Appears in the Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation, September, 2002

Energy–Efficient Design of the Reorder Buffer[1]

Dmitry Ponomarev, Gurhan Kucuk, Kanad Ghose

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

Abstract. Some of today's superscalar processors, such as the Intel Pentium III, implement physical registers using the Reorder Buffer (ROB) slots. As much as 27% of the total CPU power is expended within the ROB in such designs, making the ROB a dominant source of power dissipation within the processor. This paper proposes three relatively independent techniques for the ROB power reduction with no or minimal impact on the performance. These techniques are: 1) dynamic ROB resizing; 2) the use of low–power comparators that dissipate energy mainly on a full match of the comparands and, 3) the use of zero–byte encoding. We validate our results by executing the complete suite of SPEC 95 benchmarks on a true cycle–by–cycle hardware–level simulator and using SPICE measurements for actual layouts of the ROB in 0.5 micron CMOS process. The total power savings achieved within the ROB using our approaches are in excess of 76% with the average performance penalty of less than 3%.

1 Introduction

Contemporary superscalar microprocessors use extensive execution reordering to maximize performance by extracting ILP. One of the main dynamic instruction scheduling artifacts used in such datapath designs is the Reorder Buffer (ROB), which is used to recover to a precise state when interrupts or branch mispredictions occur. The ROB is essentially 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, thus preserving the correct (program) order of updates to the program state.

In some designs (such as the Pentium III), the physical registers are integrated into the ROB to support register renaming. An associative addressing mechanism is used in some ROBs (as in AMD K5) to determine if any entries exist in the ROB for an architectural register. This avoids the need to maintain and use an explicit rename table.

The associative lookup of ROB entries, the writes to the ROB in the course of setting up a new entry or in writing results/exception codes and in reading out data from the ROB during operand reads or commits cause a significant power dissipation. For example, a recent study by Folegnani and Gonzalez estimated that more than 27% of the total power expended within a Pentium–like microprocessor is dissipated in the ROB [4].

We study three techniques for reducing the power dissipation within the ROB. First, we consider dynamic ROB resizing. Here, the ROB is implemented as a set of independent partitions and these partitions are dynamically activated/deactivated based on the demands of applications. Second, we propose the use of fast low–power

[1] 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 IEEC at SUNY Binghamton.

comparators, as introduced in our earlier work [5], to perform the associative search of an ROB slot containing the source register. Third, we noticed that high percentage of bytes within the data items travelling on the result, dispatch and commit buses contain all zeroes. We exploit this by adding the Zero Indicator (ZI) bit to each byte of data that is read from or written to the ROB to avoid the reading and writing of these bytes, thus saving energy.

The rest of the paper is organized as follows. Section 2 discusses the related work. The superscalar datapath configuration assumed for this study and sources of power dissipation within the ROB are discussed in Section 3. In Section 4, we describe our simulation methodology. Section 5 presents the dynamic ROB resizing strategy. In Section 6, we review the energy–efficient comparator. Section 7 describes the zero–byte encoding mechanism. Simulation results are presented in Section 8 followed by our concluding remarks in Section 9.

2. Related work

Dynamic ROB resizing driven by the ROB occupancy changes was studied in our earlier work [7] in conjunction with similar resizing of the issue queue and the load/store queue. In this paper, we explore the ROB resizing in isolation and estimate the resulting power savings and effects on performance.

Alternative approaches to dynamic resizing of datapath resources were proposed by several researchers [2, 4] to reduce power dissipation of the issue queues. Unfortunately, the metrics used in these proposals to drive the resizing decisions, such as the ready bits [2] or the number of instructions committed from the “youngest” region of the issue queue [4] make it impossible to extend these techniques to the reorder buffers. Due to the space restrictions, we avoid further comparison of our work with the techniques of [2] and [4] and instead refer the reader to [7], where other limitations of the solutions proposed in [2] and [4] are documented.

In [5], we introduced a design of energy–efficient dissipate–on–match comparator and evaluated its use within the issue queue. In this paper we show how the same comparator can be used to save power in associatively–addressed ROBs. Zero–byte encoding was proposed for use in caches in [9] and issue queues in [5].

Another way of reducing the ROB complexity and energy dissipation was studied in [12], where we proposed a scheme for complete elimination of the ROB read ports needed for reading the source operands.

