To appear at HPCA'96. Please do not distribute.

Register File Design Considerations in Dynamically Scheduled Processors

Keith I. Farkas[†]
farkas@eecg.toronto.edu

Norman P. Jouppi[‡]
jouppi@pa.dec.com

Paul Chow[†]
pc@eecg.toronto.edu

[†]Dept. of Electrical and Computer Engineering
University of Toronto
10 Kings College Road
Toronto, Ontario, Canada
M5S 1A4

[‡]Digital Equipment Corporation
Western Research Lab
250 University Avenue
Palo Alto, California 94301

Abstract

We have investigated the register file requirements of dynamically scheduled processors using register renaming and dispatch queues running the SPEC92 benchmarks. We looked at processors capable of issuing either four or eight instructions per cycle and found that in most cases implementing precise exceptions requires a relatively small number of additional registers compared to imprecise exceptions. Systems with aggressive non-blocking load support were able to achieve performance similar to processors with perfect memory systems at the cost of some additional registers. Given our machine assumptions, we found that the performance of a four-issue machine with a 32-entry dispatch queue tends to saturate around 80 registers. For an eight-issue machine with a 64-entry dispatch queue performance does not saturate until about 128 registers. Assuming the machine cycle time is proportional to the register file cycle time, the 8-issue machine yields only 20% higher performance than the 4-issue machine due in part to the cycle time impact of additional hardware.

1 Introduction

A continuing trend in the design of computer systems is the use of superscalar processors that can issue ever more instructions per cycle. Traditionally, these processors have been statically scheduled with the compilers having had the task of uncovering sufficient amounts of instruction-level parallelism to take advantage of the hardware. More recently, however, an increasing number of processors are being introduced that schedule the code at run-time.

Dynamically-scheduled processors seek to increase the instruction-level parallelism by possibly issuing instructions in an order that is different from the issue order for a statically scheduled processor; we refer to the issue order for a statically-scheduled processor as the *program order*. In a dynamically scheduled processor, instructions are issued when a suitable functional unit is available, and after the resolution of data and memory-location dependences with preceding instructions. Since the issue order and hence the completion order of instructions is not deterministic, dynamically-scheduled processors require hardware to ensure that instruction ordering does not affect the behavior of applications.

Hardware is required to control the issuing instructions, to track data flow, and to recover from exceptions. A number of techniques have been used to implement this functionality. Scoreboarding, a technique first employed in the CDC 6600 [1], allows instructions to be dispatched in order but execute out of order. A similar but more powerful technique is that of reservation stations, an idea pioneered by the IBM 360/91 [2]. Implicit in the design of a reservation station is the technique of renaming registers. Register renaming involves the mapping of the registers named in the instructions, the *virtual* registers, to the actual or *physical* registers. Register renaming, in addition to eliminating write-after-write and write-after-read dependences, can also provide more temporary storage locations, which are necessary to allow many instructions to be in execution simultaneously. Although both reservation stations and scoreboards allow instructions to complete out of order, in-order completion can be implemented with the addition of a reorder buffer [1]. Reorder buffers, reservation stations and explicit register renaming hardware are used in the PowerPC 604 processor [3] to implement dynamic scheduling.

An alternate technique and one which subsumes the functionality of reorder buffers, reservation stations, and scoreboards, is *dispatch queues* with explicit register renaming hardware. With this technique, which is used in the MIPS R10000 [4], in-order completion is implemented by the register control logic. Processors using this technique have been implemented with one or more different dispatch queues for different types of instructions. In our model, we use the dispatch-queue technique and a single dispatch queue, because one queue is simpler and the dispatch queue is not the focus of our study. Figure 1 presents an overview of our model; some details are described further below.

The dispatch queue in a dynamically scheduled processor is used to maintain a pool of instructions from which the scheduling logic chooses the instructions to issue next. As instructions are issued, additional instructions are fetched

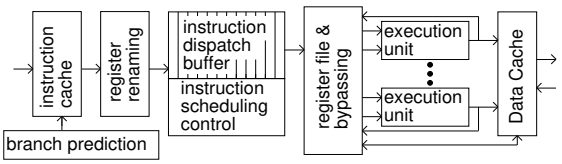


Figure 1: Overview of our dynamic scheduling implementation; only the data path is shown.

from the memory system and are inserted into the dispatch queue, in program order. As instructions are inserted into the queue, the source registers named in the instruction are mapped to physical registers, and the named destination register, if there is one, is mapped to a free physical register. If there are no free registers, then the instruction stream stalls until one becomes available.

Registers can be freed only when doing so will not prevent the processor from recovering and resuming execution after an exception occurs. With *precise exceptions*, it is required that the instructions preceding the faulting instruction in the program order be allowed to change the state of the system while those following the instruction will not; the complete set of conditions for freeing registers is discussed in Section 2.2. In this exception model, a register cannot be freed until all the instructions preceding the instruction writing this register are guaranteed to complete. Under *imprecise exceptions*, the state of the system is not maintained so exactly, thereby allowing registers to be freed earlier, and hence allowing for their more frequent reuse. Although machines with truly imprecise exceptions are rare these days in general purpose systems (since it prohibits multiprogramming and modern OS systems), we have examined a true imprecise exception model as a best case limit for other hybrid exception approaches [1].

The freeing of registers is also affected by branch prediction. Branch prediction is typically used in dynamically scheduled processors to allow the processor to move instructions across branches and thereby increase the pool of those instructions available for issue. Branch prediction, however, can negatively affect performance in two ways. First, mispredicted branches result in the execution of unnecessary instructions, giving rise to a reduction in the average useful instruction-level parallelism. And second, as discussed above but in regards to exceptions, because the direction a branch takes is not definitively known until it is executed, the physical registers that are written by instructions following the branch in program order cannot be freed until the branch is executed. Hence, the time that a register is live (i.e., in use) depends on the accuracy of the branch prediction hardware.

Another factor that directly affects the time that a register is live is the miss rate of the primary data cache. When an instruction does not find the required data in the cache, the instruction's completion is delayed until the data can be fetched. If the instruction is a load instruction, then the register that is the target for the load will need to remain live until the fetch can complete. In addition, the miss will delay the issuing of any instructions in the dispatch queue that *require the result of the load*, hence keeping the registers assigned to them live for longer.

The number of physical registers and the frequency of their reuse have a significant impact on system performance since most instructions require a destination register and instructions cannot be inserted into the dispatch queue when there are no free registers. Such instruction-stream stalls may result in the hardware not being able to keep the dispatch queue full, thereby reducing the number of instructions available for selection by the scheduler, which in turn may limit its ability to schedule the maximum number of instructions per cycle. As mentioned above, the register reuse frequency is a function of the exception model, the branch prediction accuracy, and cache misses. It is also a function of the issue width, and the number and type of functional units, for these factors affect the length of time between the insertion of an instruction into the dispatch queue and its completion. In this paper, we investigate the demand these factors place on the number of registers required. We also consider the demands these factors place on the number of register file ports, and how they affect the cycle time of the register file.

