

Reducing Register Ports Using Delayed Write-Back Queues And Operand Pre-Fetch

Nam Sung Kim and Trevor Mudge
Advanced Computer Architecture Lab
The University of Michigan
1301 Beal Ave. Ann Arbor, MI 48109-2122
{kimns, tnm}@eecs.umich.edu

ABSTRACT

In high-performance wide-issue microprocessors the access time, energy and area of the register file are often critical to overall performance. This is because these parameters grow superlinearly as read and write ports are added to support wide-issue. This paper presents techniques to reduce the number of ports of a register file intended for a wide-issue microprocessor without noticeably impacting its IPC. Our results show that it is possible to replace the 16 read/8 write port file of an eight-issue processor with an 8 read/8 write port file so that the impact on IPC is insignificant. This is accomplished with the addition of some small auxiliary memory structures. Furthermore, the access time of the smaller file plus the auxiliary structures is such that if it were the critical path a 45-50% increase in clock speed would be possible. Finally, there is an energy per access savings of about 20% and an area savings of 40%, which has the potential for further savings by shortening global interconnect in the layout. An extension to the scheme that reduces the number of write ports from 8 to 6 is also presented. It suffers modest penalty in terms of IPC, but shows further reduction in energy and area. Depending on implementation characteristics it could yield a further increase in performance.

Categories and Subject Descriptors: B.0 [Hardware]: General; C.1.1 [Computer Systems Organization]: Processor Architecture — Pipeline Processors

General Terms: Design, Performance, Measurement

Additional Keywords and Phrases: Out-of-order Processor, Register File, Write Queue, Low Power, Instruction Level Parallelism

1. INTRODUCTION

In high performance wide-issue microprocessors the register file often plays a critical role in determining the cycles time, directly through its access time and indirectly through its size. Furthermore it accounts for a significant fraction of the processor core's power consumption. These register files need to be large to support multiple in-flight instructions and

multiported to avoid stalling instruction issue. In the Alpha 21464, the register file design was several times larger than the 64 KB primary caches [12] and was split to reduce cycle time impact. Both large size and high numbers of ports result in slow access and high energy dissipation.

Figure 1 shows the effects of size and number of ports on access time, energy, and area for 128-, and 256-entry register files having various combinations of read and write ports. The numbers were calculated using CACTI 3.0 [13] assuming a 0.18 μ m technology. We modified CACTI so that it can estimate the access time, energy, and area of small memory structures such as a multiported register files, which do not require the tags found in cache memories — CACTI was originally meant for caches. We considered four configurations of register file: one with 16-read and 8-write ports; one with 12-read and 6-write ports; one with 8-read and 4-write ports; and one with 4-read and 2-write ports. These are intended to support 8-, 6- 4- and 2-issue machines respectively. The values are normalized against a 256-entry register file with 16-read and 8-write ports. Figure 1 illustrates quite dramatically the penalty paid in access time, energy, and area as the number of ports is increased.

In this work, we propose two techniques to reduce the number of register ports without impacting performance. These techniques rely on small auxiliary memory structures called a *Delayed Write-back Queue (DWQ)*, an *Operand Pre-fetch Buffer (OPB)*, and an *Operand Pre-fetch Request Queue (OPRQ)*. The DWQ is employed to reduce the number of read and write ports. The other two, the OPB and the OPRQ, are employed to reduce the number of read ports. We will show that the use of all three structures allows fewer register file ports, resulting in faster, smaller, lower power files, without significantly reducing the IPC.

The DWQ provides a source of operands recently produced from the function units. It can be implemented using a small circular FIFO queue and avoids the need to access the large register file for these recent operands. The DWQ reduces the peak need for read ports. It also reduced the need for write ports. The OPB and OPRQ pre-fetch operands in the case when an instruction has one operand ready but is waiting on the second in the instruction queue. Our simulations show this is a common occurrence and allows us to schedule the reads so that the peak need for read ports is reduced. The use of OPB/OPRQ essentially distributes read port accesses over several cycles. The combination of using a DWQ and OPB/OPRQ reduces read and write port demand more than the sum of their separate effects without impacting IPC noticeably.

An important difference between most previous research and ours is that we focus on reducing the number of register ports

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'03, June 23-26, 2003, San Francisco, California, USA.
Copyright 2003 ACM 1-58113-733-8/03/0006...\$5.00.

Effect of size and number of register ports on access time, power, and area.

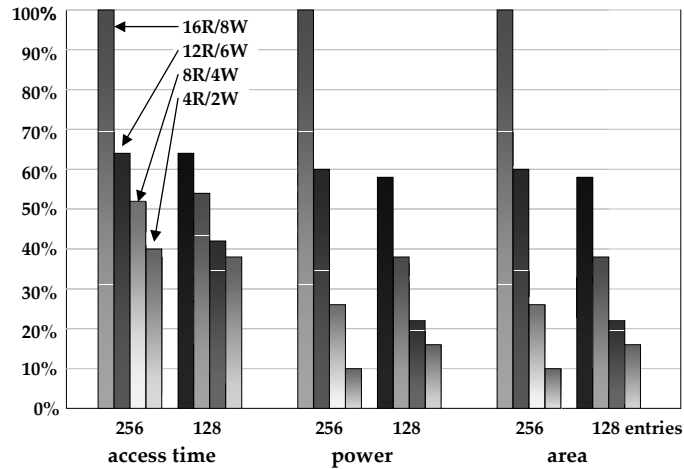


Figure 1. Shows the effect of size and number of ports on access time, power and area. The figure shows register files with 128- and 256-entries and 16-read/8-write, or 12-read/6-write, or 8-read/4-write, or 4-read/2-write ports.

rather than on reducing the number of registers. The hierarchical register organizations of [1][2][5][14] are examples of reducing the number of registers. Unlike these approaches ours avoids miss penalties associated with managing the hierarchy. In one case, [14], this management is done by the compiler. Our approach is transparent to the software.

