Appears in the Proceedings of the International Conference on Computer Design, 2003.

Distributed Reorder Buffer Schemes for Low Power

Gurhan Kucuk   Oguz Ergin   Dmitry Ponomarev   Kanad Ghose

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

Abstract

We consider several approaches for reducing the complexity and power dissipation in processors that use separate register file to maintain the commited register values. The first approach relies on a distributed implementation of the Reorder Buffer (ROB) that spreads the centralized ROB structure across the function units (FUs), with each distributed component sized to match the FU workload and with one write port and two read ports on each component. The second approach combines the use of previously proposed retention latches and a distributed ROB implementation that uses minimally ported distributed components. The second approach avoids any read and write port conflicts on the distributed ROB components (with the exception of possible port conflicts in the course of commitment) and does not incur the associated performance degradation. Our designs are evaluated using the simulation of the SPEC 2000 benchmarks and SPICE simulations of the actual ROB layouts in 0.18 micron process.

1. Introduction

Power dissipation was traditionally considered a major constraint in the domain of mobile and embedded devices, including notebook computers, PDAs, palmtops, cellular phones, and pagers where the portability requirements place severe restrictions on the size, weight and especially the battery life. In contrast to the embedded devices, high performance systems and desktop environments in general enjoy unlimited power supply from the external sources and power dissipation only affects system reliability and the design of cooling artifacts because the energy consumed by a microprocessor is converted into heat, which must be removed from the surface of a processor die. Up until recently this has not been a significant concern because large packages, fans, and heat sinks were capable of dissipating the generated heat. However, as smaller and thinner enclosures are used for packaging, and density and size of the chips continue to increase due to the use of smaller transistors and never–ending architectural innovations, it is becoming increasingly difficult to design cooling facilities without incurring significant costs. For high–performance processors, the costs of cooling solutions are rising at $1–$3 or more per watt of dissipated power [18]. The exponential rise in the power density means that the cooling costs are also rising exponentially, threatening to become a major impediment to the deployment of new systems.

Furthermore, the areal energy density distribution across a typical chip is highly skewed, being lower over the on–chip caches and significantly higher in components of the dynamic instruction scheduling logic and within the register renaming units, where a large amount of power is dissipated in the disproportionately small area. High operating temperatures, especially within the hot spots, significantly reduce lifetime and reliability of the integrated circuits because several silicon failure mechanisms, such as electromigration, junction fatigue, and gate dielectric breakdown are exacerbated at high temperatures [19]. Consequently, power and temperature management is becoming one of the most important design constraints even for high–performance systems.

Most of today's high–performance processors are implemented using dynamic out–of–order superscalar designs with heavy reliance on aggressive branch prediction and the use of large instruction windows to maximize the exploitation of the instruction–level parallelism in the sequential programs. A reorder buffer (ROB) is one of the key datapath structures in such datapath designs. Conventionally, the ROB is used to reconstruct a *precise state* – a state corresponding to the sequential program semantics that requires the processor state to be updated strictly in program order – when interrupts, exceptions or branch mispredictions occur. While the physical registers are updated as and when instructions complete, possibly out of program order, the ROB maintains results and updates the precise state strictly in program order. When a branch misprediction occurs, the instructions on the mispredicted path are squashed from the ROB, preventing the update of the architectural registers and maintaining the precise state [16].

In some microarchitectures, among them the Intel P6, the physical registers are implemented as slots within the ROB entries and the separate architectural register file (ARF) is used to implement the ISA registers that embody the precise state. An example of a processor that uses this specific datapath is depicted in Figure 1. Bold lines shown in this and subsequent datapath diagrams correspond to buses (or multiple sets of connections). At the time of writeback, 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). 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. To support the back–to–back execution of dependent instruction, the result tags are broadcast prior to the actual data, as soon as the producers are selected for execution.