Previous Work

Although many other researchers have investigated dynamically scheduled processors that used register renaming, we know of no research that has focused specifically on issues affecting the register file. And for the most part, in the literature describing these investigations, many authors have neglected to state how many physical registers were available for the renaming of virtual registers. An exception is an investigation carried out by Wall on the limits of instruction-level parallelism that included looking at the impact of varying the number of registers for a 64-issue, 2048-instruction window machine with unit operation latencies [5]. Bradlee, Eggers, and Henry investigated the performance tradeoffs of the number of registers for a RISC instruction set architecture with various kinds of compiler support, but this study was for a statically-scheduled, single-issue processor [6]. Franklin and Sohi also considered a statically-scheduled, single-issue processor in their study of register life times and the replacement of the register file with a distributed mechanism [7].

2 Simulation Methodology

The design requirements of the register file for a dynamically scheduled processor are in part defined by the functionality offered by other system components. The components of interest are: the issue width of the processor and the number of functional units, the size of the dispatch queue, the type of exceptions employed, and the memory system used to service the processor's requests for data. To investigate the relationship between the register file and these components, we simulated a number of machine configurations using scheduling rules and functional unit latencies that resemble those of a number of commercial processors including the PowerPC 604 [3], the DEC 21164 [8], the MIPS R10000 [4] and the SUN UltraSPARC [9]. Each configuration we simulated used the same hardware with the exception of the hardware required to implement the components listed above.

The processor model implements a RISC, superscalar processor whose instruction set is based on the DEC Alpha instruction set. The processor supports non-blocking loads and non-blocking stores, and allows all instructions to be speculatively executed. The processor includes separate instruction and data caches. Since our goal is to keep

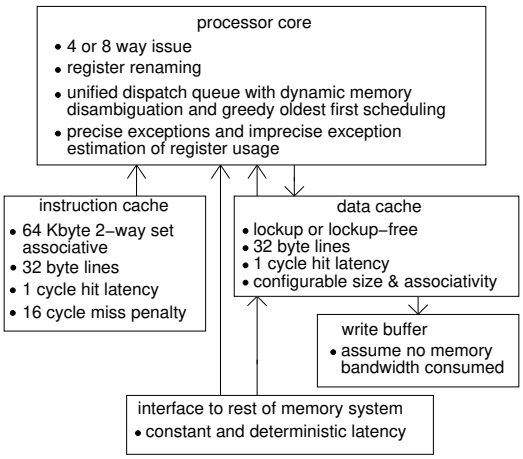


Figure 2: Overview of machine model.

constant the factors that do not directly concern the register file, we assume the servicing of instruction cache misses does not delay the servicing of data cache misses. Hence, the instruction cache has a fixed miss penalty. The data cache can be configured to be either lockup or lockup-free, and requires a deterministic and constant time to resolve cache misses.

The instruction scheduling logic includes a single dispatch queue for all functional units, and its size is configurable. In a clock cycle, the number of instructions that can be inserted into the dispatch queue is equal to 1.5 times the maximum issue width of the processor. Instructions are selected for issuing using a greedy algorithm that issues the earliest instructions in the program order first. The issue logic includes hardware to dynamically disambiguate memory addresses so as to allow memory instructions to issue before those occurring earlier in the program order. The register file includes a configurable and equal number of integer and floating-point registers. The register renaming scheme we use is modeled after the scheme used in the IBM ES/9000 [10], while the dispatch queue is similar to the fast dispatch stack of Dwyer and Torng [11].

The simulator implements both precise and imprecise exceptions. In our simulations, the only source of exceptions is mispredicted branches; arithmetic exceptions are not modeled. We use a branch prediction scheme proposed by McFarling [12] that includes two branch predictors and a mechanism to select between them. This scheme is used to predict the direction of conditional branches; all other control flow instructions are assumed to be 100% predictable. Figure 2 presents an overview of the above machine model; some of the model details are described further below.

2.1 Processor and Memory Models

The processor can issue 4 or 8 instructions per cycle, which are issue widths representative of the current state-of-the art and future processors. For the four-way issue processor, each instruction word can contain at most four operations of which there can be at most: four integer operations, one floating-point division operation, two floating-point operations, two memory operations (i.e., two loads, two stores, or one of each), and one control flow operation (i.e., branch, subroutine call or return). The issue rules for the eight-way issue processor are the same but for each of the above instruction classes, twice the number can be issued in a cycle. All integer functional units have single-cycle latencies except for the multiply unit, which is fully pipelined and has a six-cycle latency. All floating point units have three-cycle latencies and are also fully pipelined, with the exception of the floating-point divider. The floating-point divider is not pipelined and has an eight-cycle latency for 32-bit divides, and a 16-cycle latency for 64-bit divides. Finally, stores take one cycle to be resolved and there is a single load-delay slot.

The combined-predictor branch prediction scheme we model has a 12 Kbit cost and comprises a bimodal predictor and a global history predictor. The bimodal predictor employs the classical branch prediction idea of having a set of counters that indicate the direction taken by the branches that shared the counter the previous times they were executed; we use 2048 two-bit saturating counters. The global history predictor uses a shift register to generate a combined history of the direction of the last n branches. The contents of this register are exclusive ORed with the program counter word address to select one of another set of 2048 two-bit counters; these counters are used in the same way as the first set. The selection mechanism is essentially a bimodal predictor whose state reflects which branch predictor has been most correct. The global-history shift register is updated after each conditional branch is inserted into the dispatch queue using the predicted direction; the two-bit saturating counters are updated when a conditional branch is issued, that is, executed. By updating the shift register during insertion, we can take advantage of already identified branch patterns when determining the instruction to fetch next. A consequence of updating the register early, however, is that on a mispredicted branch, the shift register must be loaded with the value it contained before the mispredicted branch was inserted into the dispatch queue.

Stores are assumed to be implemented using write-around (i.e., no-write-allocate) and write-through policies with a write buffer situated between the data cache and lower levels in the memory hierarchy. Since our goal is to keep constant the factors that do not directly concern the register file, we assume that no memory bandwidth is required to retire stores in the write buffer. This assumption prevents any stalls due to a full write buffer and prevents stores from delaying the servicing of cache fetches.

The data cache can be configured to be lockup or lockup-free. The lockup-free cache employs an inverted MSHR (Miss Status Holding Register) organization [13] to process cache misses. An inverted MSHR organization can support as many in-flight cache misses as there are registers and other destinations for data in the processor. For the four-way issue processor configuration, the register file has eight read ports and sufficient write ports to prevent any write-port conflicts arising when registers are filled on the resolution of a cache miss. For the eight-way processor configuration, there is twice the number of ports. (Section 3.4 discusses read and write ports further).

Requests for blocks of data are sent via the memory interface to the next level in the memory hierarchy. The blocks of data are returned in a constant and deterministic number of cycles called the *fetch latency*. When a block is returned to the cache, the cache line is written simultaneous with the writing of the appropriate words into all

registers with loads outstanding to this block (updating all pending registers requires the multiple write ports mentioned above). This simultaneous writing is represented in Figure 2 by the arrow that bypasses the data cache. Writing a register or a cache line is assumed to take one cycle.

