Appears in the Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), 2003

Reducing Reorder Buffer Complexity Through Selective Operand Caching

Gurhan Kucuk Dmitry Ponomarev Oguz Ergin Kanad Ghose

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

ABSTRACT

Modern superscalar processors implement precise interrupts by using the Reorder Buffer (ROB). In some microarchitectures , such as the Intel P6, the ROB also serves as a repository for the uncommitted results. In these designs, the ROB is a complex multi–ported structure that dissipates a significant percentage of the overall chip power. Recently, a mechanism was introduced for reducing the ROB complexity and its power dissipation through the complete elimination of read ports for reading out source operands. The resulting performance degradation is countered by caching the most recently produced results in a small set of associatively–addressed latches ("retention latches"). We propose an enhancement to the above technique by leveraging the notion of short–lived operands (values targeting the registers that are renamed by the time the instruction producing the value reaches the writeback stage). As much as 87% of all generated values are short lived for the SPEC 2000 benchmarks. Significant improvements in the utilization of retention latches, the overall performance, complexity and power are achieved by not caching short–lived values in the retention latches. As few as two retention latches allow all source operand read ports on the ROB to be completely eliminated with very little impact on performance.

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, short–lived values

ISLPED'03, August 25–27, 2003, Seoul, Korea

1. INTRODUCTION

Contemporary superscalar microprocessors make use of aggressive branch prediction, register renaming and out–of–order instruction execution to push the performance. To recover from possible branch misspeculations, the processors typically employ a FIFO circular queue, called the Reorder Buffer (ROB), which is used to maintain information about all in–flight instructions in the program order. Such an arrangement allows the instructions on a mispredicted path to be easily identified and squashed.

Register renaming is by far the most popular technique for handling false data dependencies. In some of the register renaming implementations, for example in Intel's P6 microarchitecture, the ROB slots also serve as repositories for non–committed register values. Such a datapath is shown in Figure 1. Here, the (speculatively) generated results are first written into the ROB and eventually committed to the Architectural Register File (ARF) in program order at the time of instruction retirement.

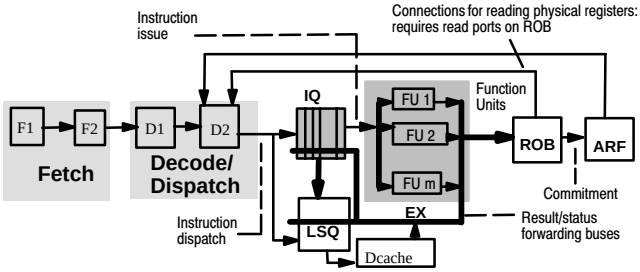

Figure 1. A P6–like superscalar datapath

The ROB, as used in a P6–style datapath [8], is implemented as a multi–ported register file. A large number of ports is needed on the ROB to support the writebacks of multiple results as well as reading the data in the course of operand reads or commitments. The large number of entries in the ROB and the non–trivial number of ports result in significant power dissipation in the course of accessing the ROB. According to some estimates, as much as 27% of the processor's total power dissipation can occur within the ROB in a P6–style processor [5]. The ROB complexity may have another impact: an additional stage D3 may have to be added after D2 to accommodate the large access time of the ROB.

Recently, a technique for simplifying the ROB complexity and its associated power dissipation was proposed in [10]. The design of [10] completely eliminates the read ports provided on the ROB for reading out the source operands. This results in a considerable reduction of the ROB complexity and its associated power

dissipation, but comes at a cost of performance loss because the issue of some instructions is delayed. In this paper, we propose an extension to the scheme of [10] that significantly reduces the performance degradation incurred in the original design and also results in further reduction of power and complexity of the ROB. Our technique leverages the notion of short–lived values – operands targeting registers that are renamed by the time the instruction producing the value reaches the writeback stage. As a result is produced it is forwarded to waiting instructions in the issue queue and written to the ROB. Simultaneously, the result is saved in a small buffer for access by instructions that are yet to be dispatched. This is necessary, as the ROB does not have any read ports for reading operands. Short–lived variables need not be saved in this buffer.

