Achieving Out-of-Order Performance with Almost In-Order Complexity

Francis Tseng   Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{tsengf,patt}@hps.utexas.edu

Abstract

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 paper introduces the braid, a compile-time identified 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. Braid processing requires identification by the compiler, minor extensions to the ISA, and support by the microarchitecture. The result from processing braids is performance within 9% of a very aggressive conventional out-of-order microarchitecture with almost the complexity of an in-order implementation.

1. Introduction

1.1. Rationale

Figure 1 plots the potential performance that is available at wider issue widths in an aggressive conventional out-of-order design, assuming a perfect branch predictor and perfect instruction and data caches. It is not unrealistic to assume future designs will have more accurate branch predictors and also larger caches. Thus, this figure measures the available untapped performance. The bars show the speedup over a conventional 4-wide out-of-order design. Changing the issue width from 4 to 8 achieves an average speedup of 44%, and changing the issue width from 4 to 16 achieves an average speedup of 83%. Some applications such as crafty, vpr, and mgrid show a speedup of 200% when issue width increases from 4 to 16.

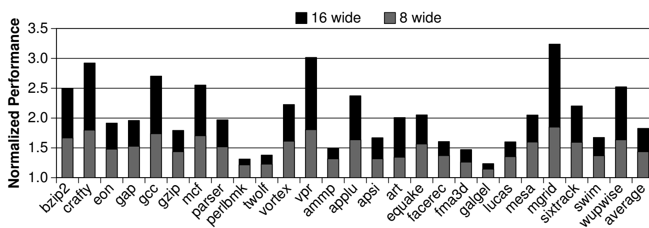


Figure 1. Potential performance of 8-wide and 16-wide designs over a 4-wide conventional out-of-order design

Although Figure 1 shows that performance is plentiful at wider issue widths, there are implications that make such designs infeasible. Traditional methods of increasing issue width do not scale. Increasing issue width beyond that of current processors drastically increases design complexity and power requirements within a processor's execution core. A processor with a wider issue width requires larger and wider structures in the execution core to support more instructions that can be processed in parallel. In particular, these structures include the instruction scheduler, the register file, and the bypass network. All of these structures pose problems because they do not scale. Even if larger and wider structures could be designed, they must be pipelined to support more aggressive clock frequencies. Pipelining structures in a critical loop of execution leads to performance loss[2]. Pipelining the scheduler with minimal performance loss has been proposed[25], but this is not feasible without adding additional complexity to the pipeline.

Rather than finding a pure hardware solution for an execution core that can support wider issue widths, the compiler can be used. Two key characteristics of values are leveraged: the fanout and the lifetime of values produced in programs. The programs in the SPEC CPU2000 benchmark suite compiled for the Alpha ISA were analyzed. A value's fanout is defined as the number of times that value is read. On average, over 70% of values are used only once, and about 90% of values are used at most twice. About 4% of values are produced but not used. These values were produced for control-flow paths that were not traversed at run time. A value's lifetime is defined as the number of instructions between a value's producer and consumer. On average, about 80% of values have a lifetime of 32 instructions or fewer. This is four processor cycles on an 8-wide processor.

The dataflow graph of the program may appear to have much irregularity. However, the majority of values have a small fanout and a short lifetime. This suggests the dataflow graph of the program may be partitioned, and the dataflow graph may have more regularity than first thought. The goal of this work is to use the compiler to partition the dataflow graph of the program into subgraphs that can be more easily processed by the hardware. Although some researchers have characterized the behavior of values[8][4], they have not partitioned the dataflow graph based on these realizations.

1.2. The Braid

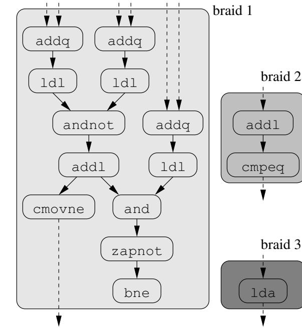
This paper introduces the braid, a compile-time identified entity which simplifies the design complexity of structures on the critical path in the execution core of a high-performance

```
1: for (j = 0; j < regset_size; j++)
2:   {
3:     register REGSET_ELT_TYPE x
4:       = (basic_block_new_live_at_end[i][j]
5:          & ~basic_block_live_at_end[i][j]);
6:     if (x)
7:       consider = 1;
8:     if (x & basic_block_significant[i][j])
9:       {
10:        must_rescan = 1;
11:        consider = 1;
12:        break;
13:      }
14:  }
```

(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, 0x1200eb2b0
```

(b)



(c)

Figure 2. Braid example: (a) snippet of source code from the `life_analysis` function in gcc (b) assembly code of basic block (c) dataflow graph of basic block

Integer	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	**average**		
braids	2.5	2.5	4.2	2.4	2.4	2.6	2.0	2.7	2.8	3.1	3.5	2.8	**2.8/1.1***		
Float Point	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	**average**
braids	2.0	5.9	4.7	2.9	2.5	2.7	2.8	5.7	3.7	2.8	4.0	3.1	6.6	3.6	**3.8/1.5***

Table 1. Braids per basic block, *excludes single-instruction braids

processor. A braid is a subgraph of the dataflow of the program that resides solely within a basic block. It exploits the small fanout and short lifetime of values communicated among instructions. Braids enable knowledge available at compile-time to be passed to the hardware, lessening the hardware's burden of extracting program dataflow information at runtime. This approach eliminates the need for complex structures in the execution core. Braids enable the use of simple instruction schedulers, reduce the size, number of ports, and accesses to the register file, and also reduce broadcast traffic. Not only do braids save design complexity by avoiding large associative structures, they also enable a higher clock frequency. This paper presents the braid microarchitecture to efficiently carry out the execution of braids. With simple augmentations to the ISA, braid information can be easily conveyed and processed by the microarchitecture.