2.2 Freeing Physical Registers

Registers can be freed only when their being freed will not prevent the processor from recovering and resuming execution after either an exception occurs during the execution of an instruction, or a branch is mispredicted. The conditions under which a register can be freed depend on whether precise or imprecise exceptions are supported. Key to the freeing of registers are the following concepts: the *completion* and *commitment* of an instruction, and the *creation*, *retiring*, and *killing* of a virtual-to-physical mapping.

An instruction is said to *complete* when it has reached the point of altering the state of the machine; branches complete when they change the program counter, stores complete when the cache is updated or the data is placed in the write buffer, and other instructions complete when their destination registers are written. Once an instruction completes, the instructions following it in the program order can make use of the result or the side-effect it produced. An instruction is said to *commit* when it has completed *and* all the instructions preceding it in program order have completed. A committed instruction will never be reissued because all of the instructions before it in the program order have completed. A completed instruction, however, will be issued again should the subsequent execution of any of the instructions preceding it in the program order give rise to an exception or a mispredicted branch. In our simulator, the maximum number of instructions that can be committed in each clock cycle is exactly twice the issue width of the processor, modeling probable hardware limitations.

When an instruction I that names a destination register, say register R_v, is inserted in the dispatch queue, this register is renamed to a physical register, say R_p^1. When this renaming occurs, we say that a virtual-to-physical mapping has been *created*. As subsequent instructions in the program order are inserted into the queue, any that use register R_v as an operand will have this register renamed to R_p^1. This mapping between R_v and R_p^1 remains active until a subsequent instruction is inserted into the queue that names R_v as a destination. At this point, another physical register R_p^2 is mapped to R_v, and the $R_v \rightarrow R_p^1$ mapping is said to have been *retired*. A retired mapping is eventually *killed* and the point at which this killing occurs depends on the exception model. The register whose mapping has been killed is free for reuse.

Conditions for Freeing Registers

To facilitate the discussion of the conditions for freeing registers, consider the scenario in which a virtual register R_v is named by an instruction I_1 as its destination register, and R_v has been mapped to a physical register R_p. Under precise exceptions, the register R_p will be freed when an instruction I_2 commits if instruction I_2 is the first instruction after instruction I_1 in program order to have register R_v as a destination. Inherent in this condition are the following two requirements: (1) Instruction I_1 has committed; (2) The instructions that use the register R_p have committed (these instructions occur later in the program order than instruction I_1 and earlier than instruction I_2).

The condition for freeing registers ensures that the exact state of the machine can be recovered at any point should an instruction suffer an exception or a branch is mispredicted[1]. When an exception does occur or a branch is mispredicted, the mappings for each virtual register must be set back to the mapping that existed *before* the execution of the instruction causing the exception and any instructions following this instruction in the program order. The resetting of the mappings entails moving a pointer in the virtual-to-physical register-map table. The physical registers will still contain the correct values because a register cannot be freed until all the instructions preceding its writer have committed. That is, until all the instructions before an instruction I commit, the operands required by I will remain in the register file. A second step in the recovery from an exception or a mispredicted branch is that all the instructions later in the program order than the instruction that caused the exception are removed from the machine. If any of these names a destination register, then the physical register is freed. These registers can be freed since the instructions that depend on them are also removed from the machine. Finally, on-going cache block fetches that were initiated by instructions that have been removed are marked so that the cache block will not be written into the cache or be used to write registers when the block returns from memory.

In our processor model, we assume that a register can be reused in the cycle after the conditions for freeing it are satisfied. We also assume that any functional units that are busy with an instruction that is removed will be available for reuse in the cycle after the exception or branch occurred.

Under imprecise exceptions, registers can be freed earlier. Again, to facilitate the discussion, consider the scenario in which a virtual register R_v is named by an instruction I_1 as its destination register, and R_v has been mapped to a physical register R_p. With imprecise exceptions, the physical register R_p can be freed when: (1) Instruction I_1 has *completed*; (2) The instructions that use register R_p have *completed*; (3) The virtual-to-physical mapping is killed by the *completion* of *any* instruction I_x that follows instruction I_1 in the program order if instruction I_x has register R_v as its destination *but only when all the branches preceding instruction I_x have completed*.

These conditions differ from those for precise exceptions in several important areas, and these are indicated by the emphasized typeface in the above list. First, the first two conditions are not subsumed by the virtual-to-physical mapping condition (condition no. 3), a result of it only being necessary for instructions to "complete" rather than "commit" (condition no. 2). Second, it is important for all preceding branch instructions to complete rather than for all preceding instructions to commit. And third, the writer of a physical register can cause the killing of *any* mappings created by preceding instructions, rather than only *the* preceding mapping. Taken together, these differences allow

[1] This statement is actually only partially true since the conditions given do not address changes in state caused by the execution of stores. To allow the recovery of the machine state, a non-merging buffer is required to hold the write data until the store instruction commits. Only at this point can the data be written into the cache or the write buffer. We do not consider stores further as this paper is concerned primarily with the register file.

Bench-mark	Data set	Com-mit instr.	4-way issue							8-way issue						
			Execute instr.			IPC		Rates (%)		Execute instr.			IPC		Rates (%)	
			total	load	cbr	issue	c'mit	load	cbr	total	load	cbr	issue	c'mit	load	cbr
compress	ref	86	126	29	14	3.06	2.09	15	14	170	42	17	4.90	2.50	10	14
doduc	small	190	209	48	12	2.75	2.49	1	10	235	56	13	4.92	3.97	1	10
espresso	ti	560	626	138	91	3.39	3.04	1	13	733	171	101	5.57	4.26	1	14
gcc1	cexp	23	27	6	3	2.80	2.35	1	19	32	8	3	4.47	3.14	1	20
mdljdp2	small	291	319	48	31	2.33	2.12	3	6	351	54	34	4.05	3.36	3	6
mdljsp2	small	350	386	82	31	2.97	2.69	1	6	429	94	33	5.25	4.28	1	6
ora	small	190	190	31	8	1.86	1.86	0	6	190	31	8	2.08	2.08	0	6
su2cor	small	417	437	107	12	3.38	3.22	17	7	460	114	13	6.24	5.65	22	7
tomcatv	ref	910	911	247	30	2.77	2.77	33	1	912	248	30	5.52	5.51	39	1

Table 1: Dynamic statistics for each benchmark for both issue widths, using 2048 physical registers, a 64 KByte, 2-way set associative lockup-free data cache with a 16 cycle fetch latency. Instruction counts are in millions; the "rates" columns give the cache load miss rate and conditional branch misprediction rate. The 4-way issue results are for a dispatch queue with 32 entries, while the 8-way issue results are for one with 64.

registers to be freed earlier, and allow the exact state of the machine to be recovered without assistance from software when a mispredicted branch does occur.

3 Performance Trends

This study is based on execution-driven simulations using an object code instrumentation system called ATOM [14], which is available for Alpha AXP workstations. The results presented correspond to simulations of nine of the SPEC92 benchmarks representing a balance between floating-point-intensive and integer-intensive applications. The benchmarks are listed in Table 1 along with some run-time characteristics for the four-way and eight-way issue processors. The column headed "Data set" specifies which of the official SPEC92 data sets were used for the simulations. In all cases, the benchmarks were compiled using the Alpha native C compiler with the global ucode optimizer enabled, and the linker was directed to perform link-time optimizations. In all cases, the instruction cache miss rate was under 1%.

