Appears in the Proceedings of the International Conference on Supercomputing, June, 2002

Low-Complexity Reorder Buffer Architecture

Gurhan Kucuk Dmitry Ponomarev Kanad Ghose

Department of Computer Science
State University of New York, Binghamton, NY 13902-6000
e-mail:{gurhan, dima, ghose}@cs.binghamton.edu
http://www.cs.binghamton.edu/~lowpower

ABSTRACT

In some of today's superscalar processors (e.g.the Pentium III), the result repositories are implemented as the Reorder Buffer (ROB) slots. In such designs, the ROB is a complex multi-ported structure that occupies a significant portion of the die area and dissipates a non-trivial fraction of the total chip power, as much as 27% according to some estimates. In addition, an access to such ROB typically takes more than one cycle, impacting the IPC adversely.

We propose a low-complexity and low-power ROB design that exploits the fact that the bulk of the source operand values is obtained through data forwarding to the issue queue or through direct reads of the committed register values. Our ROB design uses an organization that completely eliminates the read ports needed to read out operand values for instruction issue. Any consequential performance degradation is countered by using a small number of associatively-addressed retention latches to hold the most recent set of values written into the ROB. The contents of the retention latches are used to satisfy the operand reads for issue that would otherwise have to be read from the ROB slots. Significant savings of the ROB real estate as well as power savings in the range of 20% to 30% for the ROB are achieved using the proposed technique. At the same time, the fact that results are accessible in a single cycle from the retention latches actually leads to an overall improvement in the IPC of up to 3% on the average for SPEC 2000 benchmarks.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures - *pipeline processors*
B.5.1 [**Register-Transfer-Level Implementation**]: Design - *data-path design*

General Terms

Design, Performance, Algorithms

Keywords

Low-power design, low-complexity datapath, reorder buffer

1. INTRODUCTION

Contemporary superscalar microprocessors rely on aggressive execution reordering mechanisms to maximize the number of instructions committed per cycle. One of the main dynamic instruction scheduling artifacts used in such datapath designs is the Reorder Buffer (ROB) [17], which guarantees the recovery to a precise state when interrupts occur. The ROB is also used to handle branch mispredictions. It is typically implemented as a circular FIFO queue with head and tail pointers. Entries are made at the tail of the ROB in program order for each of the co-dispatched instructions. Instructions are committed from the head of the ROB to the architectural register file (ARF), preserving the correct (program) order of updates to the precise state maintained in the ARF.

Most high-performance processors take care of false data dependencies by using register renaming. In some of the register renaming implementations, the ROB slots also serve as physical registers. Such a datapath, which closely resembles Pentium III processor [9], is shown in Figure 1. Here, the results produced by functional units (FUs) are written into the ROB slots and simultaneously forwarded to dispatched instructions waiting in the Issue Queue (IQ) at the time of writeback. The result values are committed to the ARF at the time of instruction retirement. If a source operand is available at the time of instruction dispatch, the value of the source register is read out from the most recently established entry of the corresponding architectural register. This entry may be either an ROB slot or the architectural register itself. If the result is not available, appropriate forwarding paths are set up.

The ROB is generally implemented as a multi-ported register file, sometimes augmented with associative addressing capabilities. A significant power dissipation occurs during associative lookup of ROB entries, in the course of ROB writes for setting up new entries or generating result values, and in the course of reading out data from the ROB during operand reads or commits. For example, a recent study by Folegnani and Gonzalez [6] estimated that about 27% of the total power expended within a Pentium III-like microprocessor is dissipated in the ROB.

In this paper, we propose a considerably simplified ROB architecture, which exploits the fact that in a typical superscalar processor the bulk of the source operand values is obtained through data forwarding or from reading of the architectural registers containing committed register values. Only a very small percentage of sources - about 5% on the average across simulated SPEC 2000 benchmarks - are read from the ROB. Our design completely eliminates read ports on the ROB for reading out the source operand values and provides the same functionality as the traditional datapath at almost the same, and sometimes better, performance.

At the time of instruction writeback, the produced results are written into the ROB and simultaneously forwarded to dispatched, waiting instructions in the issue queue. The elimination of the ROB read ports means that the produced result is not accessible to any instruction that was dispatched since the result was written into the ROB till the result

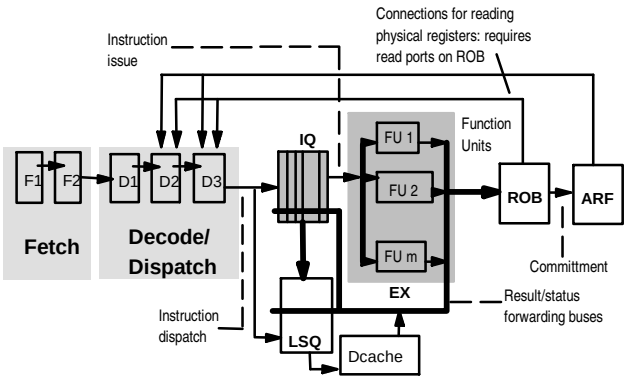


Figure 1. Superscalar datapath where ROB slots serve as physical registers

is committed and written into the ARF. We supply the operand value to instructions that were dispatched in the duration between the writing of the result into the ROB and the cycle just prior to its commitment to the ARF by simply forwarding the value (again) on the forwarding buses at the time of its commitment.

Such design potentially introduces multi-cycle "holes" in data availability, when the data is available to subsequent instructions during the writeback/forwarding stage, then it disappears for a number of cycles during its residency in the ROB and then is available again from the ARF. To counter any performance degradation that results as a consequence of such "holes", we use a small number of associatively-addressed retention latches to hold the most recent set of values written into the ROB. The contents of the retention latches are used to satisfy the operand reads for issue that would otherwise have to be read from the ROB slots.

The rest of the paper is organized as follows. Section 2 outlines the ROB structures and complexities. The motivation for the low-power and low-complexity ROB design is given in Section 3. Details of our approach for ROB complexity minimization are presented in Section 4. Section 5 describes our simulation methodology followed by the discussion of the experimental results in Section 6. Related work is described in Section 7 and we conclude in Section 8.

2. ROB COMPLEXITIES

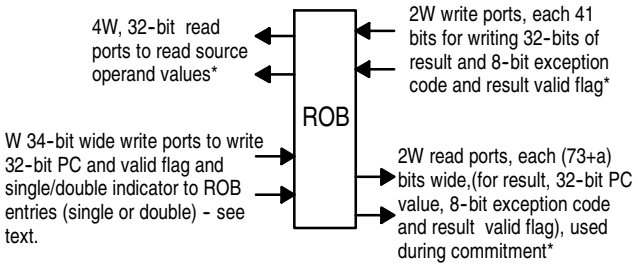
The entry established in the ROB for a dispatched instruction includes at least the following fields: (a) a result field to hold the value generated by the instruction that targets a register; some instructions do not make use of this field; (b) a bit to indicate if the result field is valid; (c) the address of the instruction ("PC value"); (d) exception codes and (e) architectural register id (used for updating the architectural register within the ARF at the time of committing the instruction.)

