The Microarchitecture of a Low Power Register File

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

The access time, energy and area of the register file are often critical to overall performance in wide-issue microprocessors, because these terms grow superlinearly with the number of read and write ports that are required to support wide-issue. This paper presents two techniques to reduce the number of ports of a register file intended for a wide-issue microprocessor without hardly any impact on IPC. Our results show that it is possible to replace a register file with 16 read and 8 write ports, intended for an eight-issue processor, with a register file with just 8 read and 8 write ports so that the impact on IPC is a few percent. This is accomplished with the addition of several small auxiliary memory structures — a “delayed write-back queue” and a “operand prefetch buffer.” We examine several configurations employing these structures separately and in combination. In the case of just the delayed write-back queue, we show an energy per access savings of about 40% and an area savings of 40%. This incurs a performance loss of just 4%. The area savings in turn has the potential for further savings by shortening global interconnect in the layout. We also show that the performance loss can be almost eliminated if both techniques are used in combination, although some area and power savings is lost.

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

General Terms: Design, Performance, Measurement

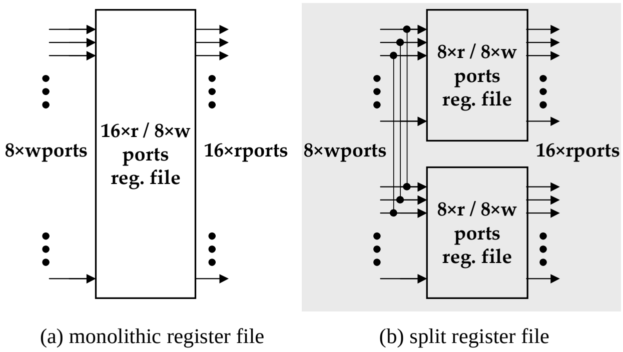
Additional Keywords and Phrases: Out-of-order Processor, Register File, Write Queue, Low Power, Instruction Level Parallelism

1. INTRODUCTION

The access time and size of register files in wide-issue processors often play a critical role in determining cycle time. This is because such files need to be large to support multiple in-flight instructions and multiported to avoid stalling the wide-issue. These factors also mean they contribute significantly to the processor’s power consumption. In the Alpha 21464, the 512-entry 16-read and 8-write (16-r/8-w) ports register file consumed more power and was larger than the 64 KB primary caches. To reduce the cycle time impact, it was implemented as two 8-r/8-w split register files [9], see Figure 1. Figure 1-(a) shows the 16-r/8-w file implemented directly as a monolithic structure. Figure 1-(b) shows it implemented as the two 8-r/8-w register files. The monolithic register file design is slow because each memory cell in the register file has to drive a large number of bit-lines. In contrast, the split register file is fast, but duplicates the contents of the register file in two memory arrays, resulting in higher power consumption and greater die area. This trade-off has been preferred in recent high-performance processor designs. We will assume the split architecture as our base-line. In either case, the high number of both register entries and ports result in high energy dissipation, slow access time and large area.

In experiments reported in a previous paper we showed 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 [7]. In such situations it is possible to avoid accessing the register file read ports by getting the required data directly from the bypass paths [8] and in so doing reduce the read port bandwidth. These results were obtained using the SPEC2000 INT and FP benchmarks and Simplescalar 3.0 [3] with an architectural configuration similar to an EV8 [5]. This is an observation that is supported by the work of a number of researchers [2][8]. We further observed that 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%. 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

Figure 1. Examples of 16 read- and 8 write-port register file implementation.



(a) monolithic register file

(b) split register file

ISLPED’03, August 25-27, 2003, Seoul, Korea.
Copyright 2003 ACM 1-58113-682-X/03/0008…$5.00.

further 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.

In this work, we re-evaluate 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)*. Both the DWQ and OPB with OPRQ are employed to reduce the number of read ports. We will show that their use allows fewer register file ports, resulting in a lower power, faster, smaller register files and 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 with limited associativity and avoids the need to access the large register file for these recent operands. The DWQ reduces the peak need for read 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 that the OPB/OPRQ combination are only marginally effective, even when coupled with a DWQ.

This work derives from earlier work reported in [7]. It differs in two respects: 1) We focus on reducing the number of read ports rather than read and write ports because current practice in register file design uses a differential bit-line structure — a shared read/write port similar to cache memories. This is a response to the need for bigger register files. 2) We use as our baseline the split register file architecture of Figure [1]. These split structures are now a common way to support a large number of read ports as in the Alpha 21464. Hence, we will be able to reduce the power and area by half if we reduce the number of register read ports by half.