The column headed "Commit instr." gives the number of instructions in the trace for each benchmark, which is equivalent to the number that commit (see Section 2.2 for the definition of *commit*). The number of committed instructions does not necessarily equal the number of instructions that are executed (i.e., issued) due to mispredicted branches. The number of executed instructions is given under the columns headed "Executed instr." with sub-columns for the number of loads ("load") and for the number of conditional branches ("cbr"). Both the number of committed instructions and the number of executed instructions are dynamic instruction counts.

The average number of instructions per cycle (IPC) for each benchmark and each issue width are given in the columns headed "IPC". The *issue IPC*, given in the sub-columns headed "issue", is the ratio of the number of instructions that are *issued* to the total (simulated) run time; the issue IPC measures the rate at which instructions are dispatched to the functional units. In our system model, the difference between the issue IPC and the maximum issue width is due to the dependences in the code and the number and type of functional units. The *commit IPC*, given in the columns headed "c'mit", is the ratio of the number of instructions that *commit* to the total run time. The difference between the issue IPC and the commit IPC is due to instructions that are incorrectly speculatively executed when following mispredicted branches. The commit IPC values we report are optimistic in part due to our assumption of a bandwidth-unconstrained memory system. To illustrate this fact, consider the commit IPC of 5.51 given in the table for *tomcatv* using the eight-way issue processor. Replacing the non-blocking cache with a blocking one, the commit IPC is reduced by 70%. Our commit IPCs are also high because we assume a larger and less restricted set of functional units than many recently announced microprocessors.

The branch misprediction rates are given along with the overall cache miss rates for loads under the columns headed "Rates". The misprediction rates shown are larger than those reported by McFarling [12] for the same branch prediction scheme and a statically scheduled processor. The increase is in part due to the use of the dispatch queue in a dynamically scheduled processor. The dispatch queue increases the time between when the prediction is made and when the predictor tables are updated with the direction taken by the branch (the prediction is made at the point of insertion into the dispatch queue while the updating occurs at the point of executing the branch instruction). Hence, predictions are based on information that may not reflect the direction actually taken by immediately preceding branches in the program order. In addition, the information used reflects the execution order rather than the program order of branches, and these two are not necessarily the same. In practice, however, we found that while the branch prediction accuracy did improve somewhat with in-order execution of conditional branches, this improvement occurred at the expense of a notable decrease in the commit IPC. Hence, we allow branches to execute out of order.

We present our investigation of factors affecting the register file design in four parts. We begin with an investigation of a processor with a large number of physical registers to evaluate how register requirements change as we vary the issue width and the size of the dispatch queue. From the results, we identify a cost effective dispatch queue size for the two issue widths. Using these dispatch queue sizes, we

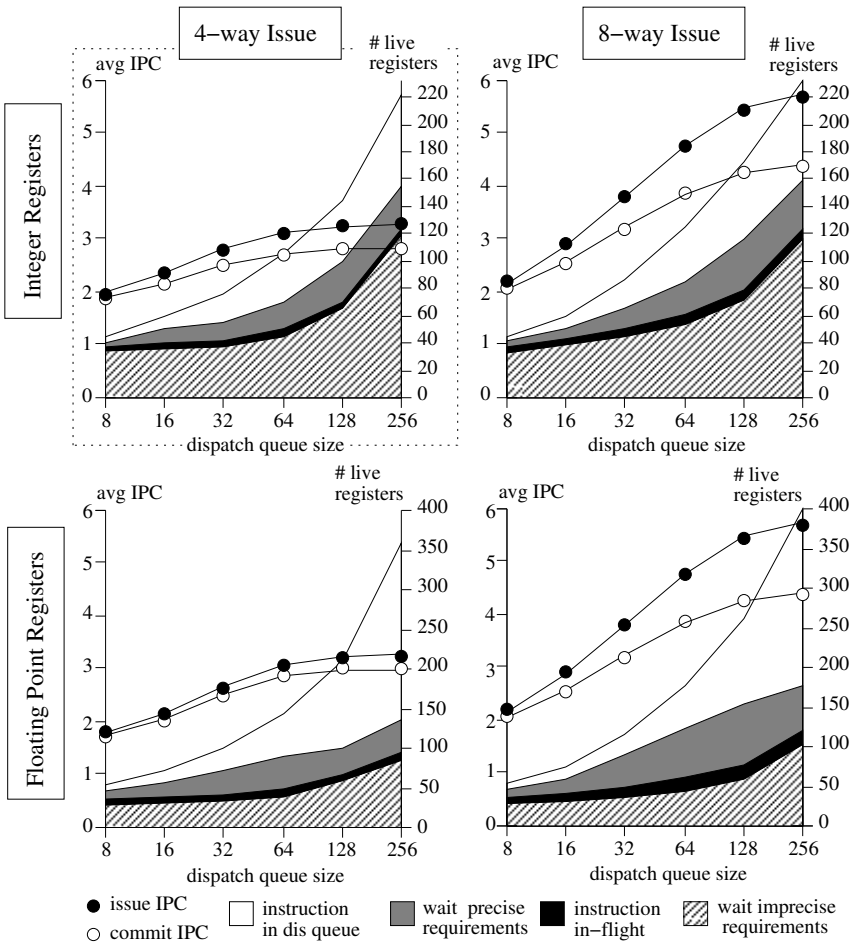


Figure 3: Average IPC and 90^{th} percentile number of live registers for all benchmarks as a function of the size of the dispatch queue. The shaded areas indicate the fraction of the live registers in each of four states.

then investigate the register requirements and performance of the two exception models. In the third part, we investigate the impact that the memory system organization has on the register requirements and on performance. Finally, in the last part, we evaluate the cycle times of the register file designs we use.

3.1 Trends for Large Register Files

To investigate the register file requirements under variations in the dispatch queue size and issue width, we configured the system model with 2048 integer and 2048 floating-point registers. These values were chosen to minimize the impact on our measurements from instruction-stream stalls caused by a lack of free registers. Such stalls occur if an instruction cannot be inserted into the dispatch queue because there are no free registers; in our simulations, such stalls accounted for much less than 1% of the run time. Using this system model, we measured the number of live registers using the 90^{th} percentile as our metric; the 90^{th} percentile indicates how many registers should be provided by the hardware to achieve nearly the same average commit IPC as is achieved with 2048 registers[2]. The 90^{th} percentile was chosen in lieu of a geometric mean or an arithmetic average owing to the non-uniform and non-Gaussian shape of the distributions.