When one has to consider higher precision results, the ROB entry for an instruction can be widened, but space would be wasted when smaller precision results are stored. Alternatively, each entry can be wide enough to accommodate single-precision results. Extended precision results will require the allocation of a number of consecutive entries in the ROB; additional ports (each narrower) will be needed in this case. This latter approach, however, wastes space within the ROB entries for storing the PC values and exception codes, since only one of the single-precision entries for an instruction that produces a higher precision result needs to store this information. For the rest of this paper, we consider a ROB organization that is wide enough to only accommodate single-precision results.

Architectural registers can be mapped to physical registers through an explicit rename table; the ROB is directly addressed in this case. An alternative to the rename table is to use an associative addressing mechanism within the ROB for locating the most recently established ROB entry for that architectural register, as was used in AMD K5, for example [16]. To the best of our knowledge, such associatively-addressed ROB is no longer used in current implementations, so we consider directly addressed ROBs in the remainder of this paper.

Irrespective of the particular implementation, the ROB integrating physical registers is an extremely port-rich structure, especially in wide-issue processors. When a relatively slow clock rate is used, it may be possible to reduce the number of ports on the ROB (and the rename table) by multiplexing the ports. Given the relatively large access time of the ROB, such multiplexing becomes increasingly difficult as the clock frequencies increase. Implementation problems arising with the reduced number of register file ports are discussed in [4].

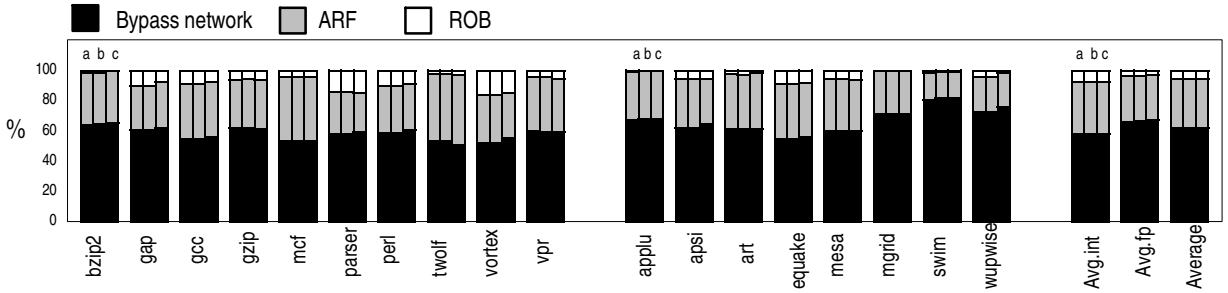
Figure 2 summarizes the port requirements for a ROB of a W-way superscalar processor. The widths (number of bits) for each port are also shown in this figure, where *a* is the number of bits in the architectural register address. Two operand widths are assumed: 32 bits and 64 bits; 64-bit operands are generated into two 32-bit architectural registers.



* allows up to 2W double length operands; information for each double operand requires 2 ports

Figure 2. ROB port requirements for a W-way superscalar datapath

The ROB complexity, as seen from the preceding discussions, can be quite substantial. The large number of ports, aside from increasing the device count linearly with the number of ports, increases the layout area for the bitcell array almost quadratically with the number of ports, as additional ports increase each lateral dimension of a bitcell linearly. This is because each ROB port requires its own set of bit lines and word select lines, accounting for the linear growth in each lateral dimension of a bitcell. Furthermore, each port also requires a dedicated address decoder and peripheral logic in the form of sense amps, prechargers and write drivers. In high-end superscalar CPUs, the ROB area can be a significant fraction of the overall die area. For example, the reorder buffer ("instruction queue") on the PA 8000 occupies about 15% of the total chip area [7]. Such complex ROB also represents a major source of total chip power dissipation - as high as 27% [6]. Finally, multiple cycles are often needed to read the data values from such large multi-ported structure [4, 5]. In the baseline datapath of Figure 1, we assumed that two cycles are needed to access the ROB. Thus, three "D-stages" are used: stage D1 is used for decoding and register renaming and stages D2 and D3 are used for accessing the ROB and moving the instruction into the issue queue.



(a) 72-entry ROB, 4-way processor (b) 96-entry ROB, 4-way processor (c) 128-entry ROB, 6-way processor

Figure 3. The origin of source operands in the baseline superscalar processor

In the result section, we also compare the performance of our new architecture with the idealized baseline datapath, assuming that the ROB can be accessed in one cycle.

3. MOTIVATIONS

It is worthwhile to consider strategies for reducing the complexity of the ROB, particularly from the standpoint of power dissipation. A detailed simulation of the SPEC 2000 benchmarks [18] on the baseline processor of Figure 1 reveals an interesting fact - only a small fraction of the source operand reads require the values to come from the ROB. Section 5 details the datapath used in this study.

In Figure 3, we show the percentage of operand reads that are satisfied through forwarding, from reads of the ARF or from reads of the ROB, for three different configurations. The three configurations differ in the number of ROB entries and the issue widths. As seen from Figure 3, for a 4-way machine with 72-entry ROB, the bulk of the operand reads (in excess of 61%) is satisfied through forwarding. Sizable percentage of operands (about 32%) comes from the ARF. These are mostly the values of stack pointer, frame pointer and base address registers. Their mappings typically do not change for large number of cycles and these registers remain valid in the ARF. Irrespective of processor configuration, the percentage of cases when source operand reads are satisfied from the ROB represents only a small percentage of all source operands. For example, for a 4-way processor with a 72-entry ROB, only 0.5% through 16% of the source operands comes from the ROB with the average of 5.9% across all simulated benchmarks. Note, that the results change slightly between the three considered configurations in a non-intuitive manner because of dynamics of instruction processing. The rather small percentage of reads of source operand values from the ROB suggests that the number of connections, i.e., the number of ports for reading source operand values from the ROB can be drastically minimized or, better yet, eliminated altogether to reduce the ROB complexity and power dissipation directly.

Completely eliminating the read ports for reading source operand values is fundamentally different from reducing the number of ports in two ways. First, the elimination of ports removes the ROB access from the critical path. Of course, other means of supplying the source operand values to the dispatched instructions must be provided, as discussed later. In contrast, even if a single port for reading out the source operands is retained, the two-cycle ROB access time (which cannot be reduced because the ROB still remains heavily-ported due to the ports needed for writeback, commitment and entry setup) still resides on the critical path. Second, complete elimination of ports avoids design complications associated with managing the register file (ROB) with reduced number of ports, as discussed in Borch et.al. [4]. A pleasant side-effect of eliminating the source read ports on the ROB is that we only read the source values that are committed from the ROB, thereby eliminating some - but not all - spurious computations using results that are generated on the mispredicted path(s).

4. LOW-COMPLEXITY ROB DESIGN

Consistent with the observation made in Section 3, we propose the use of a ROB structure without any read ports for reading source operand values. Results are written into the ROB and simultaneously forwarded to dispatched, waiting instructions on the result/status forwarding bus. Till the result is committed (and written into the ARF) it is not accessible to any instruction that was dispatched since the result was written into the ROB. Since it is not possible for dispatched instructions to repeatedly check for the appearance of a result in the ARF (and then read it from the ARF), we need to supply the operand value to instructions that were dispatched in the duration between the writing of the result into the ROB and the cycle just prior to its commitment to the ARF. As shown in Figure 4, we do this by simply forwarding the value (again) on the forwarding buses at the time it is committed. We will see shortly that there is no need to forward each and every result from the ROB at the time of commitment. Neither do we need to increase the number of forwarding paths compared to the baseline case, as the utilizations of such paths are quite low.