Prior work that has proposed methods to reduce the access time/size/power of register files often requires substantial changes to the pipeline. Examples include the need to search an active list every cycle [1], or storing register values in the instruction queue while maintaining coherence in register caches [2]. Such changes have the potential to create significant complications as noted in [11].

The register caching approach proposed in [2] maintains small register caches close to the functional units and reads register operands from one of four sources: 1) pre-read before the issue queue; 2) extended forwarding (bypass) logic, 3) register caches, and 4) the register file after a register cache miss. Because operands may be read before the issue stage it is necessary to store operands in the instruction queue. This results in an increase in the size of instruction queue entries because register values are substantially larger than the dependency tags stored in the conventional instruction queue. The additional area and wiring required for such an issue queue are the primary reasons why modern issue queues or instruction queues do not store register values. In fact, if we store the pre-read operands in the instruction queue, each instruction queue entry must have 128-bits of storage for the two 64-bit source operands. Many of these operand fields will remain unused. In our scheme, we avoid storing them in the instruction queue by instead storing them in the OPB which has far fewer entries (8~32) than a typical instruction queue (256~512). The work in [2] also introduces a forwarding buffer which can hold the recent results for 9 cycles. Again this requires a large memory structure — a 72-entry buffer for an 8-wide issue machine — which is comparable to the size of the

actual register file. Perhaps the biggest drawback for the forwarding buffer is the need for multiple access ports. Our proposal for a DWQ is similar but much smaller and it is aimed at reducing the number of ports rather than speeding up operand reads. Our experimental results show that we are able to limit the size of the DWQ to 24 entries without significant IPC loss.

In [11], Park et al. proposed two techniques to reduce the number of ports. One is to introduce an extra stage to determine whether to read the operands from the bypass or register file and thus potentially reduce the need for read ports. The other technique performs bypass prediction to select from the bypass rather than the register file — again to potentially save ports. Adding an extra stage increases branch mis-prediction penalties. Furthermore, the bypass prediction technique can mis-predict and require a pipeline stalls when there is no available read ports.

The next section of the paper provides the motivation for this work by showing some experimental statistics about the read/write port utilization. Section 3 and Section 4 describe our proposed techniques. Section 5 describes our experiment setup, estimates the access time, energy, and area impact of reducing the register file ports, and presents experiment results. Section 6 concludes the paper with some proposals for future direction for this research.

2. MOTIVATION

Our experiments using the SPEC2000 benchmarks confirm previous studies that show that wide-issue machines do not utilize the full read/write bandwidth of their register files all the time. In fact, the issue and write-back stages are often idle during several cycles due to data dependencies caused by the long latency operations such as instruction or data cache misses and floating point operations. However, it is often the case that wide-issue machines are designed to support the

worst case (e.g., 16 read ports and 8 write ports for an 8-wide issue machine) to maximize ILP and performance.

Figure 2 shows the distribution of register file *read port* utilization cycles at the issue stage in an 8-wide issue machine. In this experiment, we used both SPEC2000 INT and FP benchmarks and Simplescalar 3.0 [3] with an architectural configuration similar to an EV8 [6]. The detailed simulation specification is given in Section 5. To maximize read port utilization we used a perfect branch predictor as well. The average percentage of read port utilization cycles requiring the full 16 read ports is about 0.1%, and that of idle cycles not requiring any read port is around 54% when using both SPEC2000 INT and FP benchmarks. It is evident that we have a plenty of under-utilized cycles over which we can distribute the reading of operands from the register file and thus reduce the number of ports. For example, it is possible to pre-fetch into the instruction queue ready operands from the register file for those instructions waiting for a second operand. This pre-fetch can be scheduled during under-utilized cycles.

Figure 3 shows the distribution of register file *write port* utilization cycles at the write-back stage with the same architectural configuration used in the experiment for Figure 2. The average percentage of write port cycles requiring the full 8 write ports is about 9%, and that of idle cycles not requiring any write port is about 52% according to the experimental results for both SPEC2000 INT and FP benchmarks. It is again evident that we have plenty of under-utilized cycles to distribute the writing of results to the register file similar to the read port case. A common technique for smoothing out bursty behavior such as we see in the case of read and write port utilization is to use a queue structure. We will show it is possible to have fewer than 8 write ports without losing performance by introducing a simple circular FIFO queue between the register file write ports and function units.

Although the register file has fewer write ports, the queue will be able to complete writing the rest of queued results while the write-back stage is idle.

In both experiments, we used a perfect branch predictor, but the percentage of idle cycles at both issue and write-back stages increases if we use a real (imperfect) branch predictor, because any branch mis-predictions cause pipeline stalls, which opens up more opportunity to distribute reads from and writes to the register file.

3. THE DELAYED WRITE-BACK QUEUE

3.1 Queuing write-back to reduce read ports

Our experiments using SPEC2000 benchmarks indicated that most results from the function units are consumed by the instructions waiting in the instruction queue within a few cycles after they are produced. This is an observation that is supported by the work of a number of researchers. In such situations it is possible to avoid accessing the register file read ports and get the required data directly from the bypass paths [11] and in so doing reduce the read port bandwidth.

If we add a small memory structure, the delayed write-back queue, we can often access the write-back queue instead of accessing the register file, provided we have some way of knowing that the results are in the write-back queue. This, of course, means that many of the register file accesses can be circumvented, which allows a possible reduction of the number of read ports without losing any performance. Figure 4 illustrates the *delayed write-back queue (DWQ)* technique. The queue holds the results of instructions for next n cycles after write-back. We can provide the operands to the issue stage without accessing the register file by holding the write-back results for a few cycles. To hold the write-backs for 2 cycles,

Register file read port utilization.

