single-use vector, there may or may not be a write to the external register file. The external register file write is disabled if the single-use vector identifies the next consumer of the value to be the last consumer of the value. Once the BEU has queued the entire parent braid, it receives the child braid without waiting for the BEU to become ready. Rather than reading from the external register file for the values communicated between the two braids, the child braid is redirected to read from the internal register file.

Figures 5.3 and 5.4 plot the distribution of the external and internal register read and write accesses. In Figure 5.3, note that the percent of external and internal read accesses do not sum to 100%. This is because some external reads hit in the bypass network. The left two bars of each benchmark program show the percent of read accesses for the baseline braid microarchitecture, and the right two bars show the percent of read accesses when dynamic merging is applied. Without dynamic merging, about 50% of the register reads come from the internal register file. With dynamic merging, the percent of internal register read accesses increases to over 73%.

Figure 5.4 plots the distribution of the external and internal register write accesses. The left two bars represent the percent of register file write accesses for the baseline braid microarchitecture. Adding these two bars gives an expected 100%. The right two bars show the percent of write accesses when dynamic merging is applied. Approximately 63% of writes access the internal register file. When dynamic merging is applied, the percent of internal register write accesses increases to 67%.

Although dynamic merging is useful for decreasing the number of accesses

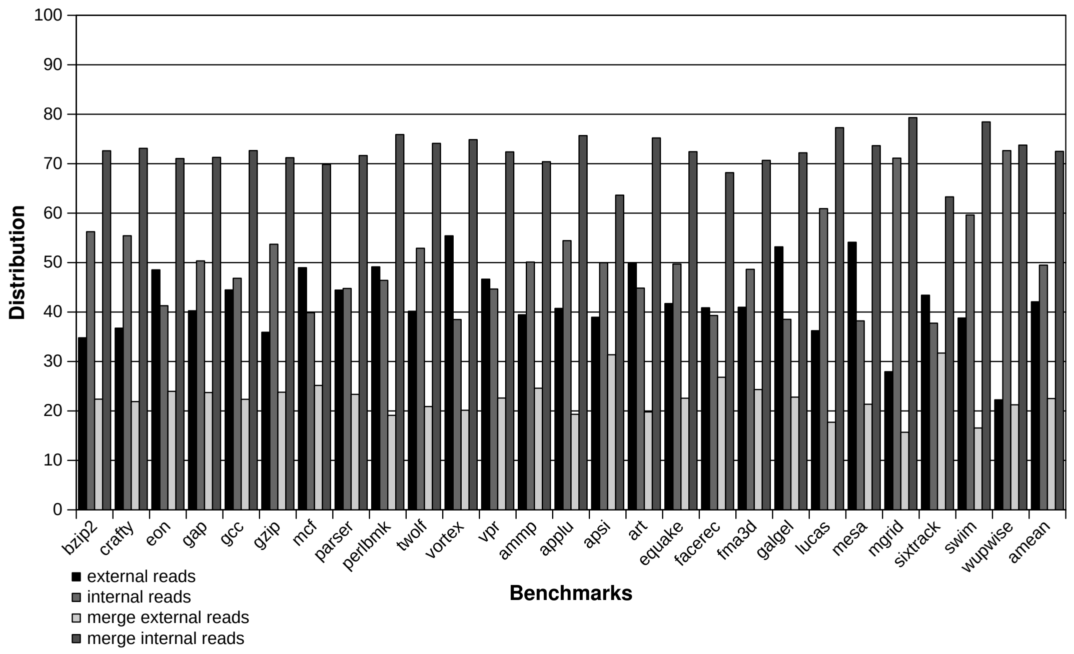


Figure 5.3: Percent of External and Internal Register Reads

to the external register file, it does not improve performance in its current form. The left and right bars of Figure 5.5 plot the performance without and with dynamic merging, respectively. On average, enabling dynamic merging reduces performance by 0.3%. This is due to load balancing problems caused by larger units of work. A braid-aware compiler can produce more balanced workloads. The slowdown is a small cost to pay for reducing external communication.

5.2 Braid Execution Unit Context Sharing

The second problem is the underutilization of the functional units. In a conventional out-of-order processor, the scheduler allows only ready instructions to execute. Instructions with operands which are not ready wait in the scheduling window. This method of processing results in the most efficient use of functional

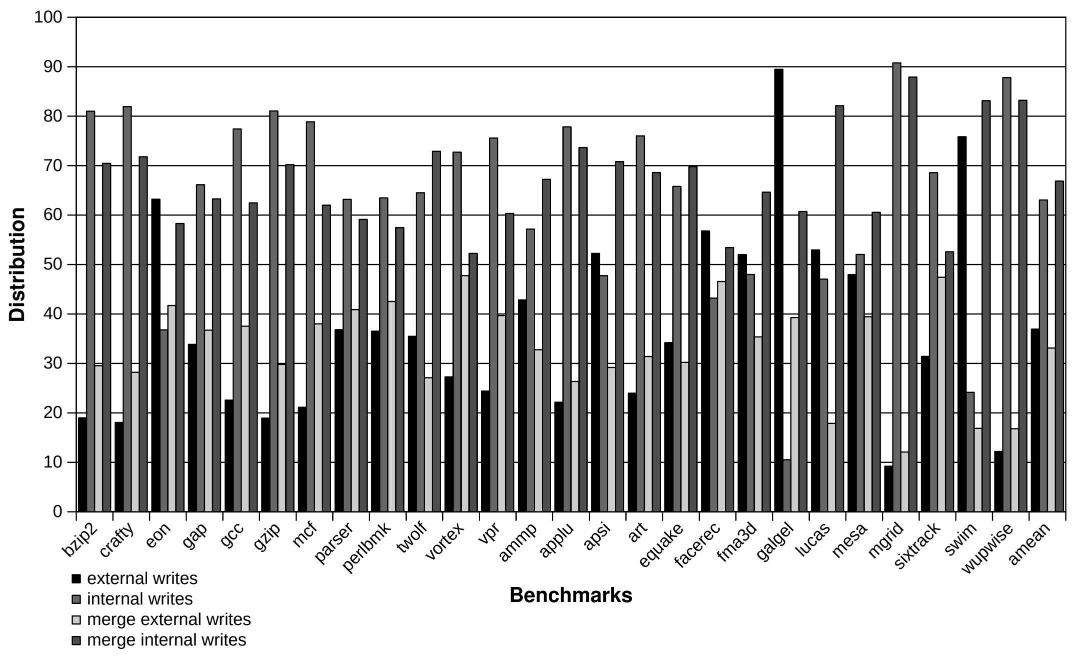


Figure 5.4: Percent of External and Internal Register Writes

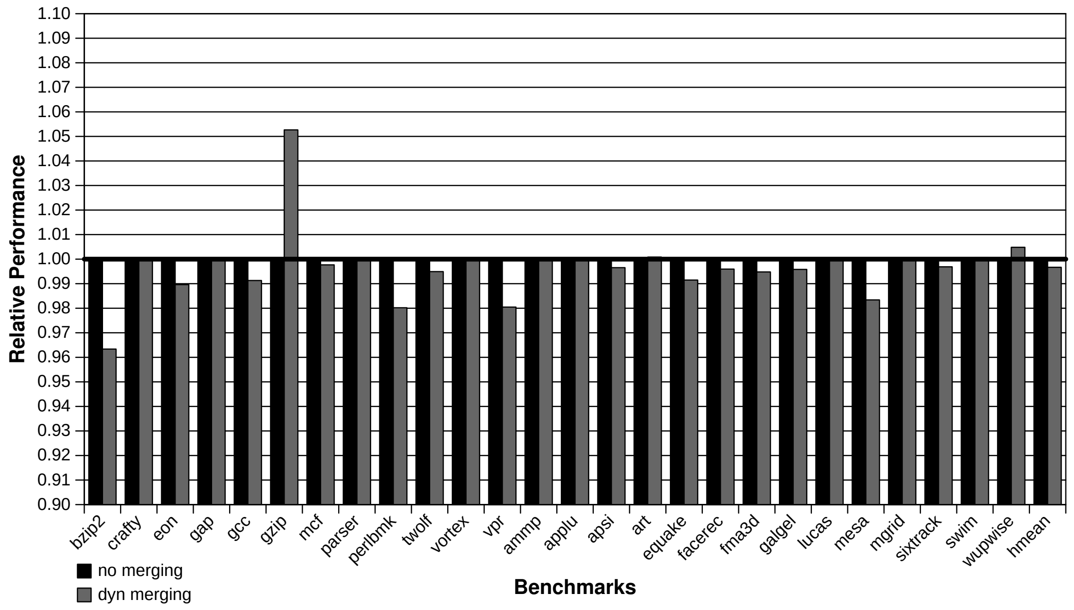


Figure 5.5: Performance Using Dynamic Merging

units because execution resources are allocated only to instructions which can use them.

The braid microarchitecture does not share the same luxury. The larger granularity of the unit of work of a braid causes this inefficiency. Once a braid is sent to a ready BEU, the execution resources in the BEU are dedicated to processing only the instructions in the braid. In the event an instruction is dependent on a value from a load miss, that instruction stalls the scheduling process. The idle functional units in the BEU are inefficient when there are ready instructions in another braid that can execute. The ready instructions in the other braid cannot use the resources of the BEU with the idled functional units.

Table 5.1 lists the percent of cycles in which all BEUs are stalled. Non-deterministic latency instructions such as load instructions that miss in the L1 can stall all the BEUs. BEUs are stalled for an average of 21.2 percent for the integer benchmark programs and an average of 16.2 for the floating point benchmark programs. This condition can be easily mitigated if an idle BEU can be made available to another braid that is ready to execute. Context sharing is a simple technique that improves the efficiency of resources in the BEU. The key idea is to temporarily set the stalled braid aside, thus freeing the context in the BEU for execution of another braid that is ready to execute.

Context sharing is a technique that addresses the issue of execution stalls. Implementing context sharing requires very minimal changes to the microarchitecture. Figure 5.6 shows the block diagram of a BEU augmented with two additional structures to support context sharing. The two shaded structures are required to maintain the state of the BEU. The first structure is the waiting instruction buffer. It is the same size as the instruction queue and is used to maintain the state of the instruction queue on a context change. The buffer is directly connected to the in-

Integer	
benchmarks	% stalls
bzip2	44.4
crafty	2.7
eon	2.0
gap	46.9
gcc	4.1
gzip	18.1
mcf	54.5
parser	57.2
perlbmk	1.7
twolf	4.8
vortex	4.1
vpr	13.8
amean	**21.2**

Floating Point	
benchmarks	% stalls
ammp	6.1
applu	0.5
apsi	3.9
art	82.7
equake	24.6
facerec	9.5
fma3d	1.3
galgel	15.9
lucas	1.7
mesa	1.5
mgrid	1.9
sixtrack	2.0
swim	75.1
wupwise	0.5
amean	**16.2**

Table 5.1: Percent of Cycles All Braid Execution Units Are Stalled

struction queue. This enables instructions to move from one structure to the other with very little effort. The second structure is the waiting internal register file. This structure is the same size as the internal register file and is used to maintain the state of the internal register file during a context change. It is also connected directly to the internal register file and allows the movement of registers between the two internal register files.