2. Braid Example and Characterization

Figure 2(a) shows a snippet of C source code from the `life_analysis` function in the gcc program from the SPEC CPU2000 benchmark suite. Figure 2(b) shows the assembly code of the basic block corresponding to lines 1 through 8 of the source code. The dataflow graph of the code in the basic block can be divided into three disjoint dataflow subgraphs indicated by the three different color shades. A pictorial representation corresponding to the dataflow graph is shown in Figure 2(c). The three different color shades correspond to the three dataflow subgraphs. The arrows indicate data dependencies. 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 is a different braid. In this example, the basic block is partitioned into three braids, each representing an operation being performed in the high-level language. The instructions in braid 1 correspond to lines 3 through 8 of the source code. This braid has six external inputs, one external output, and three internal values. The instructions in braid 2 perform the increment of the induction variable in the `for` statement of the source code. 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 known as a single-instruction braid.

A basic block consists of instructions specifying the execution of one or more operations in the high-level language. Since each one of these operations corresponds to a braid, multiple braids can exist within a basic block. Table 1 shows the average number of braids per basic block across the SPEC CPU2000 benchmark suite. On average, the integer programs have basic blocks consisting of about 2.8 braids, and the floating point programs have basic blocks consisting of about 3.8 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 is 1.1 for the integer programs and 1.5 for the floating programs. 20% of all instructions are single-instruction braids. Of the single-instruction braids, 56% are branch instructions and nops. The large number of single-instruction braids is due to the use of preexisting binaries produced by a non-braid-aware compiler.

The size of a braid is computed by counting the number of instructions in the braid. The width of a braid measures the average instruction-level parallelism of a braid. It is computed by dividing the size of a braid by the number of instructions along the longest dataflow path. Table 2 shows the average size and width of braids across the programs. On average across the integer programs, a braid has a size of 2.5 instructions and a width of 1.1. On average across the floating point programs, a braid has a size of 3.6 instructions and a width of 1.1 instructions. Larger braid sizes are expected

Integer	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	**average**		
size	3.4	3.2	2.0	2.5	2.3	3.4	2.0	2.2	2.3	2.8	2.1	2.5	**2.5/4.7***		
width	1.1	1.1	1.1	1.0	1.1	1.0	1.0	1.0	1.1	1.0	1.1	1.1	**1.1/1.1***		
Float Point	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	**average**
size	2.8	2.9	2.8	2.6	2.4	2.2	2.7	2.0	4.6	2.1	13.2	2.3	4.8	2.8	**3.6/7.6***
width	1.0	1.1	1.1	1.0	1.0	1.1	1.1	1.0	1.1	1.1	1.4	1.1	1.2	1.1	**1.1/1.2***

Table 2. Braid size and width, *excludes single-instruction braids

Integer	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	**average**		
internals	2.7	2.4	1.1	1.6	1.4	2.6	1.0	1.2	1.4	2.0	1.1	1.6	**1.7/4.0***		
ext inputs	1.9	1.7	1.5	1.5	1.6	2.1	1.5	1.5	1.4	1.7	1.7	1.7	**1.7/2.5***		
ext outputs	0.8	0.7	0.6	0.8	0.7	0.9	0.6	0.7	0.7	0.6	0.8	0.8	**0.7/1.0***		
Float Point	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	**average**
internals	2.0	2.0	2.1	1.6	1.5	1.3	2.1	1.1	4.1	1.2	14.5	1.3	4.5	2.2	**3.0/7.5***
ext inputs	1.9	1.7	1.9	1.9	1.7	1.7	2.1	1.7	2.6	1.9	5.9	1.8	3.0	1.8	**2.2/4.2***
ext outputs	0.7	0.6	0.6	0.6	0.7	0.8	0.8	0.6	0.7	0.6	1.7	0.7	0.7	0.7	**0.8/1.1***

Table 3. Braid inputs and outputs, *excludes single-instruction braids

for the floating point programs due to larger basic block sizes resulting from the streaming nature of the code. It is encouraging to see the average width of braids in the floating point programs remains similar to that of braid widths in the integer programs. When single-instruction braids are factored out, the average size of braids becomes 4.7 for the integer programs and 7.6 for the floating point programs.

Table 3 shows the number of internal and external dependencies of a braid. This result shows the potential savings of register file accesses. On average for the integer programs, 1.7 values are communicated internally. There are 1.7 external inputs and 0.7 external outputs. On average for the floating point programs, 3.0 values are communicated internally. There are 2.2 external inputs and 0.8 external outputs. The number of inputs and outputs is not unlike the operand specifications of a two-source compute instruction. When single-instruction braids are excluded, the average number of internal values increases to 4.0 for the integer programs and 7.5 for the floating point programs.