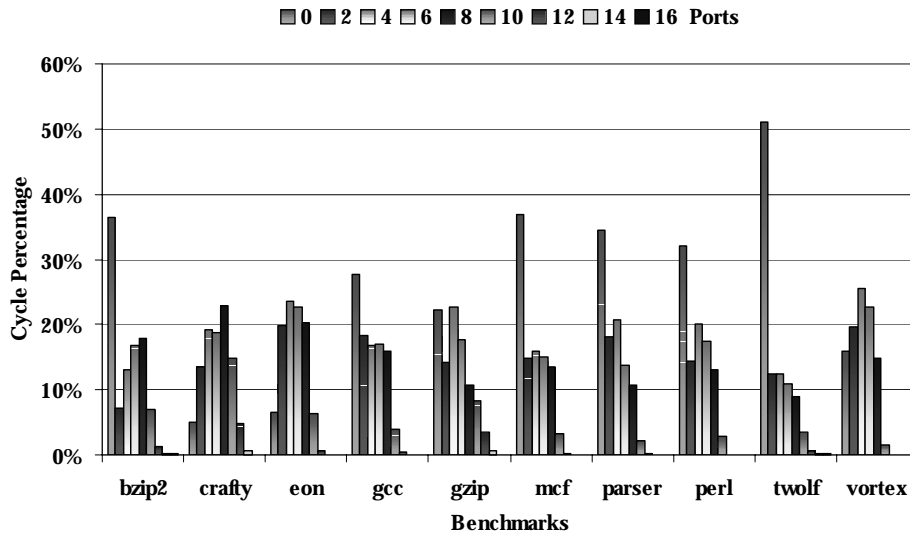


Figure 2. The distribution of register file read port utilization cycles in an 8-wide issue machine at the issue stage. The results for SPEC2000 INT benchmarks are shown in this graph.

Register file write port utilization.

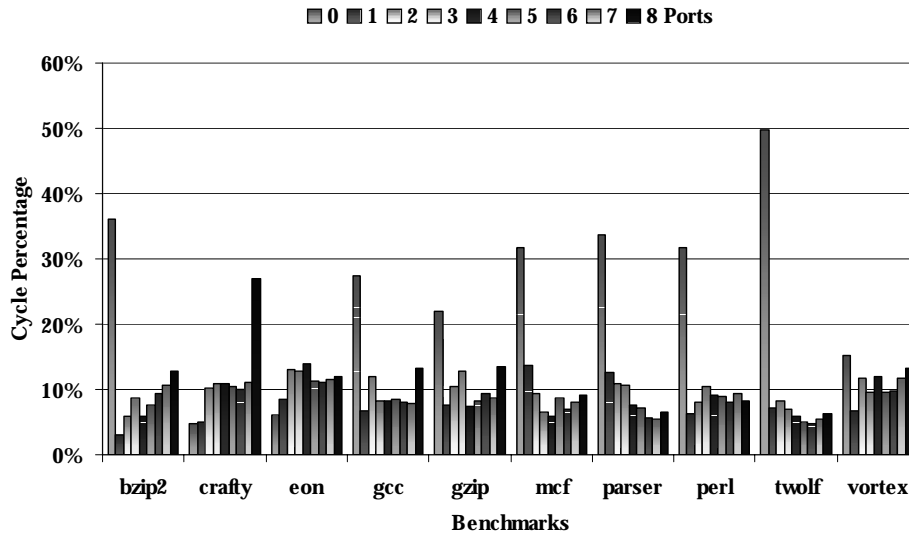


Figure 3. The distribution of register file write port utilization cycles in an 8-wide issue machine at the write-back stage. The results for SPEC2000 INT benchmarks are shown in this graph.

we need a 16-entry queue in the case of an 8-issue machine where, in the peak case, we assume that 8 results can be generated in a cycle. In addition, we write back the results both in the register file and the write-back queue concurrently to avoid consistency problems during renaming.

To determine whether the result operands are in the write-back queue and their location, we need a 2 bit counter to hold the number of cycles of delay. The count is decremented on every subsequent cycle after the value is initially loaded into the counter. The counter is simple to implement using 2 memory

elements and a mux and does not represent a major overhead. As soon as the instructions waiting for operands in the instruction queue are woken up the ready bits in the entries set the 2-bit counters. When we issue instructions we can check whether the counter value is zero or not, to determine which memory structure to access — the write-back queue or the register file as illustrated in Figure 4. In other words, the DWQ performs a very similar function to the *load and store buffer* for L1 data caches.

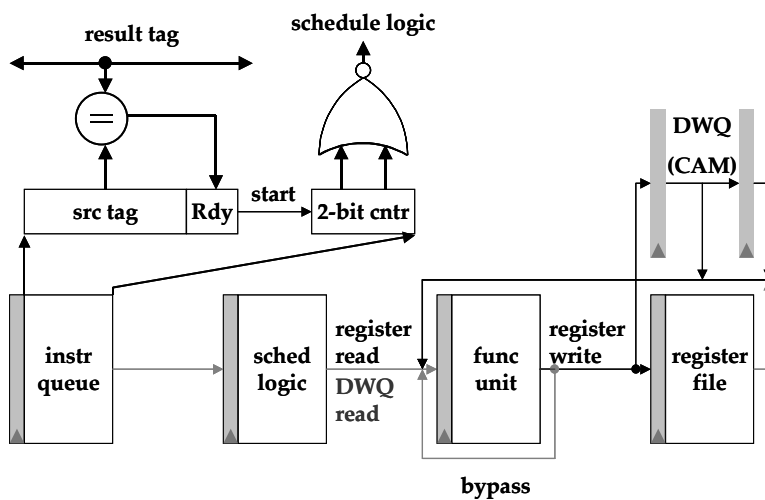


Figure 4. The block diagram of an implementation of the 2-cycle delayed write-back queue (DWQ) in the conventional out-of-order processor pipeline.