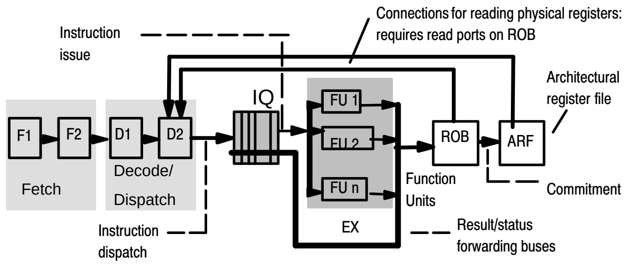
Instruction issue
Connections for reading physical registers: requires read ports on ROB
Architectural register file
IQ
F1
F2
D1
D2
FU 1
FU 2
FU n
ROB
ARF
Fetch
Decode/ Dispatch
Function Units
Commitment
Instruction dispatch
EX
Result/status forwarding buses

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

The typical implementation of a ROB is in the form of a multi–ported register file (RF). In a W–way superscalar machine where the physical registers are integrated within the ROB, the ROB has the following ports:

1) At least 2*W read ports for reading source operands for each of the W instructions dispatched/issued per cycle, assuming that each instruction can have up to 2 source operands.

2) At least W write ports to allow up to W FUs to write their result into the ROB slots acting as physical registers in one cycle.

3) At least W read ports to allow up to W results to be retired into the ARF per cycle.

4) At least W write ports for establishing the ROB entries for co–dispatched instructions.

The large number of ports on the ROB results in two forms of penalties that are of significance in modern designs. The first penalty is in the form of the large access delay that can place the ROB access on the critical path; the second is in the form of higher energy/power dissipations within the highly multi–ported RFs implementing the ROB. Since the ROB in a P6–style design can contribute a significant percentage to the overall chip power dissipation [6], it is particularly important to consider the strategies for reducing the ROB power requirements without sacrificing processor’s performance.

This paper proposes a number of alternative ways of simplifying the ROB structure and reducing its power dissipation through:

1) Implementing the ROB in a distributed manner that allows much smaller, distributed components to be used.

2) Reducing the number of ports on each distributed component.

Our earlier study [11] exploited the observation that in a typical superscalar datapath, almost all of the source operand values are obtained either through data forwarding of the recently generated results or from the reads of the committed register values. Only a small percentage of operands, which are not generated recently, need to be read from the ROB; delaying such reads have little or no impact on the overall performance. This allowed the 2*W ports for reading source operands at the time of dispatch or issue to be completely eliminated with small impact on performance [11].

In this study, we extend the approach of [11] to further reduce the overall complexity, power and delay of the ROB by implementing a centralized ROB as a distributed structure. Each distributed component of the ROB, assigned to a *single* FU, has a single write port for committing the result as opposed to W ports that are needed in a centralized implementation. Unlike recently proposed approaches that simplify a multi–ported RF by decomposing it into identical components, our ROB components are not homogeneous in size. The advantages of our scheme are as follows:

1) The size of each component can be tailored to the FU to which it is assigned.

2) Since a FU can only write at most a single result per cycle, a single write port on each component will suffice.

3) A single read port can be used on each component to commit result from the ROB component to the ARF. A simple round robin style instruction allocation within an identical set of FUs (such as a pool of integer units) can be used to maximize the probability that the instructions from different ROB components retire in a common cycle.

4) If the distribution of the ROB is used in conjunction with the scheme of [11], no read ports for reading the source operands values from the ROB are retained. This avoids the use of port arbitration logic, which is inherent in all previously proposed multi–banked schemes with reduced number of ports in each bank.

The rest of the paper is organized as follows. In Section 2 we describe the details of the proposed techniques for reducing the complexity and power dissipation of the ROB. Our evaluation methodology is presented in Section 3 followed by the discussion of the experimental results in Section 4. Section 5 describes the related work and we conclude in Section 6.

2. Reduced complexity ROB alternatives

In this section, we show a number of alternative designs that progressively reduce the ROB complexity.

