Appears in 30th International Symposium on Computer Architecture (ISCA-30), San Diego, CA, June 2003 (corrected)

Banked Multiported Register Files for High-Frequency Superscalar Microprocessors

Jessica H. Tseng and Krste Asanović
MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139
{jhtseng,krste}@lcs.mit.edu

Abstract

Multiported register files are a critical component of high-performance superscalar microprocessors. Conventional multiported structures can consume significant power and die area. We examine the designs of banked multiported register files that employ multiple interleaved banks of fewer ported register cells to reduce power and area. Banked register files designs have been shown to provide sufficient bandwidth for a superscalar machine, but previous designs had complex control structures that would likely limit cycle time and add to design complexity. We develop a banked register file with much simpler and faster control logic while only slightly increasing the number of ports per bank. We present area, delay, and energy numbers extracted from layouts of the banked register file. For a four-issue superscalar processor, we show that we can reduce area by a factor of three, access time by 25%, and energy by 40%, while decreasing IPC by less than 5%.

1 Introduction

Multiported register files and bypass networks lie at the heart of a superscalar microprocessor core, providing buffered communication of register values between producer and consumer instructions. As issue widths increase, both the number of ports and the number of registers required increase, causing the area of a conventional multiported regfile to grow more than quadratically with issue width [22]. The trend towards simultaneous multithreading also increases register count as separate architectural registers are needed for each thread. For example, the proposed eight-issue Alpha 21464 design had a regfile that occupied over five times the area of the 64 KB primary data cache [14].

Many techniques have been proposed to reduce the area, energy, and delay of multiported register files. Some approaches split the microarchitecture into distributed clusters, each containing a subset of the register file and functional units [17, 12, 8, 11, 23, 16]. These schemes have the potential to scale to larger issue widths but require complex control logic to map instructions to clusters and to handle inter-cluster dependencies. Alternatively, other approaches retain a centralized microarchitecture, but divide the physical register file into interleaved banks with fewer ports per bank [20, 2, 13]. Provided the number of simultaneous accesses to any bank is less than the number of ports on each bank, the structure can provide the aggregate bandwidth needs of a superscalar machine with significantly reduced area compared to a fully multiported regfile. These earlier banked schemes, however, require complex control logic with pipeline stalls that would likely limit the cycle time of a high-frequency design.

In this paper, we present a banked multiported register file design together with a control scheme suitable for a deeply pipelined high-frequency superscalar processor. Our control scheme does not place any register bank arbitration in the critical wakeup-select loop but instead speculatively issues potentially conflicting instructions. If any conflicts are found after issue, a pipelined recovery scheme quickly repairs the issue window and reissues conflicting instructions. In contrast to previous work [20, 2, 13], all conflicts are detected and resolved in one pipeline stage so that no write buffering or pipeline stalls are required. An important optimization is to avoid competing for register read ports for operands that will be sourced by the bypass network. We describe how we can conservatively compute a set of operands that will be sourced by the bypass network using the wakeup logic without adding to pipeline latency, and while avoiding misprediction stalls caused by schemes that optimistically predict which values will be bypassed [13]. To maintain acceptable performance, our scheme requires more ports per bank than previous work. We show through detailed circuit layouts, however, that for small banks with small numbers of ports per bank, the overall regfile size varies little with the bank port count due to the dominance of the bank multiplexing overhead. For a four issue processor we can reduce register file size by approximately a factor of three while reducing IPC by under 5%. Regfile access time is reduced by 25% and energy consumption is reduced by around 40% compared to a unified multiported structure.

The paper is structured as follows. We first describe our scheme in detail in Section 2, including the pipeline structure and required control logic. We then present area, energy, and delay numbers from detailed circuit layouts in Section 3, and performance simulation results in Section 4. We discuss the relationship of our scheme to previous work in Section 5, then conclude in Section 6.

2 Banked Regfile Design

First we consider the circuit structure of a banked multiported register file then examine the pipeline control logic.

2.1 Register Bank Structure

Figure 1 shows one example of our register banking scheme for a four-issue processor. The regfile provides a total of eight global read ports and four global write ports using four interleaved register banks, each with two local read ports and two local write ports. Compared with a conventional multiported structure, each word of register storage has fewer ports and the storage cell size is reduced dramatically. But now additional multiplexing circuitry is required to connect the local port bitlines to the global port bitlines, and there is the possibility of bank conflicts when too many global ports attempt to read or write the same bank.

Each functional unit needs two global read ports, which we term the left and right ports, to execute instructions with two register source operands. We simplify the local-global port crossbar by connecting one local port on each bank to the global left operand busses, and the other to the global right operand busses. This allows any instruction to get both operands from the same bank in one cycle, but doesn't allow the use of the local left ports to fetch global right port operands. Apart from the reduction in mux circuitry, this restriction simplifies port arbitration logic by cutting in half the number of possible contendors for a local read port.