3. Superscalar Datapath and Sources of Energy Dissipation

We consider a superscalar datapath where the ROB slots serve as physical registers. The ROB entry set up for an instruction at the time of dispatch contains a field to hold the result produced by the instruction – this serves as the analog of a physical register. We assume that each ROB entry may hold only a 32–bit long result, thus requiring the allocation of two ROB entries for an instruction producing a double–precision value. Every ROB entry contains a field for the address of the destination architectural register of the instruction allocated to this entry. A dispatched instruction attempts to read operand values either from the Architectural Register File (ARF) directly if the operand value was committed, or associatively from the ROB (from the most recently established entry for an architectural register), in case the operand value was generated but not committed. Source registers that contain valid data are read out into the issue queue entry for the instruction. If a source operand is not available at the time of dispatch in the ARF or the ROB, the address of the physical register (i.e., the ROB slot) is saved in the tag field associated with the source register in the issue queue entry for the instruction. When a function unit completes, it puts out the result produced along with the address of the destination register for this result on a forwarding bus which runs across the length of the issue queue [6]. An associative tag matching process is then used to steer the result to matching entries within the issue queue. The result is also written into the corresponding ROB slot via a result bus. Result values are committed to the ARF from the head of the ROB in program order.

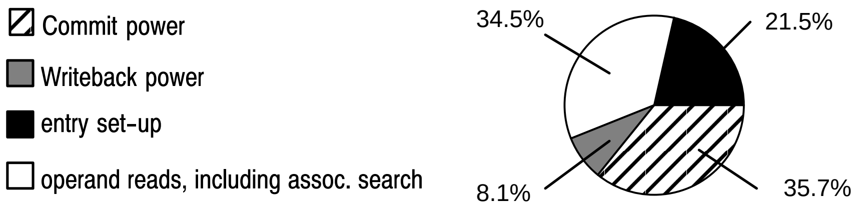

Fig. 1 Energy dissipation components of the traditional ROB (% of total)

The ROB, as used in the datapath described above, is essentially implemented as a large register file. The most recent value generated for an architectural register is located using either an associative addressing capability within the ROB or through the use of an explicit rename table. In a W–way superscalar processor, W sets of internal buses are used to establish, use and commit ROB entries. Where the ROB supports associative lookup without a rename table, 2W address buses are employed for performing an associative search for up to 2 source physical registers for each of the W dispatched instructions.

Energy dissipating events within the multi–ported register file that implements the ROB are as follows:

- Establishment of the entry for dispatched instructions. The resulting energy dissipation has the following components: the energy dissipated in writing the address of the instruction (PC value), flags indicating the instruction type (branch, store, register–to–register), the destination register id, and predicted branch direction (for a branch instruction).
- Instruction dispatch. The energy is dissipated when the address of the corresponding architectural register is driven on the address bus and comparators associated with each ROB entry compare this address against the locally stored architectural register identifiers.
- Reading the valid sources from the ROB slots that implement physical registers.
- Writing the results to the ROB from the function units at the writeback time.
- Committing the ROB entries.
- Clearing the ROB on mispredictions. Compared to all of the other ROB dissipation components, this is quite small.

Figure 1 presents the relative values of some of these energy dissipation components. Results are averaged across all SPEC 95 benchmarks, obtained from the simulated execution of the SPEC 95 benchmarks for a 4–way machine, using the configuration described in Section 4.

4. Simulation Methodology

We used the AccuPower toolset [8] to estimate the energy savings achievable within 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 the issue queue, the ROB, the load/store queue, the register files, forwarding interconnections and dedicated transfer links.

For estimating the energy/power for the ROB, the event counts gleaned from the simulator were used, along with the energy dissipations for each type of event described in section 3, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the ROB in a 0.5 micron 4 metal layer CMOS process (HPCMOS–14TB) 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 300 MHz clock. A Vdd of 3.3 volts is assumed for all the measurements.

Table 1. Architectural configuration of a simulated 4–way superscalar processor

Parameter	**Configuration**
Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry issue queue, 128 entry ROB, 32 entry load/store queue
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, 64 byte line, 4 cycles hit time.
BTB	1024 entry, 2–way set–associative
Memory	128 bit wide, 12 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), 4–way set–associative, 30 cycles miss latency
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)

The configuration of the 4–way superscalar processor used in this study is shown in Table 1. 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. For all of the SPEC 95 benchmarks, the results from the simulation of the first 200 million instructions were discarded and the results from the execution of the following 200 million instructions were used.

5. Dynamic ROB Resizing

It is well–documented [10] that the overall performance, as measured by the IPC, varies widely across applications. One approach to minimizing the power requirements of the datapath is to use a dynamic resource allocation strategy to closely track the resource demands of programs that are being executed. Because of its large size, the ROB is especially suitable candidate for such dynamic allocation. If a program exhibits modest resource requirements, the degree of the ROB overcommitment may be quite large. The ROB resizing strategy used in this paper was first proposed in [7], where the ROB was resized in combination with other resources. In this paper, we study the ROB resizing on its own, keeping the sizes of other resources at their maximum values at all times. In the rest of this section, we summarize our resizing strategy and we refer the readers to [7] for a more detailed discussion.