3. Implementation

Three components are needed to implement braid processing. First, the compiler must construct braids from the program dataflow graph. Second, the ISA must convey braid information from the compiler to the microarchitecture. Third, the microarchitecture must be aware of braids to exploit braid characteristics.

3.1. Constructing Braids at Compile Time

A braid is a compile-time identified entity. In this work, in order to mimic a compiler, binary profiling and binary translation tools were used. The profiling tool analyzes the dataflow graph of the program and records the producer and consumers of each value produced in the program. Braids are identified using a simple graph coloring algorithm. A braid is formed by selecting an instruction within the basic block and identifying the dataflow subgraph stemming from that instruction within the basic block. This is repeated until all instructions within the basic block are associated with a braid.

Next, the instructions within the basic block are arranged such that instructions belonging to the same braid are scheduled as a consecutive sequence of instructions within the basic block. This is accomplished using a binary translation tool. If the last instruction of the basic block is a branch, the braid containing the branch instruction is ordered to be the last braid in the basic block. Because the branch position is unaltered, the property of the basic block is maintained, and branch offsets remain the same.

Since the profiling tool comprehends the usage of values, it can determine values that are used within and outside of the basic block. Register allocation is performed in two passes. First, register allocation is performed for the external registers across the entire program. Second, register allocation is performed for the internal registers within a braid.

The formation of braids can be restricted by two conditions. First, the braid microarchitecture supports a limited number of internal registers. Therefore, the working set of internal values within a braid must not exceed the total number of internal registers. When the working set exceeds this threshold, the braid is broken into two braids at this boundary. Through empirical analysis, 8 internal registers are sufficient. Breaking braids using a threshold of 8 internal registers accounts for about 2% of the braids analyzed.

Second, since the rearranging of braids within the basic block rearranges instructions, the ordering of memory instructions may be violated. The majority of memory instructions access the stack so the compiler can disambiguate them. 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 rearranging braids within the basic block, the braid is broken into two braids at the location of the memory ordering violation to enable the partial ordering. This condition accounts for less than 1% of the braids analyzed. Both conditions are an artifact of using the profiling and binary translation tools on preexisting binaries. A braid-aware compiler can easily mitigate this problem.

3.2. ISA Extensions for Supporting Braids

Minor augmentations are made to the ISA to allow the compiler to effectively convey braids to the microarchitecture. Figure 3 shows the specification of a zero-destination, one-register, and two-register braid ISA instructions. A couple of bits have special meanings in a braid ISA instruction. A braid start bit (S) associated with the instruction specifies whether the instruction is the first instruction of a braid. A temporary operand bit (T) associated with each source operand specifies whether the operand obtains its value from the internal register file or the external register file. The internal destination bit (I) and external destination bit (E) associated with each destination operand specify whether the result of the instruction should be written to the internal register file, the external register file, or both files.

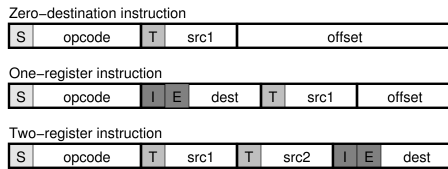


Figure 3. Braid instruction encoding

3.3. The Braid Microarchitecture

Figure 4(a) shows an overview of the braid microarchitecture pipeline. 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. Like a conventional processor, a cache line is fetched from the instruction cache. Since the instructions of a braid are grouped together within a basic block, the front-end will encounter a braid in its entirety before encountering a subsequent braid. The instructions continue through the decode stage to the allocate stage where microarchitectural resources are allocated to instructions. Instructions of a braid that write results only to internal registers do not need to be allocated an external register entry. Furthermore, these internal operands do not need to be renamed. Next, the instructions of a braid are distributed to one of the braid execution units (BEUs). A BEU can accept a new braid if it is not processing another braid. The use of the *braid start bit* in the instruction encoding greatly simplifies the identification of braid boundaries.

Figure 4(b) shows the block diagram of a BEU. The shaded regions highlight the differences from the execution core of 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. The busy-bit vector, similar to the one used in the MIPS R10000[27], tracks the availability of values in the external register file. The FIFO scheduling window is the two entries at the head of the FIFO queue. Only the instructions

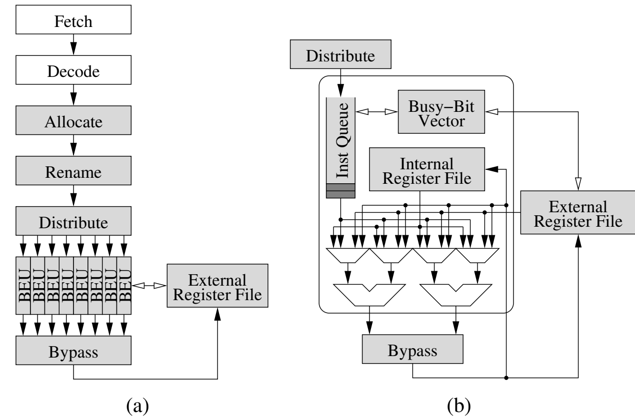


Figure 4. The braid microarchitecture: (a) block diagram of the pipeline (b) block diagram of the braid execution unit

in these two entries are examined for readiness. The sensitivity of the braid microarchitecture to the various parameters will be examined in Section 4.