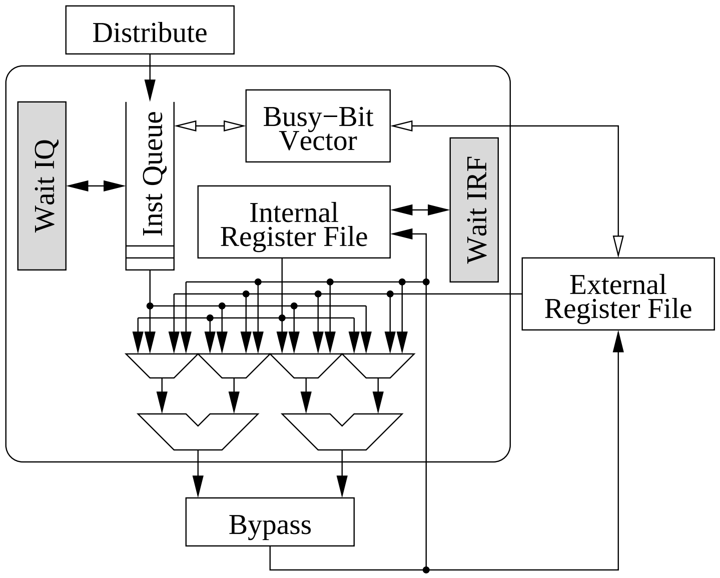


Figure 5.6: Mechanism to Support for Context Sharing

Figure 5.7 shows the state diagram for entering the context sharing mode. A counter is used to detect when a BEU can enter context sharing mode. Every cycle, this counter is incremented if a braid has been assigned to the BEU and no instructions were executed that cycle. That is, the BEU is not ready and the functional unit is idle. The counter is reset to zero otherwise. When the counter value reaches a certain threshold, context sharing is triggered. At this point, the BEU context is moved to the waiting instruction queue and waiting internal register file. The BEU context has been freed and the BEU is ready to execute a new braid. As long as there are still instructions in the new braid to be processed, context sharing mode is maintained. When all instructions in the new braid complete execution, the BEU exits context sharing mode. At this point, the context saved in the waiting buffers is restored into the instruction queue and the internal register file and processing resumes from the saved context.

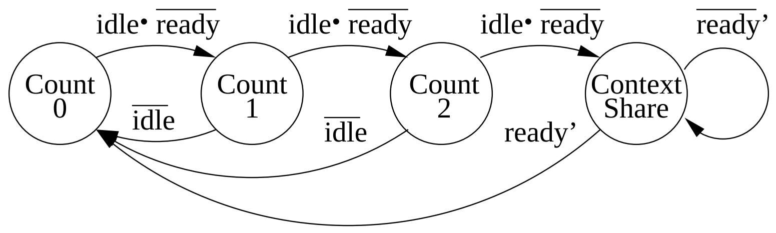

idle• $\overline{\text{ready}}$
idle• $\overline{\text{ready}}$
idle• $\overline{\text{ready}}$
$\overline{\text{ready}}$’
Count 0
Count 1
Count 2
Context Share
$\overline{\text{idle}}$
$\overline{\text{idle}}$
ready’

Figure 5.7: State Diagram for Context Sharing

Figure 5.8 shows the performance over the baseline braid microarchitecture when context sharing is enabled. On average, context sharing provides an additional 1.5% over the baseline configuration. The art and swim benchmark programs have the largest amount of stalls caused by the unavailability of BEUs. It also achieves the greatest gain using context sharing regaining between 4% and 6% of the performance.

5.3 Heterogeneous Execution Resources

The third problem is the poor utilization of the BEUs caused by single-instruction braids. These single-instruction braids are isolated instructions that do not share a dependency with any other instruction inside the basic block. The existence of these braids is a side effect of profiling preexisting binaries. Table 5.2 shows the percent of instructions that belong to single-instruction braids in the dynamic instruction stream. These single-instruction braids tie up precious resources

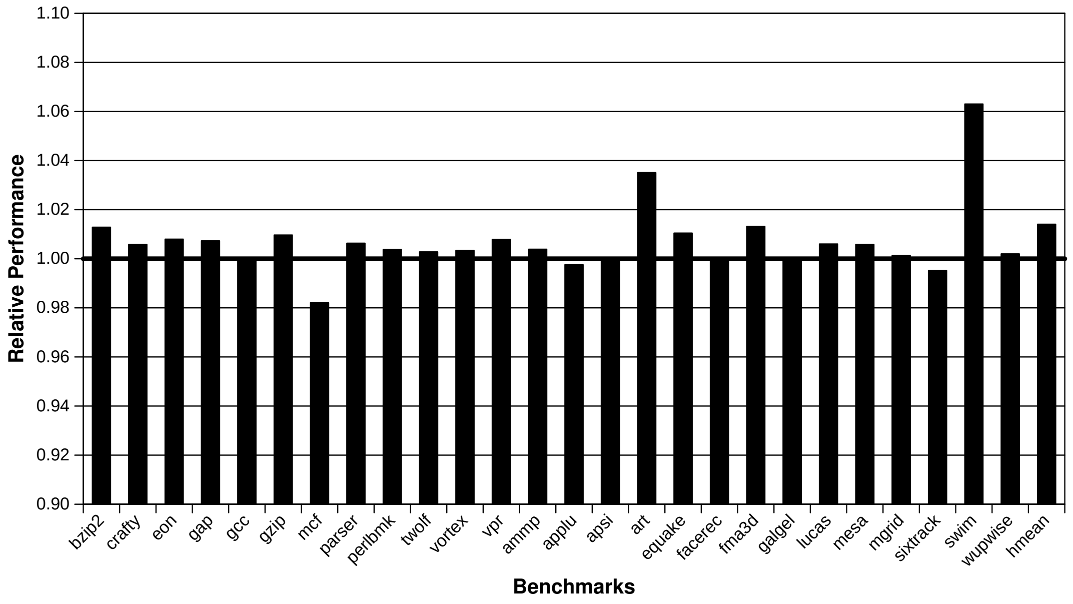


Figure 5.8: Performance Using BEU Context Sharing

that have been designed to process bigger and wider braids. Since single-instruction braids do not share the same type of processing requirements as larger braids, they should not be processed in a BEU.

Figure 5.9 shows the block diagram of the pipeline of the braid microarchitecture with the addition of a small out-of-order scheduler as indicated by the shaded block. The distribute mechanism sends single-instruction braids to this special scheduler. This scheduler is solely responsible for processing single-instruction braids.

This out-of-order scheduler is much smaller than a conventional out-of-order scheduler. The small out-of-order scheduler does not introduce significant complexity to the design. It is a small 2-wide scheduler containing not more than a few entries. This is far simpler from the design requirements of a scheduler with 32 or more entries. Figure 5.10 shows the performance when the heterogeneous

Integer	
benchmarks	percent
bzip2	15.1
crafty	18.1
eon	32.5
gap	20.8
gcc	26.4
gzip	15.4
mcf	34.8
parser	28.7
perlbmk	24.5
twolf	19.5
vortex	29.1
vpr	25.2
amean	**24.2**

Floating Point	
benchmarks	percent
ammp	13.4
applu	24.1
apsi	24.3
art	21.4
equake	21.4
facerec	28.1
fma3d	21.5
galgel	31.4
lucas	12.2
mesa	26.9
mgrid	4.4
sixtrack	27.5
swim	15.1
wupwise	22.2
amean	**21.0**

Table 5.2: Percent of Instructions from Single-Instruction Braids

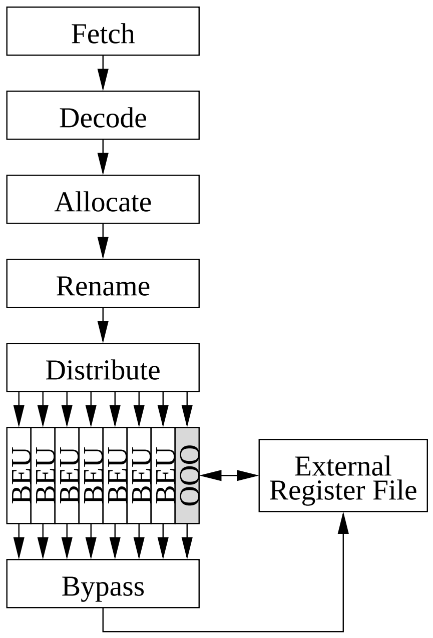


Figure 5.9: Block Diagram of Heterogeneous Execution Resources

execution resources are used. All the bars are normalized to the braid microarchitecture with seven BEUs. The left bar of each benchmark program shows the additional performance provided by one BEU. The right bar of each benchmark program shows the additional performance provided by a small out-of-order scheduler. Performance improvement increases from 2.8% to 4.0% when heterogeneous execution resources are used to handle single-instruction braids.

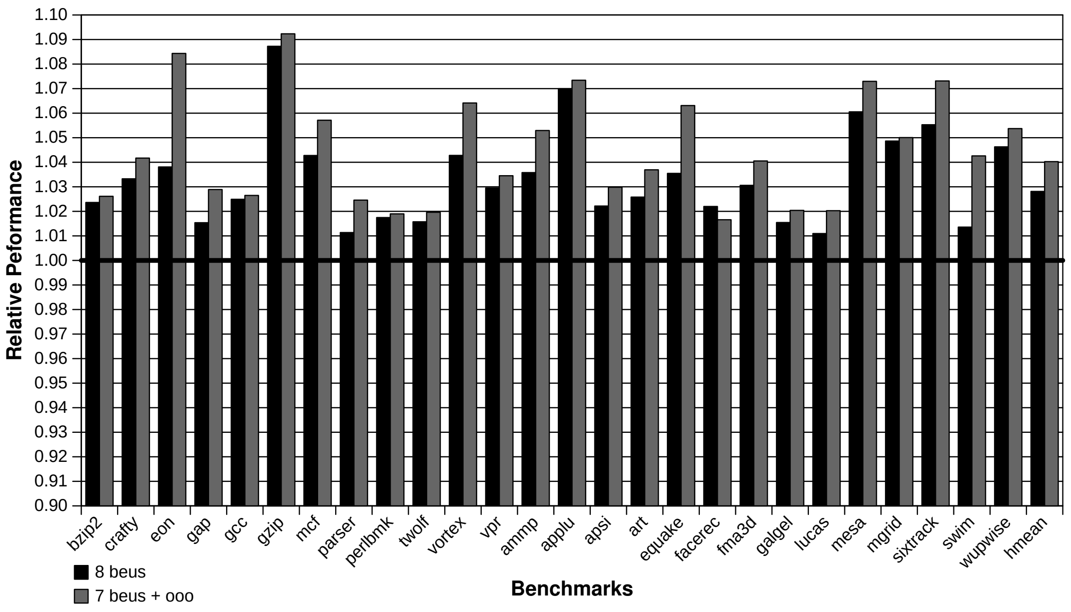


Figure 5.10: Performance Using Heterogeneous Execution Resources

Chapter 6

Hardware and Software Analysis

The braid microarchitecture is a complexity-effective alternative to an aggressive conventional out-of-order microarchitecture. This chapter presents an analysis of the hardware and software complexity in the implementation of braid processing.

6.1 Hardware

6.1.1 Renaming Mechanism

The registers associated with the external input and output operands of a braid must access the RAT to receive their proper tags. The registers associated with the internal operands of a braid do not need to be renamed. Therefore, on average, only a subset of the total registers in the fetch packet need to be renamed. The RAT is not on the critical loop and hence can be pipelined over several stages without significantly hampering performance. Braid processing provides three benefits for the process of renaming operands. First, the renaming mechanism requires fewer access ports due to the reduced bandwidth requirements. This leads to the design of a renaming mechanism with a smaller area and lower power. Second, reducing the set of registers to be renamed decreases the physical register space. A smaller register space requires fewer bits to identify each register. Fewer tag bits reduces the width of each RAT entry which also results in a smaller structure saving both area and power. Third, since fewer operands need to be renamed, fewer pipeline stages are required to support operand rename. Fewer stages lower the branch misprediction penalty.

Figure 6.1 plots the performance as a function of the number of rename ports in the braid microarchitecture. The bars in this graph are normalized to the performance of the braid microarchitecture with 16 source and eight destination rename ports. The bars for each benchmark program represent a different number of source and destination rename ports. In the braid microarchitecture, eight source rename ports and four destination rename ports are more than enough to sustain performance within 0.75% of peak performance. This slowdown increases to 1.5% when six source and three destination rename ports are used.