To support dynamic resizing, the ROB is implemented as a number of independent partitions, each of them being a self–standing and independently usable unit, complete with its own prechargers, sense amps and input/output drivers. We assumed that the number of entries in a ROB partition is 16. A number of partitions can be strung up together to implement a larger structure. The lines running across the entries within a partition (such as bitlines, lines to drive the architectural register addresses for comparison etc.) can be connected to a common through line via the bypass switches to add (i.e., allocate) the partition to the current ROB and extend its effective size. Similarly, the partition can be deallocated by turning off the corresponding bypass switches. The circular FIFO nature of the ROB requires some additional considerations in the resizing process: in some situations partitions can be activated and deactivated only when the queue extremities coincide with the partition boundaries [7].

Some efforts (such as the one in [2]) focussing on the dynamic resizing of issue queues, use the (commit) IPC values as measured on a cycle–by–cycle basis to trigger the downsizing or upsizing actions. We found, however, that dispatch rates and IPCs are not always positively correlated with the ROB occupancy, measured as the number of valid entries, and hence, with the resource demands of the applications. To illustrate this, Figure 2 shows the behavior of two floating point benchmarks from SPEC 95 suite – *apsi* and *hydro2d*. Samples of dispatch rate and the average occupancy of the ROB were taken every 1 million cycles. These samples reflected the activity within the most recent window of 1 million cycles. Results are shown for the execution of 200M instructions.

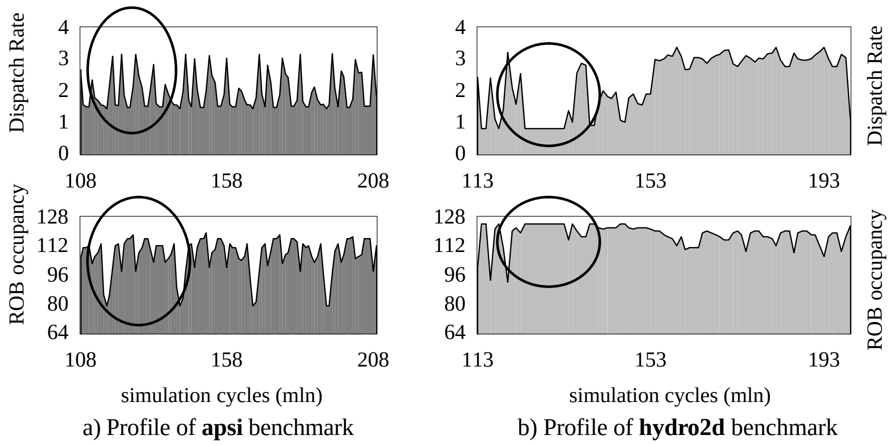

Fig. 2 Dynamic behavior of two SPEC95 floating point benchmarks.

As these representative results show, the average number of active entries within the ROB changes significantly throughout the course of execution. These graphs also suggest that it is hardly possible to efficiently control the allocation of the ROB partitions based solely on the dispatch rate (or the IPC). Our resizing strategy does not rely on the IPC or dispatch rate as a measure of resource requirements, instead we use the actual

ROB occupancy as sampled at periodic intervals as an indication of the application's needs.

The decision of downsizing the ROB is made periodically – once every *update_period* cycles. During this period, the ROB occupancy, as measured by the number of allocated entries within the active region of the ROB, is sampled several times with the frequency of once every *sample_period* cycles. The average of these samples is taken as the *active_size* in the current update period. At the end of every update period, the difference (*diff=current_size–active_size*) is computed, where *current_size* is the size of the ROB at the end of the update period. If *diff* is less than the size of an ROB partition, no resizing takes place. Otherwise, one ROB partition is turned–off. More aggressive designs – not studied here – can be used where more than one partition can be turned–off at the same time, as determined by *diff*.

By sampling the resource usage periodically instead of on a continuous basis, as done in [2,4] for the issue queue, we conserve dissipations within the monitoring logic for the downsizing. Our results also reveal that the error arising from the computations of the average occupancy by taking the average occupancy measured at discrete points (at every sample interval) is tolerable since significant energy savings are achieved using our approach. Although the current paper shows the savings on dynamic/switching power, dynamic deactivation of partitions also saves leakage power that would be otherwise dissipated within the ROB.