2.1. Fully distributed ROB

Figure 2 depicts a datapath that implements a centralized ROB in a distributed manner with one ROB component assigned to each FU. The only portion of the ROB that remains centralized is a FIFO structure that maintains pointers to the entries within the distributed components. This centralized structure has W write ports for establishing the entries in dispatch (program) order and W read ports for reading the pointers to the entries established in the distributed ROB components (ROBCs). An alternative is to implement this structure as a (wider) register file with a single read port and a single write port, with each entry capable of holding W pointers into the ROBCs. Each ROBC can be implemented as a register file, where a FIFO list is maintained. The pointer in the centralized ROB entry is thus a pair of numbers: a FU id and an offset within the associated ROBC. The ROBCs associated with FUs 1 through n in Figure 2 are shown as register files RF_1 through RF_n.

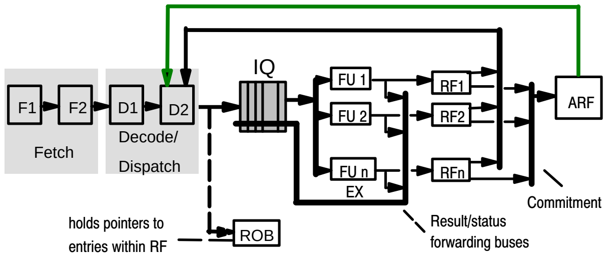

Figure 2. Superscalar datapath with fully distributed ROB.

The implementation of the speculative register storage in a structure separate from the centralized ROB is similar in spirit to the use of the rename buffers [15]. In this paper, without loss of generality, we call such a design “distributed ROB” and compare our proposed design with the baseline model that assumes that the speculative results are stored directly within the ROB slots. However, the proposed techniques are trivially applicable to any other datapath that uses separate register file for committed values, such as the datapath with the rename buffers as implemented in PowerPC 620 [15].

The process of instruction retirement now requires an indirection through the centralized ROB by following the pointers to the ROBC entries that have to be committed. This indirection delay is accommodated by using an extra commit stage in the pipeline that has a negligible impact on the overall IPC – less than 0.5% on the average across the simulated benchmarks.

As shown in Figure 2, each ROBC has a single write port that allows the result produced by the associated FU to be written without the need for any arbitration. However, since the writeback bandwidth is limited by the number of forwarding buses used to route the produced results to the waiting instructions in the issue queue, the write to the ROBC only occurs when the associated FU acquires a forwarding bus. Each ROBC has two read ports – one read port for reading source operands and another for committing results. The dispatch of an instruction is delayed if both of its operands come from the same ROBC. Likewise, if more instructions require their sources to come from the same ROBC, the dispatch of those instructions is delayed. This scheme thus requires the port arbitration mechanisms to arbitrate for the limited number of read ports on the ROBCs.

In a similar way, the instruction retirement is also blocked, if the results to be committed reside in the same ROBC. According to our simulations, the percentage of cycles when commitment blocks in this fashion is only 5.5% on the average across all the executed benchmarks, resulting in only 0.1% IPC drop on the average if a single read port for commitment is maintained on each ROBC. The size of each ROBC can be tuned to match the characteristics of its associated FU’s workload. Detailed results pertaining to the optimal size of each ROBC are presented later.

The latency of the branch misprediction recovery does not increase with the introduction of the ROB distribution. To reconstruct the correct state of the ROBCs, the tail pointer of each ROBC will have to be updated on a misprediction, along with the main ROB tail pointer. This can be done, for example, by checkpointing the tail pointers of all individual ROBCs for each branch.

2.2. Eliminating source operand read ports on the baseline ROB

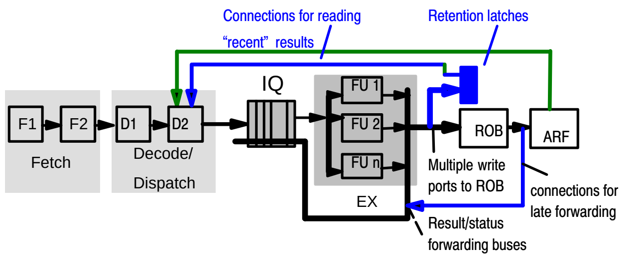

Figure 3. Superscalar datapath with the simplified ROB and retention latches.