In contrast, the design presented in [2] employed banks with a single read port. The single read port bank must connect to all global ports, and hence requires the same complexity of local-global crossbar as a dual read-port design that connects each local port to half the global ports. As shown below, there is little area saving for the single read port design versus the split dual port design once the cost of the local-global crossbar is included. Moreover, the single read port bank requires considerably more complicated control logic to handle execution of an instruction that fetches both operands from the same bank and the arbitration logic to match instructions to register bank read port decoders is twice as deep.

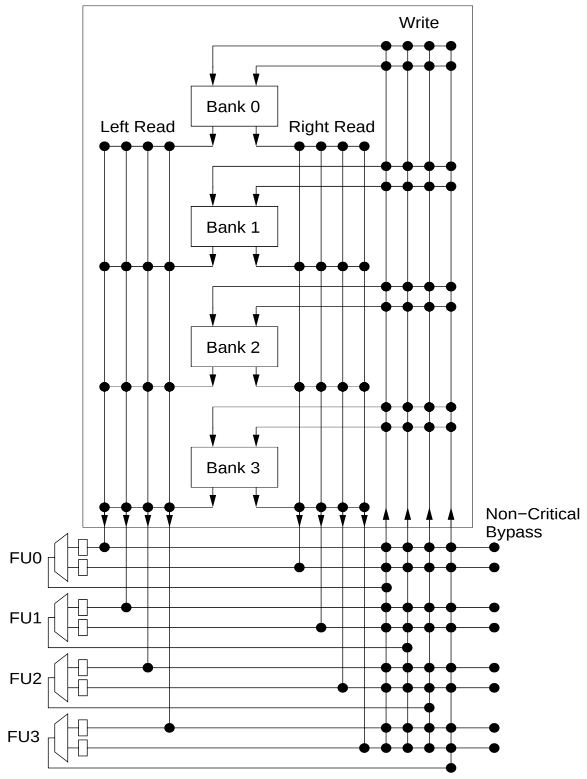

Figure 1. An eight read, four write port register file implemented using four two read, two write port banks.

The design shown has two local write ports per bank. Our results show that this is required to reduce the incidence of write bank conflicts to an acceptable level for our speculative issue control scheme. Again, as shown below, the area overhead of increasing the number of write ports from one to two is minimal as a percentage of total regfile area.

Figure 1 also shows a portion of the bypass network for single cycle latency functional units; multiple cycle units such as load units will require additional bypass paths. Register file writeback may require one or more cycles, in which case additional bypass logic is required for results that have completed but which are not yet available from the register file. These delayed bypass paths are not latency critical and can be supplied by an early stage mux that feeds into the final latency critical mux stage [9]. We rely on values sourced by the bypass network to reduce the required read port bandwidth, but to reduce control complexity, we only avoid contending for read ports for values bypassed from the immediately preceding cycle.

Another important mechanism to reduce port contention is read sharing. This allows multiple instructions to read the same physical register from the same read port [2]. Read sharing is implemented by allowing a single local read port

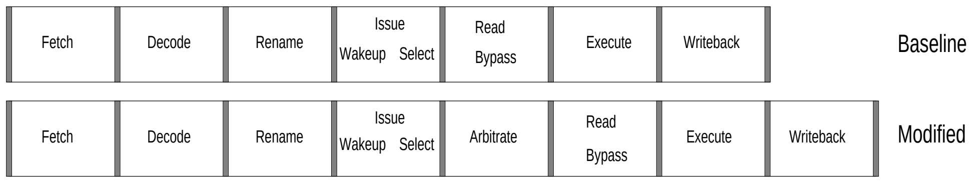

Figure 2. Pipeline structures of processor with unified register file and processor with multibanked register file. An additional cycle is added for multibanked register file for read port arbitration and muxing. Read bank and write bank conflicts are also detected in this cycle.

to drive multiple global read ports.

2.2 Control Logic

A banked multiport register file can provide sufficient bandwidth for a superscalar processor with less cost than a flat design. The main challenge is devising control logic that can handle the inevitable bank conflicts without compromising cycle time or adding excessive complexity.

Figure 2 shows the baseline processor pipeline design for the flat register file structure and the modified processor pipeline design for the banked register file structure. Instructions are decoded and renamed then placed into an instruction window. Instructions wait in the instruction window until both operands are available. The instruction window pipeline stage contains the critical wakeup-select loop [12], where the wakeup phase is used to update operand readiness and the select phase picks a subset of the ready instructions to issue. Once a single-cycle instruction is selected, its result tag is immediately broadcast to the instruction window in the next wakeup phase to allow back-to-back issue of dependent instructions, even though the selected instruction will not produce its result for several cycles.