The rest of the paper is organized as follows. Section 2 describes the scheme of [10] in more detail. In Section 3, we define short–lived values, describe mechanism to identify them and show how the notion of short–lived values can be explored to improve the scheme of [10]. Section 4 describes our simulation framework. Simulation results are presented in Section 5, we review the related work in Section 6 and offer our concluding remarks in Section 7.

2. PRIOR WORK: LOW COMPLEXITY ROB

The ROB serving as a repository of non–committed results is an extremely port–rich structure, especially in wide–issue processors. For a W–way processor, W write ports are needed on the ROB for writing the speculative results generated by the execution units, W read ports are needed for committing the results to the ARF and 2W read ports are needed for reading the source operands from the ROB, assuming that each instruction can have at most two sources. In addition, W write ports are needed to set up the ROB entries for the co–dispatched instructions. When a relatively slow clock rate is used, it may be possible to reduce the number of ports on the ROB by multiplexing the ports. Given the relatively long 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 [3].

In a recent study [10] we proposed to reduce the ROB complexity and its associated power dissipation by completely eliminating the 2W 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, if the issue queue is extended to hold the actual source data in addition to the source tags. 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 by reading the non–committed values from the ROB [10]. The percentage of source register values supplied by the ROB is low because only the register values that are produced but not committed at the time of instruction dispatch, are read from the ROB. If the source register value is not ready at the time of instruction dispatch, this value will be later captured by the issue queue as it is written by the FU producing the result.

Noting the fact that so many ROB read ports are devoted for supplying only a very small number of source operands, we proposed the use of a ROB structure without any read ports for reading the source operand values. In our scheme, results are written into the ROB and are simultaneously forwarded to dispatched instructions waiting in the issue queue. 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. This is because source operands cannot be read from the ROB, as the read ports are eliminated. 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 is simply forwarded (again) on the forwarding buses at the time it is 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 are forwarded for the second time. The values that require this second forwarding are identified by using an additional bit with each ROB entry, called the "late forwarding bit". This bit is set when an attempt is made by the dispatched instruction to read the associated operand value from the ROB. As a result, reasonable performance is sustained without increasing the number of forwarding buses.

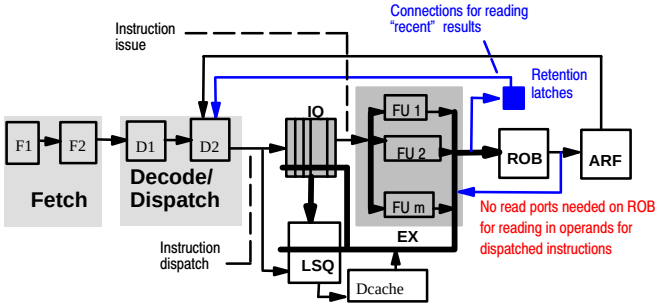

Figure 2. Superscalar datapath with the simplified ROB and retention latches

As a consequence of the elimination of the ROB read ports, the issue of some instructions is delayed. This results 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 2. 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. Notice that even if the ROB has a multi–cycle access time, we can still access the RLs in a single cycle, obviating the need for an additional stage following D2. Complete details of the scheme, including the various RL management strategies and various branch misprediction handling approaches are presented in [10].

3. IMPROVING THE DESIGN

One inefficiency of the scheme proposed in [10] stems from the fact that all generated results, irrespective of whether they could be potentially read from the RLs, are written into the latches unconditionally. Because of the small size of the RL array, this inevitably leads to the evictions of some useful values from the RLs. As a consequence, the array of RLs is not utilized efficiently and performance loss is still noticeable because many source operands can only be obtained through late forwarding, thus delaying a sizable number of instructions. If, however, one could somehow identify the values which are never going to be read after the cycle of their generation and avoid writing of these values into the RLs, then the overall performance could be improved significantly. In this section, we propose exactly such a mechanism. Our technique leverages the notion of short–lived operands – values, targeting architectural registers which are renamed by the time the instruction producing the value reaches the writeback stage.