4.1 Reducing Performance Degradation

The elimination of the ROB read ports for reading out source operand values results in "disappearance" of these values for several cycles. In particular, the value is initially available through forwarding network in the writeback cycle. As operand read ports are eliminated in the ROB, the value is not available again until it commits from the ROB to the ARF, at which point the second forwarding of the same value occurs. Certainly, the issue of some instructions that could not read this value from the proposed ROB structure will be delayed.

To compensate for any resulting performance degradation, we use a set of latches, called **retention latches**, to cache results recently written to the ROB. Each retention latch is capable of holding a single-precision result, so double precision results would need two latches. Instructions dispatched since the writing of a result to the ROB can still access the value as long as they can get it from the retention latches. In case the lookup for source operand fails to find it within the retention latches, the operand value will be eventually obtained through the second forwarding of the same value when it is written to the ARF from the ROB. The modified datapath with retention latches is shown in Figure 4.

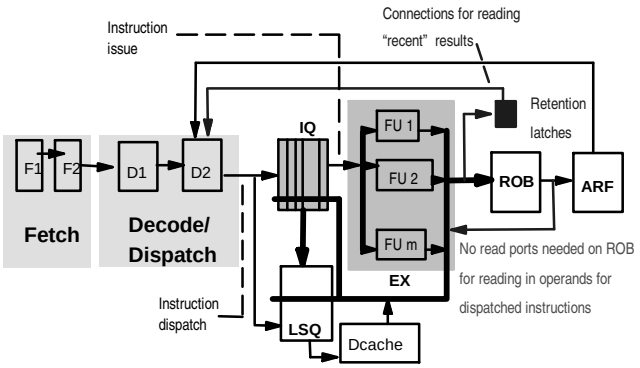


Figure 4. Superscalar datapath with the simplified ROB and retention latches

4.2 Details of the Simplified ROB

We now discuss our proposed low-complexity ROB design in some detail.

As soon as a function unit completes execution and writes its result and exception codes into the ROB, it simultaneously writes the result into one (or two, if the result is double precision) of the retention latch(es) displacing an earlier-written result (or, in the case of double precision results, the contents of two latches). We use three techniques for choosing victim latches for such writebacks: a FIFO policy, a true LRU policy and a random replacement, as detailed later.

At the time of dispatch, the ROB index of the entry for a source operand is obtained from the rename table if the "committed" flag within the entry indicates that the corresponding value is yet to be committed to the ARF. If the "result valid" bit of this ROB entry is set (meaning that the result has already been produced), this ROB index is used as a key to probe the associatively-addressed retention latches. Note that an architectural register id cannot be used as a key into the retention latches to locate the most recent value of the architectural register, since the writes into the retention latches take place out of program order in general. Consequently, the most recently written value in the retention latches may not correspond to the most recent "reincarnation" of the architectural register. On a successful match, the source operand value is obtained from the retention latches. Otherwise, data forwarding paths (for the second forwarding) are set up using the ROB index number in the usual manner. If the "result valid" bit is not set, the source will be supplied through usual forwarding at the time of its writeback.

In Figure 5, we show the relevant timing diagram, assuming a two-stage process for decoding, dispatching and source operand read stages (hereafter called D-stages) and compare it with a three-stage process for the same steps in the baseline model of Figure 1.

The datapath shown in Figure 4 has two D-stages instead of three D-stages in the baseline model, because one cycle is used for register renaming and two cycles are needed to access the complex, large multi-ported register file (implementing ROB) in the baseline model [4, 5]. In fact, this is an optimistic estimate in favor of the baseline model: other contemporary researchers assumed a higher latency of a register file access - for example, in [4], the register file access time is assumed to be three CPU cycles. Such multi-stage register file (ROB) has a negative impact on performance and/or complexity in two ways. First, branch misprediction penalties increase because of the extra stage at the front end of the pipeline; second, performance degrades significantly if multiple levels of forwarding (bypassing) are not implemented [5]. In particular, if only the first level of forwarding is used for a two-stage register file, then a data value is available at the time of writeback/forwarding, then it "disappears" for one cycle, and then it is available again from the register file. The problem with such design is significant additional complexity of the issue logic needed to schedule the instructions around the "holes". An alternative approach is to keep only the last level of bypass, thus avoiding the "holes". This effectively increases the latency of each functional unit by one cycle and has a very serious effect on IPCs - 20% across our simulated SPEC 2000 benchmarks. Similar results were reported in [5] for SPEC 95 codes. Of course, performance drop can be avoided or drastically minimized by implementing a full/multi-level bypassing mechanism. However, this entails significant complexity in the form of more forwarding busses and comparators needed to perform the tag matching for instruction wake-up.

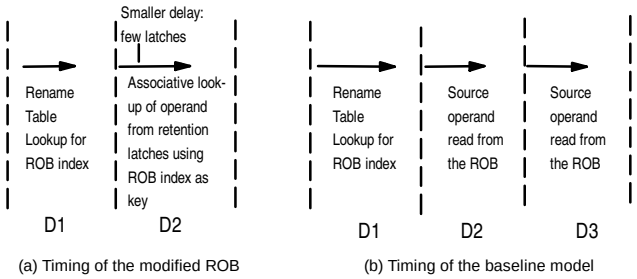


Figure 5. Assumed timing for the low-complexity ROB scheme and baseline model

In order not to hinder the baseline model, we compare the performance of our proposed architecture against three different implementations of the base case: single-cycle ROB access (idealized base case), 2-cycle ROB access with full bypass, and 2-cycle ROB access with only the last stage of bypassing.

4.3 Retention Latch Management Strategies

We study three variations of managing the retention latches. The first one is a simple FIFO scheme, where the retention latch array is used as a shifter. The writing of a result into a retention latch (or a pair of retention latches) causes the contents of one (or two) earliest written latch(es) - at the tail end of the latch array - to be shifted out and lost. The new results are written into the vacated entries at the head of the latch array. Even though the number of latches in the shifter array is small compared to the ROB size, it is possible, albeit highly unlikely, that duplicate entries keyed by a common ROB index will co-exist within the retention latches. This can, for example, happen in an extreme scenario when an instruction with a destination register is allocated to an ROB entry, and there is no other instruction with destination register (only stores and branches are encountered) until the same ROB entry is again reused for allocations. In such case, the contents of retention latches are not shifted and we end up having two retention latches keyed with the same physical register id. Branch mispredictions can also cause a ROB slot to be reallocated and two entries keyed with the same ROB index to co-exist within the retention latches. To avoid any ambiguity in finding the correct entry in such situations, we design the matching logic to return the most recently-written result in the retention latches. This is relatively easy to achieve, since the most recently-written value is always the closest to the head end of the latch array. The latch array is designed as a multi-ported structure to support simultaneous writes and reads. To support the reading and writing of up to W double precision or W single precision results from/to the retention latches, they are

implemented to support 4W associative addressing ports (requiring 4W comparators per latch) and to support 2W write ports.