The FIFO scheduler is a two-wide in-order scheduler. Each cycle, the external operands of instructions in the FIFO scheduling window consult the busy-bit vector for their availability. The busy-bit vector contains one bit for each entry of the external register file. This entire structure is very small and requires only 8 bits of storage. An instruction can receive its operands from one of four locations: the internal register file, the external register file, the instruction queue in the case of immediate values, or the bypass network. An instruction can write its result in the internal register file, the external register file, or both files. Similar to a conventional out-of-order design, results that are produced by the functional unit are buffered on a bypass network while they are being written to the external register file.

The internal register file maintains the current working set of internal operands of a braid within a BEU. It is very small with only 8 entries. 8 entries are sufficient to satisfy the working set of the internal values for the majority of braids. Since only two instructions can access the internal register file in any cycle, the internal register file is implemented with 4 read ports and 2 write ports. Because the values in the internal register file are temporary, they are safely discarded once the last instruction of a braid executes. These values do not need to be written to the external register file.

Aside from the execution core, the remainder 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 run time.

3.4. Control Flow, Recoveries, and Exception Handling

Braids are formed at compile time and do not span basic block boundaries. This is done to maintain implementation

simplicity and to avoid unnecessary code duplication. Allowing braids to span control-flow boundaries introduces consistency problems due to the existence of control-flow merge and fork points. If a braid is allowed to span across two basic blocks, the guarantee made for the internal and external registers in the braid may no longer hold at run time if a different path is traversed. By restricting braids to reside completely within the basic block, problems associated with changing run-time control-flow paths are avoided.

Recovering from a branch misprediction is a simple procedure with the support of a checkpoint recovery mechanism. Because checkpoints are already created for branch instructions, no additional hardware or storage is required to support braid processing. 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. Thus, internal register values do not need to be stored in the checkpoint. When a recovery initiates, the processor restores the checkpoint of the state prior to the branch misprediction and initiates 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 but one of the BEUs are disabled. All instructions are sent to the predetermined BEU. Since a BEU contains an in-order scheduler, forcing instructions to one BEU turns the processor into a strict 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. Due to the rarity of exceptions in general-purpose processing, simplicity was chosen over speed for handling them.

4. Experimental Results and Analysis

4.1. Simulation Methodology

The experiments were performed on a cycle-accurate, execution-driven simulator. To show how braids can be useful for the design of future aggressive processors, the experiments were performed on 8-wide configurations. Table 4 shows the default configuration of an aggressive conventional out-of-order microarchitecture and the braid microarchitecture. Both configurations share a similar front and back-ends. The front-end is capable of processing up to three branches per cycle in order to represent an aggressive future design. There is another reason why an aggressive front-end is needed. Since this paper focuses on the design of the execution core, the execution core must be stressed. An aggressive front-end accomplishes this by not constraining the delivery of instructions to the execution core (see for example, Salverda and Zilles[19]). There is a 64KB instruction cache, a 64KB data cache, a 1MB L2, and a memory access latency of 400 cycles.

Common Parameters	
Instruction Cache	64KB, 4-way associative, 3-cycle latency
Branch Predictor	perceptron with 64-bit history, 512-entry weight table
Fetch Width	8 instructions, processes up to 3 branches per cycle
Issue Width	8 instructions
L1 Data Caches	64KB, 2-way associative L1 with 3-cycle latency
L2 Cache	1MB, 8-way associative unified L2 with 6-cycle latency
Main Memory	400-cycle latency
Out-of-Order Parameters	
Misprediction Penalty	minimum 23 cycles
Allocate	8 operands
Rename	16 source operands, 8 destination operands
Scheduler	8 distributed 32-entry out-of-order schedulers
Functional Unit	8 general-purpose units
Register File	256 entries with 16 read ports, 8 write ports
Bypass Network	3 levels, 8 values per cycle
Braid Parameters	
Misprediction Penalty	minimum 19 cycles
Allocate	4 operands
Rename	8 source operands, 4 destination operands
BEU	8
FIFO	32-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, 2 write ports per BEU
External Register File	8 entries with 6 read ports, 3 write ports
Bypass Network	1 level, 2 values per cycle

Table 4. Default processor configurations

The out-of-order configuration has a minimum branch misprediction penalty of 23 cycles. The allocator is capable of processing 8 instructions per cycle. The renaming mechanism is capable of processing 16 source operands and 8 destination operands per cycle. There are 8 distributed 32-entry out-of-order schedulers and 8 general-purpose functional units. There is a 256-entry monolithic register file with 16 read ports and 8 write ports. The bypass network consists of 3 levels where each level is capable of supporting 8 values every cycle.

Due to its design simplifications, the braid microarchitecture has a pipeline that is shorter by four stages than the comparable conventional design. The savings come from a shorter rename stage and a shorter register access stage. The braid microarchitecture configuration has a minimum branch misprediction penalty of 19 cycles. The allocator is capable of processing 4 instructions per cycle. The renaming mechanism is capable of processing 8 source operands and 4 destination operands per cycle. There are 8 BEUs. Each BEU contains a 32-entry FIFO queue with a 2-entry instruction scheduling window. Each BEU also has an 8-entry busy-bit vector, two functional units, and a small 8-entry internal register file with 4 read ports and 2 write ports. There is an 8-entry global external register file with 6 read ports and 3 write ports. The bypass network consists of 1 level support-

ing 2 values per cycle. For the experiments, clock frequency was not modified.