Short–lived values do not have to be written into the RLs, because all instructions that require this value as a source will receive it through forwarding. Since the destination register is renamed by the time of instruction writeback, all dependent instructions are already in the scheduling window and the produced value is supplied to these instructions in the course of normal forwarding. Since the instruction queue entries have the data slots in addition to the result tags, the produced values are latched in the queue entries associated with the dependent instructions. Consequently, the values of short–lived operands will never be sought from the ROB or from the RLs. It is thus safe to avoid writing such short–lived values into the RLs, thus preserving the space within the RL array for the results that could actually be read. The short–lived values still, however, have to be written into the ROB to maintain the value for possible recovery in the cases of branch mispredictions, exceptions or interrupts. In the rest of this section, we describe the hardware needed for the identification of short–lived values and quantify the percentage of short–lived results generated across the simulated execution of the SPEC 2000 benchmarks.

We first outline how the register renaming is implemented in processors that use the ROB slots to implement repositories for non–committed results. The ROB slot addresses are directly used to track data dependencies by mapping the destination logical register of every instruction to the ROB slot allocated for this instruction, and maintaining this mapping in the *Register Alias Table (RAT)* until either the value is committed to the ARF, or the logical register is renamed. If an instruction commits and its destination logical register is not renamed, the mapping in the RAT changes to reflect the new location of the value within the ARF. Each RAT entry uses a single bit (the "location bit") to distinguish the locations of the register instances. If the value of this bit is set for a given architectural register, then the most recent reincarnation of this register is in the ROB, otherwise it is in the ARF. When a value is committed and no further renaming of the architectural register targeted by this value occurred, the location bit in the RAT is reset. With this in mind, we now describe the mechanism for identifying short–lived values.

3.1. Detecting Short–Lived Results

The extensions needed in the datapath for the identification of short–lived operands are very simple, they are similar to what is used in [12] for early register deallocation and in [2] for moving the data between the two levels of the hierarchical register file. We maintain the bit vector, called *Renamed*, with one bit for each ROB entry. An instruction that renames destination architectural register X sets the *Renamed* bit of the ROB entry corresponding to the previous mapping of X, if that mapping points to a ROB slot. If, however, the previous mapping was to the architectural register itself – i.e., to a committed value, no action is needed because the instruction that produced the previous value of X had already committed. These two cases are easily distinguished by examining the location bit in the RAT. *Renamed* bits are cleared when corresponding ROB entries are deallocated. At the end of the last execution cycle, each instruction producing a value checks the *Renamed* bit associated with its ROB entry. If the bit is set, then the value to be generated is identified as short–lived.

Figure 3 shows an example of identifying a short–lived value. The instruction ADD renames the destination register (R1) previously written by the LOAD instruction. Assume that the ROB entries numbered 31 and 33 are assigned to the instructions LOAD and ADD respectively. When the ADD is dispatched, it sets the *Renamed* bit corresponding to the ROB entry 31 (the previous mapping of R1), thus indicating that the value produced by the LOAD *could be* short–lived. When the LOAD reaches the writeback stage, it examines *Renamed[31]* bit and identifies the value it produces as short–lived. Notice, however, that the indication put by the ADD is just a hint, not necessarily implying that the value produced by the LOAD will be identified as short–lived. For example, if the LOAD had already passed the writeback stage by the time the ADD set the value of *Renamed[31]* bit, then the LOAD would have not seen the update performed by the ADD and the value produced by the LOAD would not be identified as short–lived.