In examining the relationship between the dispatch queue and the number of registers, it is useful to categorize the registers that are live into one of four categories. The live registers in our system may be (1) assigned to instructions residing in the dispatch queue, (2) assigned to instructions presently being executed (i.e., in-flight), (3) waiting for the imprecise exception register-freeing requirements to be met, (4) waiting for the precise exception register-freeing requirements to be met. Applying this categorization to the

[2]The 90^{th} percentile is determined by first having the simulator record how many registers were live in each cycle of a benchmark's execution. Then, this distribution of cycle counts for each register value is normalized by the (simulated) run time of the benchmark, giving normalized cycle counts of between zero and one. Next, the normalized distribution for all benchmarks of a given system model are averaged together. Finally, we determine the number of registers needed to cover 90% of the resulting distribution. This approach was adopted to prevent the distribution of a single benchmark from dominating the combined distribution.

integer and floating point registers, and the two issue widths yields four sets of data. Figure 3 presents these four sets of data as graphs[3]. We begin by discussing the graph corresponding to the integer registers of the four-way issue processor (the upper left-hand graph which is enclosed in a dotted box).

This graph presents the average issue IPC, the average commit IPC and the number of live registers as a function of the dispatch queue size using the baseline cache configuration of a 64 KByte, two-way set-associative lockup-free cache with a 16-cycle fetch latency. As shown by the curve marked with the solid circles, the issue IPC approaches the issue width of the processor as the size of the dispatch queue is increased. This trend is a by-product of the increased scheduling flexibility afforded by the larger pool of instructions available for scheduling, with the effect of allowing the scheduler to avoid many functional unit conflicts and data dependences. The commit IPC, as shown by the curve marked with open circles, also increases with increasing dispatch queue sizes, but at a slower rate than that of the issue IPC. This difference in rates is due to the issuing of instructions that don't commit because a preceding branch in the program order was mispredicted. Also note that the gap between the issue and commit IPC is significantly larger for the eight-way issue processor than the four-way issue processor. This fact is a result of the eight-way issue processor speculatively executing more instructions, and these instructions are subject to branch misprediction.

Turning to the other curves in the graph, the upper boundary of the "instruction in the dispatch queue" (white) region corresponds to the number of registers live under the precise exception model, and the size of the "wait precise requirements" (stippled) region indicates the number of additional registers required to support precise exceptions over imprecise exceptions. Note that there are at least 32 live registers. This value is the minimum number of live registers under both exception models for any program that references all virtual registers, as do most useful programs. With fewer than 32 physical registers, the system will become deadlocked since, to free a physical register, at some point there must be two physical registers assigned to the same virtual register, and there are 31 virtual registers that can be renamed (the zero register is not renamed). This situation is needed to effect the killing of the virtual-to-physical mapping, as discussed in Section 2.2.

The shape of the upper boundary of the white region shows that the number of live registers increases with increasing dispatch queue size. There are two primary reasons for this relationship. First, as the number of entries in the dispatch queue increases, instructions will likely be issued in an order less and less similar to the program order; in other words, there will be more out of order issue. When these instructions complete, their destination registers cannot be freed until the conditions for the exception model are met. Since these conditions involve the completion of instructions earlier in the program order, the registers will remain live for longer. Hence, there will be an increase in the number of live registers waiting for precise and imprecise freeing requirements to be met. The increase under imprecise exceptions is illustrated in the graph by the lined region; the increase under precise exceptions is illustrated by the stippled region. Observe that the lined region exhibits a more substantial increase in size with larger dispatch queues than does the stippled region. The stippled region represents registers assigned to instructions that have already satisfied the requirements for imprecise exceptions. Since these requirements involve the completion of all preceding conditional branches in the program order, instructions in the wait-for-precise-requirements category must have had any preceding conditional branches already complete. This fact reduces the out-or-orderness of instructions in the wait-for-precise-requirements category, and hence, the number of registers pending completion.

Second, as the number of entries in the dispatch queue increases, more instructions remain in the dispatch queue longer. Since registers are allocated to these instructions when they are inserted into the queue, there will be an increase in the number of live registers. In addition, the slight increase in issue IPC gives rise to only a slight increase in the number of instructions in flight; this effect is illustrated by the black region. This trend is also present in the other three graphs in the figure, that is, those corresponding to the floating point register file and the eight-way issue processor. Observe that the doubling of the issue width results in less than a doubling of the number of registers associated with instructions in flight. This fact is due to the less than doubling in the issue IPC that occurs with a doubling in the issue width.

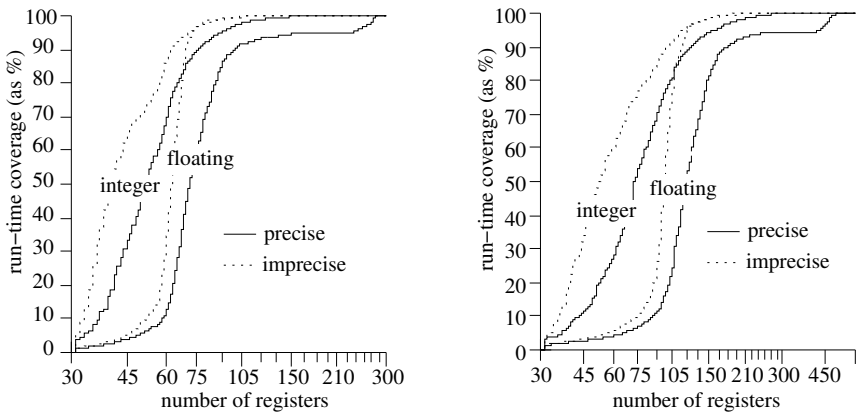
Finally, observe that as the dispatch queue size increases, there is a value at which the average commit IPC approaches its asymptotic value. For the four-way issue processor, this point occurs around a dispatch queue of 32 entries whereas for the eight-way issue processor, it is around 64 entries. Moreover, once a certain number of dispatch queue entries is reached, a greater proportion of the increase in the number of live registers is attributable to the instructions residing in the dispatch queue. Taken together, these two trends suggest that a dispatch queue of 32 entries is most cost-effective for the four-way issue processor, and one of 64 entries is most cost-effective for the eight-way issue processor.

3.2 Precise versus Imprecise Exceptions

The non-zero size of the precise region in Figure 3 suggests that the use of the precise exception model requires more registers than is required under the imprecise exception model. To examine this trend in more detail, we begin by presenting register usage histograms of the floating point registers for *tomcatv* for both exception models. Figure 5 shows these histograms as run-time coverage curves. In this figure, the x-axis specifies the number of registers live at each cycle during the execution of the benchmark while the y-axis indicates the percentage of the total number of cycles with at most the indicated number of registers live. For example, on an 8-issue machine under the precise exception model, for 70% of the run time there were 150 or fewer floating-point registers live.

Observe that the floating-point register count at which the imprecise exception model reaches 100% coverage is $\approx$130 while the same point for precise exceptions is $\approx$500. Also observe that the curve representing imprecise excep-

[3] In Figure 3 and all subsequent figures that present averages for all benchmarks, the curves for the integer registers include data from all benchmarks whereas the floating-point register curves only include data from the floating point intensive benchmarks.



(a) 4−way issue processor (b) 8−way issue processor

Figure 4: Average register usage histograms under both the precise and imprecise exception models using a lockup-free cache. The four-way issue processor histograms correspond to a dispatch queue with 32 entries while those for the eight-way issue processor correspond to a dispatch queue of 64 entries.