The experiments were carried out using the programs from the SPEC CPU2000 benchmark suite. These programs were compiled with gcc 4.0.1 for the Alpha EV6 ISA with the `-O2` optimization flag. All programs were run to completion using the MinneSPEC reduced input sets[12].

4.2. Register File and Bypass Restrictions

The first set of experiments examines the sensitivity of performance to the number of external registers in both the conventional out-of-order microarchitecture and the braid microarchitecture. Figure 5 plots the performance of a conventional out-of-order microarchitecture as a function of the number of register file entries. The results are normalized to the performance of the same configuration with a 256-entry register file. Using 32 registers causes an 8% degradation in performance, and using 16 registers causes a 21% degradation in performance. This result becomes more severe as branch prediction and caches improve.

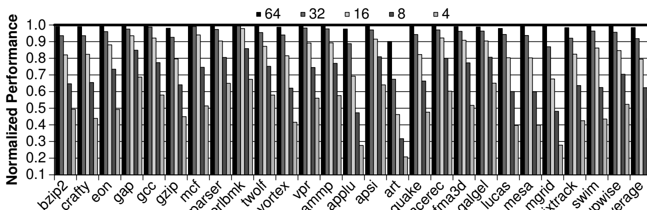


Figure 5. Out-of-order performance varying the number of registers

Figure 6 plots the performance of the braid microarchitecture as a function of the number of external register file entries. Again, the results are normalized to the performance of the braid microarchitecture with a 256-entry external register file. Since the majority of registers access the internal register files, there are fewer accesses to the external register file. Performance is not significantly affected until 4 registers are used. It can be seen that using a small 8-entry external register file is sufficient to maintain similar performance as a 256-entry register file.

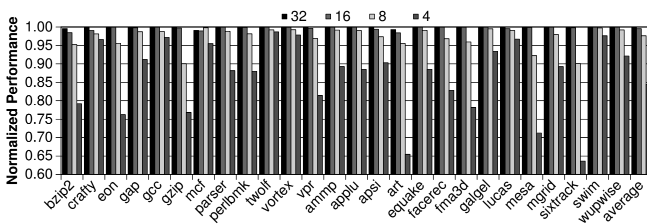


Figure 6. Braid microarchitecture performance varying the number of external registers

Figure 7 plots the performance of the braid microarchitecture as a function of the number of read and write ports on the external register file. Each bar is a configuration representing the number of read and write ports on the external register file. The results are normalized to the performance of the braid microarchitecture with an external register file with 16 read ports and 8 write ports. With as few as 6 read ports and 3 write ports, performance is well within 0.5% of performance from using a full set of read and write ports. The programs mesa and sixtrack are more sensitive to port scaling because these programs contain phases where instructions have a larger number of external registers saturating the external read and write ports.

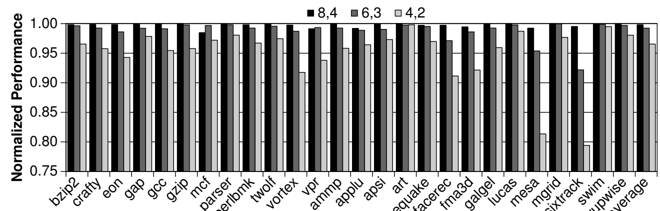


Figure 7. Braid microarchitecture performance varying the number of external register file ports

Figure 8 plots the performance as a function of the number of bypass paths supported in the braid microarchitecture. The results are normalized to the braid microarchitecture using a full set of bypass paths in the bypass network. Since internal values do not require bypassing, the number of bypass values is greatly reduced. Designing a microarchitecture supporting at most 2 bypass values per cycle achieves performance that is within 1% of using a full set of bypass paths.

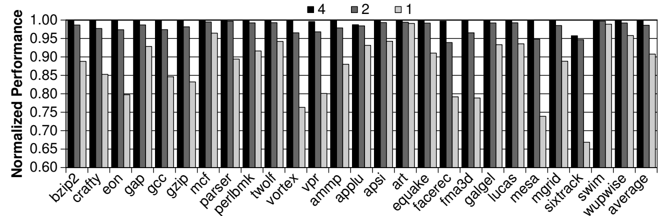


Figure 8. Braid microarchitecture performance varying the number of bypass paths

4.3. Execution Core Parameters

Various design parameters within the execution core were analyzed for 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 show the effects of varying each design parameter. Each experiment starts with the braid microarchitecture with the default configuration. There are 8 BEUs where each BEU contains a 32-entry FIFO queue with a 2-entry in-order scheduling window and 2 functional units. In each experiment, one parameter is varied while holding the other parameters constant. The results are normalized to the performance of an 8-wide conventional out-of-order microarchitecture.

Figure 9 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. Increasing the number of BEUs increases dataflow-level parallelism. Another benefit

from increasing the number of BEUs has to do with instructions that may stall due a dependency with a long-latency instruction. Supporting more BEUs allows ready braids to slip pass stalled braids. However, the number of BEUs should not increase to the point where complexity limits the communication across BEUs.