The number of read ports required can be reduced significantly if operands that will be sourced from the bypass network do not also compete for access to the register file. Determining which operands will be bypassed would at first appear to require an additional sequential bypass check after issue but before port arbitration. To avoid this increase in pipeline latency, the check can be folded into the wakeup phase. Previous work has described an optimistic bypass hint scheme [13] where an extra hint bit is added to each operand of instructions waiting in the issue window. The hint bit is cleared if the operand was ready before the instruction entered the issue window. If the operand becomes ready while the instruction is in the window, the hint bit is set. When the instruction is selected, this operand will not contend for a read port as it is likely to be sourced from the bypass network. The disadvantage with this scheme is that it is only a prediction which can be incorrect, requiring several additional repair cycles for recovery.

We instead adopt a conservative bypass bit scheme, which is always correct. We store the bypass bit for each instruction window operand in a latch that is loaded with the result of the wakeup tag match on every cycle. If an instruction is selected in the same cycle that a tag match caused it to wake up, the bypass bit will be set indicating that the value is available from the bypass network. If the instruction is not selected for issue, the bypass bit latch will be cleared by the failing tag match on the next wakeup phase. The bypass bit is conservative because it is only set for values that will be ready in the cycle before this instruction executes. Where there are several pipeline stages feeding the bypass mux (e.g., when register file access takes multiple cycles) this scheme will still compete for read ports even though these operands will be sourced from later pipeline stage bypasses. In practice, we find this lost opportunity causes negligible performance impact. To reduce datapath complexity, a microarchitecture might not support bypass from every functional unit. In this case, the wakeup tag search can be modified to broadcast a signal indicating whether the operand can be bypassed. This value can then be latched into the bypass bit on a successful tag match.

Any register operand of an issued instruction which doesn't have the bypass bit set must contend for read ports. A conventional pipeline has a fixed mapping of issued instruction operands to register file ports, but with a banked scheme it is necessary to mux operand addresses into the available register file ports. By splitting the read ports into left and right sets, corresponding to the left and right register operands, we halve the size of the arbiters and muxes needed. An N-way superscalar needs only an N-way arbitration and mux for the left operands, and a parallel N-way arbitration and mux for the right operands. To further reduce port allocation delay, it is possible to move the register address decode function ahead of the arbitration. Rather than mux the encoded bank address into the decoders on each local bank which places bank address decode in series with arbitration, we can decode each operand address into a unary word select in parallel with arbitration, then mux the unary word selects into the bank word line drivers once

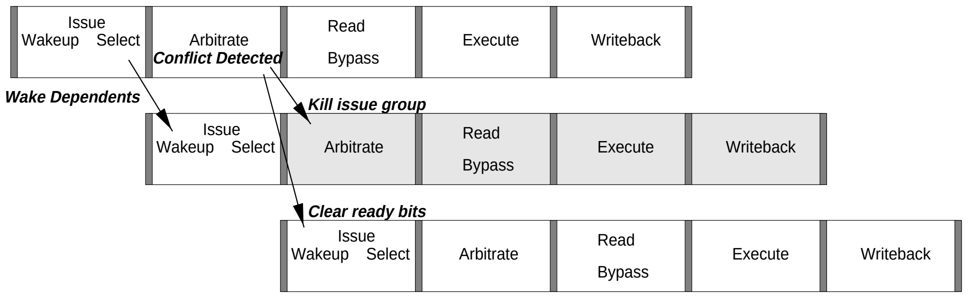

Figure 3. Pipeline diagram showing repair operation after conflicts are detected. The wakeup tag search path is used to clear ready bits of instructions that had a conflict causing them to be reissued two cycles later. Any intervening instruction issues are killed.

the arbitration result is known. In this way, only the N-way arbitration and mux are added to the pipeline latency. In Figure 2 and in our evaluation, we allocate an extra pipeline stage to this arbitration and mux, but we believe the actual penalty would be much lower in practice.

The arbitration stage also detects read bank conflicts, when too many read accesses try to access the left or right side of a single bank. The method by which the pipeline state is repaired after a conflict is shown in Figure 3. A second group of instructions following the ones that encounter a conflict will have been speculatively issued into the pipeline in parallel with detection of the conflict. This second group is killed along with the instructions in the first group that were not granted a read port. The wakeup phase that would have been used to broadcast the tags of the second group of instructions is now used to repair the issue window by broadcasting the tags and reseting the ready and issued bits for the destinations of the killed instructions in the first group. Issue will now resume correctly in the select phase of this pipeline stage. This approach adds only a mux into the tag broadcast path of the critical wakeup phase. Previous work has either placed additional arbitration logic in the select path to avoid conflicts or required that pipeline stages be stalled [20, 2, 13]. Both approaches complicate critical timing loops [3]. Stalling in particular is usually prohibited on high frequency pipeline designs due to the difficulty of generating and routing a global stall signal to a large number of pipeline registers. Using reissue to handle conflicts can potentially decrease the effective size of the instruction window by requiring that instructions remain in the window until all possible reissue conditions have passed. We assume the bank conflict replay time is subsumed within the longer replay time required to handle reissue on data cache load misses, so there is no change in effective window size.