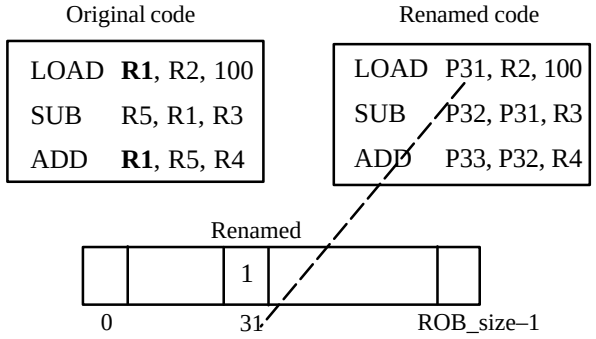

Figure 3. Identifying short–lived values

At the time of writeback, only the values that were not identified as short–lived are inserted into the RLs. To understand why such separation of values is useful, it is instructive to examine the results presented in Figure 4, which shows the percentage of short–lived result values for realistic workloads. Across the SPEC 2000 benchmarks, about 87% of all produced values were identified as short–lived on the average in our simulations, ranging from 97% for *bzip2* to 71% for *perl*. Details of our simulation framework are given in Section 4.

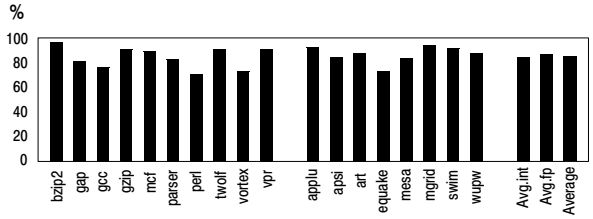

Figure 4. Percentage of short–lived values

The advantages of selective result caching in the RLs are two–fold. First, the performance improves as the RLs are used more efficiently. Alternatively, fewer RLs can be used to maintain the same performance. Second, the design is even more energy–efficient as fewer writes to the RLs take place. In addition, as the percentage of variables not identified as short–lived is small, fewer write ports to the RLs are needed to sustain performance. We now describe the main steps involved in the resulting datapath.

3.2. Instruction Renaming and Dispatching

A group of logically consecutive instructions is decoded and the corresponding RAT entries are updated. The *Renamed* bit vector is updated as described earlier in this section. For each dispatched instruction, the source operands are handled as follows:

If the source operand is in the ARF (as indicated in the "location bit" within the RAT), it is read out from the ARF and stored in the issue queue entry of the instruction.

If the result has not been committed, as evident from the setting of the location bit for the RAT entry for source, an attempt is made to read the operand from the RL array associatively using the ROB slot corresponding to the source operand as the key. In parallel, the bit indicating if the ROB entry for the source holds a valid result ("result valid" bit) is read out from the ROB. (For faster lookup, the vector of result valid bits and the vector of "late forwarding" can be implemented as individual smaller, faster arrays.) Three cases arise in the course of the associative lookup of the RLs:

Case 1: If the source value is in one of the RLs, it is read out and stored in the issue queue entry for the instruction.

Case 2: If the value was not found in the RL array but the result was already produced and written to the ROB ("result valid" bit for the ROB entry was set), the corresponding tag field for the operand within the issue queue entry for the instruction is set to the ROB index read out from the RAT. The late forwarding bit of the ROB entry is also set in this case. This allows the waiting instruction to read out the result when it is committed from the ROB in the course of *late* forwarding.

Case 3: If the value was not found in the RLs and it is yet to be produced, as evident from the setting of the "result valid" bit of the ROB entry, the corresponding tag field for the operand within the issue queue entry for the instruction is set to the ROB index read out from the RAT. In this case, the source operand will be delivered to the waiting instruction in the course of *normal* forwarding.

To complete the dispatching process, the index of the bit in the *Renamed* vector that was altered by the dispatched instruction, is saved in the ROB entry for the instruction. This allows the state of the *Renamed* vector to be restored on branch mispredictions.

3.3. Instruction Completion and Commitment

During the last execution cycle, the *Renamed* bit associated with the destination ROB slot is examined. If it is set, the result is simply written to the ROB entry in the writeback cycle. If the *Renamed* bit is not set, the result is additionally written to a pre–identified victim entry in the RL. Our past results show that either FIFO or LRU replacement schemes work well even for a small number of retention latches [10].