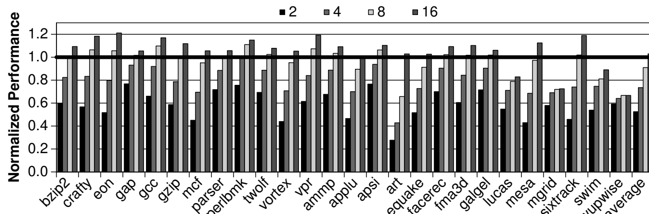


Figure 9. Braid microarchitecture performance varying the number of BEUs

The FIFO queue in each BEU is an instruction waiting buffer. An instruction of a braid waits in the queue until it shifts to the head of the FIFO. The size of this structure does not inhibit design scalability. It must simply be large enough to buffer the instructions of a braid. Figure 10 plots the performance as a function of the number of entries in the FIFO queue. 32 entries are enough to achieve most of the performance. This is because 99% of braids consist of 32 instructions or fewer. Without the proper amount of buffering, a long braid will unnecessarily stall the braid distribution mechanism because all the instructions in the braid cannot be sent to a queue. This event will eventually stall the front-end.

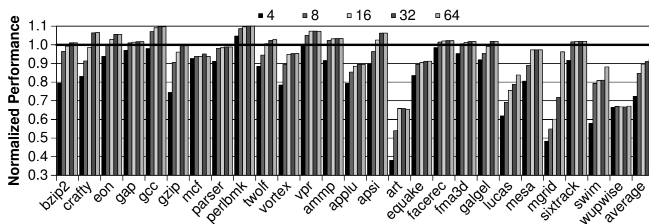


Figure 10. Braid microarchitecture performance varying the number of FIFO entries

The following experiment examines the sensitivity to the scheduling window size in each BEU. Figure 11 plots the performance as a function of the window size. It is encouraging to see the bars rise steeply from 1 to 2, and then reaching a plateau from 2 and on. This result confirms that ready instructions are likely located at the head of the FIFO queue. The floating point programs swim and mgrid have slightly wider widths on average and are more sensitive to the size of the scheduling window.

The following experiment examines the instruction-level parallelism of a braid. In the previous experiment, only the scheduling window size is varied. Figure 12 plots the performance as a function of both the scheduling window size and the number of functional units. Each bar corresponds to the same number of scheduling window entries and functional units. This graph shows a similar trend as the graph in Figure 11. This result confirms that instruction-level parallelism within the braid is limited to 2 and that using more than 2 functional units is not greatly beneficial to performance.

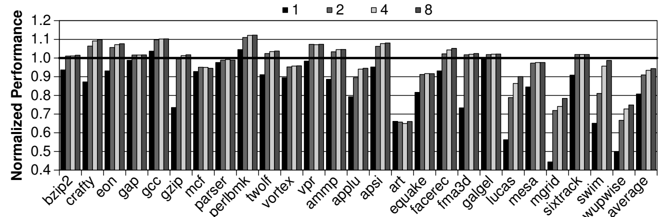


Figure 11. Braid microarchitecture performance varying the FIFO scheduling window size

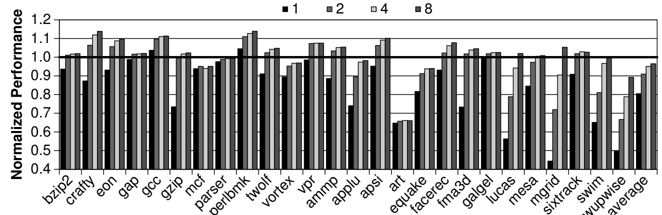


Figure 12. Braid microarchitecture performance varying the number of functional units per BEU

4.4. Comparison to Various Paradigms

Figure 13 compares the performance of four microarchitectural paradigms at three issue widths. Each stacked bar from bottom to top represents the performance of an in-order, FIFO-based dependence steering, braid, and out-of-order microarchitectures. The set of bars for each program from left to right represent the performance of 4-wide, 8-wide, and 16-wide designs. The results of using a dependence-based steering algorithm[17] are presented to illustrate a simple and implementable algorithm with a design complexity that is comparable to braids. Three observations can be made from this graph. First, significant performance gain is 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. This includes the performance gains from the shortened pipeline. Third, the gap between the braid and out-of-order microarchitectures closes as the issue width increases.

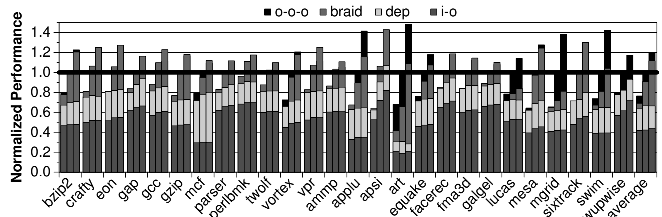


Figure 13. Performance of in-order, dependence based, braid, and out-of-order at 4, 8, and 16-wide designs

4.5. Equal Functional Unit Resources

Figure 14 plots the performance of the braid microarchitecture assuming the same number of functional units as a con-

ventional 8-wide out-of-order microarchitecture. The results are normalized to the performance of the braid microarchitecture with the default configuration which has 16 functional units. The configuration of the left bar is the braid microarchitecture with 4 BEUs, each containing 2 functional units. The configuration of the right bar is the braid microarchitecture with 8 BEUs, each containing 1 functional unit. In each of these cases, there is a total of 8 functional units. Supporting more BEUs, if it can be designed, is more beneficial to performance than providing fewer but wider BEUs.