A common cause of read bank conflicts is when multiple instructions in an issue group try to read the same physical register [2]. The read port arbiter can detect this sharing and remove the conflict by setting enable signals such that a local port drives multiple global ports. Our register file structure only allows sharing on either the left ports or the right ports, and requires a second local port if a physical register is read from both sides.

Write bank conflicts are also detected in the arbitration stage of the pipeline, with the same pipeline repair mechanism used to recover from conflicts. All instructions that pass the arbitration phase can write back with no conflicts. This approach avoids register bank write buffers [2] which increase the size of the bypass network and which require pipeline stalls when buffers fill. It is also much simpler than schemes that delay physical register allocation until writeback to avoid conflicts [13]. The main disadvantage of the approach proposed in this paper is that sufficient write ports must be provided such that write bank conflicts do not cause performance degradation. We found that increasing the number of write ports to two per bank removed most write bank conflicts, with only a small penalty in overall register file area.

3 Register File Layouts

We have completed layouts of various sized banked register files to determine their area, delay, and energy. These were laid out in a 0.25 μm CMOS process from TSMC. The storage cells are a standard six transistor SRAM design, with differential write ports and single-ended read ports.

Metal 1 is used for local bitlines within a bank and metal 2 for word lines. The local ports from each bank then connect to the global bitlines running over the cells in metal 3. Most previous work has assumed that a large conventional multiported register file would have each port on a storage cell connected directly to the global bitline. With more metal layers, it is desirable to employ a hierarchical bitline structure, where each port on a cell connects to a local bitline which in turn connects to the global bitline [1, 9].

64×32b, 8 read ports, 4 write ports				
Area	8r4w	2r2w	2r1w	1r1w
1 banks	100.00%	-	-	-
4 banks	110.95%	28.99%	24.29%	22.88%
8 banks	122.55%	37.06%	31.97%	30.19%
Packing	1	2	2	2

Table 1. Relative area of different multibanked regfile designs in comparison to the unified design. Packing is the number of local bit cells packed per global bit column.

On each access, only one local bitline is connected to the global bitline. The parasitic drain capacitances of the storage cells in other banks are not driven, reducing delay and energy dissipation. Another benefit is that signal-to-noise ratio improves in the presence of leakage currents from off cells [1]. Hierarchical bitlines will reduce the relative advantage of a multibanked design. To save area, we employ a single-ended global write bitline which is converted to a differential local bitline using a local inverter. To further save area, we pack two local storage cells into one global bit column where possible. This has the disadvantages that a 2:1 column mux is required which adds area and delay, and that twice as many local bitlines are discharged on each access increasing energy usage.

Table 1 shows the relative area of a variety of 64×32-bit 8 read ports and 4 write ports multibanked register file designs in comparison to a unified design and Figure 4 shows the detailed area breakdowns. Figure 5 provides a graphical comparison of the floorplan of a few representative register file designs.

For the designs with 8 read ports and 4 write ports per storage cell, moving to hierarchical bitlines adds area because of the interconnection overhead. An additional 11% area overhead is needed for 4 local bitlines (16 words per local bitline) and an additional 23% moving to 8 local bitlines (8 words per local bitline). Bank conflicts do not occur in these designs.

As the number of local ports per bank is reduced, area drops dramatically. Compared to the baseline design, the designs with four banks are around one quarter the size, and the designs with eight banks are around one third the size. Apart from the reduction in storage cell size, designs with smaller numbers of ports per bank have significantly less address decoder area than the highly multiported designs. Each bank has fewer decoders with narrower addresses. The design with four banks, each with one read port, can not sustain eight global read port accesses and relies on the bypass network to supply the missing read operands.

Both Figure 4 and Figure 5 also show that multiplexing

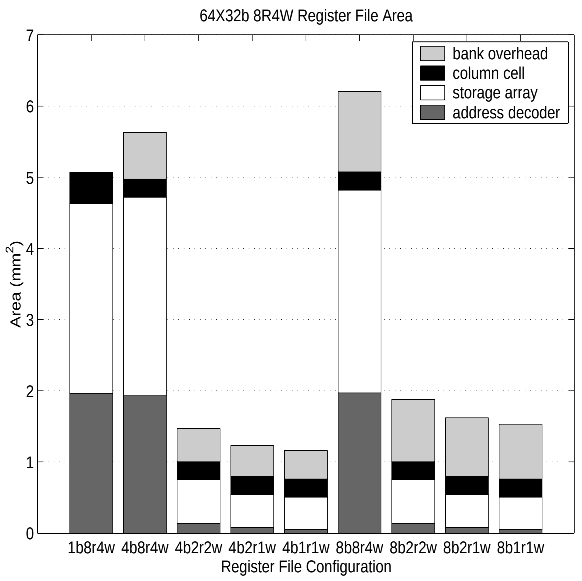

Figure 4. Area breakdown of different 64×32b 8 read-ports and 4 write-ports register file designs. For example, 1b8r4w refers to the baseline implementation—using one bank with eight read ports and four write ports.