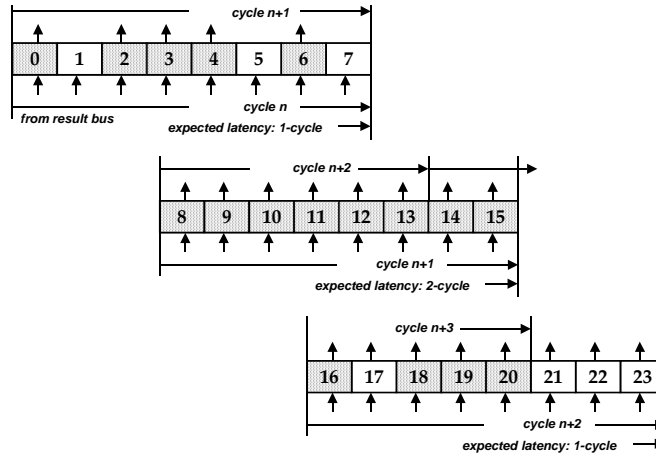


Figure 5. The timing diagram of 24-entry DWQ that accepts up to 8 writes from the functional units and outputs up to 6 writes to the register file.

3.2 Queuing write-back to reduce write ports

As we saw in Section 2, the percentage of the write-back cycles requiring the full write port bandwidth is very small. In the case that we simply reduce the number of write ports, the number of instruction issues are limited by the number of result buses, which are equal to the number of register write ports. However, we may reduce this potential limitation on performance by queuing the write-backs and delaying some when there are not enough write ports free.

To implement this technique, we may use the same DWQ presented in Section 3.1. But the actual write-backs to the register file will be completed a few cycles later depending on the number of queue entries, register write ports, and write-backs rather than after a fixed delay as was the case in Section 3.1. In addition, we need to broadcast write-back latencies for each write-back with the result tags to update the rename mapping table, and to load counter values for the ready operands in the instruction queue.

Figure 5 shows the timing diagram of a 24-entry DWQ that feeds a register file of 6-write ports. The DWQ can accept up to 8 writes, but only moves up to 6 results to the register file every cycle. We start moving the write-back results 1 cycle after writing the results to DWQ. During this 1 cycle delay, the DWQ identifies empty slots (blank boxes in Figure 5) and calculates 6 DWQ non-empty slots (shaded boxes) from which to write into the register file. The expected latency can be calculated by dividing the number of non-empty slots (entries) by the number of the write ports in the register file. The logic to do this requires 11 gates with a 4-gate delay in the case of a 6 write port register file. We obtained this design using the Synopsys Design Compiler™ and the Artisan™ TSMC 0.18μm technology standard cell library. The DWQ structure can be implemented with a simple 24-entry SRAM structure that has 8 write ports and 6 read ports — the read addresses correspond to the shaded boxes. We need 24 entries rather than 8, because of there may be some entries that stay in the DWQ for 2 cycles.

4. OPERAND PRE-FETCHING

4.1 The operand pre-fetch buffer

The experiments in [8] indicate that in 80% ~ 90% of the cases one of the operands for an instruction is ready when it enters the instruction queue after renaming. However, most of the instructions cannot be issued because the other operand is not ready and thus they wait in the instruction queue for this second operand. If an operand is ready we are able to identify the location (or address) of the physical register during renaming, which means that we may pre-fetch some of them while the instructions is in the instruction queue waiting for the second operand. This removes the potential for read-port congestion that would occur if we were to wait until both operands were ready before sending them to issue slots. If we can stagger the reading in the case where one becomes available before the other we can utilize the read ports more efficiently. In particular, there is a potential to reduce the number of register file read ports without impacting performance to an unacceptable degree. Our proposed technique to reduce the number of read ports by pre-fetching ready operands employs an *operand pre-fetch buffer (OPB)* to store the pre-fetched operands, and a status bit, the *pre-fetch flag (PF)*, in the instruction queue entry to specify whether the operand is in the OPB or the register file. In the issue stage we check the PF bit to determine where we should send the operand addresses to retrieve the operand.

Figure 6 shows a block diagram for an implementation of the OPB in a conventional out-of-order processor pipeline. When the *rename logic* dispatches instructions to the instruction queue, it feeds the physical register number of the ready operands to the *pre-fetch logic*, which requests register ports from the *resource scheduling or select logic*. The resource scheduling logic allocates a result bus, connected to a register file write port, to a functional unit. We need this allocation mechanism because there are usually more functional units than the number of result buses or register file write ports. In addition, it should also be extended to the register file read buses and ports because there might be more register read bus and port requests than available resources. However, the same resource contention situation happens in assigning result buses, because there are more functional units than available

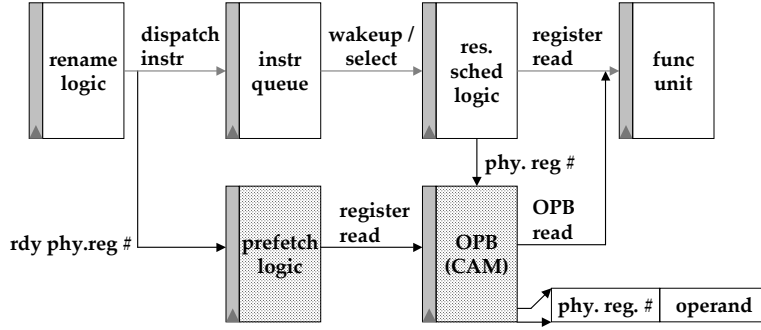


Figure 6. The block diagram of an implementation of the operand pre-fetch request buffer with OPB in the conventional out-of-order processor pipeline.

result buses. Therefore, we use the same mechanism for assigning the register file read buses.

If a register read bus and port is granted to the pre-fetch logic, it accesses the register file and store the pre-fetched operands to the OPB with the *source tags* which are just the physical register numbers. Each OPB entry contains the physical register number and an operand. It can be implemented with a fully associative memory, which is inexpensive because it requires only a small number of entries (16~32) with a small number of read and write ports (e.g., 8-read/8-write ports).

In addition, we need an additional pre-fetch flag bit associated with each source operand field in each instruction queue entry. When both operands for an instruction are ready and woken up, the instruction queue request function units, result buses and register ports be sent to the scheduling logic. If there are available functional units, result buses and register read ports, the selection logic issues the instruction and read operands from the register file or the OPB according to the pre-fetch bit status of each source tag.