The second variation of the retention latch structure is one where the latch contents are managed in a true LRU fashion. A true LRU scheme can be implemented since the number of latches is small (8 to 16). The LRU management policy allows only recently-used result values - and ones that are likely to be used again - to be retained within these latches. The LRU management policy is offering a better performance on the average across the simulated benchmarks; we discuss this phenomenon in detail in Section 6. As in the case of FIFO latches, it is conceivable, that two different entries, keyed with the same ROB index can reside within the retention latches. This happens when a result value continues to be frequently used from the retention latches and when the ROB slot is committed and eventually allocated to another instruction that establishes a duplicate entry in the retention latches. In contrast to the FIFO latches, the logic needed for selecting the most recently-produced value is more complicated, because the value can be located anywhere in the latch array. For this reason,we avoid the need for any disambiguating logic by deleting the entry for a ROB slot from the retention latches when its contents are committed to the ARF. We do this by associatively addressing the retention latches at the time of committing and marking the matching entry, if any, as deallocated. As in the case of the FIFO retention latches, branch mispredictions can introduce duplicate entries in the retention latches, keyed with a common ROB index. We avoid any consequential ambiguity in such an instance by invalidating the existing entries for ROB indices that are to be flushed because of the misprediction. This later invalidation is accomplished in a single cycle by adding a "branch tag" to the instructions and to the retention latch keys to invalidate retention latch entries that match the tag of the mispredicted branch. A simpler solution is to flush the entire set of retention latches in the case of branch misprediction. As shown in Section 6, this alternative degrades performance to a very little extent.

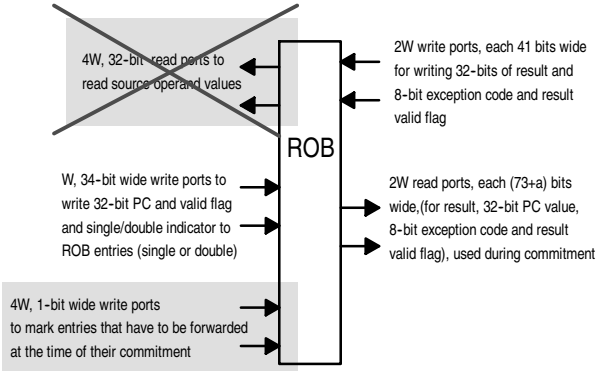
The third conceivable option is to use retention latches with random replacement policy. While this scheme is inferior in performance to both FIFO and LRU latches (as we demonstrate in Section 6), none of the design complications associated with LRU latches is eliminated: the corresponding entries still need to be flushed from the latches at the time of result commitment and some form of flushing need to occur on branch mispredictions.

4.4 Optimizing the Retention Latches

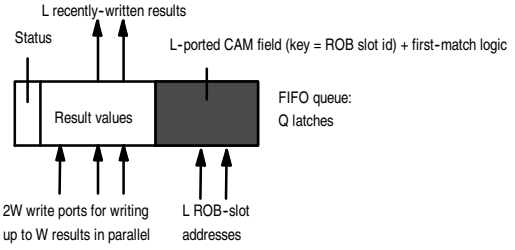
As seen from the data in Figure 3, the percentage of reads satisfied from the ROB is quite low. Consequently, the pressure on the retention latches is also low. Thus, the number of read ports to the retention latches can be drastically cut down from 4W single-precision result wide ports to only a few (1 or 2) read ports. Some of the instructions will be delayed because of contention over the limited number of retention latch ports, but this has little impact on performance, as discussed in Section 6. While implementing the register file (however small it is) with reduced number of ports can be problematic in general [4], in this particular case an instruction competing for the retention latch port will attempt to do so only once. If an attempt to acquire a port fails, an instruction is moved to the issue queue anyway, and obtains the result at the time of the second forwarding. We do not attempt to redispatch this instruction, since it would incur significant complexity and the effect on performance is not clear. The oldest-first arbitration scheme for the use of limited number of ports to the retention latches is used among the co-dispatched instructions, that is, the earliest instruction in program order has the highest priority for using a port. Note, that only the number of read ports to the retention latches is reduced, the number of write ports stays at its maximum level (2W for our architecture).

4.5 Reducing Forwarding Bus Contention

If all results were forwarded for the second time during their commitment from the ROB to the ARF, it would have created a significant pressure on the result/forwarding buses potentially requiring the allocation of extra buses to sustain performance. For this reason, only results whose values were actually sought from the ROB need to be forwarded at the time of commitment. ROB entries that need to be forwarded are marked as follows. An attempt to look up a source whose value is only within the ROB ("committed" bit not set and "result valid" bit set in the rename table entry) will cause the ROB entry to be marked; this requires 4W additional 1-bit wide ports to the simplified ROB.



(a) Low-Complexity ROB



(b) Generic retention latch structure

Figure 6. Low-Complexity ROB and retention latches
(ROB simplifications/changes highlighted)

It may so happen that a ROB entry marked for commitment may forward a result in vain, as the value sought may have been retrieved from the retention latches. Additional logic for avoiding this spurious traffic is overly complicated and not worth the investment as the selective forwarding traffic is already quite low, as seen from Figure 3. A result may thus show up on the forwarding bus twice - once (and always) when it is written to the ROB from a FU and possibly again when it is committed to the ARF from the ROB. All sources for the forwarding buses - including, in this case, the ROB, compete as usual for access to the buses. Our experiments showed that there is little difference in performance for various priority assignment schemes, mainly because forwarding at the time of commit occurs infrequently, as detailed below. In all presented experiments, the forwarding at the time of writeback was given the highest priority in accessing the forwarding buses.

Such selective forwarding of committed results generates only a small additional traffic on the forwarding buses. In our simulations, we observed that the percentage of such forwarding is limited to 3.5% on the average across all the benchmarks and for all the configurations studied, depending on the number of retention latches used as well as

the management policy. That is, about 3.5% of the generated results need to be forwarded for the second time. In terms of forwarding bus usage, only 0.05 buses are utilized per cycle on the average by the second forwarding. This allows us to use the existing forwarding buses without any noticeable degradation in performance. More results supporting this claim are presented in Section 6.

If selective forwarding as described is not used, the performance degradation resulting from the use of the existing set of forwarding buses can be substantial, because forwarding rate would increase by a factor of two; the only way to avoid performance loss in such a case would be to use additional forwarding/result buses.

Figure 6 depicts the details of the proposed ROB and the associated retention latches.

Table 1. Architectural configuration of simulated processors

Parameter	Configuration	
Machine width	4-wide fetch, 4-wide issue, 4-wide commit	6-wide fetch, 6-wide issue, 6-wide commit
Window size	32 entry issue queue, 96 entry ROB, 32 entry load/store queue	48 entry issue queue, 128 entry ROB, 48 entry load/store queue
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)	6 Int Add (1/1), 2 Int Mult (3/1) / Div (20/19), 3 Load/Store (2/1), 6 FP Add (2), 2FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I-cache	32 KB, 2-way set-associative, 32 byte line, 2 cycles hit time	
L1 D-cache	32 KB, 4-way set-associative, 32 byte line, 2 cycles hit time	
L2 Cache combined	512 KB, 4-way set-associative, 128 byte line, 4 cycles hit time	
BTB	1024 entry, 4-way set-associative	
Branch Predictor	Combined with 1K entry Gshare, 10 bit global history, 4K entry bimodal, 1K entry selector	
Memory	128 bit wide, 60 cycles first chunk, 2 cycles interchunk	
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency	

5. SIMULATION ENVIRONMENT

We used the AccuPower toolset [15] to evaluate the effects of the proposed architecture on the performance, power dissipation and the overall complexity of the ROB. The widely-used Simplescalar simulator [1] was significantly modified (the code for dispatch, issue, writeback and commit steps was written from scratch) to implement *true hardware level, cycle-by-cycle* simulation models for such datapath components as the ROB (integrating a physical register file), the issue queue, and the rename table. The studied configurations of superscalar processors are shown in Table 1.