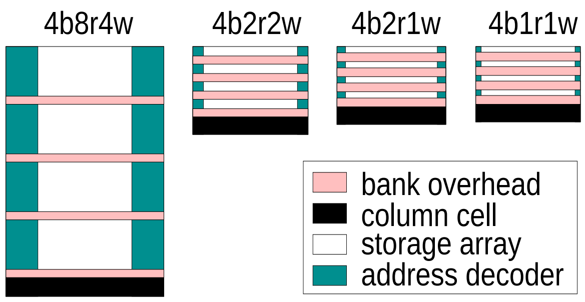

Figure 5. Area comparison of four different 64×32b regfiles for a quad-issue processor. The clear regions represent the storage cells while the lighter shaded regions represent the overhead circuitry in each bank. The black shading at the bottom is the area required for the global bitline column circuitry. The medium-dark shading to the side is the area for address decoders.

64×32b, 8 read ports, 4 write ports				
Delay	8r4w	2r2w	2r1w	1r1w
1 bank	100.00%	-	-	-
4 bank	92.38%	79.05%	79.05%	81.90%
8 bank	83.33%	74.76%	74.76%	77.14%
Energy	8r4w	2r2w	2r1w	1r1w
1 bank	100.00%	-	-	-
4 bank	61.98%	57.93%	56.90%	40.54%
8 bank	61.41%	58.62%	57.55%	40.71%

Table 2. Relative delay and energy numbers of different 64×32-bit eight global read port and four global write port register file designs.

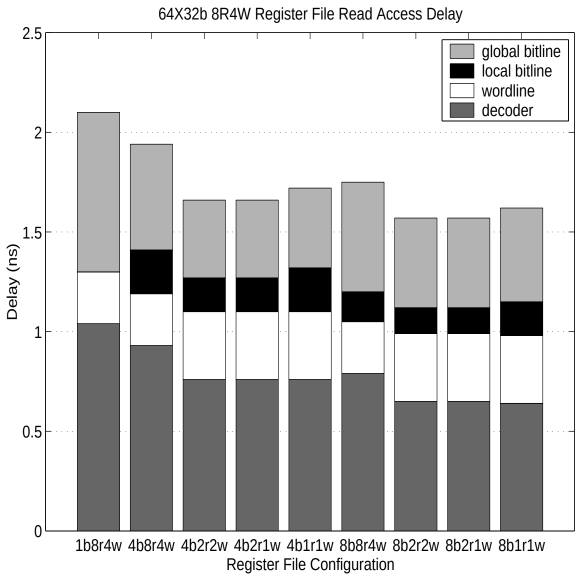

Figure 6. Read access delay breakdown of different 64×32-bit 8 read-ports and 4 write-ports register file designs.

overhead dominates with few ports per cell. Designs with two read ports per bank are not much larger than designs with a single read port per bank given that the single read port must connect to all global read ports whereas each of the two read ports only connects to half of the global read ports. Increasing the number of write ports from one to two adds only 16–20% in area.

Table 2 lists normalized delay and energy measures for the different multibanked register file designs compared to the unified design. Figure 6 and Figure 7 show the detailed delay and energy breakdowns of these designs. Delay and energy numbers were obtained from HSPICE simulations of extracted layout with 2.5V supply voltage. We obtained relative leakage energy numbers by calculating the total width of leaking transistors assuming that 70% of stored values are zero, and the bit cell ports were optimized to reduce read energy for zero values [19].

For the fully ported storage cell designs, using hierarchical bitlines reduces energy by almost 40% and cuts delay by 8–17%. The lesser-ported bank designs have a slightly greater reduction in delay, at around 20% faster for the two read, two write port case. The energy reduction is also slightly greater for the lesser-ported cells compared with just using hierarchical bitlines on the fully-ported cells. The delay and energy reductions are not as great as might be expected from the area reduction as the packing of two local storage cells per global bit column slows the wordline drive and adds a column mux stage, and also causes twice as many bitlines to discharge on a read. It would be possible to reoptimize the smaller ported cells for even smaller delay and energy, but this would add considerable additional area.

4 Performance Evaluation

To determine the performance impact, we modified the Simplescalar simulator [4] to keep track of a unified physical register file organized into banks. We did not modify the register renaming strategy, simply taking the next available registers off a single FIFO free list regardless of bank allocation. The machine configurations are shown in Table 3. For designs with multibanked register files, we modeled the additional cycles required for read and write bank conflicts and pipeline repair. To account for the extra arbitration cycle, we increase branch misprediction latency by one cycle. The baseline design has a three-cycle latency for branch misprediction while other designs with multibanked register files have a four-cycle latency. We performed simulations of four and eight bank configurations for a four-issue machine.