An important difference between most previous research and ours is that we focus on reducing the number of register ports rather than on reducing the number of registers. The hierarchical register organizations of [1][2][4][12] are examples. Unlike these approaches ours avoids miss penalties associated with managing the hierarchy. In one case, [12], 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 [8]. In [11], they partitioned the register file into smaller sub-banks containing fewer read and write ports. This approach requires an additional pipeline stage to arbitrate the register file ports, which results in an additional performance penalty common to other multi-banked register file approaches [1][4][8].

The remainder of the paper begins by describing the proposed techniques in Section 2 and Section 3. Section 4 describes our experiment setup, estimates the access time, energy, and area impact of reducing the register file ports, and presents experiment results. Section 5 concludes the paper with some proposals for future direction for this research.

2. THE DELAYED WRITE-BACK QUEUE

As we mentioned in Section 1, most results from the function units are consumed by the instructions waiting in the instruction queue within a few cycles after they are produced. If we add a small memory structure — our proposed DWQ — in many cases we can 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 results in a further possible reduction of the number of read ports without losing any performance.

Figure 2 shows how the DWQ fits into the stages usually found in a superscalar wide-issue processor. The results produced from the functional units are written-back to both the register file and DWQ. The FIFO-type DWQ holds the results for next *n* cycles — 2 cycles in 2 after write-back. To hold the write-backs for 2 cycles, 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. The queue has two entries capable of holding 8 results. If the results are reused in two cycles or less they can be retrieved from the DWQ without accessing the register file. As mentioned above, we write back the results both to the DWQ and the register file to prevent register state inconsistency in the case a branch mis-prediction or an exception. Should either occur the contents of the DWQ will be flushed.

To determine whether the result operands are in the write-back queue and their location, a 2 bit counter is needed. 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 significant overhead. As soon as the instructions waiting for the operand 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 2. In other words, the DWQ performs a very similar function to the *load and store buffer* for L1 data caches.

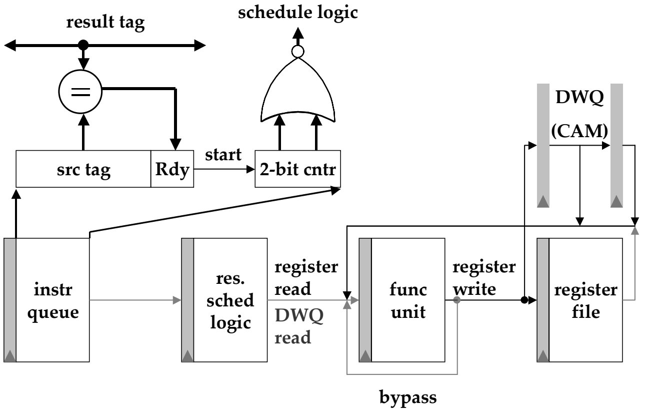


Figure 2. The block diagram of an implementation of the 2-cycle delayed write-back queue (DWQ) in the conventional out-of-order processor pipeline.

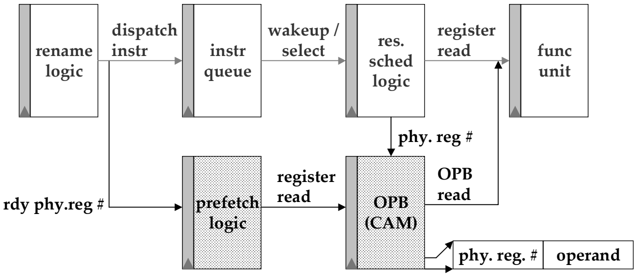


Figure 3. The block diagram of an implementation of the operand pre-fetch request buffer with OPB in the conventional out-of-order processor pipeline.

To implement the DWQ, we choose to use a 8-read/8-write port fully associative memory. Because the number of entries is small (16) the associativity is not a serious penalty, and it makes other aspects of the implementation simpler and easier. We will discuss the energy, access time, and area overhead caused by the DWQ in Section 4.3. We can implement the DWQ with a plain SRAM memory supporting FIFO functionality, but this requires 3 more broadcast network interconnects per a write-back between the function unit and the instruction queue, as well as 3 more bits per an instruction queue entry.