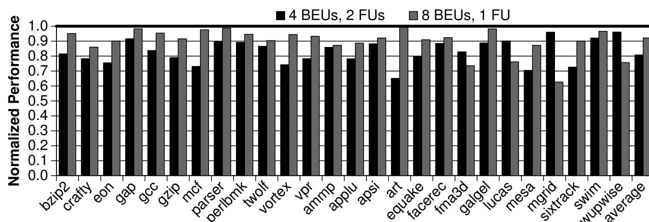


Figure 14. Performance using an equal number of functional units

5. Discussion

5.1. Analysis of Complexity

In the braid microarchitecture, the allocate and rename stages of the pipeline can be designed to support lower bandwidth than a conventional out-of-order processor. Instructions which write only to internal operands do not need to be allocated an external register entry. Internal source and destination operands do not need to be renamed.

The out-of-order instruction scheduler is one of the commonly-studied structures due to its design complexity[17]. The braid microarchitecture avoids this complexity by using FIFO schedulers. The FIFO scheduler does not broadcast tags to the entire structure. Since there is less switching activity, FIFO schedulers consume less power. Supporting an out-of-order scheduler in the BEU has been considered. This is possible at the cost of complexity. However, Figures 11 and 12 indicate that an out-of-order scheduler is not necessary since braids are on average very narrow.

Salverda and Zilles concluded that it is difficult to achieve out-of-order performance by fusing in-order cores[20] due to the need of a sophisticated steering mechanism. The braid microarchitecture replies on the compiler to identify sets of dependent instructions. This identification greatly simplifies the distribution mechanism. No steering decisions based on dependencies are needed to be made at run time.

It is well known that increasing the number of entries and ports of the register file has a negative effect on area and power[29]. Doubling the number of register ports doubles the number of bit-lines and doubles the number of word-lines causing a quadratic increase in area[7]. In the braid microarchitecture, the register file is partitioned into an external register file and multiple internal register files. Furthermore, each register file contains only 8 entries with a very minimal number of read and write ports. The partitioning and port reduction greatly reduce the total area required by the register files. Values in the internal register file are naturally discarded once a braid finishes execution in the BEU. Values do not propagate between the internal and external register files. Consequently, fewer accesses to the external register file imply fewer values that are sent to the bypass network.

There are more functional units in the braid microarchitecture than the conventional out-of-order microarchitecture. The lack of complexity in the BEU allows 8 BEUs to occupy less space than an 8-wide out-of-order design, while having 16 functional units. The majority of the programs can take advantage of these extra functional units.

The busy-bit vector within each BEU tracks the readiness of values in the external register file. Since the external register file is only 8 entries, each busy-bit vector requires only 8 bits. Because there is an average of 2 external values produced every cycle across all of the BEUs, synchronizing the busy-bit vector across the BEUs is not at all difficult.

The less complex design of the braid microarchitecture shortens the number of pipeline stages reducing the branch misprediction penalty. The performance gained from a shorter pipeline is on average 2.19%. Since the structures on the critical path have been simplified, it is reasonable to assume the processor can be clocked at a higher frequency which was not evaluated in this paper. With the frequency compensation, the braid microarchitecture can achieve performance that is even closer to that of the aggressive out-of-order design.

5.2. Future Direction

A compiler uses registers to convey values between instructions in the program. Due to a limited set of registers, the compiler inserts spill and fill code to save and restore registers to memory when the working set of values exceeds the number of registers. Because the analysis in this paper follows a profiling approach, only data values propagated through the register space have been considered. There are also data values that are propagated through memory. If a braid-aware compiler is used to generate braids, the more efficient management of the register space will reduce the amount of spill and fill code.

Clustering is a technique that is orthogonal to the functionality of the braid microarchitecture. Clustering can be applied to the braid microarchitecture by grouping a subset of the BEUs together. Values synchronized between BEUs within a cluster can be fast whereas values synchronized between BEUs across clusters will be slower.

6. Related Work

Although the braid microarchitecture shares similarities with other proposals in the literature, no single scheme achieves the combined benefits of the braid microarchitecture.

6.1. Block-Based Processing

Most of the proposals in the literature follow a style of processing where the basic block or multiple of the basic blocks are treated as a unit of processing. All instructions within the basic block is treated with equal priority. They enter a scheduling window regardless of their data dependencies. In the braid microarchitecture, a set of tightly connected instructions related by their data dependencies travel together as a unit of processing. By distinguishing between different dataflow subgraphs within a basic block, the braid microarchitecture is able to more efficiently carry out instruction processing.