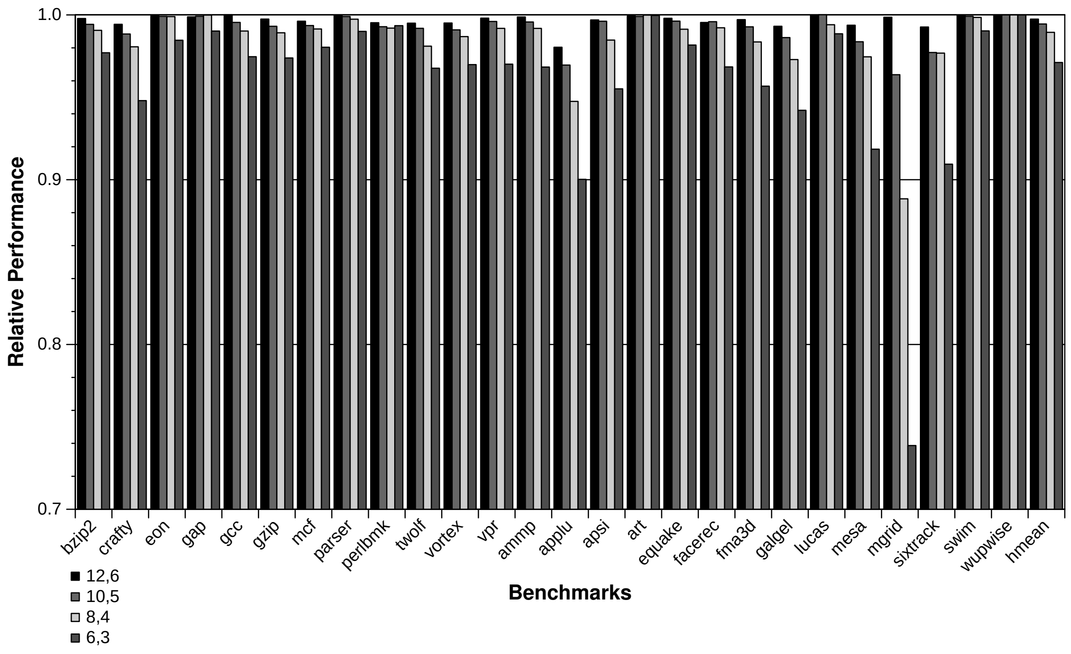


Figure 6.1: Performance Sensitivity to the Number of Source and Destination Rename Ports

6.1.2 Scheduler

In a w-wide out-of-order design, a monolithic scheduler with n entries each requires $n \times w \times 2$ full tag comparators. n entries are examined each cycle to identify instructions that are ready to execute. If the monolithic scheduling window is partitioned into separate windows, wake-up wire delay is reduced but the number of full comparators remains the same. When processing braids, the microarchitecture has been passed a carefully allocated unit of work that can be executed efficiently by a BEU. The long and narrow characteristics of a braid subgraph eliminate the need for a dynamic out-of-order scheduler. Only two instructions need to be considered for execution at the head of the instruction queue using an in-order scheduler.

6.1.3 Busy-Bit Vector

The busy-bit vector maintains the availability of external registers similar to that found in the design of in-order processors. It is an 8-bit vector where each bit represents the availability of an external register. This vector is replicated in each BEU and kept in synchronization. The instruction scheduler queries the busy-bit vector in its BEU to determine the external source operand availability of an instruction. Since the scheduler examines up to two instructions, this structure had four read ports. Given the small size of this structure, adding ports to this structure is not a problem.

A conventional out-of-order processor broadcasts a much larger set of operand tags than the braid microarchitecture. Because of this, the braid microarchitecture uses narrower comparators. A more challenging task is the process of updating busy-bit vectors. When an instruction is scheduled for execution, it broadcasts its tag to other BEUs. The other BEUs examine the broadcast tag and update the corresponding bit field in their busy-bit vector. Since a BEU contains two functional units, simultaneous broadcast of external result tags can take place. Since most braids do not produce two external results in the same cycle, a design simplification is introduced by restricting each BEU to broadcast only one tag per cycle. Therefore, each busy-bit vector requires eight write ports. Each busy-bit vector also requires eight comparators. Since the external register space is small, a 3-wide bus is all that is needed for each tag. Furthermore, BEUs are very compact unlike typical clusters which are spread apart. This compactness allows a broadcast signal to span the width of the pipeline in one cycle.

6.1.4 Internal Register File

An average of 50% of all register file accesses are to/from the internal register files due to the partitioning of the register space. Each internal register file contains eight entries. Since each BEU can execute up to two instructions per cycle, each internal register file has four read ports and two write ports. The internal register file is disjoint from the external register file. The internal register file of one BEU is also disjoint from those in other BEUs. Values never propagate between any of the register files. Values in the internal register file do not need to be maintained for the execution of a subsequent braid and are naturally discarded once a braid finishes execution in the BEU. The reduced number of accesses and ports to the internal register file allow a design with a smaller area and lower power requirements. The entire working set of values in the program is supported by the many disjoint internal register files.

6.1.5 External Register File

The remaining 50% of register values that do not access the internal register file access the external register file. This 16-entry register file is twice as large as the internal register file and also accessed by all BEUs. It is designed with six read ports and three write ports. Like the internal register files, the external register file also has a small number of entries and ports compared to the register file in a conventional microarchitecture. This results in structure with a smaller area and lower power requirements. The smaller area also eliminates the pipelined access to the register file, allowing access to be performed in a single cycle. The resulting shorter pipeline also lowers the misprediction penalty.

Figure 6.2 plots the performance as a function of the number of read and write ports of the external register file in the braid microarchitecture. Each bar is a two-tuple representing the number of read and write ports of the external register file. The bars in this graph are normalized to the performance of the braid microarchitecture with an external register file that has 16 read ports and 8 write ports. As the number of read and write ports is decreased, there is negligible slowdown. With a few as six read ports and three write ports, the braid microarchitecture obtains performance within 0.3% of the performance from using a full set of read and write ports is achieved. Mgrid is especially sensitive to port scaling because it has a large number of external inputs and outputs as shown in Table 2.4. There is a 1.5% slowdown going to four read ports and two write ports.

Figure 6.3 plots the performance as a function of the number of entries in the register file in a conventional out-of-order microarchitecture. The bars in this graph are normalized to the performance of a conventional out-of-order microarchitecture with 256 registers. Using 32 registers in an out-of-order design causes 8% degradation in performance, and using 16 registers causes 21% degradation in performance.

Figure 6.4 plots the performance as a function of the number of entries in the external register file in the braid microarchitecture. The bars in this graph are

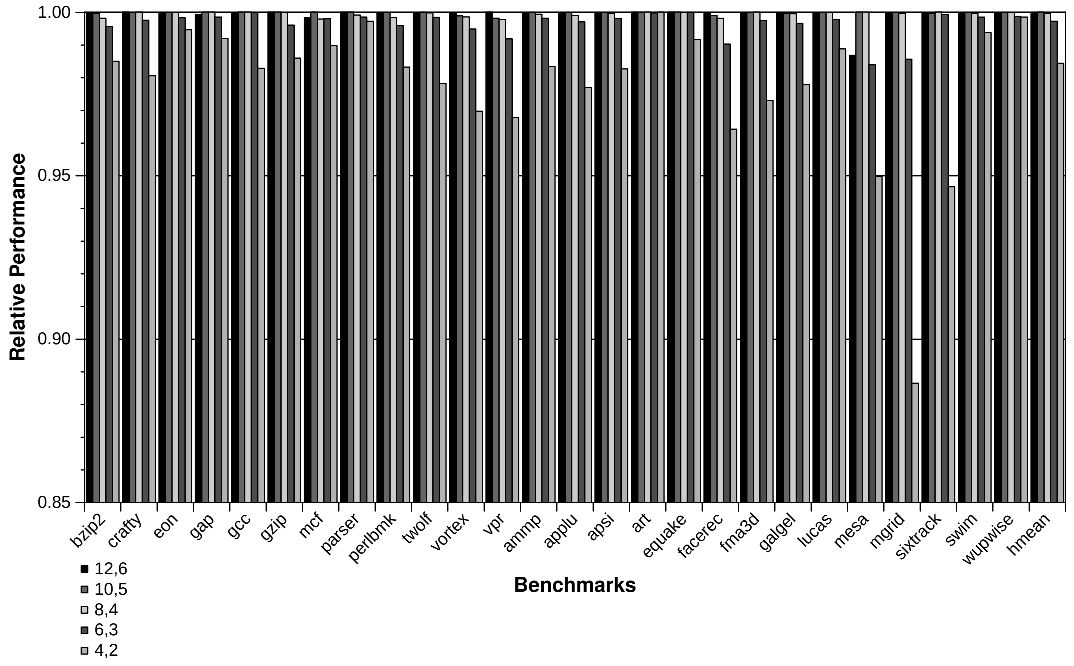


Figure 6.2: Performance Sensitivity to the Number of Register File Read and Write Ports

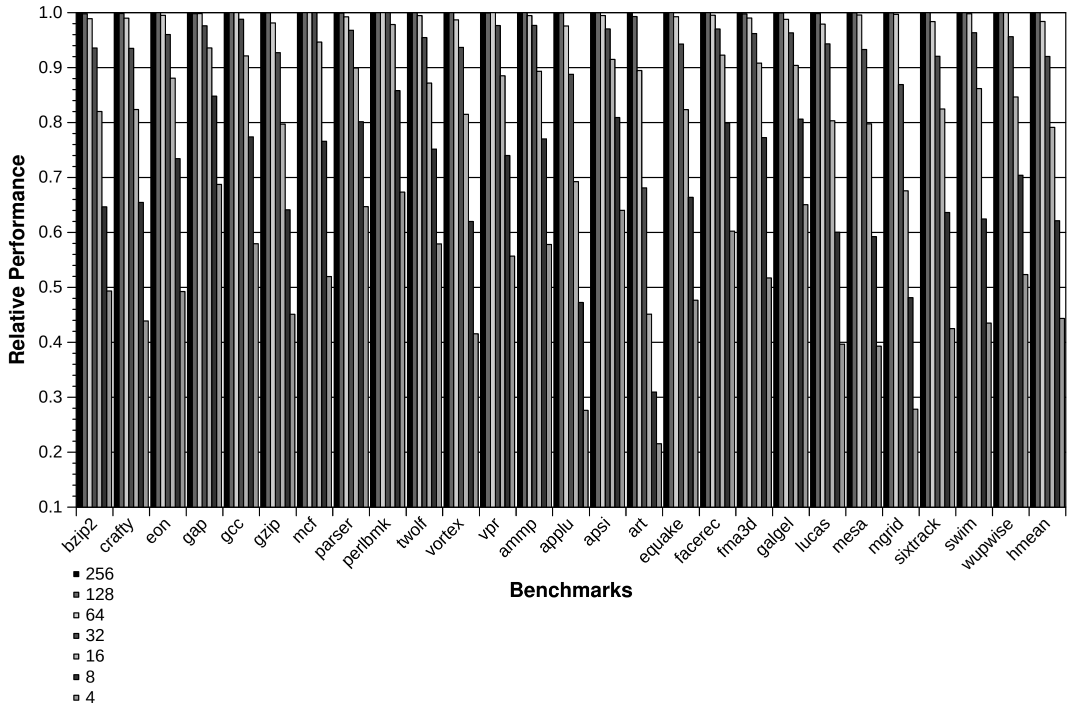


Figure 6.3: Performance Sensitivity to the Number of Registers in an Out-of-Order Microarchitecture

normalized to the performance of the braid microarchitecture using a 256-entry external register file. Since most of the operands access the internal register files, there is less pressure on the external register file. Reducing the number of external registers does not significantly affect the performance of the braid microarchitecture until reaching four registers. It can be seen using a small 8-entry external register file is sufficient to maintain the performance within 1% of the performance from using a 256-entry register file.