3. OPERAND PRE-FETCHING

3.1 The operand pre-fetch buffer

The experiments in [6] 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 them cannot be issued because the other operand is not ready and thus they wait in the instruction queue for this second operand to become available. 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 perhaps 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 where the operand is (OPB or register file) if there are any available read ports not used by the issue stage when we place an instruction into the instruction queue. In the issue stage we check the PF bit to determine where we should send the operand addresses to retrieve the operand.

Figure 3 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 to 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 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 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 remains inexpensive if it requires only a small number of entries (16~32) with a small number of read and write ports (e.g., 8-r/8-w 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.

3.2 The operand pre-fetch request queue

The operand pre-fetch technique proposed in Section 3.1 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 4 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.

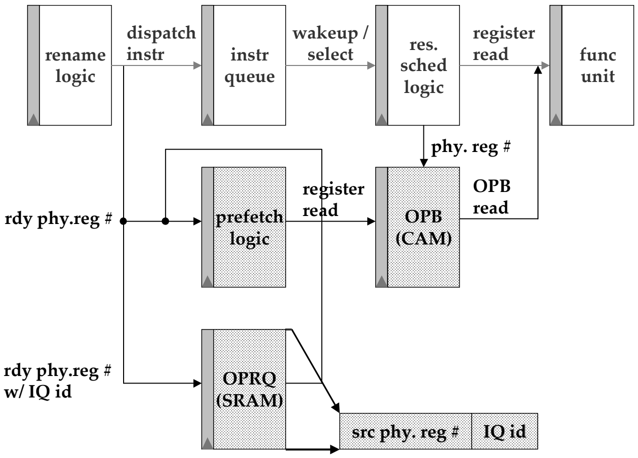


Figure 4. The block diagram of an implementation of the operand pre-fetch request buffer with OPB in the conventional out-of-order processor pipeline.

4. EXPERIMENTAL EVALUATION

4.1 Methodology

Our evaluation methodology combines detailed processor simulation to obtain performance analysis and event counts, with analytical modeling for estimating access time, energy, and area for the register files with various combinations of read and write ports. The SimpleScalar toolset [3] is 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 [5].

The register update unit employed in the standard version of SimpleScalar was replaced with instruction queues and a reorder buffer. In addition, we modeled the conflicts that results from having a finite number of read and write ports (SimpleScalar assumes an infinite number of both). Finally, we 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.

4.2 Impact on IPC of reducing the ports

Equation (1) shows the reduction in loss of performance *(LossReduction)* when using the proposed techniques. This metric shows relative performance improvement of the proposed techniques against register files with 8-read ports. We also show the straight performance loss (*PerfLoss*) metric of (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.

$$LossReduction = \frac{IPC_{\text{8-read reg file w/ DWQ}} - IPC_{\text{8-read reg file}}}{IPC_{\text{16-read reg file}} - IPC_{\text{8-read reg file}}} \quad (1)$$

able 1. Simulation Parameters

Parameters	Values
fetch queue / speed	32 instructions / 1x
ROB size	512 entry
IQ size	256 entry
LSQ size	64 entry
INT ALU / Mul-Div	8 / 2/ 2
FP ALUs / Mult-Div	4 / 2
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

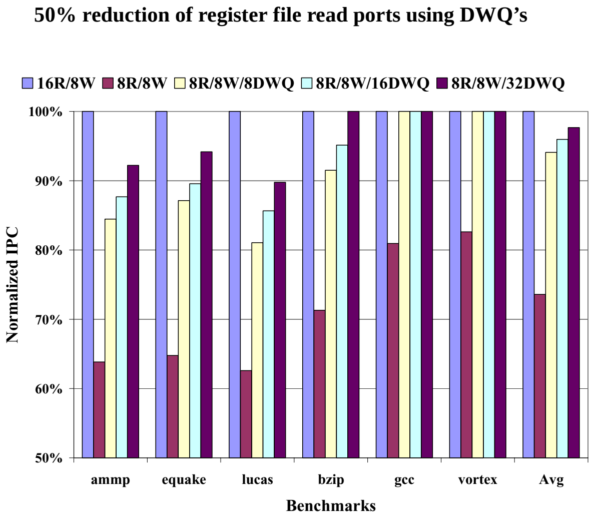


Figure 5. The impact on IPC of halving the number of read ports when using DWQ’s.