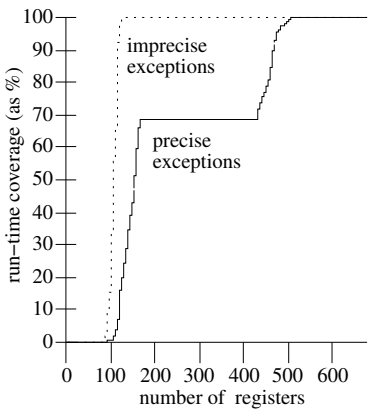


Figure 5: Impact of the exception model on the floating pointer registers for *tomcatv* with an 8-way issue processor, a 64 entry dispatch queue, and a lockup-free cache.

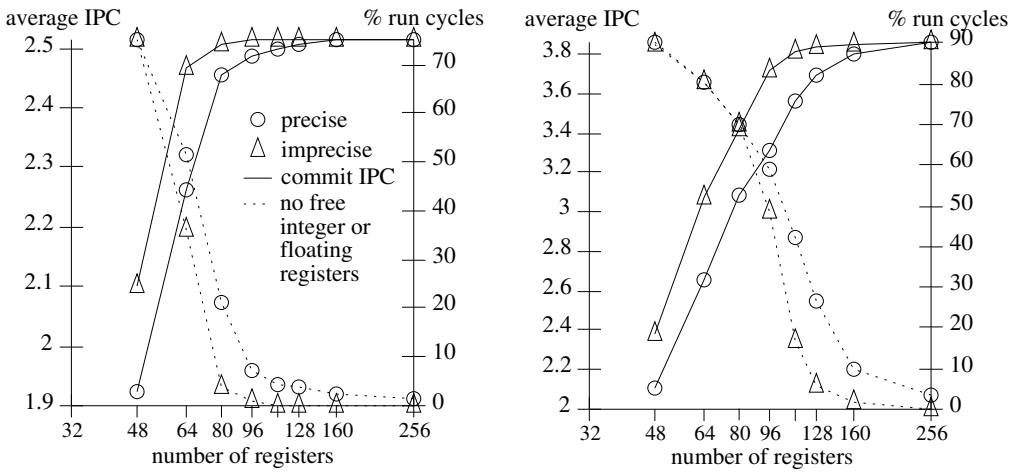
tions has shifted towards zero, an indication that fewer registers are required with this exception model. With precise exceptions, the corresponding curve exhibits a flat region between 150 and 400 registers, signifying that there were rarely 150 to 400 registers live, and that the register usage distribution is bimodal. The second modality, which is centered around 450, is a result of the more strict conditions for freeing registers under the precise exception model. Even though the dispatch queue has only 64 entries, the need for 500 registers in the precise model shows that at some points there is at least one instruction in the dispatch queue which is 500 instructions out of sequence (i.e., there is an instruction in the dispatch queue which occurs at least 500 instructions later in the program order than the earliest instruction). And because the 499 intervening instructions cannot be committed until the earliest instruction completes, any registers assigned to these instructions cannot be freed.

Although *tomcatv* represents an extreme case for register usage, the average 100% coverage points for all benchmarks are still significant, as shown in Figure 4. This figure gives the run-time coverage for both issue widths and register files; the curves were obtained by averaging the run-time coverage curves for each benchmark. As shown in the figure, 90% coverage is achieved with 90 registers for the four-way issue processor and 150 registers for the eight-way issue processor. Unfortunately, providing even 90 registers with sufficient numbers of read and write ports could be prohibitively expensive.

To evaluate the impact on performance of using a smaller and more realistic number of registers, we simulated the benchmarks with different register-file sizes while keeping the dispatch queue size constant. The results of this evaluation are presented in Figure 6 for both issue widths and both exception models. As shown by the solid lines in the figure, the commit IPC increases with larger register files, but the degree of improvement diminishes at the larger sizes. This trend is due to the decreased pressure on the registers with larger register-file sizes. The register pressure is represented in the figure as dotted lines; these lines give the percentage of the run-time for which there were no free registers. Observe that with larger register files, there are usually free registers available, a fact that accounts both for the leveling off of the performance, and for the similar performance under both exception models. With smaller register files, there is a more significant performance difference between the two exception models. The performance difference arises because under the imprecise model, on average, registers are live for shorter amounts of time.

3.3 Memory System Effects

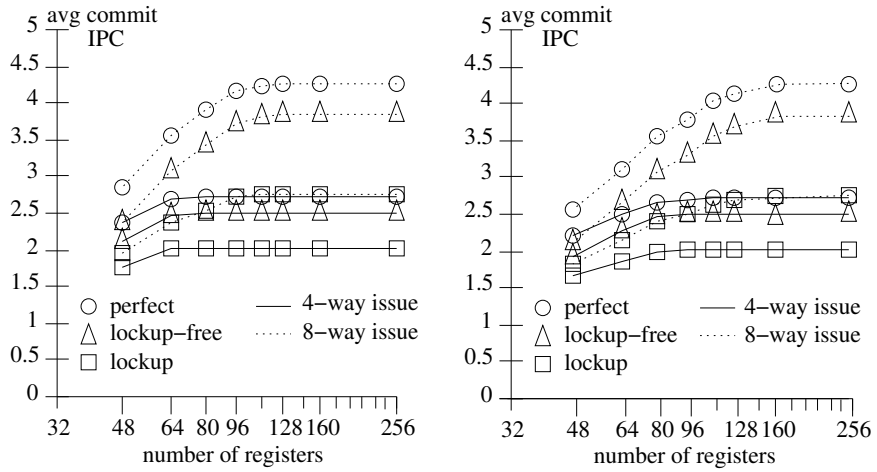
In the preceding sections, we discussed how the number of live registers is affected by the size of the dispatch queue, the issue width of the processor, and the exception model. Another factor that directly affects the number of live registers is the data cache miss rate. When a load instruction does not find the required data in the cache, its completion is delayed until the data can be fetched from memory. As a result, the register target of the load will need to remain live for longer. In addition, any instructions that use the result of the load cannot be issued until the load



(a) 4-way issue processor

(b) 8-way issue processor

Figure 6: Average IPC for the benchmarks and the percentage of the run time during which there were no free registers. The dispatch queue size was held constant and the number of registers varied.



(a) Imprecise exception model

(b) Precise exception model

Figure 7: Average commit IPC for three data cache organizations using a 32 entry dispatch queue for the four-way issue processor, and a 64 entry dispatch queue for the eight-way issue processor.

completes, thereby increasing the live time of their source and destination registers. Finally, the average time a register is live will also increase because by delaying some instructions from being issued, it is more likely that fewer instructions will be issued in program order. Thus it will take longer to meet the exception model requirements for freeing registers.