In a recent work [11] we proposed to reduce the ROB complexity and its associated power dissipation by completely eliminating the 2*W read ports needed for reading the source operands from the ROB. The technique was motivated by the observation that only a small fraction of the source operand reads require the values to come from the ROB. For example, for a 4–way machine with a 72–entry ROB, about 62% of the operands are obtained through the forwarding network and about 32% of the operands are read from the ARF. Only about 6% of the sources are obtained from the ROB reads [11]. Exploiting the fact that so many ROB ports are used to supply so few operands, we. proposed the use of a ROB structure without any read ports for reading the source operand values. Consequently, 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. 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, the value was simply forwarded (again) on the forwarding buses at the time it was committed. To avoid the need to perform this late forwarding for each and every committed result, only the values that were actually sought from the ROB were forwarded for the second time. As a result, reasonable performance was sustained without increasing the number of forwarding buses.

As a consequence of the elimination of the ROB read ports, the execution of some instructions is delayed, resulting in some performance degradation. To compensate for most of this performance loss, we used a set of latches, called retention latches (RLs), to cache a few recently produced results. The datapath that incorporates the RLs is shown in Figure 3. Instructions dispatched since the writing of a result to the ROB could still access the value as long as they could get it from the RLs. The ROB index of an instruction that produced the value was used as a key for associative searches in the RLs. In case the lookup for source operand failed to locate the data within the RLs, the operand value was eventually obtained through the late forwarding of the same value when it was committed to the ARF, similar to what is described in [20]. Complete details of the scheme, including the various RL management strategies and various branch misprediction handling approaches are presented in [11]. An important optimization to the retention latch management was introduced in [12], where the results values are selectively cached in the retention latches based on the possibility of their future use. This optimization can also be used in conjunction with the scheme that we propose in this paper.

2.3. Using retention latches with a distributed ROB

The distributed ROB implementation shown in Figure 2 can be further simplified through the incorporation of a set of retention latches – the read ports used for reading operands from the ROBCs, RF_1 through RF_n, can now be completely eliminated. The resulting datapath is shown in Figure 4.

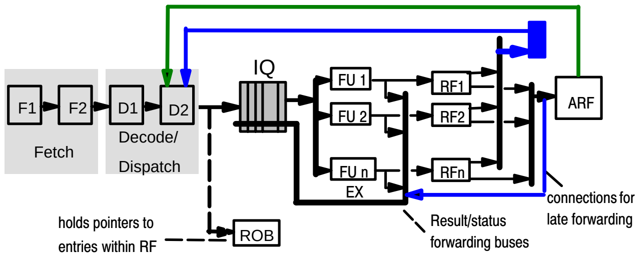

Figure 4. Superscalar datapath with the distributed ROB and retention latches.

Dispatched instructions are allocated to the appropriate type of FUs. Where multiple instances of the required type of FU exist, instructions are allocated within the identical instances in a round robin fashion thus distributing the load on the ROBCs and minimizing the potential port conflicts in the course commitment. When a FU j completes, it writes the relevant entry within its ROBC, RF j, using the dedicated write port. Simultaneously, the result is forwarded on the result/status bus. Late forwarding uses these same forwarding connections when results are committed from the ROBCs (i.e., RF_1 through RF_n in Figure 4) to the ARF using the only read port on these components. As in the case of a centralized ROB without source read ports and retention latches, a common set of W forwarding buses is able to handle normal and late forwarding in a W–way machine.