$$PerfLoss = IPC_{\text{16-read reg file}} - IPC_{\text{8-read reg file (or w/ DWQ)}} \quad (2)$$

Figure 5 shows the impact on IPC of reducing the number of read ports by half when a DWQ is included. In this experiment, 8- (1-deep FIFO), 16- (2-deep FIFO), and 32-entry (4-deep FIFO) DWQs were studied. The experimental results indicate that 8-, 16-, and 32-entry DWQs *reduce* the performance loss by about 78%, 85%, and 91% 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. However, because the access time of the 32-entry 8-r/8-w fully-associative memory is slower than one of the 512-entry 8-r/8-w register file that we are trying to replace with

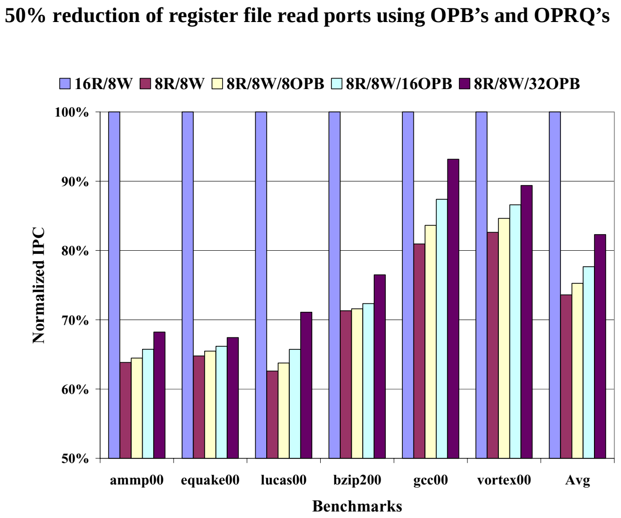

50% reduction of register file read ports using OPB's and OPRQ's

Figure 6. The impact on IPC of halving the number of read ports when using OPB's with OPRQ's.

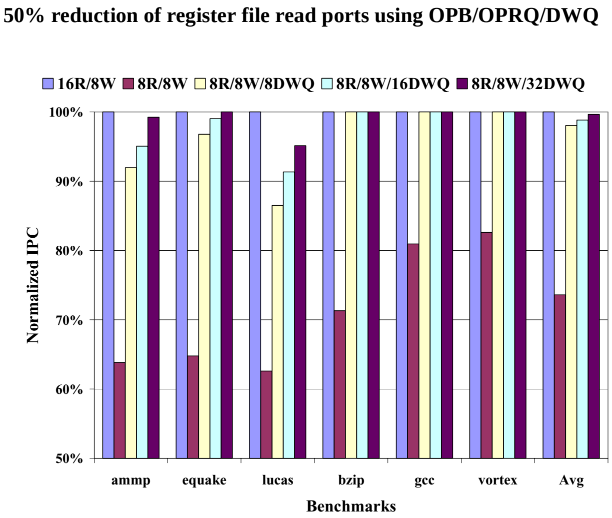

50% reduction of register file read ports using OPB/OPRQ/DWQ

Figure 7. The impact on IPC of halving the number of read ports when using OPB's with OPRQ's and DWQ's.

something smaller, we will not consider the 32-entry DWQ further.

Figure 6 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 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% 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.

If we compare the results in Figure 6 with those in Figure 5 (compare averages) we see that the DWQ achieves greater performance improvements than the OPBs with OPRQs. However, there is some opportunity to improve the performance by combining both the OPB/OPRQ and DWQ techniques. This can be seen from the results of Figure 7, where we show the impact on IPC when we employ both. 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% 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 can help the DWQ-only configuration.

Table 2. The average performance comparisons.

Memory	LossReduction	PerfLoss
8-r/8-w ports RF	-	27%
w/ 16 DWQ	85%	4%
w/ 16 OPB	15%	22%
w/ 16 DWQ and 16 OPB	96%	1%

Table 2 shows the summary of the average performance comparisons for the proposed techniques. We just show the case for a 16 entry DWQ. The combined DWQ and OPB with OPRQ technique virtually eliminates the performance loss of reducing register read-ports. The reduced number of ports gives us a lower power, and smaller register file, which, in turn, has the potential to improve the clock rate. The detailed access time, energy, and area calculations will be discussed in Section 4.3.