The only additional step needed during instruction commitment has to do with the late forwarding mechanism. If the late forwarding bit is set, a reservation is requested for a forwarding bus. The commitment process stalls until such a bus is available. When the forwarding bus becomes available, the result being committed is forwarded – for a second time – as it is written to the ARF.

3.4. Handling Branch Mispredictions

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. This alternative degrades performance negligibly [10]. The *Renamed* bit vector is also restored to the state prior to the branch along with the RAT by walking down the ROB entries along the mispredicted path. Alternatively, more aggressive checkpoint–restore mechanisms can be used.

In the rest of the paper we quantify the performance gains and the additional energy savings achievable through the exploitation of short–lived values.

4. SIMULATION METHODOLOGY

The widely–used Simplescalar simulator [1] was significantly modified 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, the register alias table and the ARF. The configuration of a simulated processor is shown in Table 1.

Table 1. Architectural configuration of the simulated processor

Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry issue queue, 96 entry ROB, 32 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)
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, 8 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, 100 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

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 100 million instructions were used for all benchmarks.

For estimating the energy/power, 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 for the use of a 2 GHz clock. A V_{dd} of 1.8 volts was assumed for all the measurements.

5. RESULTS

Figure 5 shows the commit IPCs of various schemes that use the RLs. The first bar shows the IPCs of the baseline model. The second bar shows the IPCs of the original scheme presented in [10], where each and every result is written into the RLs. 8 RLs managed as a FIFO are assumed to be in use unless stated otherwise. The average IPC