This design allows for the use of a fully distributed ROB where the conflicts over both read and write ports on the ROB are avoided, with the exception of rare occasions of port conflicts during commitments. The read ports needed on the ROB for reading out the sources for instruction issue are completely eliminated by using the technique of [11]. As each FU is assigned to a dedicated ROBC, all produced results can be written into speculative storage within the ROBCs with no need for write port arbitration (although the arbitration logic for the use of the forwarding buses is still necessary, just like it is in the baseline machine).

3. Evaluation framework and methodology

The widely–used Simplescalar simulator [1] was substantially modified to implement realistic models for such datapath components as the ROB (integrating physical register file), the issue queue, and the rename table. The studied configuration of superscalar processor is shown in Table 1.

We simulated the execution of 8 integer SPEC 2000 benchmarks *(gap*, *gcc*, *gzip*, *parser*, *perlbmk*, *twolf*, *vortex* and *vpr*) and 6 floating point SPEC 2000 benchmarks (*applu*, *art*, *mesa*, *mgrid*, *swim* and *wupwise*). 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 estimating the energy/power of the ROB, the ROBC and the retention latches, 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, the ROBC, 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. A V_{dd} of 1.8 volts was assumed for all the measurements.

Table 1. Configuration of a simulated processor

Parameter	Configuration
Machine Width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry IQ, 96 entry ROB (double results hold two entries), 32 entry LSQ
FUs 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)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 2 cycles hit
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit
L2 Cache combined	512 KB, 4–way set–associative, 128 byte line, 4 cycles hit
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 assoc., 30 cycles miss

4. Results and discussions

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

4.1. Evaluation of the distributed ROB

In order to sufficiently size each of the ROBCs, we first recorded the maximum number of ROBC entries that are simultaneously in use (allocated to hold the results of in–flight instructions) by each functional unit, provided that the number of entries in the ROBCs is unlimited and the instruction window is constrained by the sizes of the ROB and the issue queue. The results of these experiments with unlimited number of the ROBC entries are shown in the leftmost part of Table 2. As seen from these results, even the maximum demands on the number of the ROBC entries are quite modest across the simulated benchmarks. Considerable variations in the number of used ROBC entries are observed for the MUL FUs, both integer and floating point. The ROBCs for integer and floating point ADD and LOAD FUs are utilized fairly uniformly across the benchmarks. Notice that in Table 2, a single column is used to represent four integer ADD units and four floating point ADD units. This is because the presented statistics is identical for all four individual functional units due of the round–robin allocation of instructions.

Table 2. The max. number of entries used within each ROBC and the percentage of blocked cycles

	Maximum demands on the number of ROBC entries					Percentage of cycles when dispatch blocks for 8_4_4_4_16 configuration				
FU type	Int. Add	Int. Mul/ Div	FP Add	FP Mul/ Div	Load	Int. Add	Int. Mul/ Div	FP Add	FP Mul/ Div	Load
gap	17	8	0	0	28	0.6	0.3	0	0	3.0
gcc	20	4	1	2	30	0.6	0	0	0	2.2
gzip	19	2	0	0	30	1.5	0	0	0	0.9
parser	15	3	0	0	28	1.4	0	0	0	5.5
perl	17	6	4	12	28	0.1	0.1	0	0	5.1
twolf	16	4	3	5	30	0.5	0	0	0	14.0
vortex	16	4	0	0	28	0	0	0	0	1.5
vpr	15	2	5	11	27	2.6	0	0	0.3	9.2
applu	16	0	2	5	22	1.0	0	0	0	0.1
art	12	0	8	16	26	0.1	0	0	0.4	5.2
mesa	12	10	6	11	26	0.4	1.4	0.6	4.4	2.2
mgrid	10	8	2	2	30	0	4.8	0	0	2.7
swim	20	1	2	2	15	4.8	0	0	0	0
wupw.	15	0	3	4	22	2.6	0	0	0	0.9
Int avg.	16.9	4.1	1.6	3.8	28.6	0.9	0.1	0	0	5.2
FP avg.	14.2	3.2	3.8	6.7	23.5	1.5	1.0	0.1	0.8	1.9
Average	15.7	3.7	2.6	5.0	26.4	1.2	0.5	0	0.4	3.8