To evaluate the impact of the memory system organization, we simulated the following three cache organizations: a cache with an assumed 100% hit rate, referred to as the *perfect* cache, and two 64 Kbyte, 2-way set-associative, caches both with a 16 cycle miss penalty, one being lockup-free and the other not. We assumed that the lockup-free cache could initiate as many new cache line fetches as necessary from the next lower level in the memory hierarchy in each cycle. Figure 7 presents the average commit IPC for each issue width and each of these cache organizations as a function of the number of registers; Figure 7(a) presents a set of curves for imprecise exceptions, while Figure 7(b) presents a set for precise exceptions. Observe first the familiar concave shape of the IPC curves, and second, that under precise exceptions, more registers are required to obtain a similar performance than is required under imprecise exceptions. Note also that the lockup cache organization achieves significantly worse performance for both issue widths. This shows that the benchmarks require a cache

with at least some lockup-free support. Finally, the performance curves for different memory system models tend to saturate at roughly the same register count for a given issue width and exception model. For example, the performance of an 8-way issue machine with imprecise exceptions saturates for 96 registers or more, independent of the memory system model.

Further insight into the performance difference between the three cache organizations is provided by the register usage histograms obtained when these three organizations are employed in a system with 2048 registers. We have chosen to present the integer register histograms for *compress* as they clearly show the differences between the organizations owing to the significant cache miss rate of *compress*. The histograms are presented in Figure 8 as run-time coverage curves. As in Figure 5, the x-axis specifies the number of registers live at each cycle during the execution of the benchmark while the the y-axis indicates the percentage of the total number of cycles with at most the indicated number of registers live. Comparing the shapes of the curves for the perfect cache (the solid line) to the curve for the lockup-free cache (the dotted line), we note that the lockup-free cache requires more registers to obtain the same run-time coverage as the perfect cache. In addition, the smaller slope of the lockup-free cache curve indicates that the live registers are concentrated in a wider range. This range becomes smaller when a lockup cache is used, suggesting that the additional registers required for the lockup-free cache are a result of allowing multiple outstanding cache misses and cache probes to occur during the servicing of the misses. The curve for the lockup cache, however, is similar in shape to that for the perfect cache, but the curve for the lockup cache shows that the majority of the number of registers is concentrated in a more narrow region (between 55 and 75), suggesting that there is less variance in the register requirements. This reduction in the register requirements may be due to more in-order issuing of instructions and thus less variance in the time required to meet the conditions for freeing registers.

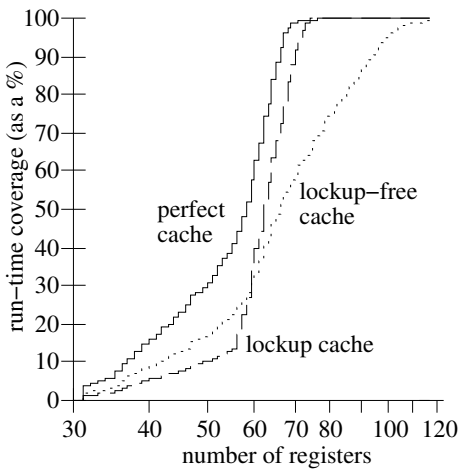


Figure 8: Cumulative register usage histogram for *compress* showing how many registers are live as a percent of run time. The system modeled used precise exceptions, a 4-way issue processor, 32 entry dispatch queue, and 2048 registers.

3.4 Timing Model

In a wide-issue dynamically scheduled processor, there are a number of critical paths that will likely determine the cycle time. These paths include the dispatch queue, the register renaming unit, and the register file. The implementation size and complexity of these structures tend to scale together since it is desirable that none of these structures offer an disproportionate amount of functionality. For example, if many additional ports are added to the register renaming tables, additional ports to the register file will probably be needed as well. Similarly, if many additional entries are added to the dispatch queue, additional registers in the register file will probably be needed as well.

The register renaming unit and the dispatch queue are subject to wide variations in implementation architecture and circuitry, while the register file design space is more limited. Independent of how the dispatch queue and renaming unit are implemented, however, they will have structures similar to those found in the register file (such as numbers of ports or numbers of entries). Hence, we assume the register file cycle time scales similarly to their cycle times, and therefore to that of the machine as a whole.

We present an evaluation of the register file cycle times required for the four-way and eight-way issue processor systems. This evaluation assumed a lockup-free cache organization and a 32 entry dispatch queue for the four-way issue processor, and a 64 entry dispatch queue for the eight-way issue processor. We simulated a number of register file designs each differing in the number of read and write ports and the number of registers. The number of read and write ports was set by the issue width of the processor while the register file sizes correspond to those used in Figure 6. For these simulations, we modified the cache access and cycle time model of Wilton and Jouppi [15] to generate cycle times for multiported register files using the register file cell shown in Figure 9. This cell uses two bitlines per write port and one bitline per read port. One wordline is required per port. We assumed a 0.5μm CMOS technology.

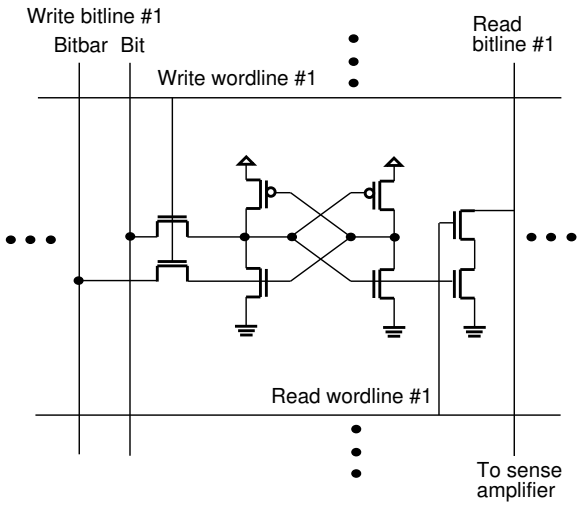
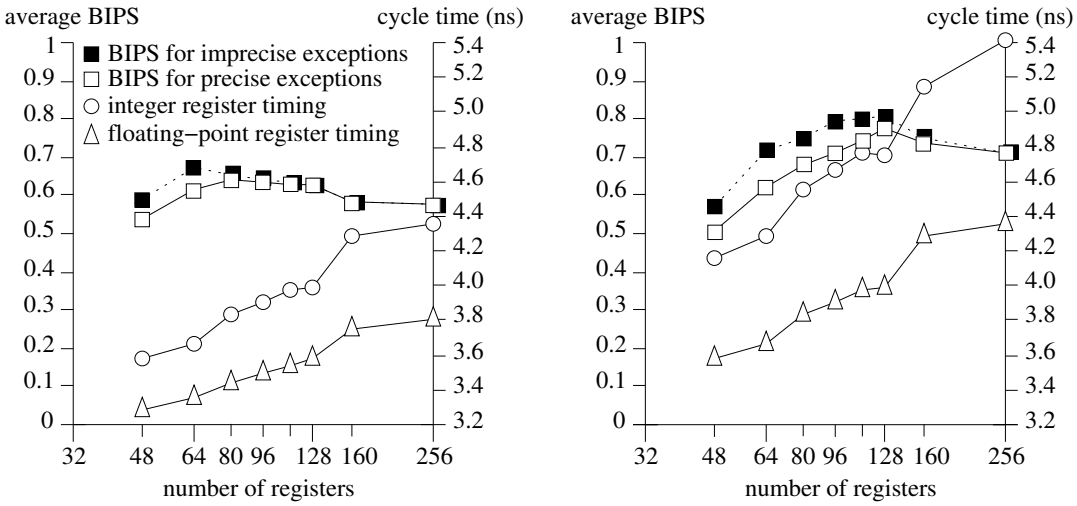


Figure 9: Multiported register file cell.