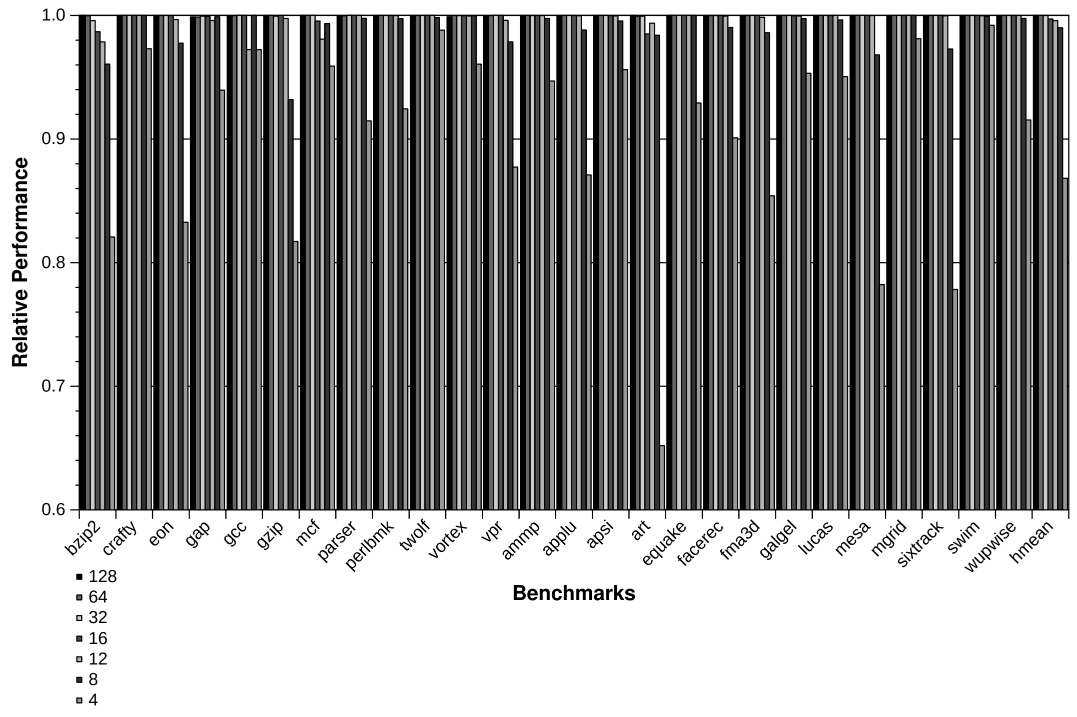


Figure 6.4: Performance Sensitivity to the Number of External Registers in the Braid Microarchitecture

6.1.6 Bypass Network

An instruction specifies whether source operands receive a value from the internal register file or the external register file. However, an operand that requires a value from the external register file may receive its value from the bypass network due to data forwarding. Figure 6.5 plots the distribution of locations where source operands are read. Bypassed values are read 8% of the time. Internal register file values are read 44% of the time. External register file values are read 48% of the time.

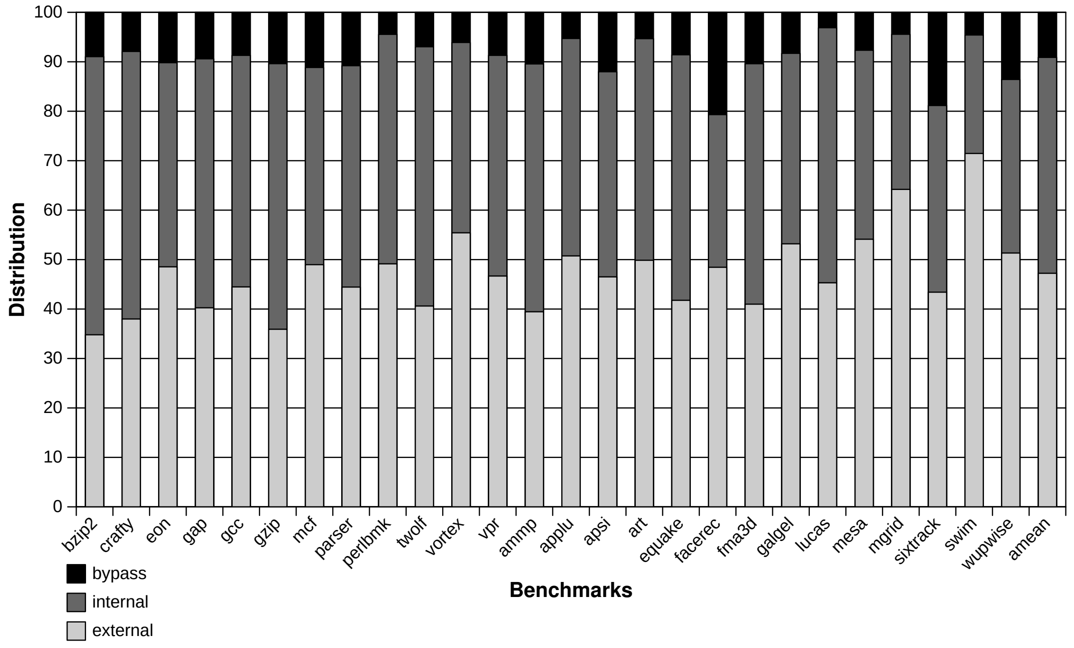


Figure 6.5: Distribution of Source Operand Locations

The bypass network in a conventional out-of-order design requires multiple levels due to pipelined writes to the register file. In the braid microarchitecture, the bypass network contains only one level because it takes one cycle to complete a write to the external register file. This is due to the fewer number of entries and ports of the external register file. Since there are fewer external values written to the external register file, there are also fewer values that require the bypass network. Figure 6.6 plots the performance as a function of the number of bypasses paths in the braid microarchitecture. The bars in this graph are normalized to the performance of the braid microarchitecture with 16 bypass paths. The number of paths corresponds to the average number of supported value bypasses per cycle. Since internal values do not require bypassing, the number of bypassed values is greatly reduced. Supporting the capability of bypassing four values per cycle in the braid microarchitecture does not hamper performance, and supporting two values per cycle obtains performance that is well within 0.6% of the performance from using a full set of bypass paths. A 6.5% performance drop takes place when only one value is supported.

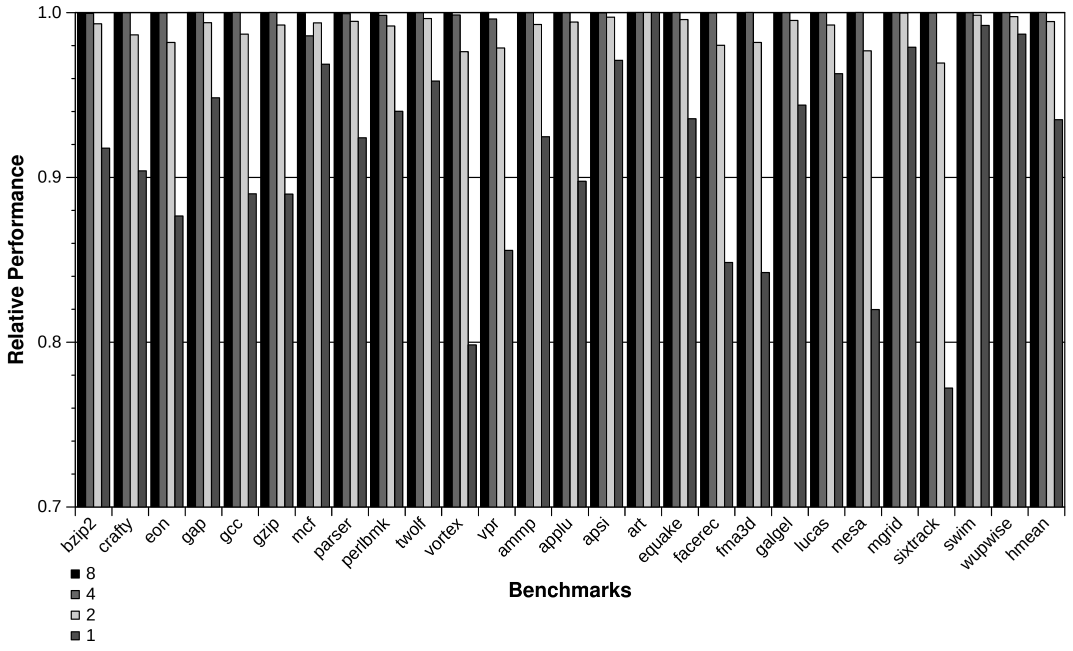


Figure 6.6: Performance Sensitivity to the Number of Bypass Network Paths

6.1.7 Reducing Pipeline Stages

The braid microarchitecture reduces the number of stages in the pipeline. First, the operand rename stage is reduced by two stages. This is due to the lower bandwidth requirements of renaming external registers. Second, register file access is reduced by two stages. This is due to the fewer number of entries and ports of the external register file. In all, the pipeline is shortened by four stages reducing the branch misprediction penalty by four cycles. Figure 6.7 plots the extra performance as a result of the shorter pipeline. The integer benchmark programs gain more from the shorter pipeline. This is because integer programs have a higher branch misprediction rate. On average, the shorter pipeline results in 2.6% performance boost for the benchmark programs.

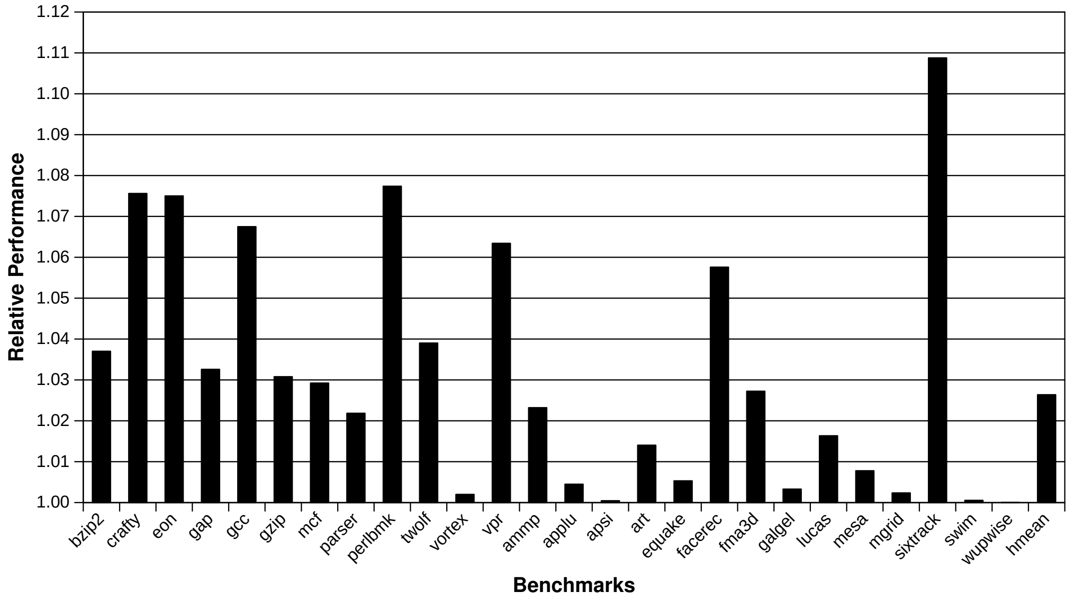


Figure 6.7: Performance from Reducing Pipeline Stages

6.1.8 Clock Frequency

The structures on the critical path of a processor include the renaming mechanism, the instruction scheduler, and the bypass network. The braid microarchitecture simplifies each of these structures resulting in a design requiring less area. Thus, the critical path is shortened allowing the design to run at a higher clock frequency. With the benefit of this additional frequency compensation, the braid microarchitecture can achieve even higher performance closing the performance gap between itself and an aggressive out-of-order microarchitecture.

6.1.9 Perfect Front-End and Memory System

The braid microarchitecture targets the design simplification of the structures in the execution core. To see the effectiveness of the braid microarchitecture, the execution core must be fully used. The following set of experiments plots the performance from eliminating the effects of imperfect instruction and data delivery mechanisms.