The averages from Table 2 were then used to determine the number of entries in each ROBC in the following manner. Based on the results of Table 2, the initial values for the sizes of each ROBC were derived from the averages presented in Table 2 by taking the corresponding averages from the last row of Table 2 and rounding them to the nearest power of two. In an attempt to keep the sizes of the ROBCs to the minimum and ensure that the total size of the ROBCs does not increase the size of the centralized ROB, the exception was made for the ROBCs of the integer ADD units, whose initial sizes were set as 8 entries each. Specifically, 8 entries were allocated for each of the ROBCs dedicated to integer ADD units, 4 entries were allocated for the ROBC of integer MUL/DIV unit, 4 entries were used for the ROBCs of floating–point ADD units, 4 entries were used for floating point MUL/DIV unit and 16 entries were allocated for the ROBC serving the LOAD functional unit to keep the result values coming out of the data cache. In the subsequent discussions, the notation "x_y_z_u_v" will be used to define the number of entries used in various ROBCs, where x, y, z, u, and v denote the number of entries allocated to the ROBCs for each of the integer ADD units, integer MUL unit, floating point ADD units, floating point MUL unit and the LOAD unit, respectively. Using this notation, the initial configuration of the ROBCs is described as "8_4_4_4_16".

To gauge how reasonable was our choice for sizing the various ROBCs, we measured the percentage of cycles when instruction dispatching blocked because of the absence of a free ROBC entry for the dispatched instruction. The rightmost part of Table 2 indicates that the number of the ROBC entries allocated for integer ADD, integer MUL/DIV, floating–point MUL/DIV and LOAD functional units could be increased to reduce the number of dispatch stalls. As a consequence of these stalls and the conflicts over the read ports of the ROBCs, the average performance across all benchmarks degrades by 4.8% in terms of committed IPCs, as shown in Figure 5. The floating–point benchmarks are more effected by this with the average IPC degradation of 6.5%. Integer codes experience a loss of only 2.5%.

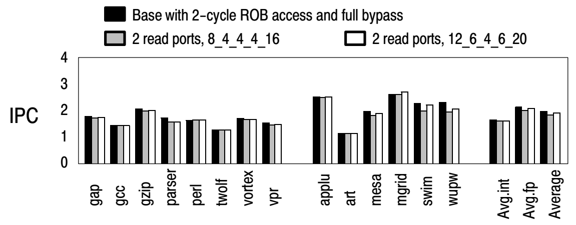

Figure 5. IPCs of 8_4_4_4_16 and 12_6_4_6_20 configurations of the ROBCs compared to the baseline model. 2 read ports are used for each ROBC – one port is used for source operand reads and the other port is used for commits. Access to the baseline ROB is assumed to take 2 CPU cycles. Full bypass network is assumed.

To improve the performance, consistent with the observations from Table 2 and Figure 5, we increased the sizes of all ROBCs with the exception of the ROBCs dedicated to floating–point ADD units. Specifically, the ROBC configuration "12_6_4_6_20" was considered. The average performance degradation across all simulated benchmarks was measured as 2.4% with the average drop for floating point benchmarks reduced to 3.2%. A small performance improvement of 3.8% was observed for *mgrid* due to the faster access time to the source operands (2 cycles in the centralized ROB scheme vs. 1 cycle with the distributed latches).

4.2. Evaluation of the distributed ROB with retention latches

We now evaluate the effects of integrating the set of retention latches into the datapath with ROBCs. To minimize the design complexity, the simplest configuration of eight 2–ported FIFO retention latches was used in our experiments. Recall that by themselves, this combination of retention latches only degrades the performance of the baseline datapath by about 0.2% on the average. Figure 6 shows the IPCs of the processor that uses this small set of retention latches in conjunction with "12_6_4_6_20" ROBCs. The average performance drop is reduced to 1.7% and each of the ROBCs only has a single port that is still needed for commitment. The performance of floating–point benchmarks improved slightly. Performance degradation within integer benchmarks is 1.5% on the average compared to the baseline organization.