4.2 The operand pre-fetch request queue

The operand pre-fetch technique proposed in Section 4.1 may pre-read the operands only if there are available ready operands in the physical register file, register read ports, and operand pre-fetch buffer space. All these conditions must be met in a cycle — the dispatching cycle of the instruction. This is not the common case, because we have more register port congestion when we reduce the number of register ports. In other words, it reduces the chances to pre-fetch operands by limiting the operand pre-fetches to the dispatch cycle. However, we can improve the chances of pre-fetching operands by adding an *operand pre-fetch request queue (OPRQ)*. When we do not have either any available register read ports or operand pre-fetch space at the dispatch cycle of the instruction, we send the physical register address of the ready operand to the operand pre-fetch request queue.

Figure 7 shows the block diagram of an implementation of the OPRQ combined with the OPB. In this technique, we push the ready physical register number and the instruction queue entry pointer of the dispatched instruction into the OPRQ at the dispatch cycle when we do not have available resources for pre-fetching. The OPRQ monitors availability of necessary resources with information given from the scheduling logic. Whenever the pre-fetch conditions are met, it requests operand

pre-fetches to the pre-fetch logic by sending the physical register number from the head entry of the OPRQ. As soon as the pre-fetch logic successfully finishes reading the requested operand, it updates the associated pre-fetch flags in the instruction queue with the instruction queue entry pointer from the OPRQ.

5. EXPERIMENTAL EVALUATION

5.1 Methodology

The evaluation methodology combined detailed processor simulation to obtain performance analysis and event counts, with analytical modeling for estimating access time, energy, and area for the register files having various combinations of read and write ports. The SimpleScalar toolset [3] was employed to model an out-of-order speculative processor with a two-level cache. The simulation parameters, listed in Table 1, roughly correspond to those of a present-day high-end microprocessor such as the Alpha 21464.

We replaced the register update unit found in the simulator with instruction queues and a reorder buffer. In addition, we modeled the congestion that results from a finite number of read and write ports (SimpleScalar assumes an infinite number of both), and added models for the DWQ, OPB, and OPRQ. Our benchmarks came from the SPEC2000 INT and FP benchmarks and were compiled with GCC 2.6.3 using “-O2” optimizations and statically linked library code. We ran 200 million instructions for each simulation after fast forwarding 20 billion to warm up the systems under study. This allowed us to complete the simulations in a reasonable time while avoiding results that might be biased by startup effects.

5.2 Impact on IPC of reducing read and write ports

(EQ 1) shows a *performance loss reduction* metric for evaluating results when using the proposed techniques. This metric shows relative performance improvement of the proposed techniques against the register files (RFs) of 8-read ports, and we also use the *straight performance loss* metric shown in (EQ 2) for each experimental result to show absolute performance degradation of register files with fewer ports against the register file of full 16-read and 8 write ports.

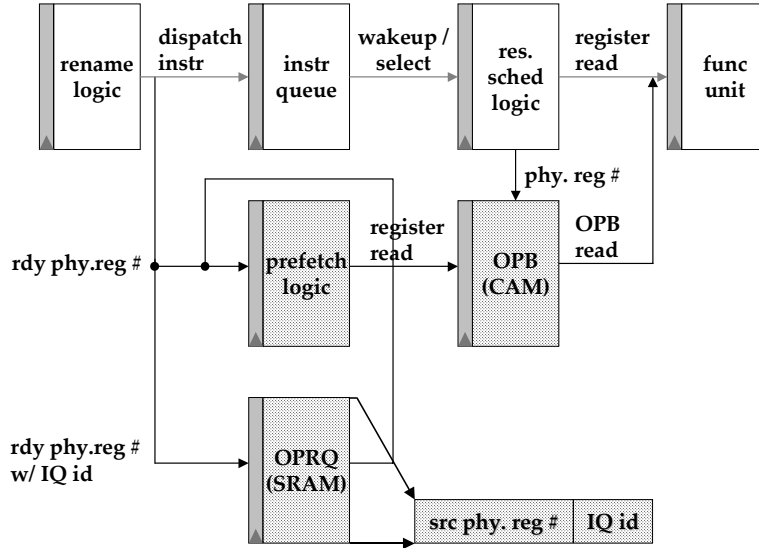


Figure 7. The block diagram of an implementation of the operand pre-fetch request buffer with OPB in the conventional out-of-order processor pipeline.

Performance loss reduction =

$$\frac{\text{IPC of 8-read RF w/ OPB} - \text{IPC of 8-read RF}}{\text{IPC of 16-read RF} - \text{IPC of 8-read RF}} \times 100 \quad (\text{EQ 1})$$

Straight performance loss =

$$\text{IPC of 16-read RF} - \text{IPC of 8-read RF (or w/ OPB)} \quad (\text{EQ 2})$$

Figure 8 shows the impact on IPC of reducing the number of read ports by half when including a delayed write queue (DWQ). In this experiment, 8- (1-cycle delay), 16- (2-cycle

delay), and 32-entry (4-cycle delay) DWQs were studied. The experimental results indicate that 8-, 16-, and 32-entry DWQs *reduce* the performance loss by about 78%, 85%, and 91% (black arrows at right in figure) compared to an 8-read port register file. The experiments also show just a straight 6%, 4%, and 2% performance loss against a 16-read port register file.