The first experiment considers the use of a perfect branch predictor. This branch predictor is always correct and thus never fetches instructions on the wrong path. Figure 6.8 plots the performance using perfect branch prediction over realistic branch prediction. The bars in this graph are normalized to the performance of a conventional out-of-order design with realistic branch prediction. The left two bars of each benchmark program represent the performance of the braid microarchitecture with realistic and perfect branch prediction. The right two bars represent the performance of a conventional out-of-order microarchitecture with realistic and perfect branch prediction. Except for a couple benchmarks, the change in performance is not too significant. On benchmark programs where the conventional out-of-order processor achieves a performance gain, the braid microarchitecture also achieves a comparable performance gain.

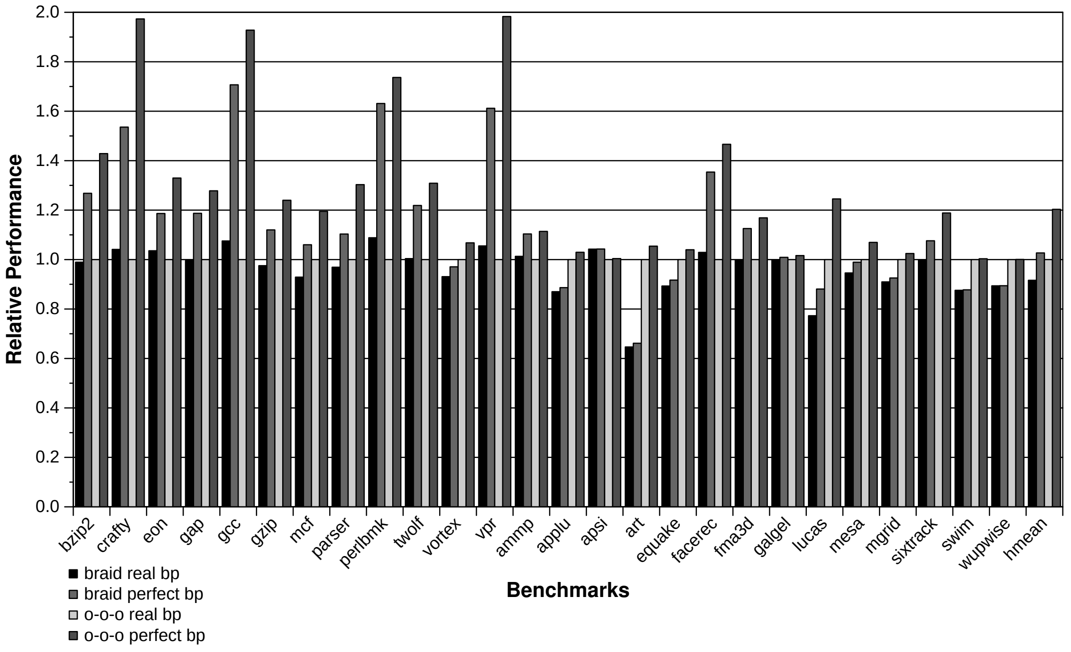


Figure 6.8: Performance Using Perfect Branch Prediction

If instructions are not fetched at a fast enough rate, branch predicting is not as beneficial. The following experiment assumes the use of a perfect branch predictor as well as the use of perfect instruction and data caches. Figure 6.9 plots the performance of the realistic and perfect instruction and data delivery mechanisms. The bars in this graph are also normalized to the performance of a conventional out-of-order design with realistic branch prediction. This graph shows that the braid microarchitecture scales as well as the conventional microarchitecture with more aggressive branch predictors and memory systems.

While the memory system is important for performance, this dissertation does not address memory system issues. The problems to be solved for the memory system are orthogonal to the problems to be solved for the execution core. Figure

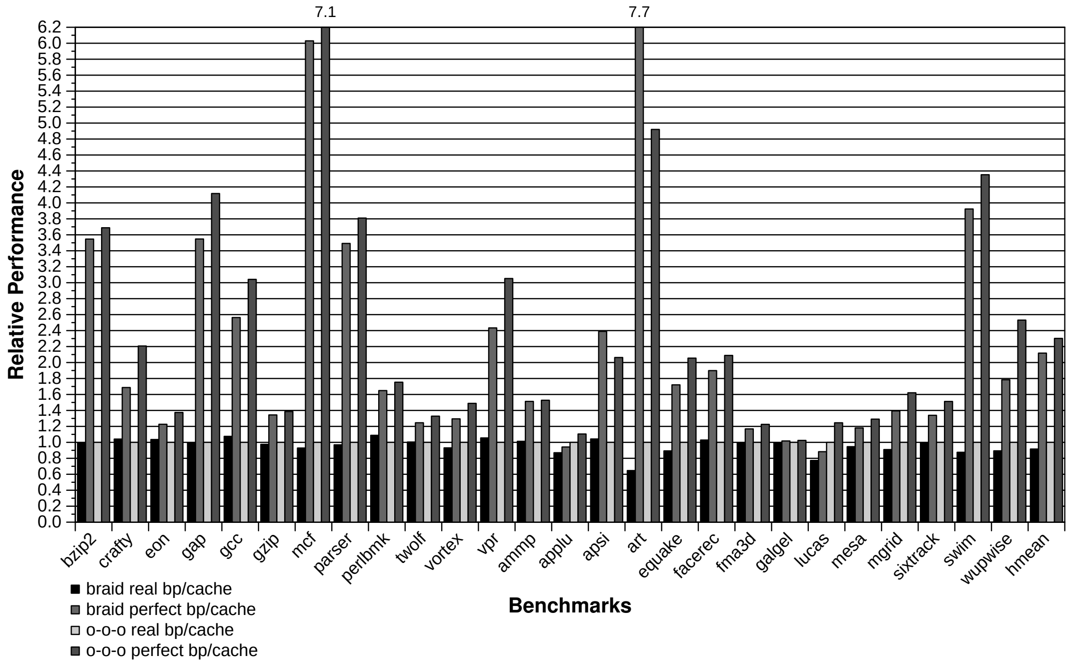


Figure 6.9: Performance Using Perfect Branch Prediction and Perfect Instruction and Data Caches

6.10 provides results of the braid microarchitecture as a function of the number of cycles to main memory. All the bars are normalized to the baseline braid microarchitecture. Results for 100, 200, and 400 access cycles to memory are shown. If the number of cycles to main memory is halved from the baseline, 30% additional performance can be achieved, and if the number of cycles is halved again, an additional 20% additional performance can be achieved.

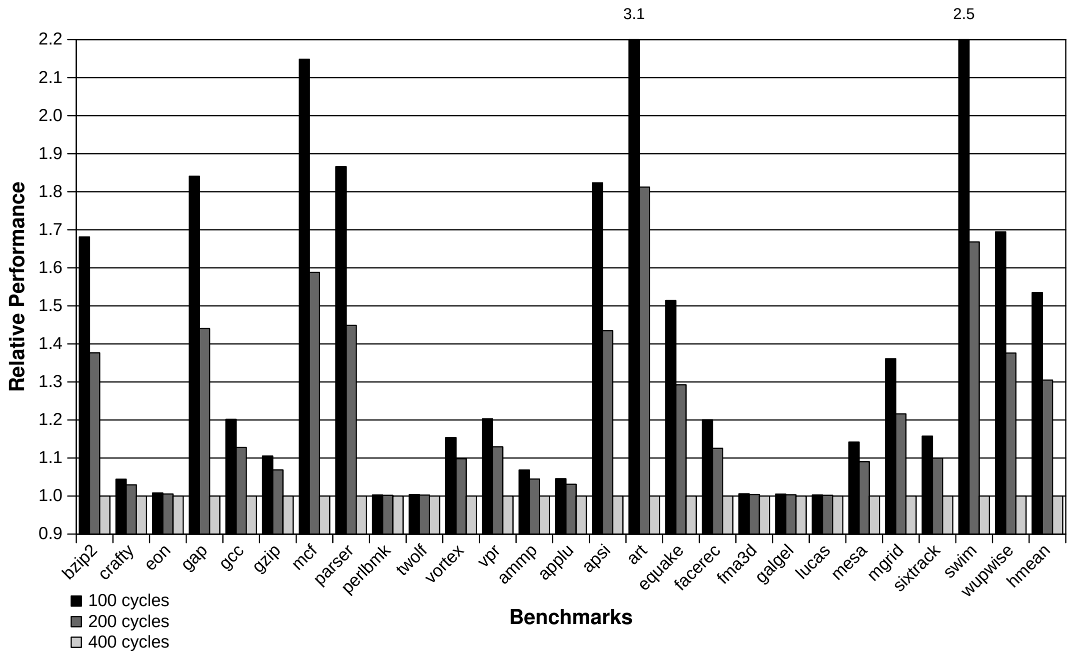


Figure 6.10: Performance Sensitivity to the Number of Cycles to Main Memory

6.2 Software

6.2.1 Strands versus Braids

A strand as proposed by Kim and Smith is a chain of dependent instructions [40]. It has as width of exactly one, whereas a braid has an average width of 1.3. Restricting strands to be a 1-wide dependent chain of instructions greatly limits the number of instructions that can be a part of a strand. This is because the irregular dataflow graph of program does not easily breakdown into long strands. When the 1-wide restriction is lifted, larger dataflow subgraphs can be formed. Even though a braid is larger than a strand in size, a braid maintains an average width that is close to one.

6.2.2 Tradeoffs of Dataflow Graph Size

At one extreme, the entire dataflow graph of a program can be considered as one large braid. In this model, all values are internal. At the other extreme, the dataflow graph can be broken into single-instruction braids. In this model, all values are external. At both of these extremes, the scheduler, the register file, and the bypass network resemble those found in a conventional out-of-order microarchitecture. These components become a design challenge at wide issue widths and will become impractical to implement in future designs. An efficient design point lies in between the two extremes. Braids convert a seemingly irregular dataflow graph into regular dataflow subgraphs achieving a balance between the use of external and internal operand values.

6.2.3 Spill Code

A compiler uses registers to pass values between instructions in the program. Due to a limited set of architectural registers, the compiler uses a technique to temporarily free registers by storing them to memory and restoring them when they are needed again. It accomplishes this by inserting spill and fill code when the working set size of values exceeds the number of registers. In the profiling analysis used in this dissertation, only data values propagated through registers were considered. There is lost opportunity when using the profiling technique because it does not account for data values propagated through the memory system via spill and fill code. With a braid-aware compiler, larger braids can be formed reducing the amount of spill and fill code. This should lead to register allocation requiring fewer external registers.

6.2.4 Software Compatibility

The braid microarchitecture is capable of running legacy applications at lower performance. This is accomplished by treating all register operands in the program as external register operands. The renaming mechanism provides external register tags to the architectural registers. Instructions can be distributed to BEUs using a simple steering policy like round robin. Instruction queues in the BEU buffer instructions until they become full. The instructions within each BEU follow in-order scheduling. All operands access the external register file and the internal register files remain unused. The performance of running legacy applications on the braid microarchitecture is not optimal, resembling that of an in-order design.

6.2.5 Instruction Set Architecture Annotations and Code Bloat

The Alpha ISA was augmented to support braid processing. Each register field is five bits wide. The braid microarchitecture specifies eight external registers and eight internal registers. This means that four bits are required to specify a source register operand (three to specify the register name and one to specify the temporary operand bit). A destination register requires five bits (three to specify the register name and two to specify external/internal destination bits). To specify the braid start bit in the case of zero-destination or one-source register instructions, an extra bit is borrowed from the displacement field or an unused field in the instruction encoding. By carefully making use of the available bits in the instruction, there is no code bloat in the Alpha binary.

The x86 ISA provides more flexibility for passing information to the microarchitecture through the use of instruction prefixes. This ISA supports prefix bits which can provide the needed braid annotations for operands of the instruction without modifying the actual operand field. One or more prefixes can be appended to the instruction at the start of each braid specifying the operands and characteristics of the braid. This method is less intrusive to the individual instructions of the original binary but adds additional code to the binary.

If instruction prefixes are not available and there are not enough available bits in the instruction to encode braid information, branch and link instructions can be used [23] to provide braid information. A branch and link instruction is inserted immediately before the starting instruction of a braid. When the branch and link instruction is encountered, program control is transferred to another region of memory where information can be provided for the upcoming braid. After the braid information is obtained, program control is returned to the first instruction of the braid. This technique of providing braid information is also less intrusive to the individual instructions of the original binary but adds additional code to the binary.