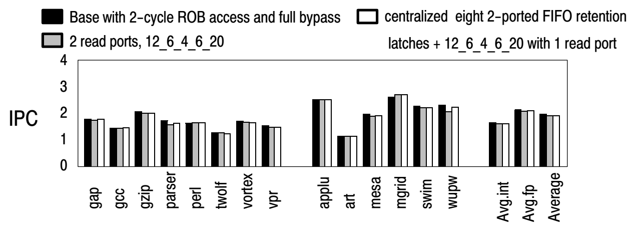

Figure 6. IPC of the "12_6_4_6_20" ROBC configuration with retention latches compared to baseline configuration and the scheme with distributed ROB

4.3. Implications on ROB power dissipations

Figure 7 shows the power savings achieved within the ROB by using the techniques proposed in this paper. The configuration with just the retention latches results in 23% of ROB power savings on the average [11]. The ROB distribution results in more than 49% reduction in energy dissipations within the ROB. The savings are attributed to the use of much shorter bitlines and word–select lines and the absence of sense amps and prechargers for the small ROBCs. If the retention latches are integrated into the datapath with the ROBCs, the power dissipation is actually slightly increases (47% power savings) for two reasons. First, the retention latches introduce the extra source of power dissipation, and second, the performance increases with the use of the retention latches, resulting is slightly higher energy dissipation per cycle. But complexity is still reduced, because the read ports and connections from all ROBCs are replaced by a small centralized buffer with just a few ports.

5. Related work

There is a growing body of work that targets the reduction of register file ports. Alternative register file

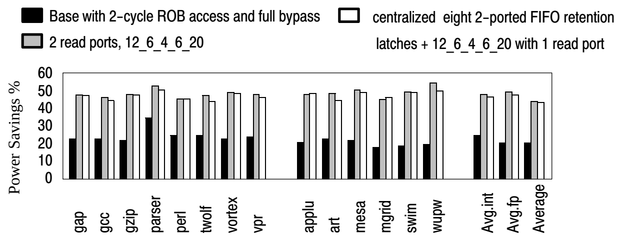

Figure 7. Power savings within the ROB

organizations have been explored primarily for reducing the access time and energy, particularly in wire–delay dominated circuits [4, 5, 13, 14, 17]. Replicated [10] and distributed [21, 22] register files in a clustered organization have been used to reduce the number of ports in each partition 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 way of reducing the complexity of the physical register file by using a two–level implementation, along with multiple register banks is described by Balasubramonian et.al. in [3] for a datapath that integrates physical register file and ARF in a structure separate from the ROB. The complexity and power reduction comes from the use of banks with a single read port and a single write port in each bank, thus resulting in a minimally–ported design. The simplified ROB designs presented in this paper result in smaller IPC drop compared to the results of [3], mainly because the conflicts over the write ports are not introduced and the conflicts over the read ports are eliminated by using retention latches.

In [13], Tseng and Asanovic proposed a multi–banked register file design that avoids complex arbitration logic by using a separate pipeline stage (following the selection and preceding the register file read), which detects bank conflicts and reschedules the instructions that are the victims of the bank conflicts, as well as the prematurely issued dependents of such instructions. Our scheme involves no such complications as *all* port conflicts are eliminated.

In [17], the conflicts over write ports in a multi–banked register file are avoided by delaying the physical register allocation until the time of the actual instruction writeback. The scheme incurs considerable additional complexity because of the need to manage virtual tags used to maintain data dependencies.

In [25], the number of register file read ports is reduced by using a bypass hint. The speculative nature of the bypass hint results in some performance loss caused by the need to stall the instruction issue on bypass hint mispredictions. In [26], the peak read port requirements are reduced by prefetching the operands into an operand pre–fetch buffer.