The braid microarchitecture and the trace processor[18] have important differences. First, a trace consists of a set of basic blocks identified at run time and does not distinguish between different dataflow subgraphs. All instructions in a trace are scheduled in an out-of-order window. Second, the trace processor relies on run-time capturing and marshaling of traces. Traces do not have the same benefits as braids. The braid microarchitecture leverages the compiler-identified subgraphs requiring no run-time analysis of code. Although the trace processor also uses local and global registers, the concept of partitioning the register space was previously proposed in the context of a statically tagged ISA[24]. Local and global registers in the trace processor must be identified at run time. The braid microarchitecture uses in-order schedulers to process compiler-identified dataflow subgraphs and leverages compiler-identified internal and external registers.

Multiscalar[23] shares similar characteristics but also has important differences with the braid microarchitecture. Both paradigms process a piece of work identified by the compiler. The unit of work in Multiscalar is a task which consists of a set of basic blocks. Tasks are assigned to processing units arranged in a ring formation. Processing units use out-of-order schedulers. The compiler also conveys which register values need to be forwarded to other processing units and which values are no longer needed through the ISA. In contrast, the braid microarchitecture processes dataflow subgraphs as a unit of work. The topology of the braid microarchitecture does not require multiple hops to communicate values.

The Block-Structured ISA[15] uses the compiler to generate code blocks with embedded data dependency information. This simplifies design of hardware dependence checking logic. Braids provide further simplification of hardware complexity by explicitly identifying dataflow subgraphs and internal values within the basic block.

6.2. Strands, Dependency Chains, and Subgraph Processing

The term *strand* was first coined by Marquez et al. to mean a dataflow subgraph that terminates at a long-latency instruction or a branch instruction in the superstrand microarchitecture[13]. A strand is considered for execution only when all of its operands are ready. Kim and Smith redefined the strand to mean a chain of dependent instructions[10]. In this version of the strand, the result from one instruction solely feeds the input of the next instruction. Local communication within a strand takes place through a single accumulator in the processing element. Strands terminate when an instruction's output is needed by more than one instruction. Sassone and Wills proposed a mechanism for identifying strands at run time[21]. These strands are analyzed in a fill unit and stored in a strand cache. The strict width and fanout requirements of strands restrict their length which reduces their benefits.

Narayanasamy et al. proposed dependency chain processing in a clustered microarchitecture[16]. Dependency chains are dataflow subgraphs of the program formed from hot traces via profiling. Code duplication is used to further enlarge chains. Hot traces are often very dependent upon input set and can lead to more costly branch recoveries since state must be rolled back. Since basic blocks can also frequently end up in multiple hot paths, this approach can introduce significant code duplication in the program binary.

A number of other proposals have suggested the use of dataflow subgraphs to amplify pipeline resources. These proposals track the frequent dataflow subgraphs at compile or run time and use them to speed up the subsequent execution of the subgraph. Bracy et al. proposed dataflow mini-graphs[3]. In mini-graphs, there is a predefined set of ALUs in the microarchitecture. Mini-graphs consist only of instructions that map directly onto the predefined ALUs. The braid microarchitecture is more flexible in the composition of dataflow subgraphs. Macro-op scheduling proposed by Kim and Lipasti[11] is a technique that increases the effective size and throughput of the execution core structures by treating multiple instructions as one fused unit. This technique requires bookkeeping at run time to identify the macro-op instructions. The braid microarchitecture relies on the compiler to identify dataflow subgraphs without run-time assistance.

6.3. Register File

González et al. proposed virtual-physical registers as a technique for mimicking a larger register set by delaying the allocation of registers[9]. Butts and Sohi proposed the prediction of the last use of registers[4]. Martin et al. [14] used the compiler to provide dead value information by making assertions in the program that certain registers will not be used again. The braid microarchitecture provides similar benefits by implicitly freeing internal registers when a braid has finished processing in a BEU.

A number of proposals have examined providing higher bandwidth to the register file via various organizational changes. These include register file banking[26], 2-level hierarchical register file[1][28], register file caching[5][6][2], register read and write specialization[22]. All of these proposals require the hardware management of values. Braids enable the register file to be efficiently partitioned into inter-

nal and external register files at compile time.

7. Conclusions

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. Scaling key structures in the execution core drastically increase design complexity and power requirements. This paper uses a combined compiler and microarchitecture approach to enable wider issue widths while reducing design complexity. The observation that most values have a small fanout and a short lifetime led to the formation of a compile-time entity called the braid. Using the braid as a unit of processing simplifies the design of various critical structures in the execution core of a high-performance processor. The braid microarchitecture uses simple FIFO schedulers for instruction scheduling. The register space is partitioned into a small external register file and several small internal register files. Fewer accesses to the external register file results in a bypass network with a lower bandwidth requirement. The result is performance within 9% of a very aggressive conventional out-of-order microarchitecture with the complexity closer to the complexity of an in-order implementation. The simplifications in the execution core shorten the pipeline lowering the branch misprediction penalty. The simplifications also result in a design that is capable of supporting a higher clock frequency for further performance improvement.

8. Acknowledgments

We gratefully acknowledge the Cockrell Foundation and Intel Corporation for their support. Francis Tseng's stipend was provided by an Intel fellowship. We thank Santhosh Srinath and Veynu Narasiman for their help. We also thank the other members of the HPS research group and Derek Chiou for providing comments.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. González, J. González, and M. Valero. Virtual-physical registers. In *Proceedings of the 4th IEEE International Symposium on High Performance Computer Architecture*, pages 175–184, 1998.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *HPCA '08: Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, February 2008.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.