Chapter 7

Related Work

Although the braid microarchitecture shares some similarities with other proposals in the literature, no single scheme achieves the combined benefits of the braid microarchitecture. This chapter compares and contrasts the braid microarchitecture with various proposals.

7.1 Basic Block-Based Processing

Most of the proposals in the literature follow a style of processing wherein instructions within a basic block are processed equally to one another. That is, instructions of a basic block are issued into a scheduling window and scheduled for execution without taking into account their data dependencies. In the braid microarchitecture, the instructions of a braid travel through the distribution and scheduling process as a unit. The instructions in one scheduling window are not a random set of instructions but a set of tightly connected instructions related by their data dependencies. With the information to identify the different dataflow subgraphs within a basic block, the braid microarchitecture can carry out instruction scheduling without complex hardware.

7.1.1 Trace Processing

The braid microarchitecture and the trace processor [59] have important differences. First, the trace processor does not distinguish between different dataflow subgraphs to simplify instruction scheduling. A trace in the trace processor consists of a set of dynamic contiguous basic blocks identified at runtime. When a trace is issued, all instructions are treated the same and enter the same instruction scheduler. The out-of-order scheduler operates on this entire set of instructions even though there can be multiple disjoint dataflow subgraphs. The braid microarchitecture uses a different in-order scheduler to process each of the predetermined dataflow subgraphs. Second, the trace processor requires runtime capturing and marshaling of traces. Traces do not share the same benefits as braids. The braid microarchitecture leverages the compiler-identified subgraphs requiring no runtime analysis of instructions. Although the trace processor also uses local and global registers, the concept of partitioning the register space was first proposed by Sprangle and Patt [68]. Local and global registers in the trace processor are identified at runtime in the trace preprocessing unit. Vajapeyam and Mitra [72] proposed a similar technique of processing traces. In the braid microarchitecture, the compiler identifies external and internal registers at compile time.

7.1.2 Multiscalar

Multiscalar [65] shares similar characteristics with the braid microarchitecture but also has important differences. Both paradigms process a piece of work that is identified by the compiler. The unit of work in Multiscalar is the task which is a very large piece of work consisting of a set of basic blocks. In contrast, the braid microarchitecture uses the braid as a unit of work. In Multiscalar, each processing unit is an out-of-order processor. The processing units are arranged in a ring formation. Since tasks are assigned to one processing unit, the scheduler in that unit must consider all the instructions in that unit for execution. Unlike the braid microarchitecture, there is no notion of dataflow subgraphs in Multiscalar. In Multiscalar, the compiler identifies which register values need to be forwarded to other processing units on the ring and which values are no longer needed. This is done through the use of a bit-mask conveyed though the ISA. In the braid microarchitecture, most of the register communication is performed through internal registers. Internal registers are localized to a braid and kept in a small register file. Internal registers are implicitly freed when they are overwritten or when the entire braid has been processed. Multiscalar uses a larger granularity of work and a processing unit topology that increases the latency of communications between processing units. These issues are avoided in the braid microarchitecture.

7.1.3 rePLay

The rePLay framework [52] is another processing paradigm that operates on large units of work called frames. A frame is a trace-like entity consisting of many basic blocks which are captured dynamically. It is essentially a very large trace where the entire trace is asserted to be from the correct path of execution. The goal of the rePLay framework is to identify large chunks of work rather than identifying disjoint dataflow subgraphs. The rePLay optimization engine is used to analyze frames as they are captured dynamically to improve the efficiency of the code. The optimization engine adds additional complexity to the pipeline and increases power requirements.

7.2 Strands, Dependency Chains, and Subgraphs

7.2.1 Strands

The term strand was first coined by Marquez et al. [45] in the superstrand microarchitecture. A strand is a dataflow subgraph which terminates on two conditions. The first condition is a long-latency instruction, and the second is a branch instruction. Even though identifying strands for processing can simplify hardware complexity, certain design choices limited the performance of the superstrand microarchitecture. First, the superstrand microarchitecture allows a strand to execute only when all of its operands are available. Unless strands are extremely short, there is a high probability that most strands do not have inputs only at the top of the dataflow subgraph. There will also be inputs feeding into the middle of the dataflow subgraph. Therefore, waiting for all operands to be available is too performance limiting. Second, the scheduler must monitor a variable number of ready instructions in order to determine if a strand can execute. This increases the design complexity of the scheduler. Third, the heuristics used to form strands do not partition the dataflow in such a way to reduce inter-processing unit communications. Terminating a strand at a long-latency operation or a branch operation is too simple because the dependencies of a strand may not terminate on these instructions.

The term strand was redefined by Kim and Smith [40] in the context of instruction-level distributed processing (ILDP). This definition of the strand identifies a dataflow subgraph that consists of a single chain of back-to-back instructions. In a strand, the result of one instruction solely feeds the input of the next instruction. The instruction-level parallelism or width of a strand is exactly one. A microarchitecture was proposed to leverage the single-wide chains. Local communication of a strand takes place through a single accumulator in the processing element to which the strand was steered. Global communication of a strand takes place through duplicated register files. A compiler is used to identify strands. A strand terminates at an instruction that does not produce a value or produces a value needed by more than one instruction. Although the hardware of a processing element is extremely simple in design, ILDP does not achieve the full benefits of braids. Since, a strand has an instruction-level parallelism of one, it is difficult to find long strands in the program dataflow. By not requiring a fixed width in the braid microarchitecture, larger dataflow subgraphs are formed even though width remains very close to one. Since braids are larger units of work, there is more benefit in processing them.

Sassone and Wills [61] proposed a mechanism that identifies strands dynamically and stores them in a strand cache. A fill unit [51], similar to one used in trace cache design, captures instructions. The fill unit also identifies strands by maintaining an operand table that tracks temporary operands of instructions. The use of the fill unit adds complexity to the design and increases power requirements. Sassone et al. [62] later examined the use of strands in the embedded processor space. These strands are identified at compile time and are called static strands. A method of encoding static strands in the program binary with minimal changes to the original ISA is presented. The static strand is an extension of the strand in ILDP with one difference. Sassone introduced a restriction on the makeup the internal strand instructions making static strands smaller than the strands in ILDP. As mentioned already, restricting the number of instructions in a dataflow subgraph lessens the benefits of processing it.

7.2.2 Dependency Chains

Narayanasamy et al. [48] proposed a clustered microarchitecture that operates on dependency chains. A dependency chain is a dataflow subgraph identified by the compiler. It is not as restrictive as strands. Dependency chains are formed via profiling analysis. The hot traces of the program are selected, and their dataflow subgraphs are converted into dependency chains. As an optimization of the microarchitecture, Narayanasamy incorporated code duplication to enlarge dependency chains. Although dependency chains help reduce design complexity, they introduce some difficulties in the microarchitecture. First, dependency chains rely heavily on hot paths. Since hot paths are heavily dependent upon input set as well as program phase behavior [39], the dependency chains identified using one profiling input set may not hold for another input set. A branch misprediction in the middle of a long dependency chain means rolling back state to the very beginning of the dependency chain. Thus, processing dependency chains can lead to costly branch recoveries. Second, when code duplication is used, the same basic block can often end up in multiple dependency chains which causes significant code duplication in the program binary, decreasing the efficiency of the instruction cache.

7.2.3 Subgraphs

There have been other proposals of dataflow subgraph processing. Kim and Lipasti [41] proposed macro-op scheduling. A macro-op is a fused entity containing two dependent instructions. A macro-op is used to simplify the design of the scheduler by allowing the scheduler to be pipelined across two cycles. A macro-op is identified dynamically through detection logic similar to a fill unit in the trace cache design [51]. Each macro-op is assigned one pointer that is used to identify the macro-op in the pipeline. Processing one pointer rather than two instructions allows more efficient use of execution core resources. Macro-op scheduling simplifies the design of the scheduler but does not remove the use of the out-of-order scheduler. Processing macro-ops does not provide the full benefits provided by the braid microarchitecture. The braid microarchitecture uses in-order schedulers as well as identifies internal register values for reducing register file complexity. The identification of macro-ops requires the use of a fill unit which adds complexity to the design and increases power requirement.

Bracy et al. [13] proposed using dataflow mini-graphs to amplify the bandwidth of various microarchitecture structures. A dataflow mini-graph is a dataflow subgraph that is identified at compile time. Each mini-graph must meet a specific set of requirements. It must have two inputs, one output, at most one memory reference, and at most one control instruction. Each mini-graph is referenced via a handle. The use of the handle allows the amplification of many structures in the pipeline. A set of ALUs, arranged in a pipelined fashion where one ALU feeds the next, is used to process the mini-graphs. Since a mini-graph can consist only of instructions that map directly onto the predefined ALUs, the size and composition of mini-graphs are restricted by the number and type of ALUs. The braid microarchitecture neither restricts the size nor the type of instructions in a braid. Thus, the braid microarchitecture maximizes the reduction of external register communication.

Clark and his colleagues [23] [22] [21] proposed the processing of dataflow subgraphs on a configurable compute accelerator in general purpose and embedded processors. Subgraphs can be identified dynamically or statically. If done statically, the compiler using profile analysis identifies frequently executed subgraphs with a predefined number of inputs and a predefined number of outputs. A set of functional units is instantiated and arranged in such a way to speed up the processing of the dataflow subgraph. The set of functional units is called a configurable compute accelerator. A special branch and link instruction inserted into the program notifies the processor that it is about to execute a subgraph. A separate structure provides the necessary information required to execute the subgraph including the inputs and the control signals needed by the configurable compute accelerator. The configurable computer accelerator is an interesting idea but designing a specific accelerator for general purpose computing is not practical because each program has a different set of characteristics and dataflow. Configurable computer accelerators are more applicable to embedded processors where a specific task is continuously being performed.

7.3 Register File

7.3.1 External and Internal Registers

Sprangle and Patt [68] proposed the concept of separate external and internal register sets in the context of a statically tagged ISA. The compiler specifically produces code that avoids output dependencies by writing results of instructions within the basic block to different registers. This eliminates the dependency checking logic simplifying the renaming process in the microarchitecture. Furthermore, a bit associated with each source operand specifies whether the value is read from the internal or external register file. Another bit associated with the destination operand specifies where the result should be written. Values in the internal registers are valid only within the context of the basic block. Although the identification of external and internal registers implicitly identifies dataflow subgraphs within the basic block, dataflow subgraphs were not presented as a technique of simplifying the instruction scheduler. Later proposals also exploit the concept of separate external and internal register sets [59] [72] [48] [40]. These proposals call internal registers local and external registers global.

7.3.2 Increasing Effective Register File Size

The braid microarchitecture provides a larger effective register set without physically adding more entries. A number of proposals have also suggested techniques to increase the effective size of the register file. These proposals differ from the braid microarchitecture in their implementation.

González et al. [32] proposed virtual-physical registers. Rather than allocating a register at the time the instruction enters the pipeline, this idea delays the allocation until a value is actually produced. This creates the effect of a larger physical register file allowing more simultaneous in-flight instructions.

Lozano and Gao [44] observed that short-lived values made up a significant portion of the values produced in a program. A technique is proposed that avoids allocating a physical register to a short-lived value. Short-lived values are maintained in buffers. Thus, this technique avoids committing short-lived values to the register file. This is done with the help of the compiler. Since short-lived values do not occupy register file space, there is an effective increase in the size of the register file. Ponomarev et al. [56] followed this concept and presented a runtime approach.

Martin et al. [46] used the compiler to provide dead value information by making assertions in the program that certain registers will not be used again. The dead value information allows the processor to free registers earlier. Thus, this technique also increases the effective size of the register file.

Another technique to increase the size of the register file is register file packing proposed by Ergin et al. [27]. It is observed that most of the values are narrow meaning that the most significant bits of the values contain no information. To exploit this, multiple narrow results are packed into a single physical register to effectively mimic a larger register set.