Figure 9 shows the impact on IPC of reducing the number of read ports by half with and without an operand pre-fetch buffer (OPB). In an 8-wide issue machine we compare a file with 16-read ports to one with 8-read ports. In the case with 8-read ports we show the IPCs for 8-read ports without an OPB, and then with 8, 16, and 32 OPBs. For each OPB size, we use

Table 1. Simulation Parameters

Parameters	Value
fetch / issue / decode / commit width	8 instructions each
fetch queue / speed	32 instructions / 1x
branch prediction	perfect branch predictor
ROB size	512 entry
instruction queue size	256 entry
LSQ size	64 entry
integer ALUs / multi-divs / memory ports	8 / 2 / 2
floating point ALUs / multi-divs	4 / 2
functional unit latencies	INT: mul 3, div 20, all others 1 FP: adder 2, mul 4, div 12, sqrt 24
memory bus width / latency	8 bytes / 80 and 8 cycles for the first and inter chunks
inst. / data TLBs	128 entry / 32 entry in each way, 8KB page size, fully-associative, LRU, 28-cycle latency
L1 caches	64KB, 4-way, 64B blocks, LRU, 1 cycle latency for the inst / 2 cycle for the data, write-back
L2 unified cache	4MB, 8-way, 128B line block, LRU, 12 cycle latency

50% reduction of the number of register file read ports with DWQ

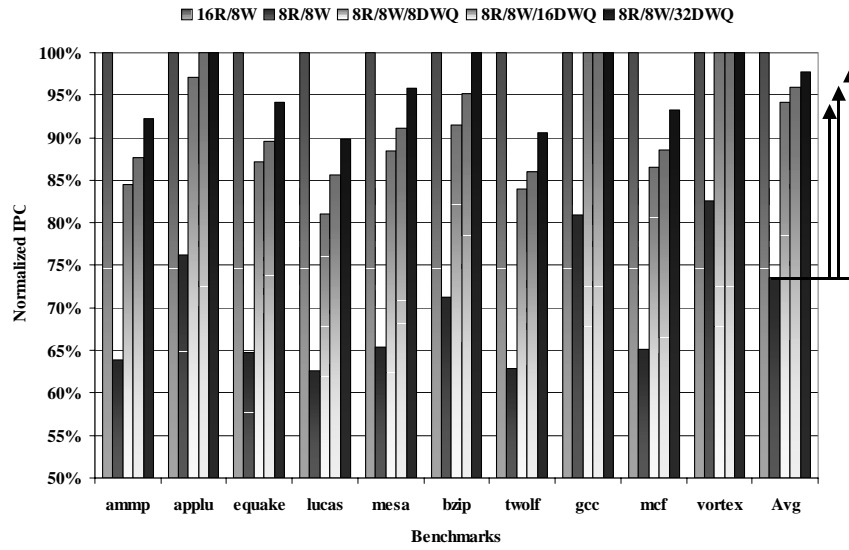


Figure 8. The impact on IPC of halving the number of read ports when using an delayed write queue.

In this experiment 8-, 16-, and 32- entry DWQs were used. A subset of benchmarks is shown in this graph, but the average number was obtained from the entire benchmark suite.

50% reduction of the number of register file read ports with OPB and OPRQ

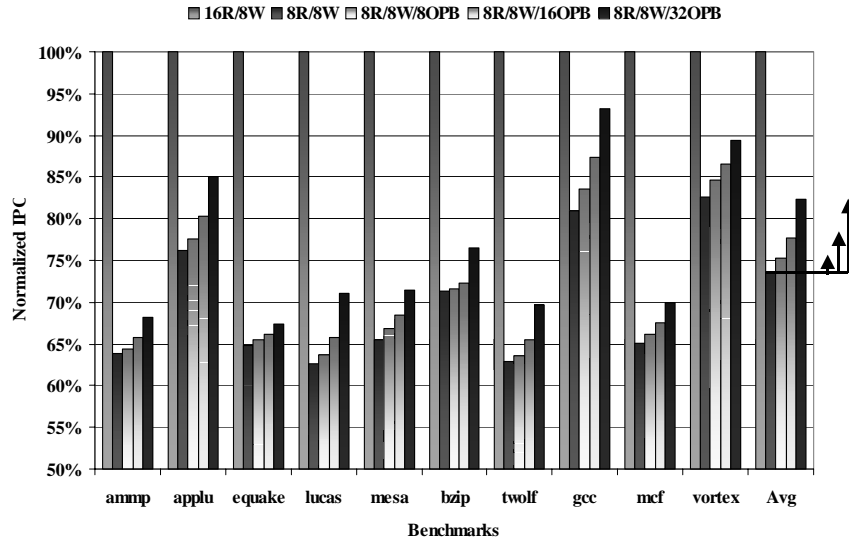


Figure 9. The impact on IPC of halving the number of read ports when using an operand pre-fetch buffer.

In this experiment 8-, 16-, and 32-entry OPBs were used with 16-, 32-, and 64-entry OPRQs respectively. The OPRQs were always twice the size as the OPBs. We assume an 8 issue machine. A subset of benchmarks are shown in this graph, but the average number was obtained from the entire benchmark suite.

50% reduction of the number of register file read ports with OPB, OPRQ and DWQ

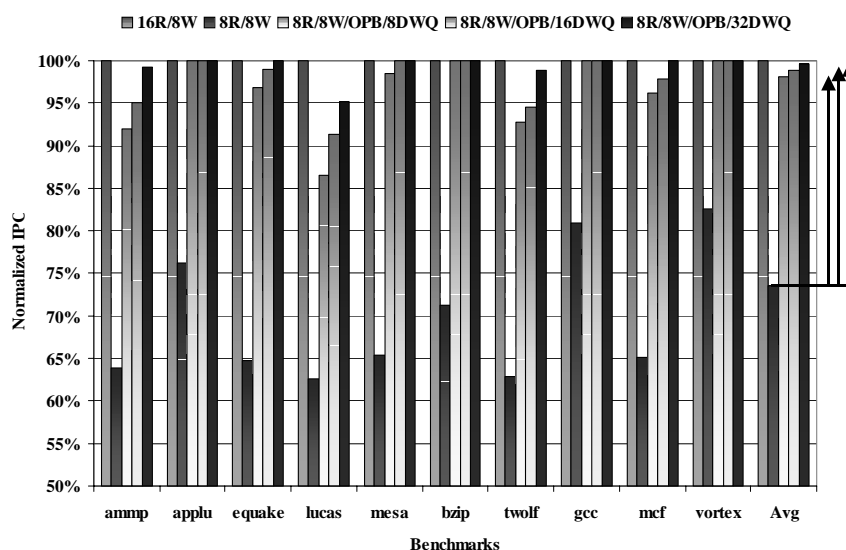


Figure 10. The impact on IPC of halving the number of read ports when using a DWQ combined with an OPB/OPRQ.