(a) Four−way issue processor

(b) Eight−way issue processor

Figure 10: Register file timing and estimated machine performance. The four-way issue processor used a 32 entry dispatch queue while the eight-way issue processor used a 64 entry dispatch queue.

Using this model, we determined the cycle time for each of the integer and floating-point register files as a function of the size of the register file. For the four-way issue processor, we assumed the integer register file had 8 read ports and 4 write ports, whereas the floating-point register file had half as many (because only half as many floating-point instructions can be issued per cycle in our model); twice the number of ports were assumed for the eight-way issue processor. The results of the evaluation are presented graphically in Figure 10. This figure presents for both issue widths two register file timing curves and two estimated performance curves.

In the two graphs shown in the figure, the cycle time of the floating-point register file is given by the curve marked with triangles, while the cycle time of the integer register file is given by the curve marked with circles. Note that the cycle time of the floating point register file is always smaller than the integer register file, a speed difference that is attributable to the floating-point register file having half the number of ports as the integer register file. The register file cycle times for the four-way issue processor also show a smaller increase as the number of registers is doubled than the increase which occurs with a doubling of the issue width for the same register file size. This relationship is due to the cycle time of a large register file being more strongly affected by a doubling of the number of register file ports rather than a doubling of the number of registers. For a register file, doubling the number of ports doubles the number of wordlines and bitlines (quadrupling the register file area in the limit), but doubling the number of registers only doubles the number of wordlines (doubling the register file size in the limit).

The two graphs also show an estimate of machine performance assuming that the machine cycle time scales proportionally to that of the integer register file. Performance is measured in billions of instructions per second (BIPS) and is derived by dividing the average commit IPC from Figure 6 by the cycle time of the register file in question. The performance obtained under precise exceptions is shown by the curves marked with white squares while that obtained under imprecise exceptions is shown by the curves with black squares. For both issue widths, the imprecise model has a small performance advantage with small numbers of registers. However, in the four-way issue processor, there is little performance difference between the two exception models with register file sizes greater than 80. For the eight-way issue processor, this point occurs at 160 registers, a result of the need for more registers due to the more out-of-order issue of instructions in the eight-way issue processor.

The performance curves in Figure 10 all exhibit performance maxima at moderate numbers of registers. For register files smaller than these maxima, the average BIPS falls off, a result of instruction-stream stalls. For register files larger than these maxima, the increasing register file cycle time negatively impacts the machine cycle time and hence, overall performance. We also note that the maximum performance only improves by 20% when moving from the 4-issue machine to the 8-issue machine. A major reason for this fact is the large increase in the cycle time that is mandated by the larger and more complex register file. Although the data presented in this figure is for a dynamically scheduled processor, a VLIW processor with centralized integer and floating-point register files would also be subject to performance limits similar to Figure 10. Hence, there is a need for new decentralized architectures, such as the proposed Multiscalar architecture[16].

4 Conclusions

We have investigated a number of issues in the design of register files for dynamically scheduled superscalar processors. From these investigations we draw the following conclusions.

First, the additional register requirements for providing precise exceptions in these processors is relatively small. The imprecise model we simulated only reduced the av-

erage number of registers required by the four-way issue machine by at most 20% with a dispatch queue of 32 entries. The difference between the precise and imprecise models was larger for the eight-way issue machine, since to get good utilization of the eight-way issue machine, the instructions must be executed more out of program order. The eight-way issue machine using imprecise exceptions required an average of 37% fewer registers than one using precise exceptions with a dispatch queue size of 64. Because the register file cycle time is more heavily dependent on the number of register file ports than the number of registers, and in view of all the other hardware required by a dynamically scheduled superscalar processor, the additional registers required to support precise exceptions are a small cost.

Second, the combination of dynamic scheduling and aggressive non-blocking load support can achieve performance quite close to that of systems with single-cycle direct memory access (a perfect memory system). Although the processor at times could use many hundreds of registers, we found that limiting the number of registers to 80 (both integer and floating-point) for the four-way issue machine and 128 for the eight-way issue machine, resulted in performance that was only a few percent lower than that of a machine with an unlimited number of registers.

Third, we extended a cache memory access and cycle time model to model register file cycle times. Although there are many critical paths in a dynamically scheduled superscalar processor, the worst may have timing that scales similarly to that of register files with complexity. Therefore we approximated the scaling of machine cycle time with complexity to be proportional to the scaling of the required register file cycle time. Since the register file becomes slower as the number of registers increases, and the resulting IPC tends to saturate, the overall machine performance has a maxima with the above noted number of registers. In addition, since the register file cycle time is also strongly dependent on the number of ports, we conclude from our simulations that the use of centralized integer and floating-point register files may yield only a 20% performance improvement for an eight-way issue processor over a four-way issue processor.

Acknowledgments

The research described in this paper has been partially funded by the National Sciences and Engineering Research Council of Canada and by Digital Equipment Corporation. We thank Alan Eustace for the answers to numerous questions as we used the ATOM simulation infrastructure. We also thank Brad Calder, Annie Warren, and the other WRLites for both helping out and putting up with the simulations. Finally, we thank Digital Equipment Corporation for providing us with the Alpha AXP workstations.

References

[1] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 2nd edition, 1995.

[2] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM 360 Model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[3] S. Peter Song. Power PC 604. *In the proceedings of Hot Chips VI*, August 1994.

[4] John Brennan. T5: A high-performance superscalar MIPS processor. *In the proceedings of MicroProcessor Forum*, October 1994. See also http://www.mips.com/HTMLs/T5_B.html.

[5] David W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, Digital Equipment Corporation Western Research Lab, November 1993.

[6] David Bradlee, Susan Eggers, and Robert Henry. The effect on risc performance of register set size and structure versus code generation strategy. *Proceedings of the 18th Intl. Symp. on Computer Architecture*, pages 330–339, 1991.

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

[8] John Edmondson and Paul Rubinfeld. An overview of the 21164 Alpha AXP Microprocessor. *In the proceedings of Hot Chips VI*, August 1994.

[9] Anant Agrawal. Utrasparc: A 64-bit, high-performance SPARC processor. *In the proceedings of MicroProcessor Forum*, October 1994.

[10] J.S. Liptay. Design of the IBM Enterprise System/9000 high-end processor. *IBM Journal of Research and Development*, 36(4):713–731, July 1992.

[11] Harry Dwyer and H. C. Torng. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, 1992.

[12] Scott McFarling. Combining branch predictors. *DEC WRL Technical Note TN-36*, 1993.

[13] Keith I. Farkas and Norman P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st Intl. Symp. on Computer Architecture*, pages 211–222, 1994.

[14] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages*, March 1994.

[15] Steven J. E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Equipment Corporation Western Research Lab, July 1994.

[16] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Proceedings of the 22st Intl. Symp. on Computer Architecture*, pages 414–425, 1995.