We chose a subset of SPEC2000 and Mediabench benchmarks compiled with optimization for the PISA instruction set. The Mediabench benchmarks were used to provide some higher IPC codes that we would expect to cause greater register file traffic. The Mediabench codes were run to completion. For the SPEC2000 numbers, we used the methodology described in [15] to select a fast-forward period and sample length. We first simulate the baseline case which uses a unified register file design and does not cause any register file port conflicts. Then we analyze the performance of various multibanked register file schemes.

Figure 8 and Table 4 show the resulting absolute and relative IPC numbers obtained for the four issue machine with register file of size 64. We label each configuration as *#banks*/*#reads*/*#writes*/*bypass*/*sharing*, where *#banks* is the number of banks, *#reads* is the number of local read ports,

L1 I-cache	32KB 2-way, 64-byte lines, 2 cycles
L1 D-cache	32KB 2-way, 32-byte lines, 2 cycles
L2 unified cashe	1MB 4-way, 64-byte lines, 10 cycles
Memory system ports available to CPU	2
Branch misprediction latency	3 (baseline), 4 (others)
Fetch, dispatch, commit width	4
Integer ALUs	4
Integer multi/div	1

Table 3. Simplescalar simulation configurations.

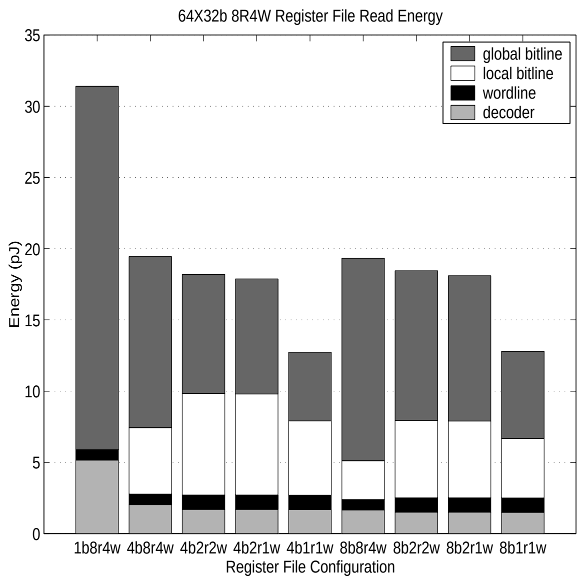

Figure 7. Read energy consumption breakdown of different 64×32b 8 read-ports and 4 write-ports register file designs.

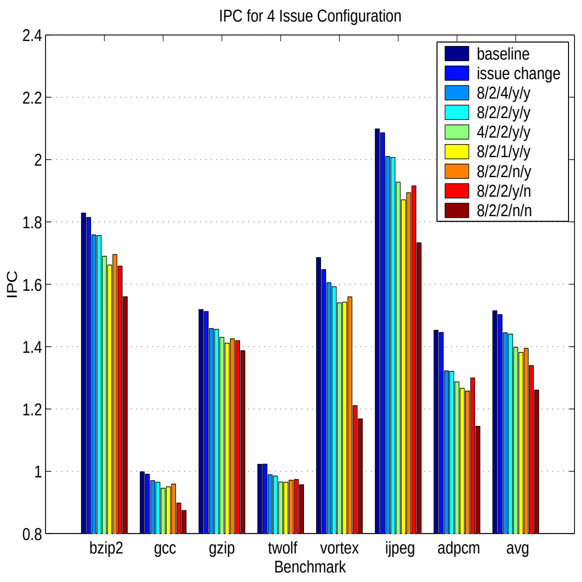

Figure 8. IPCs for the 4-issue pipeline with register file of size 64.

#writes is the number of local write ports, *bypass* indicates if values bypassed from the last execution cycle avoid competing for register ports, and *sharing* indicates if local read ports can drive multiple global read ports to implement read sharing.

We have also included a configuration labelled *issue 8/2/2/y/y* which shows the results if the select logic were changed to avoid register bank conflicts at issue time, and where both bypassing and port sharing are used to reduce conflicts in a system with eight 2R2W banks. In this case, performance is degraded by less than 1% compared to the baseline. This scheme would have a slower wakeup-select loop which would likely limit clock frequency. The row labelled (8/2/2/y/y) shows the performance drop when we instead issue instructions without considering conflicts and kill conflicting instructions. Performance drops another 4–6%, but it is possible this configuration could have a higher cycle time to make up for this difference.