In this experiment a 16-entry OPB and 32-entry OPRQ, and 8-, 16-, and 32-entry DWQs were used. A subset of benchmarks was shown in this graph, and the average number was obtained from the entire benchmark.

OPRQs having twice as many entries as the OPB. The experimental results indicate that 8-, 16-, and 32-entry OPBs *reduce* the performance loss by about 6%, 15%, and 33% (black arrows at right in figure) compared to a system with just an 8-read port register file. The experiments also show a straight 25%, 22%, and 18% performance loss against 16-read port register file. There is some improvement to be had by using 32 OPBs rather than 16. However, considerations of access time, energy, and area overhead, mean we will limit ourselves to a 16-entry OPB with a 32-entry OPRQ for the rest of the experiments: it is necessary that the impact of the auxiliary structures is small.

The delayed write queue achieves quite a significant performance improvements compared to just using the OPBs with OPRQs. However, we can improve the performance by combining both OPB/OPRQ and DWQ techniques. The combination is more than additive for the following reason: The performance limit of the OPB technique in the reduced read ports situation was mainly caused by lack of the available read ports. The OPRQ can pre-fetch operands only if there exists an unused register read port. But we have already reduced the number of read ports. This increases the register read port traffic, and reduces the chances of accessing an unused read ports by the OPRQ. However, the DWQ techniques frees up read ports, because the DWQ read ports take place of the register file read ports. This gives the OPRQ more opportunities to pre-fetch operands through the available register file read ports.

In Figure 10, we present the impact on IPC when we employ both the OPB/OPRQ and the DWQ techniques. In this

experiment, we again used a 16-entry OPB with a 32-entry OPRQ. The experimental results indicate that 8-, 16-, and 32-entry DWQs when combined with the OPB/OPRQ *reduce* the performance loss by about 93%, 96%, and 99% (black arrows at right in figure) when compared to just an 8-read port register file. The experiments also show just a 2%, 1%, and ~0% performance loss against a 16-read port register file. Furthermore, this combined technique shows about a 15%, 11%, and 8% IPC improvements against the DWQ only technique, illustrating that the OPB technique is quite effective even when the number of DWQ entries is small.

Figure 11 shows the IPC impact of reducing write ports by 2 (a 25% reduction in the number of write ports). In this experiment, we used 16 read ports with 6 write ports and 24-, and 48-entry DWQs. We also show the IPC impact of halving the number of read ports with 6 write ports. The experimental results indicate that 24-, and 48-entry DWQs *reduce* the performance loss by 57% and 74% (black arrows at right in figure) compared to a 16-read/8-write port register file. The straight performance losses compared to a 16-read/8-write port file are 3% and 2%. The results also show that a 24-entry DWQ with the 16-entry OPB reduce the performance loss by 81% compared to an 8-read/6-write port register file and 4% reduction against 16-read/8-write port register file.

The reduced number of ports gives us a faster, smaller, and lower power register file, which, in turn, has the potential to improve the clock rate. If we assume that the register file is in the critical path we can compare the impact with another metric — instructions per second (IPS). The detailed access

25% reduction of the number of register file write ports with DWQ

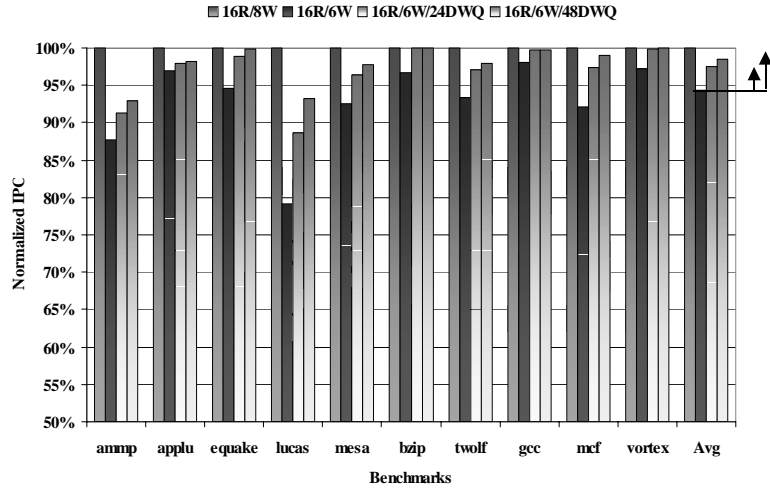


Figure 11. The IPC impact of reducing the write ports by 2 — a 25% reduction in write ports.

In this experiment a 24- and a 48-entry DWQ were used. A representative subset of benchmarks was shown in this graph, and the average number was obtained from the entire benchmark.

memory components	access time (ns)	energy (nJ)	area (cm ²)
512-entry 16-read/8-write register file	2.9	15.5	0.88
512-entry 8-read/8-write register file	1.8	7.4	0.40
512-entry 8-read/6-write register file	1.6	6.2	0.29
16-entry 8-read/8-write fully assoc. OPB	2.0	2.2	0.07
16-entry 8-read/8write fully assoc DWQ	2.0	2.2	0.07
16-entry 8-read/8write DWQ	0.8	2.1	0.05
24-entry 8-read/8write DWQ	0.9	2.3	0.08
32-entry 8-read/8-write OPRQ	0.7	0.8	0.01

Table 2. Access time, energy, and area estimation of memory components using our modified CACTI 3.0 with 0.18 μ m technology.

We modified CACTI to make it estimate access time, energy, and area of small memory structures such as register files, the DWQ, and the OPRQ, which do not require tag memory structures.

time, energy, and area calculations will be discussed in Section 5.3.