We simulated the execution of 10 integer *(bzip2, gap, gcc, gzip, mcf, parser, perlbmk, twolf, vortex* and *vpr*) and 8 floating point (*applu, apsi, art, equake, mesa, mgrid, swim* and *wupwise*) benchmarks from SPEC 2000 suite. Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. The results from the simulation of the first 1 billion instructions were discarded and the results from the execution of the following 200 million instructions were used for all benchmarks.

For estimating the energy/power for the key datapath components, the event counts gleaned from the simulator were used, along with the energy dissipations, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the ROB and the retention latches in a 0.18 micron 6 metal layer CMOS process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition. The register file that implements the ROB was carefully designed to optimize the dimensions and allow the use of a 2 GHz clock. A V_{dd} of 1.8 volts was used for all the measurements.

6. RESULTS AND DISCUSSIONS

In this section, we evaluate the implications of the proposed techniques in terms of performance, power dissipation and the overall complexity of the ROB.

6.1 Performance

We begin by showing the effects of eliminating the ROB read ports for reading out source operand values on the performance. Figure 7 shows the IPCs of a 4-way machine with 96-entry ROB. Results are shown for four configurations: three baseline cases and the machine without any ROB read ports. (In the rest of the paper by saying "no ROB read ports" we mean "no ROB read ports for reading out source operand values". Of course, ROB read ports needed for instruction commitment are still retained). The first bar of Figure 7 shows IPCs of the idealized baseline model with a single cycle ROB access time. We call this configuration *Base 1*. The second bar shows IPCs of a more realistic baseline machine with two-cycle ROB access and a full bypass network; this is referred to as *Base 2*. The third bar shows IPCs of a machine that does not implement a full bypass and only maintains the last level of bypassing logic, effectively extending the latencies of functional units by one cycle. Finally, the fourth bar shows the IPCs of the proposed machine with no ROB ports. We assume that write accesses to the ROB still take two cycles.

Compared to the idealized baseline machine *Base 1*, the configuration with no ROB read ports performs 9.6% worse in terms of IPCs on the average. (We computed the average performance drop/gain by computing the drops/gains within individual benchmarks and taking their average). Across the individual benchmarks, the largest performance drop is observed for *swim* (36.7%), *parser* (16.9%) and *equake* (13%). There are many reasons why the performance drop is not necessarily proportional to the percentage of sources that are read from the ROB in the baseline model. Performance is dependent on the number of cycles that an instruction, whose destination was sought from the ROB as a source, spends in the ROB from the time of its writeback till the time of its commitment. This directly effects the duration of "holes" that are created by eliminating the read ports from the ROB. Another important factor is the criticality of these sources for the performance of the rest of the pipeline.

Some benchmarks experience almost no performance degradation, such as *mgrid* (1.05%) and *bzip2* (1.3%). What is remarkable about these results, is that even compared to the idealized base case, the total elimination of the ROB read ports for reading out the source operand values results in only less than 10% performance degradation on the average. This data supports the basic tenet of this paper, which is that the performance loss is quite limited even if the capability for reading out the sources from the ROB is not present. A 10% performance drop is the absolute worst case and, as we show later, the use of retention latches significantly improves performance and, combined with faster access time of the results, actually provides a better-performing architecture in some cases.

Compared to the baseline model with 2-cycle ROB access and full bypass, the configuration with zero ROB ports results in 6.5% drop

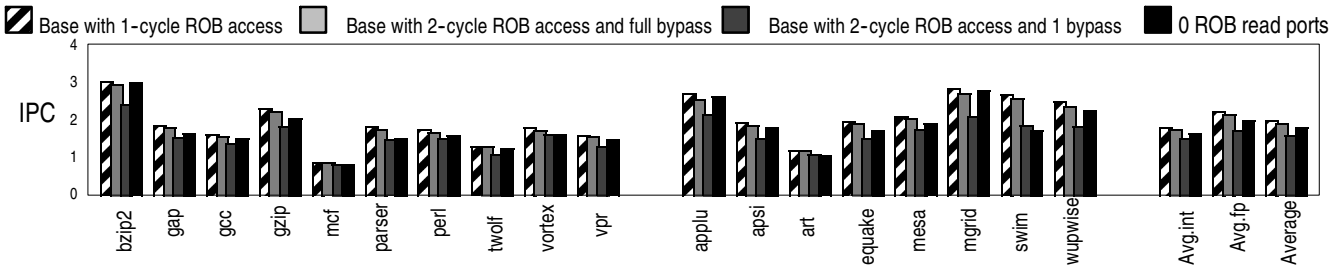


Figure 7. IPCs of baseline configurations and configuration without ROB read ports for reading out source operand values

in IPCs on the average. The drop is smaller than in the previous case because of the extra cycle that is added to the front end of the base case pipeline, thus increasing the branch misprediction penalties. A few benchmarks (*applu, bzip2* and *mgrid*) actually showed small performance improvements.

Not surprisingly, the system with no ROB read ports performed significantly better than the baseline model with only the last level of bypassing – performance gains are in excess of 11% on the average across all benchmarks. Only three benchmarks (*vortex, art* and *swim*) still exhibited some performance degradation.

Table 2 shows the performance improvements achieved by using retention latches. The first two columns of Table 2 show the IPCs of two optimistic baseline configurations – *Base 1* and *Base 2*. We do not consider the baseline model with partial bypass in the rest of the paper, because of its poor performance. Results in the first two columns are similar to those shown in Figure 7 and they are given here only for convenience. The next column shows the IPCs for the configuration with zero ROB read ports – again, these results were already graphed in Figure 7. The following three sets of columns show the performance of the architecture that uses retention latches managed as a FIFO, retention latches with LRU replacement policy and retention latches with random replacement policy, respectively. The "x-y" notation specifies the number of retention latches (x) and the number of read ports (y) to these latches in each case. The number of write ports to the retention latches was assumed to be eight in all simulations to support simultaneous writeback of four double-precision values. Results of Table 2 were obtained by simulating a 4-way machine with 96-entry ROB.