Overall, we found the (8/2/2/y/y) configuration to perform well for this design point, and we chose this as our center point in perturbing other parameters. Reducing the number of banks to four (4/2/2/y/y), lowers performance by another 3–4%. We can also see that moving from 1 to 2 write ports (8/2/1/y/y, 8/2/2/y/y) improves performance by more than 4% but having more than 2 write ports per bank (8/2/4/y/y and 8/2/8/y/y) adds only another 0.3%. This is expected given that average IPCs are rarely above 2, and some instructions do not write registers.

Omitting the bypass optimization (8/2/2/n/y) reduces performance by over 3%. Omitting the sharing optimization reduces performance by around 7%. We confirmed the observation in [2] that groups of load and store instructions dependent on the stack pointer tend to issue together, probably

Type	bzip2	gcc	gzip	twolf	vortex	ijpeg	adpcm	average
baseline	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
issue 8/2/2/y/y	99.3%	99.2%	99.6%	100.0%	97.7%	99.4%	99.5%	99.2%
8/2/4/y/y	96.2%	97.2%	96.0%	96.7%	95.2%	95.8%	91.0%	95.4%
8/2/2/y/y	96.1%	96.7%	95.8%	96.3%	94.5%	95.6%	90.9%	95.1%
4/2/2/y/y	92.4%	94.7%	94.1%	94.4%	91.4%	91.9%	88.6%	92.3%
8/2/1/y/y	90.9%	95.2%	92.9%	94.4%	91.5%	89.2%	87.2%	91.2%
8/2/2/n/y	92.7%	96.1%	93.9%	95.0%	92.5%	90.2%	86.6%	92.1%
8/2/2/y/n	90.7%	89.9%	93.5%	95.2%	71.8%	91.3%	89.5%	88.4%
8/2/2/n/n	85.3%	87.6%	91.3%	93.5%	69.3%	82.6%	78.8%	83.2%

Table 4. Normalized IPC % for a quad-issue machine with 64 physical registers. Configurations are labelled as (#banks)/(#local read ports)/(#local write ports)/(bypass skipped?)/(read sharing?). Results are normalized to the IPC of the baseline case (unified with eight read and four write ports). Results for a configuration with bank arbitration in the issue logic (issue 8/2/2/y/y) are also shown.

at procedure call/return points. We also noticed that branch instructions dependent on the same register issuing together were another common source of value sharing. Omitting both bypassing and value sharing (8/2/2/n/n) lowers performance by 12–14%.

5 Related Work

Several techniques can reduce the difficulty of providing a large, fast multiported register file. One approach, used in the Alpha 21264 [11] and 21464 [14] designs, is to divide the functional units among two clusters and provide copy of all registers in each cluster. This approach halves the number of read ports required on each copy of the regfile, but requires the same number of write ports on both regfiles to allow values produced in one cluster to be made available in the second cluster. An extension of this approach is to develop a clustered microachitecture that divides the registers among a number of clusters [17, 12, 8, 23, 16]. Clustered microarchitectures also allow the instruction window to be divided among clusters and can potentially scale to larger issue widths at high clock frequencies. Clustering reduces the number of ports on each partition of the register file, but requires inter-cluster communication when a value is needed from a different cluster. A critical issue in the design of such systems is the heuristics used to map instructions to clusters. The primary disadvantages of a clustered microarchitecture is the complexity of the inter-cluster control logic and the additional area required to achieve performance similar to a centralized architecture.

The approach adopted here and in previous work [20, 2, 13] retains a centralized and non-duplicated register file, but constructs this from multiple interleaved register banks. The challenge with this approach is managing the complexity and added latency of the control logic needed to handle read and write bank conflicts and the mapping of register ports to functional units.

A banking scheme that uses the bypass network to reduce read port usage is described in [20] but no description of the bypass check or read conflict resolution logic is given. Write conflicts are handled by delaying physical register allocation until writeback, at which point registers are mapped to non-conflicting banks. The primary motivation for delayed allocation was to limit the size of the physical register file, but this can lead to a deadlock situation requiring a complex recovery scheme [20].

The scheme presented in [2] handles read bank conflicts by only scheduling groups of instructions without conflicts. As confirmed by our results above, this reduces the IPC penalty but adds significant logic into the critical wakeup-select loop. The authors also assume that bypassability can be determined during wakeup, but do not detail the mechanism used [2]. A design with single-ported read banks is evaluated, but this requires more complex issue logic and functional unit datapaths to allow instructions where both operands are from the same bank to issue across two successive bank read cycles. As we show in the register file layouts above, there is little area overhead in moving from single read ports to split dual read ports per bank once multiplexing overhead is considered. Write port conflicts are handled by buffering conflicting writes [2], which increases the size of the bypass network. Functional unit pipelines must also be stalled when conflicting writes build up. Our scheme never has any write stalls because write bank conflicts are detected after issue and conflicting instructions are killed. We add an additional write port per bank to maintain acceptable performance. We believe the reduction in control logic complexity and bypass mux size justifies the increase (16–20%) in overall register file area.