5.3 Impact on Energy and access time of reducing the number of ports

Table 2 shows access time, energy, and area estimation of the auxiliary memory structures for our microarchitectural modification of the register file required to support the reduced port register files. We used a modified version of CACTI 3.0 [13] and assumed 0.18 μ m technology. In particular, we modified CACTI so that it can estimate the access time, energy, and area of small memory structures such as a register file, the DWQ, and the OPRQ, which do not require tag memory. We assume that the register file has 512 registers with 16-read/8-write ports as our reference baseline [7]. We have two options for the DWQ. We can implement it with a

fully associative memory to avoid complicating the address broadcast bus. Or we can implement it with a regular memory structure with the broadcast bus. Our figures reflect the first choice. For the OPB, we assume that it has an 8-read/8-write fully associative memory with 16 entries. We also assume that each entry of the 16-entry OPRQ has 16 bits.

Table 3 shows the access time, energy, and area impact of reducing the number of read ports by half. All results are normalized against those of 512-entry register file having 16-read and 8-write ports. These results show that we can improve access time, energy, and area overhead with the proposed techniques. The configuration that impacts the performance least has 8-read and 8-write ports with a 16 entry OPB, a 16 entry DWQ, and a 32 entry OPRQ. It shows just a ~1% loss. If we examine the savings (see the last but one line of Table 3) we see that we are able to build a register file that has the performance of a 16-read and 8-write port file, but that permits

register file architectures	access time	energy	area
512-entry 16-read/8-write register file	100%	100%	100%
512-entry 8-read/8-write register file	62%	48%	45%
512-entry 8-read/6-write register file	55%	40%	34%
8-read/8-write with 16 OPB/32 OPRQ	68%	67%	54%
8-read/8-write with 16 OPB/32 OPRQ/16 DWQ	68%	81%	60%
8-read/6-write with 16 OPB/32 OPRQ/24 DWQ	68%	69%	50%

Table 3. Access time, energy, area impacts of reducing read and write ports

To obtain access time, we used the slowest access time among the memory structures for “with 16 OPB / 32 OPRQ” and “with 16 OPB / 32 OPRQ / 16 DWQ” cases in Table 3.

a ~47% increase in clock speed (1/access time savings), while reducing the energy per access by 20% and saving 40% in area. The area savings also has the potential to reduce the global interconnect between other components.

6. CONCLUSION AND FUTURE WORK

In this paper, we develop two techniques for reducing the number of register file ports without impacting IPC noticeably. The techniques are based on: 1) a delayed write-back queue; and 2) an operand pre-fetch technique comprised of an operand pre-fetch buffer and request queue. We described the implementations of both techniques. They rely on the addition of small auxiliary memory structures (DWQ, OPB, and OPRQ) to reschedule accesses to the register file so that the maximum number of ports is rarely needed. These structures further reduce the need for ports by supplying recently written register values directly to the processor pipelines.

There are several follow-up pieces of research that can be done. First, the effect of the techniques on timing, and hence instructions per second, could be made by including more details about technology. Second, the effect of using real branch predictors could be studied. Our expectation is that using an imperfect branch predictor reduces the pressure on register ports, because of the bubbles introduced by the mispredictions. Our proposal may allow one to exploit this to further reduce ports. There would also be more chances to pre-fetch operands if we have less use of the register ports. Third, one could use our delayed write back and operand pre-fetch techniques to improve performance for register files that require multicycle accesses. The delayed write-back queue and operand pre-fetch buffer are small memory structures that can be accessed in a single cycle, which means that multiple-cycle register file accesses can be replaced with accesses to a fast single-cycle delayed write-back queue or operand pre-fetch buffer. Running the register file slowly may allow more savings in energy and size. Unlike the hierarchical register file approach, such a solution does not have any coherence problems.

References

- [1] Balasubramonian, R., et al. Reducing the complexity of the register file in dynamic superscalar processors. *Proc. of the 34th Int. Symposium on Microarchitecture (MICRO 34)*, Dec. 2001.
- [2] Borch, E., et al. Loose loops sink chips. *Proc. of the 8th Int. Symposium on High Performance Computer Architecture*, Feb. 2002.
- [3] Burger, D., and T. Austin. The SimpleScalar Toolset, Version 2.0. *Tech. Rept. TR-97-1342*, Univ. of Wisconsin-Madison, June 1997.
- [4] Compaq Computer Corporation. Alpha 21264 microprocessor hardware reference manual. July 1999.
- [5] Cruz, K., et al. Multiple-banked register file architectures. *Proc. of the Int. Symposium on Computer Architecture*, Jun. 2000.
- [6] Diefendorff, K.,. Compaq chooses SMT for Alpha. *Microprocessor Report*, Dec. 1999.
- [7] Emer, J.,. EV8: The post-ultimate Alpha. *Keynote at Int. Conference on Parallel Architecture and Compilation Techniques*, Sep. 2001.
- [8] Ernst, D., et al. Efficient dynamic scheduling through tag elimination. *Proc of the Int. Symposium on Computer Architecture*, May 2002.
- [9] Farkas, K., et al. Register file design considerations in dynamically scheduled processors. *Proc. of the 2nd Int. Symposium on High Performance Computer Architecture*, Jan. 1996.
- [10] Gonzalez, A., et al. Virtual-physical registers. *Proc. of the 4th Int. Symposium on High Performance Computer Architecture*, Feb. 1998.
- [11] Park, I., et. al. Reducing register ports for higher speed and lower energy. *Proc. of the 35th Int. Symposium on Microarchitecture (MICRO 35)*, Nov. 2002.
- [12] Preston, R., et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. *ISSCC Digest and Visuals Supplements*, Feb. 2002.
- [13] Shivakumar, P., et al. An integrated cache timing, power, and area model. *WRL Research Report*, Feb. 2002.
- [14] Zalamea, J., et al. Two-level hierarchical register file organization for VLIW processors. *Proc. of the 33th Int. Symposium on Microarchitecture (MICRO 33)*, Dec. 2000.