Table 2. IPCs of various ROB configurations

	Base 1	Base 2	0 ROB read ports	Retention Latches with FIFO replacement					Retention Latches with LRU replacement					Retention Latches with random replacement	
				8-1	8-2	8-16	16-2	16-16	8-1	8-2	8-16	16-2	16-16	8-16	16-16
bzip	3.05	2.95	3.01	3.03	3.03	3.03	3.03	3.03	3.03	3.04	3.04	3.04	3.04	3.03	3.03
gap	1.87	1.82	1.65	1.80	1.81	1.82	1.84	1.84	1.85	1.85	1.87	1.87	1.87	1.74	1.76
gcc	1.62	1.56	1.53	1.60	1.60	1.61	1.61	1.62	1.61	1.61	1.62	1.62	1.62	1.59	1.61
gzip	2.31	2.25	2.04	2.25	2.26	2.26	2.29	2.29	2.28	2.30	2.30	2.30	2.31	2.25	2.28
mcf	0.88	0.88	0.82	0.83	0.84	0.84	0.86	0.87	0.86	0.86	0.87	0.88	0.88	0.86	0.86
parser	1.84	1.76	1.53	1.68	1.71	1.72	1.79	1.80	1.72	1.79	1.80	1.82	1.83	1.66	1.71
perl	1.76	1.67	1.60	1.65	1.66	1.66	1.70	1.71	1.70	1.72	1.72	1.74	1.75	1.63	1.65
twolf	1.29	1.29	1.24	1.27	1.27	1.27	1.28	1.28	1.27	1.29	1.29	1.29	1.29	1.27	1.28
vortex	1.80	1.73	1.62	1.72	1.73	1.75	1.76	1.79	1.75	1.76	1.77	1.78	1.79	1.75	1.77
vpr	1.59	1.56	1.49	1.53	1.55	1.56	1.56	1.57	1.54	1.57	1.58	1.58	1.58	1.54	1.56
applu	2.71	2.56	2.64	2.66	2.66	2.66	2.67	2.67	2.66	2.66	2.66	2.68	2.68	2.66	2.67
apsi	1.94	1.88	1.81	1.88	1.89	1.90	1.90	1.91	1.89	1.89	1.90	1.92	1.93	1.89	1.89
art	1.21	1.18	1.07	1.15	1.16	1.16	1.18	1.18	1.16	1.16	1.16	1.18	1.18	1.12	1.15
equake	1.97	1.90	1.71	1.75	1.88	1.88	1.88	1.88	1.74	1.84	1.95	1.84	1.97	1.66	1.73
mesa	2.10	2.04	1.93	2.04	2.04	2.05	2.06	2.06	2.06	2.06	2.06	2.06	2.08	2.03	2.05
mgrid	2.85	2.71	2.82	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.84	2.84
swim	2.70	2.59	1.71	2.64	2.64	2.64	2.65	2.66	2.65	2.69	2.69	2.69	2.69	2.62	2.65
wupwise	2.50	2.39	2.26	2.26	2.26	2.26	2.26	2.26	2.26	2.50	2.50	2.50	2.50	2.26	2.29
Int avg.	1.80	1.75	1.65	1.74	1.75	1.75	1.77	1.78	1.76	1.78	1.79	1.79	1.80	1.73	1.75
FP avg.	2.25	2.16	1.99	2.15	2.17	2.18	2.18	2.18	2.16	2.21	2.22	2.21	2.24	2.14	2.16
Average	2.00	1.93	1.80	1.92	1.94	1.94	1.95	1.96	1.94	1.97	1.98	1.98	1.99	1.91	1.93

As seen from the table, even eight single-ported retention latches managed as a FIFO reduce the performance penalty to about 4.1% on the average compared to the baseline configuration *Base 1*. The most dramatic improvement is observed for *swim:* 36.7% performance drop is reduced to 2.2%. With the exception of *equake* (11.7%), the IPC drop of all benchmarks was measured to be below 10% in this case. Compared to the baseline model *Base 2*, the average IPC drop is only 0.9%, and several benchmarks (*bzip2, gcc, applu* and *mgrid*) actually have performance improvement of up to 5%. Fully-ported set of eight FIFO retention latches brings the performance drop down to 3% on the average compared to *Base 1*, and results in a small performance gain of about 0.2% compared to *Base 2*. As seen from this example and other results presented in Table 2, the number of ports to the retention latches have only marginal impact on the IPCs for the majority of the simulated benchmarks.

Performance can be further improved by increasing the number of latches, although increasing the number of latches beyond 16 can make it problematic to perform the associative lookup of the latches and read the source operand values (in the case of a match) in one cycle. The use of sixteen 16-ported FIFO latches results in 1.9% performance loss compared to *Base 1* and 1.4% performance gain compared to *Base 2*. In this configuration, *wupwise* experienced the largest performance drop among the individual benchmarks - 9.6% compared to *Base 1* and 5.4% compared to *Base 2*.

Further performance improvement is achieved by using the set of retention latches with LRU replacement policy. The motivation here is that the same value stored in one of the latches can be used more than once. According to [5], around 10-15% of the sources are consumed more than once, the fact corroborated by our experiments. One obvious example is the value of a base register used by neighboring load or store instructions or the use of values of the stack pointer and the frame pointer. Table 2 shows the performance of the system with 8 and 16 LRU retention latches with various number of ports. The average performance loss of the system with 8 single-ported LRU latches is 3.1% compared to *Base 1*. Compared to *Base 2*, there is a performance gain of 0.1% on the average. Example of *equake* shows that the LRU scheme does not necessarily outperform FIFO scheme, especially if the number of ports to the latches is limited. This is because some of the values that are consumed only once are retained in the latches by LRU policy at the expense of other potentially usable sources that would have otherwise been kept in the latches if FIFO strategy was used. In general, FIFO latches require larger sizes to cope with the capacity misses whereas eight LRU latches are sufficient to hold frequently used operands in most of the cases. The configuration with sixteen 16-ported LRU latches comes as close as 0.4% in performance to *Base 1* configuration and improves the performance by about 3% on the average with respect to baseline configuration *Base 2*. This provides a measurable improvement over the performance of sixteen 16-ported FIFO latches.

The last two columns of Table 1 show the performance of a configuration that uses retention latches with random replacement policy. Eight fully-ported retention latches with random replacement policy perform 3.5% worse than eight fully-ported LRU latches and 1.6% worse than eight fully-ported FIFO latches. For sixteen fully-ported latches, the random replacement strategy performs 3% worse than LRU and 1.6% worse than FIFO. This makes the use of retention latches with random replacement an unattractive choice.

We also studied the performance effects of simpler retention latch management in the cases of branch mispredictions for LRU latches, where the contents of the entire set of retention latches are flushed on every misprediction instead of selective invalidation of retention latch entries. Figure 8 shows the results for eight fully-ported LRU latches, comparing the organizations, where the contents of the retention latches keeping the results of the instructions executed on a mispredicted path are flushed and the organization with complete flushing of retention latches on every branch misprediction. Recall, that if a complete retention latch flushing is not implemented in the case of LRU latches, it is necessary to flush the contents of the latches selectively. The performance drop due to the complete flushing of the latches is 1.5% on the average. The largest drop among individual benchmarks was observed for *equake* (10.8%) and *gap* (4.8%). Notice that this optimization is not applicable to FIFO latches, because the logic needed to select the most recently-produced result corresponding to an ROB index is still needed and thus no additional logic is required to support branch misprediction recovery if FIFO latches are used.