The braid microarchitecture provides a larger effective register set through the identification of external and internal registers. Internal registers of a braid are valid only within the braid. Once the braid has finished processing in a BEU, the internal registers are implicitly freed. By freeing registers early, the braid microarchitecture emulates a larger effective register set. This is accomplished without extra instructions or identification at runtime. The more efficient use of the register space allows the microarchitecture to support a larger set of registers than a conventional design.

7.3.3 Increasing Register File Size and Access Bandwidth

A number of proposals have suggested techniques to allow register file designs higher access bandwidth as well as larger sizes. Seznec et al. [63] analyzed the physical constraints of a large register file and proposed a microarchitectural organization to increase access bandwidth. In register write specialization register read specialization, the execution core is divided into four quadrants. Each quadrant is connected to a subset of the register file. In this model of processing, an instruction in a quadrant can only read from and write to the register file subset connected to that quadrant. Although the number of read and write ports to the register file is reduced, a more complex register renaming mechanism is required to support this model of processing. The renaming mechanism has to be aware of the instruction window load in each of the quadrants to be able to load balance between quadrants. The braid microarchitecture lessens register file access bandwidth while also lessening register file and rename bandwidth.

Banking is a technique that improves access bandwidth and energy requirements. A number of proposals have suggested banking the register file. Pericàs et al. [54] proposed a microarchitecture with a front-end register file. After rename, an instruction can retrieve its operand if that operand is available in the front-end register file. Rather than implementing enough ports to support the worst case access scenario, banking is used to save complexity and power. Tseng and Asanović [71] proposed using register file banking for a high-frequency processor design. A simple mechanism allows instructions with conflicting accesses to reschedule. Wallace and Bagherzadeh [73] show that banking can reduce the register file requirements of a superscalar processor to that of a scalar processor. Ayala et al. [8] uses the compiler for bank assignment of registers in order to reduce energy consumption. While banking is useful to improve bandwidth, the hardware techniques mentioned add extra hardware complexity by requiring some form of table look-up and update. The static banking decision may not be representative of runtime behavior. The braid microarchitecture does not require banking because access to the register files is distributed. Internal register files can be accessed by at most two instructions locally within the BEU. The external register file can be accessed by at most three instructions.

Another technique to improve access bandwidth is the use of 2-level register files. Typically, the first level has a few entries with many ports, and the second level has many entries with few ports. Zalamea et al. [77] proposed a design where the compiler explicitly manages the movement of values in the hierarchy. Balasubramonian et al. [9] and Cruz et al. [25] discuss a hardware solution to the problem of managing values between levels. This is accomplished by monitoring the usage of registers at the rename stage of the pipeline. Butts and Sohi [18] improves upon this scheme by explicitly tracking the number of uses via a degree of use predictor. Yung and Wilhelm [76] and Borch et al. [12] proposed the use of a small buffer next to the functional unit that caches recent results. This buffer can provide operand values and complements the main register file. Oehmke et al. [49] proposed the virtual context architecture where the register file is treated as a cache of a larger memory-mapped logical register space. The microarchitecture injects loads and stores to perform fills and spills on demand when the working set size of registers exceeds the physical register space. This technique allows the implementation of microarchitecture schemes requiring a large register footprint. All the schemes mentioned require either additional instructions or extra hardware to manage the movement of values in the hierarchy. The braid microarchitecture simplifies the use of registers by explicitly marking external and internal registers in an instruction. No additional instructions are needed to manage values, and no values are tracked at runtime.

Register file replication as implemented on the Alpha 21264 [38] is another technique to increase access bandwidth by creating an exact copy of the register file. In this setup, each register file supports half of the required bandwidth. Together, both register files provide the full bandwidth required by the execution core. This technique requires duplicating the register file. It saves the design complexity of the read ports but does not reduce the complexity of write ports.

While the mentioned techniques have been proposed to increase the access bandwidth of the register file, these techniques require additional instructions in the pipeline or additional hardware structures to track values in the pipeline. The braid microarchitecture achieves the needed bandwidth without the additional overhead. Since the compiler partitions the register space into external and internal registers, each register file contains a smaller number of entries and a small number of ports. The partitioning allows each register file to be accessed independently of others. Furthermore, internal results do not need to be written back to the external register file. The techniques mentioned above do not share this capability.

7.4 Compiler Identified Dependencies

Some techniques for allowing the compiler to explicitly specify instruction dependencies have been proposed. These proposals differ from the braid microarchitecture in the way the dependency information is conveyed.

The Block-Structured ISA proposed by Melvin and Patt [47] uses the compiler to generate code blocks that simplify processing. The compiler embeds instruction dependency information in the header of the code block. The microarchitecture to implement the block-structured ISA has a much simpler dependence checking logic since many instruction dependencies are made explicit by the compiler. A technique called block enlargement is used to increase the size of the code block. This works by duplicating blocks and consolidating the duplicated blocks with subsequent blocks. A larger code block provides more code movement and optimization potential within the block. It also offers a higher instruction fetch rate than fetching conventional basic blocks. Braids provide further simplification of hardware complexity by explicitly identifying dataflow subgraphs within the basic block.

The Intel Itanium 2 is a VLIW processor that implements the IA-64 instruction set [3]. The processor fetches two bundles every cycle where a bundle consists of three instructions. Each bundle contains a template which explicitly specifies the dependencies between instructions within the bundle and dependencies between other bundles. It is the task of the compiler to form bundles that obtain the greatest runtime benefit.

Similar to the braid, the block structured and IA-64 ISAs allow the compiler to specify dependency information in the program binary to avoid using complex dependence checking hardware in the microarchitecture. The braid ISA accomplishes this in a different way. Rather than specifying the dependencies for every instruction, the braid ISA implicitly identifies instruction dependencies by grouping instructions into braids. Processing braids eliminates the dependence checking hardware for all but two entries in the instruction queue because up to two instructions in a braid are considered for execution every cycle. Thus, it is not necessary to encode exact dependency information which incurs high code overhead. Furthermore, encoding dataflow subgraph information in the braid does not incur any additional code overhead.

7.5 Steering

Many proposals have suggested clustered designs which rely on a steering mechanism to distribute instructions. Front-end steering mechanisms make decisions by analyzing the current state of the machine which includes operand dependencies, cluster availability, and load balance. Palachara et al. [50] proposed FIFO-based instruction schedulers to simplify the design of the execution core. The algorithm tracks the dependencies of an instruction and steers instructions based on their dependencies. Farkas et al. [28] proposed the multicluster architecture where instructions are steered based on their logical register names. Copy instructions are used to transport values from one cluster to another on demand. Front-end steering mechanisms are relatively simple in terms of design complexity. They make decisions based on the current state of the processor. The decisions do not take into consideration instructions not yet fetched which can yield a suboptimal steering decision.

A more optimal steering mechanism considers future instructions by examining the steering decisions of previous instructions via feedback. Baniasadi and Moshovos [10] proposed an adaptive steering technique. Each cluster has a table of 2-bit counters which tracks how appropriate the cluster is for an instruction to be steered. Counters are updated based on past executions of the instruction. Canal et al. [19] proposed a steering mechanism which operates on instruction slices. A slice is the dataflow tree leading to a load or a branch instruction. Slices can be identified statically or dynamically but the authors advocated the runtime approach. All instructions of a slice are steered to the same cluster. Fields et al. [30] proposed using dynamic critical path analysis to help balance instruction distribution between clusters. Bhargava and John [11] used the retirement fill unit to analyze the past history of instructions. Hint bits are inserted into the trace to identify inter-trace data dependencies. Feedback-directed steering mechanisms perform better than front-end steering mechanisms. This is because feedback-based mechanisms track the past history of instructions. To accomplish this, hardware structures are needed to maintain and analyze the executed instructions. This adds design complexity to the pipeline and increases power requirements.

The braid microarchitecture is not clustered but shares similar characteristics as a clustered microarchitecture. All instructions in a braid are sent to the same BEU. However, unlike a traditional clustered microarchitecture, the distribute mechanism in the pipeline does not decide which instructions belong to which braid. Braids are identified at compile time. The only decision made by the distribute mechanism is the identification of BEUs that are ready to accept braids.

7.6 Scheduling

A number of proposals have suggested solving the complexity problems associated with a monolithic scheduler. Palachara et al. [50] proposed a microarchitecture which uses simple FIFO schedulers. A FIFO scheduler examines readiness of the instruction at the head of the FIFO rather than all the instructions in the window. Even though the scheduling process is easier, the complexity of this approach is shifted to the steering mechanism.

Kemp and Franklin [37] proposed a method of decentralizing the dynamic scheduling hardware called PEWs. The decoder is given the responsibility of sending an instruction to the PEW producing its source operands. Since dependent instructions are generally placed in the same PEW, most of the register traffic is obtained though intra-PEW forwarding.

Lebeck et al. [43] proposed a technique that uses the scheduling resources more efficiently. The instructions in the dataflow tree stemming from a load miss in the window are moved to a waiting instruction buffer. When the miss is satisfied, the same instructions in the dataflow tree are inserted back into the scheduling window. By managing the scheduling window resources more efficiently, the scheduling window can be designed with fewer entries to save power.

Brekelbaum et al. [14] proposed the use of hierarchical scheduling windows, each consisting of a small, fast window and a large, slow window. All instructions first enter the slow window. Latency critical instructions are moved to the fast window using a selection heuristic. Instructions are classified as either latency tolerant or latency critical. The use of hierarchical scheduling windows allows the design of a short scheduling loop for the small window which handles the short latency instructions and a longer scheduling loop for the large window which handles the long latency instructions.

Raasch et al. [57] proposed a segmented issue queue to tolerate high clock frequencies. The design dynamically constructs subtrees of the dataflow called chains which typically start with a load instruction. Chains flow from segment to segment and are controlled by a combination of data dependencies and predicted operation latencies. Chains reach the final segment when their inputs are ready.

The mentioned proposals simplify the complexity of the scheduling window by partitioning the window and intelligently managing instruction placement within the windows. Although the window can be more efficiently used, managing instruction movement within the scheduling window requires hardware resources to track instructions. The braid microarchitecture relies on the compiler to form units of work called braids. A braid is scheduled out of a FIFO queue. The braid microarchitecture simplifies both instruction placement and instruction scheduling.

Chapter 8

Conclusions and Future Directions

8.1 Conclusions

Increasing the performance of a single-core processor is a challenging and difficult task due to complexity issues and power requirements. This dissertation introduced an entity called the braid which allows the processor to scale to wider issue widths by simplifying the design complexity of structures in the execution core of a high-performance processor.

Braids partition the register space into external and internal registers. This enables the use of small partitioned register files. The characteristics of the braid dataflow subgraph enable the use of simple FIFO schedulers. Port requirements are reduced for a number of structures including the renaming mechanism, the external and internal register files, and the bypass network.

Three limitations of the braid microarchitecture are identified and a solution is presented to address each. Dynamic merging is proposed to address the limitation on braid size. This technique increases the percent of internal register reads from 50% to 73%. Context sharing is proposed to address the underutilization of braid execution resources by long-latency instructions. This technique improves performance by 1.5%. The use of heterogeneous execution resources is proposed to address the poor utilization of braid execution resources by single-instruction braids. This technique improves performance over using a set of similar braid execution units.

The internal register files maintain the internal values of a braid which represent 50% of the read accesses and 65% of the write accesses. On an 8-wide design, the result from executing braids is performance within 9% of a very aggressive conventional out-of-order microarchitecture with the complexity of an in-order implementation. The simplifications to the execution core enable lower power requirements, a shorter pipeline, and a higher clock frequency. Using braids is a viable approach to the design of future high-performance processors.

8.2 Future Directions

8.2.1 Braid-Aware Compiler

The braid and its implementation represent a new processing paradigm. There is no known commercial or research compiler that makes optimization decisions based on dataflow subgraphs. A braid-aware compiler produces more useful braids that are targeted towards processing by the underlying braid microarchitecture. An optimizing braid-aware compiler should make the following considerations, most of which can be addressed through code transformations.