The bypass hint scheme proposed in [13] also makes use of the wakeup tag search to determine bypassability. However, because the bypass hint is not reset every cycle, the hint is optimisitc and can be incorrect if the source instruction has written back to the register file before the dependent instruction is issued. The authors propose stalling the pipeline in the case that optimistic bypass hints cause read port bandwidth to be oversubscribed [13] but stalls are difficult to implement in a high frequency pipeline without compromising cycle time. In contrast, our bypass bit is conservative and is only set if the bypass will occur from the immediately preceding cycle. Our scheme does not save register port bandwidth for operands that will be bypassed from later bypass stages, but avoids the possibility of a stall. The design in [13] does not use banked reads to avoid increasing the complexity of the select logic. The select logic still has to select no more instructions that there are available read ports after considering the bypass hint bits. In contrast, our scheme issues instructions without considering bypassability or conflicts and relies on rapid port arbitration and non-stalling pipeline repair to reduce the pipeline latency impact. This enables the use of read banking to reduce cell size.

A number of other approaches have been described to reduce the complexity of a large multiported register file. Registers can be cached to reduce average access latency [6, 3]. Register caching can add considerable control complexity to an architecture, as register caches have much worse locality than conventional data caches and determining the appropriate values to cache is non-trivial. Register caching is motivated by the increasing access penalty of conventional multiported structures as port counts and register counts increase. Multibanking counteracts this increase and reduces total area.

The preceding work has focused on the design of a high bandwidth register file for dynamically scheduled superscalar processors with a single logical register file. Other work has examined the use of partitioned register files made visible to software. The SPARC architecture [21] has overlapping register windows where software explicity switches between sets of registers. In-order superscalar implementations of the UltraSPARC exploit the fact that only one register window is visible to implement a dense multiported structure [18]. Clustered VLIW machines make the presence of multiple register file banks visible to software, and the compiler is responsible for mapping instructions to clusters [10]. Vector machines have also long been designed with interleaved register file banks that exploit the regular access patterns of vector instructions to provide high bandwidth with few conflicts [5, 7].

6 Conclusion

A banked register file design can provide the bandwidth needed by a superscalar processor but with reduced area, delay and energy. By using layouts of all components in the register file, we see that for a small number of ports per bank, overall register file size grows slowly as ports are added because area is dominated by bank interconnect. We exploit this fact by using more bank ports to reduce the IPC impact of a simpler pipelined control scheme that allows higher frequency operation. For a quad-issue processor, register file size is reduced by over a factor of three while reducing IPC by under 5%. Access time is reduced by 25% and access energy by 40%. These reductions in register file delay and power can potentially be used to increase clock rate, leading to a more complexity-effective design.

7 Acknowledgments

This work was funded in part by an NSF graduate fellowship, DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and an equipment donation from Intel Corporation.

References

[1] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar. A low-leakage dynamic multi-ported register file in 0.13 μm CMOS. In *ISLPED*, pages 68–71, 2001.

[2] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO-34*, December 2001.

[3] E. Borch, E. Tune, S. Manne, and J. S. Emer. Loose loops sink chips. In *HPCA*, pages 299–310, Boston, MA, February 2002.

[4] D. Burger and T. Austin. The Simplescalar Toolset, version 2.0. Technical report, University of Wisconsin-Madison, New York, June 1997.

[5] Unisys Corporation. Scientific processor vector file organization. U.S. Patent 4,875,161, October 1989.

[6] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA-27*, pages 316–325, 2000.

[7] DEC. Vector register system for executing plural read/write commands concurrently and independently routing data to plural read/write ports. U.S. Patent 4,980,817, December 1990.

[8] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko G. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In *MICRO-30*, pages 149–159, 1997.

[9] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully-bypassed 6-issue integer datapath and register file on an Itanium microprocessor. *IEEE Journal Solid-State Circuits*, 37(11):1433 – 1440, November 2002.

[10] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA-10*, pages 140–150, 1983.

[11] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[12] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA-24*, pages 206–218, June 1997.

[13] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO-35*, Istanbul, Turkey, November 2002.

[14] R. P. Preston et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *ISSCC Digest and Visuals Supplement*, February 2002.

[15] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM Research Report, Yorktown Heights, New York, October 2000.

[16] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *MICRO-35*, Istanbul, Turkey, November 2002.

[17] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA-22*, 1995.

[18] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *HICSS*, January 1995.

[19] J. Tseng and K. Asanović. Energy-efficient register access. In *Proc. of the 13th Symposium on Integrated Circuits and Systems Design*, Manaus, Brazil, September 2000.

[20] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proc. PACT*, October 1996.

[21] D. L. Weaver and T. Germond. *"The SPARC Architecture Manual/Version 9"*. Prentice Hall, February 1994.

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

[23] V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Trans. on Computers*, 50(3):268–285, March 2001.