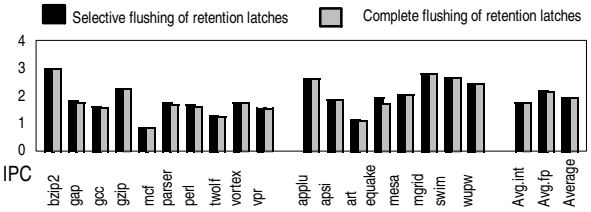


Figure 8. Effects of simplified retention latch management in the case of branch mispredictions (LRU latches)

Finally, we evaluated the impact of reducing the number of ROB ports and using the retention latches on the performance of a more aggressive, 6-way superscalar CPU. For the sake of brevity, we only compare the performance of a 6-way machine against the idealized base case - *Base 1*. The configuration with zero ROB read ports performs 9.8% worse than *Base 1* model. The largest IPC drop was observed for *equake* (24.6%) and *parser* (16%). Notice that the performance of *swim* only decreased by about 11% compared to *Base 1* configuration in contrast to more than 36% drop observed for 4-way machine. The reason for the smaller drop is the larger window size assumed for a 6-way machine that makes stalls at the time of dispatch less frequent if either issue queue or the ROB is full. Simulations of a 6-way machine with the window size identical to that of a 4-way machine showed performance drop in *swim* of about 34% - almost as high as what is observed for a 4-way machine. Indeed, one of the reasons for the performance drop due to the elimination of read ports is the potential ROB saturation, because the oldest in-flight instruction cannot get one of its sources for a large number of cycles. This inevitably causes the ROB to become full and consequently results in pipeline stalls. As instruction window size increases, this problem is somewhat alleviated.

We also studied one FIFO and one LRU configuration for the 6-way machine. In both configurations we used twelve fully-ported retention latches. The average performance degradation with the use of FIFO retention latches was measured as 3.2% with the largest drop for *wupwise* (10.4%) and *equake* (6.6%). The average performance drop with the use of LRU retention latches was measured as 1% with the maximum drop of 5.2% observed for *art* and 2.4% for *apsi*. These results are almost identical to what was achieved for a 4-way processor with eight fully-ported retention latches.

6.2 Power Dissipation and Complexity

Figure 9 shows the percentage of ROB power savings for the simulated benchmarks for various ROB configurations studied in this paper. These results are for a 4-way machine with 96-entry ROB. On the average across all benchmarks, the power savings compared to the fully-ported ROB are 30% for the simplified ROB with no read ports, 23.4% for eight 2-ported FIFO latches, 22.2% for eight 2-ported LRU latches, 21.1% for sixteen 2-ported FIFO latches, and 20.2% for sixteen 2-ported LRU latches. Results are consistent across integer

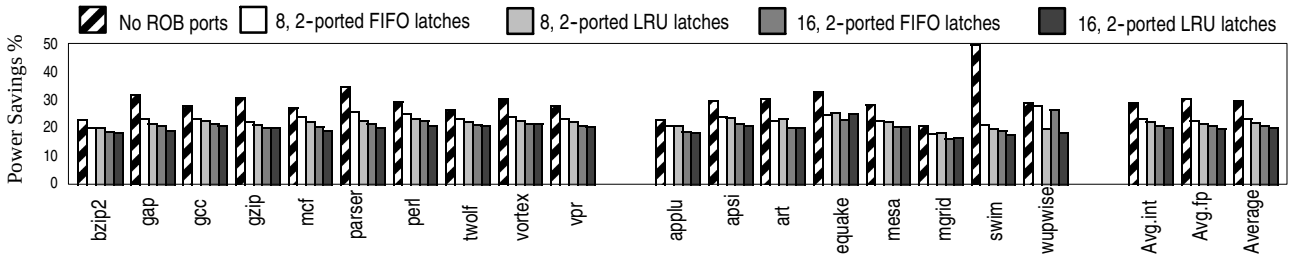


Figure 9. Power savings within the ROB

and floating-point benchmarks. A large power reduction for *swim* in the case with no read ports is explained by the large IPC drop which reduces the number of instructions accessing the ROB per cycle. Power dissipation within the LRU latches can be reduced by the use of low-power comparators that dissipates energy predominantly and only on full matches as suggested in [2] and implemented in [11], but we avoided the evaluation of such optimization to avoid "coloring" our current results.

Power savings are lower when the retention latches are used because of two factors: first, retention latches themselves dissipate some extra power and second, performance increase as a result of using retention latches also leads to higher power dissipation. Nevertheless, noticeable overall power savings are still retained. In terms of total chip power dissipation (recalling that the ROB is about 27% of it), savings are on the order of 6-8% depending on the number of retention latches used.

To get an approximate idea of the reduction in the complexity of the ROB, we estimated the device counts of the baseline ROBs and the simplified ROBs and their retention latches. The device counts for the ROB include the bit arrays, the decoders, the prechargers the sense amps. All devices in the retention latches are counted. For a 96-entry ROB of a 4-way superscalar processor, the device count savings from getting rid of the ROB read ports are in the range of 23% to 26% depending on the number of retention latches used, the number of ports to these latches and the replacement policy used. The ROB area reduction due to the elimination of read ports for reading out the source operand values is about 45%, taking into account the area occupied by the retention latches.

7. RELATED WORK

Lozano and Gao [12] observed that about 90% of the variables in a datapath are short-lived, in the sense that they are exclusively consumed during their residency in the ROB. This seems to imply that most of the dependencies are satisfied in the course of forwarding or reads from the ROB and thus seems to contradict the results presented in Figure 3. However, the percentage presented by Lozano and Gao is for the instances of destination registers. In contrast, we don't keep track of the register instances that are accessed, but instead show in Figure 3 the total number of accesses to the produced register values, irrespective of the instance. For this reason, the percentage of source reads that are satisfied from the ARF is higher than the percentage of long-lived variables as reported by Lozano and Gao. For example, instances of the stack pointer and the global frame pointer are counted as just two instances by Lozano and Gao; in the results of Figure 3, we have tens to hundreds of accesses to these two instances and each such access is accounted for in Figure 3. Another way to look at it is that Lozano and Gao consider the statistics for destination registers, while our analysis is for the source register values; each destination register can be used as a source by multiple instructions and this is precisely the case with most registers that are accessed from the ARF.

The idea of using the retention latches in the context of the ROB is similar in philosophy to forwarding buffer described by Borch et.al. in [4] for a multi-clustered Alpha-like datapath. Our solution and Borch et.al.'s solution both essentially extend the existing forwarding network to increase the number of cycles for which source operands are available without accessing the register file. A set of forwarding buffers retains the results for instructions executed in the last nine cycles. Nine stages of forwarding logic are employed to supply these results to dependent instructions. Borch et.al. further extend this idea by using per-cluster register file caches (CRC - Clustered Register Cache) to move the register file access out of the critical path and replace it with the faster register cache access. Each CRC only stores operands required by instructions assigned to that cluster and operands needed by a dependent instruction that is unlikely to get these operands by other means.

There is a growing body of work that targets the reduction of register file ports. Alternative register file organizations have been explored primarily for reducing the access time (which goes up with the number of ports and registers), particularly in wire-delay dominated circuits [4, 5, 13]. Replicated register files in a clustered organization have been used in the Alpha 21264 processor [10] to reduce the number of ports in each replica and also to reduce delays in the connections in-between a function unit group and its associated register file.