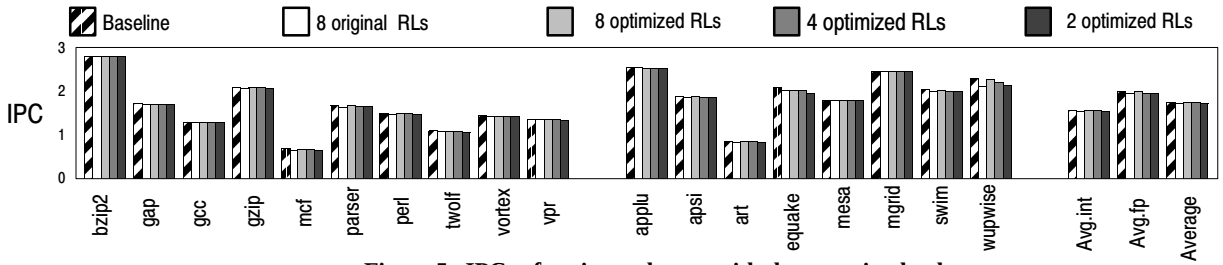

Figure 5. IPCs of various schemes with the retention latches

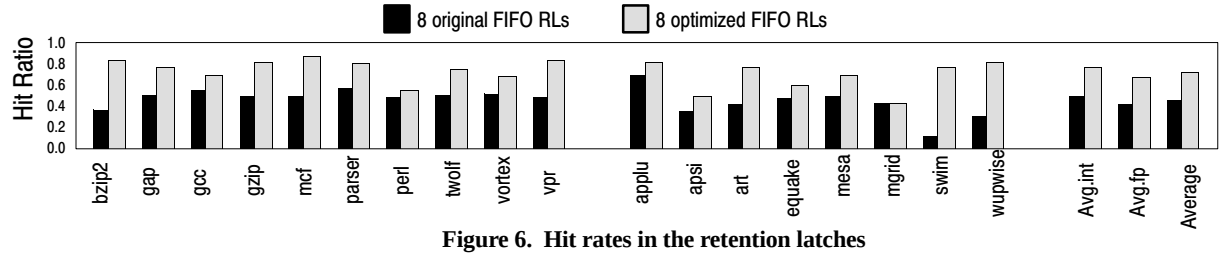

Figure 6. Hit rates in the retention latches

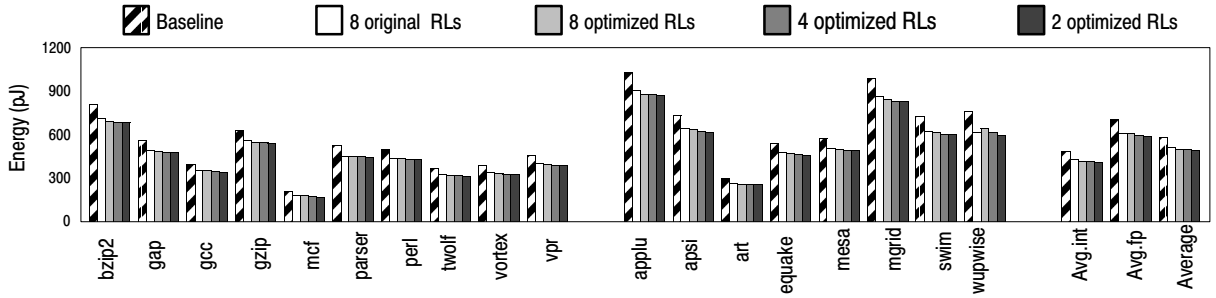

Figure 7. Energy per cycle in the ROB and the retention latches

difference between these two schemes is 1.6%. The highest IPC drop is observed for *wupwise* with 7.4% followed by *mcf* with 2.9%. The last three bars show the IPCs for the schemes, where the short–lived result values are not written into the RLs (hereinafter called "optimized RLs"). The bars correspond to the use of 2, 4 and 8 optimized RLs respectively, each managed as a FIFO. If the short–lived values are not cached in the RLs, the use of 2 RLs achieves virtually the same performance as that of the original scheme with the use of 8 RLs. Further reduction in complexity in the RL array is possible by limiting the number of write ports on the optimized RLs, as we only need two write ports to buffer two results. The use of 4 and 8 optimized RLs achieves the IPCs within 0.9% and 0.4% of the baseline machine respectively. The use of the optimized RLs is especially beneficial for certain benchmarks, such as *wupwise*. In the original scheme of [10], even the allocation of 32 RLs results in the performance degradation of more than 7% for *wupwise*. In contrast, the use of 8 optimized RLs reduces the performance loss to 0.4% with respect to the baseline machine. We assume that RLs are managed as a FIFO in all considered schemes. *We gave the benefit of the doubt to the baseline machine and assumed that all accesses to the baseline ROB take a single cycle. If the access to the ROB is pipelined across 2 or more cycles, our techniques actually result in a performance gain!*

Figure 6 shows the percentage of cases when the search for the data in the RLs results in a hit. As seen from Figure 6, the average hit rate in the RLs increases from 46% to 73% if short–lived values are not cached. Across the individual benchmarks, the hit rate of *swim* improves by about 6.5 times (from 12% to 77%), and the hit rate of *wupwise* improves by almost 3 times (from 31% to 82%). The relatively low hit rate to the RLs in the original scheme is a result of the suboptimal use of the RLs, as most of the entries hold the short–lived results.

Figure 7 shows the ROB energy reduction achieved by the use of the optimized RLs, as well as by the use of the original scheme of [10]. The scheme with 2 optimized RLs results in 16% ROB energy reduction compared to about 12% energy savings achieved with the use of 8 RLs managed as proposed in [10]. Recall that these two schemes have comparable loss in performance – about 1.6%. The power overhead due to the need to access the bit–vector *Renamed* was fully accounted for in our measurements.

6. RELATED WORK

It has been noticed by several researchers that most of the register instances in a datapath are short–lived. In [6], Franklin and Sohi report the statistics about the useful lifetime of register instances. They conclude that if a perfect knowledge about the last use of each register instance is available, then delaying the writes to the register file until 30 more instructions are dispatched into the pipeline can eliminate the need to perform as much as 80% of these writes.