4.3 Impact on energy of reducing the ports

Table 3 shows energy, access time, and area estimation of 512-entry 16-r/8-w, and 8-read/8-write port register files and auxiliary memory structures for our microarchitectural modification of the register file required to support the reduced port register files. We constructed the 512-entry 16-r/8-w port register file using two 8-r/8-w port ones, as mentioned earlier (i.e., a split register file structure). This is our baseline. We used a modified version of CACTI 3.0 [10] and assumed 0.13µm technology. In particular, we modified CACTI so that it can estimate the energy, access time, and area of memory structures such as a register file, the DWQ, and the OPRQ, which do not require tag memory in cache memories. 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 OPRQ has 16 bits for the 16-entry OPRQ.

Table 3. Energy, access time and area of the register files and the individual auxiliary memory structures.

Memory	Energy (nJ)	Access time (ns)	Area (cm^2)
16-r/8-w ports RF	8.82	1.47	0.34
8-r/8-w ports RF	4.41	1.47	0.17
16 DWQ	1.07	1.43	0.04
16 OPB	1.07	1.43	0.04
32 OPRQ	0.37	0.59	0.01

Table 4 shows the energy, and area impact of reducing the number of read ports by half. All results are normalized against those of 2 split 512-entry register files having 8-read and 8-write ports. These results show that we can save 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 DWQ, a 16 entry OPB, and a 32 entry OPRQ. It shows just a ~1% loss. If we examine the savings (see the last line of Table 4) we see that we are able to build a register file that has the performance of a 16-read and 8-write port file while reducing the energy per access by 22% and saving 26% in area. The area savings also has the potential to reduce the global interconnect between other components.

If we are prepared to suffer a 4% loss in IPC the configuration that just employs a 16 entry (2-deep FIFO) gives nearly a 40% savings in energy and area. This could translate into significant savings in instructions per second, if the register file were on the critical path

5. CONCLUSION

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 multi-cycle 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.

Table 4. Normalized energy, area and performance of the register files with auxiliary memory structures.

Configuration	Energy	Area	PerfLoss
16-r / 8-w ports RF	100%	100%	-
8-r / 8-w ports RF	50%	50%	27%
w/ 16 DWQ	62%	61%	4%
w/ 16 OPB & 32 OPRA	66%	64%	22%
w/ 16 DWQ, 16 OPB, & 32 OPRQ	78%	74%	1%

Acknowledgement: This work was supported by an Intel Graduate Fellowship, by DARPA contract number F33615-00-C-1678, by SRC 2001-HJ-904, and by NSF Grant CCR-0073497.

6. REFERENCE

[1] R. Balasubramonian et al. Reducing the complexity of the register file in dynamic superscalar processors. *Proc. Ann. IEEE/ACM Symp. on Microarchitecture*, Dec. 2001.

[2] E. Borch et al. Loose loops sink chips. *Proc. of the 8th Int. Symp. on High Performance Computer Architecture*, Feb. 2002.

[3] D. Burger and T. Austin. *The SimpleScalar Toolset Version 2.0.* Tech. Rept. TR-97-1342, Univ. of Wisconsin-Madison, June 1997.

[4] K. Cruz et al. Multiple-banked register file architectures. *Proc. Int'l Symp. on Computer Architecture*. June 2000.

[5] J. Emer. EV8: The post-ultimate Alpha. *Keynote at PACT*, Sep. 2001.

[6] D. Ernst et al. Efficient dynamic scheduling through tag elimination. *Proc. of the 8th Int. Symp. on High Performance Computer Architecture*, May 2002.

[7] N. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand prefetch. *Proc. Int'l Conf. on Supercomputing*, June 2003.

[8] I. Park et. al. Reducing register ports for higher speed and lower energy. *Proc. Ann. IEEE/ACM Symp. on Microarchitecture*, Nov. 2002.

[9] R. Preston et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. *Proc. ISSCC Digest and Visuals Supplements*, Feb. 2002.

[10] P. Shivakumar et al. *An Integrated Cache Timing, Power, and Area Model.* WRL Research Report, Feb. 2002.

[11] J. Tseng and K. Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. *Proc. Int'l Symp. Computer Architecture.* June 2003, to appear.

[12] J. Zalamea et al. Two-level hierarchical register file organization for VLIW processors. *Proc. Ann. IEEE/ACM Symp. on Microarchitecture*, Dec. 2000.