While replicated register files [10] or multi-banked register files with dissimilar banks (as proposed in [5], organized as a two-level structure - cached RF - or as a single-level structure, with dissimilar components) are used to reduce the register file complexity, additional logic is needed to maintain coherence in the copies or to manage/implement the allocation/deallocation of registers in the dissimilar banks. A recent work at reducing the complexity of the physical register file uses a two-level implementation, along with multiple register banks [3]. The complexity reduction comes from the use of banks with a single read port and a single write port in each bank. The two-level structure allows the first level (banked) register file to be kept small (and fast), with the higher speed compensating for IPC drops arising from limiting the number of ports. In [3], additional work attempting a reduction in the complexity of the register file is also described, including solutions used in VLIW datapaths and novel transport-triggered CPUs.

The idea of caching recently produced values was also used in [8] (hereafter called "the VAB scheme"). At the time of instruction writeback, FUs write results into a cache called Value Aging Buffer (VAB). The register file, holding both speculative and committed register values, is updated only when entries were evicted from the VAB. Furthermore, when the required value is not in the VAB, a read of the register file is needed, requiring at least some read ports for reading the source operands. Unless a sufficient number of register file ports is available or the number of entries in the VAB is sufficiently large, the performance degradation can be considerable. In addition, in the VAB scheme, the multi-cycle register file access is still an intimate part of the issue process. In contrast to this, if the required

value is not found in the retention latches or in the ARF in our scheme, we still do not read the ROB, thereby eliminating any read ports on the ROB for reading source operands. By separating the ARF from the ROB, we satisfy a large percentage of the dependencies from forwarding, the retention latches, and the ARF, requiring only a small percentage of sources to be obtained from the ROB. Deferring these reads does not have any significant impact on performance. Further, in our scheme, the register file (ROB) access does not form a component of a critical path.

Some other differences between the VAB scheme and the proposed technique are as follows. Since the recent results may not necessarily have been found in the register file, misprediction handling and interrupt handling with the VAB were somewhat involved; misprediction and interrupt handling required selective lookup of values from the VAB for generating a precise state. In our scheme, results are written *to both* the retention latches and the ROB; we simply clear the entire set of the retention latches or invalidate their contents selectively on a branch misprediction. In fact, as shown in Figure 8, clearing all of the retention latches on a misprediction, which is much more simpler to implement, produces almost the same level of performance as the selective clearing scheme. Unlike the VAB scheme, there is no need to copy anything from the retention latches into the ROB. On a miss to the VAB, the necessary accesses to the large (integrated) register file can dissipate considerable energy. Such dissipations are eliminated in our scheme; we only access the relatively small ARF and the retention latches.

To the best of our knowledge, the only work that has addressed ROB power minimization directly attempts to reduce the ROB power by using a multi-segmented organization, where segments are deactivated dynamically if they are not used [14]. The scheme of [14] is orthogonal to our proposed technique and can be used in conjunction with our scheme to achieve additional power savings.

8. CONCLUDING REMARKS

The reorder buffer in some modern superscalar processors is a complex multi-ported structure with multi-cycle access time. Much of this complexity stems from the need to read and write source operand values and to commit these values to the architectural register file. We proposed a scheme to eliminate the ports needed for reading the source operand values for dispatched instructions. Our approach for eliminating the source read operand ports on the ROB capitalizes on the observation that only about 5% or so of the source operand reads take place from the ROB. Any performance loss due to the use of the simplified ROB structure was compensated for by using a set of retention latches to cache a few recently-written values and allowing the bulk of the source operand reads to be satisfied from those latches, from forwarding and from reads of the architectural registers. Our technique also removes the multi-cycle ROB access for source operands read from the critical path and substitutes it with the faster access to the retention latches. As a result, we have an overall performance improvement of up to 3% (depending on the latch management strategy and the number of latches used) on the average across the simulated benchmarks. For the configurations studied, we achieved a reduction of about 25% in the device counts of the ROB, the ROB area reduction of about 45% and the overall chip power reduction of 6-8%.

In conclusion, our organization seems quite attractive as it simultaneously lowers the device count, operand access time and power dissipation in the ROB with a slight gain in performance.

9. ACKNOWLEDGMENTS

We thank Oguz Ergin for helping us with the datapath power estimation. We also thank anonymous reviewers for their valuable comments. This work is supported in part by DARPA through contract number FC 306020020525 under the PAC-C program, the NSF through award no. MIP 9504767 & EIA 9911099, and by IEEC at SUNY-Binghamton.

10. REFERENCES

[1] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases (through version 3.0).

[2] Brooks, D.M., Bose, P., Schuster, S.E. et. al., "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors", IEEE Micro Magazine, 20(6), Nov./Dec. 2000, pp. 26-43.

[3] Balasubramonian, R., Dwarkadas, S., Albonesi, D., "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", in Proc. of the 34th Int'l. Symposium on Microarchitecture (MICRO-34), 2001.

[4] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips", in Proceedings of Int'l. Conference on High Performance Computer Architecture (HPCA-02), 2002.

[5] Cruz, J-L. et. al., "Multiple-Banked Register File Architecture", in Proceedings 27th Int'l. Symposium on Computer Architecture, 2000, pp. 316-325.

[6] Folegnani, D., Gonzalez, A., "Energy-Effective Issue Logic", in Proceedings of Int'l. Symp. on Computer Architecture, July 2001.

[7] Gwennap, L., "PA-8000 Combines Complexity and Speed", Microprocessor Report, vol 8., N 15, 1994.

[8] Hu, Z. and Martonosi, M., "Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics", in Workshop on Complexity-Effective Design, 2000.

[9] Intel Corporation, "The Intel Architecture Software Developers Manual", 1999.

[10] Kessler, R.E., "The Alpha 21264 Microprocessor", IEEE Micro, 19(2) (March 1999), pp. 24-36.

[11] Kucuk, G., Ponomarev, D., Ghose, K., and Kogge, P.M., "Energy-Efficient Instruction Dispatch Buffer Design", in Int'l. Symp. on Low Power Electronics and Design (ISLPED'01), August 2001.

[12] Lozano, G. and Gao, G., "Exploiting Short-Lived Variables in Superscalar Processors", in Proceedings of Int'l Symposium on Microarchitecture, 1995, pp. 292-302.

[13] Llosa, J. et.al., "Non-consistent Dual Register Files to Reduce Register Pressure", in Proceedings of HPCA, 1995, pp. 22-31.

[14] Ponomarev, D., Kucuk, G., Ghose, K., "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources", in Proc. of the 34th Int'l. Symposium on Microarchitecture (MICRO-34), December 2001.

[15] Ponomarev, D., Kucuk, G., and Ghose, K., "AccuPower: an Accurate Power Estimation Tool for Superscalar Microprocessors", in Proc. of 5th Design, Automation and Test in Europe Conference (DATE-02), March, 2002.

[16] Slater, M., "AMD's K5 Designed to Outrun Pentium", Microprocessor Report, vol.8, N 14, 1994.

[17] Smith, J. and Pleszkun, A., "Implementation of Precise Interrupts in Pipelined Processors", in Proc. of Int'l. Symposium on Computer Architecture, pp.36-44, 1985.

[18] Standard Performance Evaluation Corporation, "Spec2000", 2000. http://www.spec.org.