In [23], Savransky, Ronen and Gonzalez proposed a mechanism to avoid useless commits in the datapath that uses the ROB slots to implement physical registers. Their scheme delays the copy from the ROB to the architectural register file until the ROB slot is reused for a subsequent instruction. In many cases, the register represented by this slot is invalidated by a newly retired instruction before it is needed to be copied. Such a scheme avoids about 75% of commits, thus saving energy. An alternative way to reduce the number of data movements in the course of commitments and writebacks was proposed in [24], where we exploited the observation that the majority of the generated result values are short–lived. We stored such short–lived results in a small dedicated register file and avoided their writeback to the ROB and commitment to the ARF in most of the cases. Both schemes can be used in conjunction with the techniques proposed in this paper to achieve further power savings.

6. Concluding remarks

This paper described several techniques to reduce the complexity and the power dissipation of the ROB. We introduced the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit. All conflicts over the read ports are eliminated by removing the read ports for reading out the source operand values from the distributed ROB completely and using the combination of a small set of associatively–addressed retention latches and late result forwarding to supply the results to the waiting instructions in the issue queue.

Our designs result in extremely low performance degradation of 1.7% on the average across the simulated SPEC 2000 and significant reduction in the ROB complexity and power dissipation. On the average, the ROB power savings of as high as 49% are realized. Taking into account that the ROB, as used in the considered datapath style, can dissipate as much as 27% of the total chip energy [6], our techniques result in about 13% of total CPU power reduction with only 1.7% average drop in performance.

7. Acknowledgements

This work was 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.

8. 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., “Low Complexity Reorder Buffer Architecture”, in Proc. of Int’l. Conference on Supercomputing, June, 2002, pp.57–66.

[12] Kucuk, G., Ponomarev, D., Ergin, O., Ghose, K., “Reducing Reorder Buffer Complexity Through Selective Operand Caching”, in Proc. of ISLPED, 2003.

[13] Tseng, J., Asanovic, K., “Banked Multiported Register Files for High Frequency Superscalar Microprocessors”, in Proc. ISCA, 2003

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

[15] Levitan, D., Thomas, T., Tu, P., “The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor”, CompCon, 1995.

[16] 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.

[17] Wallase, S., Bagherzadeh, N., “A Scalable Register File Architecture for Dynamically Scheduled Processors”, in Proc. PACT–96.

[18] Gunther, S., Binns, F., Carmean, D., Hall, J., “Managing the Impact of Increasing Microprocessor Power Consumption”, *Intel Technology Journal,* Q1, 2001.

[19] Small, C., “Shrinking Devices Put a Squeeze on System Packaging”, EDN, Vol, 39, N4, pp. 41–46, Feb.17, 1994.

[20] Sohi, G., Vajapeyam, S., “Instruction Issue Logic for High–Performance, Interruptable Pipelined Processors”, in Proceedings of ISCA, 1987.

[21] Canal, R., Parserisa, J.M., Gonzalez, A., “Dynamic Cluster Assignment Mechanisms”, in Proceedings of HPCA, 2000.

[22] Farkas, K., Chow, P., Jouppi, N., Vranesic, Z., “The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, in Proceedings of Int’l. Symp. on Microarchitecture, 1997.

[23] Savransky, G., Ronen, R., Gonzalez, A., “Lazy Retirement: A Power Aware Register Management Mechanism”, in Workshop on Complexity–Effective Design, 2002.

[24] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., “Reducing Datapath Energy Through the Isolation of Short–Lived Operands”, in Proceedings of PACT, 2003.

[25] Park, Il., Powell, M., Vijaykumar, T., “Reducing Register Ports for Higher Speed and Lower Energy”, in Proc. of the 35th International Symposium on Microarchitecture, 2002.

[26] Kim, N., Mudge, T., “Reducing Register Ports Using Delayed Write–Back Queues and Operand Pre–Fetch”, in Proc. of Int’l Conference on Supercomputing, 2003.