The exploitation of short–lived variables continued with the work by Lozano and Gao [11], who observed that about 90% of the generated

result values are short–lived, in the sense that they are exclusively consumed during their residency in the ROB. Lozano and Gao then proposed mechanisms to avoid the commitment of such variables to the architectural register file and also avoid register allocation for such variables. Their approach is based on a compiler analysis, where the ROB slots are effectively exposed to the compiler in the form of symbolic registers for storing the short–lived variables.

The idea of using the retention latches is similar in philosophy to forwarding buffer described by Borch et.al. in [3] for a multi–clustered Alpha–like datapath. Both solutions essentially extend the existing forwarding network to increase the number of cycles for which source operands are available without accessing the register file. A similar technique is used in [16], where a delayed write–back queue is used to reduce both the number of read and write ports on the register file.

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, particularly in wire–delay dominated circuits [2, 3, 4]. Replicated register files in a clustered organization have been used in the Alpha 21264 processor [9] 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. In [15], 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 [16], the peak read port requirements are reduced by prefetching the operands into an operand pre–fetch buffer.

The idea of caching recently produced values was also used in [7]. At the time of instruction writeback, FUs write results into a cache called Value Aging Buffer (VAB). The register file is updated only when entries were evicted from the VAB. The scheme has an inherent performance loss since the VAB and the register file are accessed serially.

In [14], Savransky et.al. 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. The overhead of the scheme of [14] is in the form of the additional mapping table to keep track of the place in which the last non–speculative copy of each architectural register is stored. In [13], the ROB power is reduced by using a multi–segmented organization, where segments are deactivated dynamically if they are not used. The schemes of [14] and [13] are orthogonal to the proposed technique and can be used in conjunction with our scheme to achieve additional power savings.

7. CONCLUDING REMARKS

The reorder buffer in some superscalar processors is a complex multi–ported structure, representing a significant source of power dissipation. 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. In this paper, we proposed a modification to an earlier introduced scheme for elimination of the ports needed for reading the source operand values from the ROB.

The proposed extension improves the scheme of [10] by avoiding the caching of short–lived values in the retention latches. This technique is quite effective as it increases the hit ratio of the retention latches from 46% to 73%. Consequently, the average performance loss is reduced from 1.6% to only 0.4% when 8 retention latches managed as a FIFO are used. If the same level of performance is maintained, then the proposed scheme can use as few as 2 retention latches (with two write ports), thus allowing to further reduce the overall complexity and power dissipation.

8. ACKNOWLEDGMENTS

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.

9. 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] 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.

[3] 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.

[4] Cruz, J–L. et. al., ”Multiple–Banked Register File Architecture”, in Proceedings 27th Int’l. Symposium on Computer Architecture, 2000, pp. 316–325.

[5] Folegnani, D., Gonzalez, A., “Energy–Effective Issue Logic”, in Proceedings of Int’l. Symp. on Computer Architecture, July 2001.

[6] Franklin, M., Sohi, G., “Register Traffic Analysis for Streamlining Inter–Operation Communication in Fine–Grain Parallel Processors”, in International Symposium on Microarchitecture, 1992.

[7] Hu, Z. and Martonosi, M., “Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics”, in Workshop on Complexity–Effective Design, 2000.

[8] Intel Corporation, “The Intel Architecture Software Developers Manual”, 1999.

[9] Kessler, R.E., ”The Alpha 21264 Microprocessor”, IEEE Micro, 19(2) (March 1999), pp. 24–36.

[10] Kucuk, G., Ponomarev, D., Ghose, K., “Low Complexity Reorder Buffer Architecture”, in Proceedings of Int’l Conference on Supercomputing, June, 2002, pp.57–66.

[11] Lozano, G. and Gao, G., “Exploiting Short–Lived Variables in Superscalar Processors”, in Proceedings of Int’l Symposium on Microarchitecture, 1995, pp. 292–302.

[12] Martinez, J., Renau, J., Huang, M., Prvulovich, M., Torrellas, J., “Cherry: Checkpointed Early Resource Recycling in Out–of–order Microprocessors”, in Proceedings of the 35th International Symposium on Microarchitecture, 2002.

[13] 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.

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

[15] 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.

[16] 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.