- Width should be taken into consideration. Braids should be generated to target a width of two throughout the entire length of the braid.
- Braids should be long but without an extraneous amount of external communications.
- Braids should terminate at an instruction which produces a result that is used by many instructions. An example of this is stack or global pointer calculation instructions.
- Instruction duplication should be used to reduce external communications.

- Braids should be constructed so that the same set of external operands is maintained across control-flow boundaries to simplify the merging mechanism.
- Braids should be formed such that external inputs are hoisted to the top of the braid.
- Single-instruction braids should be eliminated.

8.2.2 Other Compiler Hints

The braid microarchitecture depends on the compiler to provide compile-time information to simplify runtime processing. The compiler partitions the dataflow graph of the program so that each dataflow subgraph can be processed in an in-order fashion. As a result of the partitioning, the communication of values is also partitioned. The compiler is not limited to providing only this information and can do much more to improve the processing capability of the microarchitecture.

One example of a simple annotation is the identification of dead register values. There have been runtime techniques proposed to identify the last use of registers [17]. Through dataflow analysis, the compiler can determine when values are dead. The compiler can associate a bit with an operand use to identify its last use. With this information, the microarchitecture can free registers early to increase the effective register size. This information is also useful to make dynamic merging more effective.

Some other examples of simple annotations that can be provided by the compiler include indicating the instruction makeup of a braid, identifying the locations of external inputs and external outputs of a braid, identifying the likely branch target of a branch instruction in a braid, and identifying if a braid contains any long latency instructions. Providing information to the microarchitecture provides more opportunity for improving performance.

8.2.3 Atomicity

The braid microarchitecture does not treat braids as an atomic unit of execution. It is natural to extend the support of atomicity to braids, because there is a clear delineation of braid boundaries and external inputs and outputs. Supporting atomicity can further reduce the complexity of the microarchitecture. For example, in the current implementation of the braid microarchitecture, a braid requires multiple entries allocated to it in the reorder buffer. If atomicity is supported, one handle is required in the reorder buffer to point to the braid. Thus, resource efficiency increases.

8.2.4 Clustering

This dissertation does not assume a clustered design. Clustering is a technique used to improve bandwidth and complexity of certain microarchitecture structures. Clustering centers on the concept of fast and slow communication paths. Communication within the cluster is fast, and communication between clusters is slow. Clustering can be applied on top of the braid microarchitecture to further simplify its design. A number of BEUs can be grouped together to form a cluster. Clustering will require more complex distribution heuristics to distribute braids that are likely to communicate to the same cluster.

8.2.5 Multi-Core Adaptability

The braid microarchitecture targets the design simplification of a single-core processor. It can just as easily be adapted in a multi-core design. Since the memory system has not changed, designing a microarchitecture to support braids uses the same coherence protocol as a conventional microarchitectural design. Since the braid microarchitecture targets future designs, it could be used as a building block for multi-core systems.

8.2.6 Cache Accessibility

Future high performance processors will have large caches to exploit locality. Techniques that partition the cache [58] are not compatible with the braid microarchitecture because they rely on distributing memory instructions to the functional units adjacent to the cache that the instructions are likely going to access. This does not work for the braid microarchitecture because a braid may contain multiple memory instructions that must be sent to the functional units of different BEUs. Furthermore, allowing each BEU to have its own access path to the cache may be difficult because of the cache organization.

This problem is best solved with the compiler. The compiler must generate some braids containing no memory instructions and some braids containing memory instructions. The microarchitecture can then direct braids that contain memory instructions to BEUs with the necessary cache access ports.

Bibliography

[1] Gnu compiler collection. http://gcc.gnu.org.

[2] Welcome to SPEC. http://www.specbench.org/.

[3] *Intel IA-64 Architecture Software Developer's Manual*, January 2000.

[4] P. S. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *MICRO 28: Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 36–45, 1995.

[5] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 423, Washington, DC, USA, December 2003. IEEE Computer Society.

[6] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–128, Washington, DC, USA, December 2004. IEEE Computer Society.

[7] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, May 1992.

[8] J. L. Ayala, M. López-Vallejo, and A. Veidenbaum. A compiler-assisted banked register file architecture. In *Proceedings of the 3rd Workshop on Application Specific Processors*, 2004.

[9] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. A high performance two-level register file organization. Technical Report TR-745, 2001.

[10] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *MICRO 33: Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 337–347, December 2000.

[11] R. Bhargava and L. K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 264–274, June 2003.

[12] E. Borch, E. Tune, S. Manne, and J. S. Emer. Loose loops sink chips. In *Proceedings of the 8th IEEE International Symposium on High Performance Computer Architecture*, pages 299–310, February 2002.

[13] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 18–29, December 2004.

[14] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *MICRO 35: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 27–36, November 2002.

[15] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report TR-1308, University of Wisconsin - Madison Technical Report, July 1996.

[16] M. Butler and Y. Patt. An area-efficient register alias table for implementing HPS. In *Proceedings of the 4th International Conference on Parallel Processing*, pages 611–612, 1990.

[17] J. A. Butts and G. S. Sohi. Characterizing and predicting value degree of use. In *MICRO 35: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 15–26, Los Alamitos, CA, USA, November 2002. IEEE Computer Society Press.

[18] J. A. Butts and G. S. Sohi. Use-based register caching with decoupled indexing. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 302–313, June 2004.

[19] R. Canal, J.-M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Proceedings of the 6th IEEE International Symposium on High Performance Computer Architecture*, pages 133–142, February 2000.

[20] R. S. Chappell, P. B. Racunas, F. Tseng, S. P. Kim, M. D. Brown, O. Mutlu, H. Kim, and M. K. Qureshi. The SCARAB microarchitectural simulator. Unpublished documentation.

[21] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, Washington, DC, USA, June 2005. IEEE Computer Society.

[22] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2004.

[23] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customisation. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[24] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, 2004.

[25] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–324, June 2000.

[26] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Computer Architecture News*, 3(4):126–132, 1974.

[27] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 304–315, Washington, DC, USA, 2004. IEEE Computer Society.

[28] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *MICRO 30: Proceed-*

ings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, pages 149–159, December 1997.

[29] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the 4th IEEE International Symposium on High Performance Computer Architecture*, pages 40–51, 1998.

[30] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, New York, NY, USA, June 2001. ACM Press.

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

[32] A. González, J. González, and M. Valero. Virtual-physical registers. In *hpca98*, pages 175–184, 1998.

[33] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, October 1996.

[34] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Intel Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001.

[35] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *ISCA '87: Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, 1987.

[36] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in power5 – ibm's next generation power microprocessor, August 2003. Hot Chips 15 presentation.

[37] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *International Conference on Parallel Processing*, pages 239–246, August 1996.

[38] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of the 16th IEEE International Conference on Computer Design*, pages 90–95, October 1998.

[39] H. Kim, M. A. Suleman, O. Mutlu, and Y. N. Patt. 2d-profiling: Detecting input-dependent branches with a single input data set. In *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, March 2006. IEEE Computer Society.

[40] H.-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, Anchorage, AK, USA, May 2002.

[41] I. Kim and M. H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 277–290, December 2003.

[42] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[43] A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. Patwardhan. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, May 2002.

[44] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *MICRO 28: Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–302, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[45] A. Marquez, K. Theobald, X. Tang, and G. Gao. A superstrand architecture. Technical Report Technical Memo 14, University of Delaware, Computer Architecture and Parallel Systems Laboratory, December 1997.

[46] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *MICRO 30: Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 125–135, Washington, DC, USA, 1997. IEEE Computer Society.

[47] S. Melvin and Y. N. Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–297, 1991.

[48] S. Narayanasamy, H. Wang, P. Wang, J. Shen, and B. Calder. A dependency chain clustered microarchitecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2005.

[49] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to fake 1000 registers. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM*

International Symposium on Microarchitecture, pages 7–18, Washington, DC, USA, 2005. IEEE Computer Society.

[50] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[51] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[52] S. J. Patel and S. S. Lumetta. rePLay : A hardware framework for dynamic program optimization. Technical Report CRHC-99-16, University of Illinois Technical Report, December 1999.

[53] Y. Patt, W. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *MICRO 18: Proceedings of the 18th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 103–107, December 1985.

[54] M. Pericàs, R. González, A. Cristal, A. V. Veidenbaum, and M. Valero. An optimized front-end physical register file with banking and writeback filtering. In *Proceedings of the 4th Workshop on Power-Aware Computer Systems*, pages 1–14, December 2004.

[55] J. Pierce and T. Mudge. Wrong-path instruction prefetching. *IEEE Micro*, 00:165, 1996.

[56] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. Reducing datapath energy through the isolation of short-lived operands. In *Proceedings of the 12th*

IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques, page 258, Washington, DC, USA, 2003. IEEE Computer Society.

[57] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 318–328, May 2002.

[58] P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *ics03*, pages 22–31, June 2003.

[59] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith. Trace processors. In *MICRO 30: Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997.

[60] P. Salverda and C. Zilles. Dependence-based scheduling revisited: A tale of two baselines. In *Proceedings of the 6th Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2007.

[61] P. G. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–17, December 2004.

[62] P. G. Sassone, D. S. Wills, and G. H. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, 2005.

[63] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *MICRO 35: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 383–394, Los Alamitos, CA, USA, November 2002. IEEE Computer Society Press.

[64] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[65] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[66] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, volume 00, page 0123, Los Alamitos, CA, USA, May 2002. IEEE Computer Society.

[67] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Technical Report TR-1420, University of Wisconsin - Madison Technical Report, October 2000.

[68] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *MICRO 27: Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 143–147, 1994.

[69] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *MICRO 33: Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[70] J. E. Thornton. Parallel operation in the control data 6600. pages 32–39, 2000.

[71] J. H. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 62–71, New York, NY, USA, 2003. ACM Press.

[72] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, 1997.

[73] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the 5th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, page 179, Washington, DC, USA, October 1996. IEEE Computer Society.

[74] O. Wechsler. Inside intel core microarchitecture: Setting new standards for energy-efficient performance. *Intel Technology Journal*, 10(2), 2006.

[75] K. C. Yeager. The MIPS R10000 superscalar microprocessor. In *MICRO 29: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 28–41, December 1996.

[76] R. Yung and N. C. Wilhelm. Caching processor general registers. In *Proceedings of the 13th IEEE International Conference on Computer Design*, pages 307–312, Washington, DC, USA, 1995. IEEE Computer Society.

[77] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for vliw processors. In *MICRO 33: Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–146, New York, NY, USA, December 2000. ACM Press.

[78] V. V. Zyuban and P. M. Kogge. The energy complexity of register files. In *Proceedings of the 1998 International Symposium on Low Power Electronic Design*, pages 305–310, August 1998.

[79] V. V. Zyuban and P. M. Kogge. Inherently low-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–286, March 2001.

Vita

Francis Tseng was born in Pingtung, Taiwan on November 3, 1976, the son of Chu-Lan Chang and Wan-An Tseng. Not long after attending primary school in Taiwan, his family moved to the US. In 1995, he graduated from Brighton High School and enrolled in the Computer Engineering program at the University of Michigan in Ann Arbor. He graduated with a Bachelor of Science and Engineering degree in 1999. In the same year, he followed his advisor, Professor Yale N. Patt to The University of Texas at Austin where he enrolled in the Ph.D. program in Computer Engineering. In 2006, he married Teresa H. Lai.

At the University of Texas at Austin, he served as a teaching assistant for the *Introduction to Computing* and the *Computer Architecture* courses. He was nominated by the Electrical and Computer Engineering Department for the Outstanding TA/AI Award in 2000. His studies were supported in part by an Intel Graduate Research Fellowship and by an Intel Ph.D. Fellowship. While in college, he had held summer internships at Netscape Communications Corporation, Intel Corporation, Advanced Micro Devices, Cray, and HAL Computer Systems.

Permanent address: 7600 Wood Hollow Dr Apt 915
Austin, TX 78731

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.