The second phase of the resizing strategy is implemented to scale the ROB size back up once the demands of the application begin to require more resources. Here, we use the ROB overflow counter (*rob_overflow*), which counts the number of cycles when dispatch is blocked because of the absence of a free entry in the ROB to hold the new instruction. This counter is initialized to zero every *update_period* cycles and is incremented by one whenever instruction dispatch is blocked because of the unavailability of a free ROB entry. Once this overflow counter exceeds a preset value (*overflow_threshold*), the ROB size is incremented by turning one currently inactive partition on. After that, the counter is re–initialized. Notice, that for performance reasons, the process of increasing the size of the ROB is more aggressive than the process of shrinking the ROB. To avoid the occurrence of several such incremental updates in one update period, we reset the update period counter when the increase of the ROB size is triggered. This is to avoid the instability in the system associated with the fact that ROB is sampled for different maximum sizes in the same update period.

6. Use of Energy–Efficient Comparators

As shown in Figure 1, the power dissipated within the ROB during instruction dispatch, particularly during the associative look–up for the ROB slot containing the source register, is a significant portion of the total ROB power. We assume that the architectural register file consists of 32 integer registers and 32 floating point registers, thus requiring 6–bit comparators to perform the associative look–up. The typical comparator circuitry used for the associative architectural register address matching in the ROB is a dynamic pulldown comparator or a set of 8–transistor associative bitcell pulling down a common precharged line on a mismatch in any bit position. All of these comparators dissipate energy on a *mismatch* in any bit position. The energy is thus wasted in comparisons that do not locate matching entries, while little energy (in the form of precharging) is spent in locating matching entries.

The data collected for the simulated execution of SPEC 95 benchmarks on our system indicates that during the associative lookup of source values from the ROB, only up to 6 of the comparators in valid entries within the ROB produce a match per instruction on the average. This is clearly an energy–inefficient situation, as more energy is dissipated due to mismatches (compared to the number of matches) with the use of traditional comparators that dissipate energy on a mismatch. This situation can be remedied by using Domino–style compatators, that predominantly dissipate energy only on a full match. Such a comparator design was proposed in [5]. For the ROB, we use a variation of the circuit shown in [5], that compares two 6–bit comparands. The basic circuit is a two–stage domino logic, where the first stage detects a match of the lower 4 bits of the comparands. The following stage does not dissipate any energy when the lower order 4 bits do not match.

Table 2. Comparator Statistics

Number of bits matching —>	% of total cases		
	2 LSBs	4 LSBs	Match
Avg, SPECint 95	23.0	14.0	12.0
Avg, SPECfp 95	26.0	16.0	11.0
Avg, all SPEC 95	25.0	16.0	11.0

LSB = least significant bits

Table 2 shows how comparand values are distributed and the extent of partial matches in the values of two adjacent bits in the comparands, averaged over the simulated execution of SPEC 95. The lower order 2 bits of both comparands are equal in roughly 25% of the case, while the lower order 4 bits match 16% of the time. A full match occurs 11% of the time on the average. Equivalently, a mismatch occurs in the lower order 2 bits of the comparator 75% of the time (=100 – 25). The behavior depicted in Table 2 is a consequence of the localized usage of the architectural registers in a typical source code. As a result, the likelihood of a match of the higher order bits of the register addresses (i.e., the comparands) is higher. Our comparator design directly exploits this fact by limiting dynamic energy dissipation due to partial matches to less than 16% of all cases when the lower order 4 bits match; no energy dissipation occurs in the more frequent cases of the higher order bits matching. The overall energy dissipation due to partial or complete mismatches is thus greatly reduced.

7. Use of Zero–byte Encoding

A study of data streams within superscalar datapaths revealed that significant number of bytes are all zeros within operands on most of the flow paths (ROB to the issue queue, issue queue to function units, functions units to issue queue and ROB, ROB to ARF etc.). On the average, about half of the byte fields within operands are all zeros for SPEC 95 benchmarks. This is really a consequence of using small literal values, either as operands or as address offsets, byte–level operations, operations that use masks to isolate bits etc. Considerable energy savings are possible when bytes containing zero are not transferred, stored or operated on explicitly. Other work in the past for caches [8], function units [11] and *scalar* pipelines [3] have made the same observation. We extend these past work to the ROB. By not writing zero bytes into the ROB at the time of dispatch and writeback, energy savings result as fewer bitlines need to be driven. Similarly, further savings are achieved during commit by not reading out implied zero valued bytes. This can be done by storing an additional Zero Indicator (ZI) bit with each byte that indicates if the associated byte contains all zeros or not. The circuit details of the zero encoding logic used within the ROB is similar to that of [8] and [5].

8. Results and Discussions

Power savings achieved within the ROB using techniques proposed in this paper are summarized in Figures 3–5. Figure 3 shows the effects of the energy–efficient comparator on the power dissipated during the process of comparing the architectural register addresses. The total ROB energy reduction is 41% on the average across all SPEC95 benchmarks in this case. However, the total ROB power is reduced by only about 13% because the comparator does not have any effect on the energy dissipated during entry setup as well as during writeback and commitment (Figure 4).

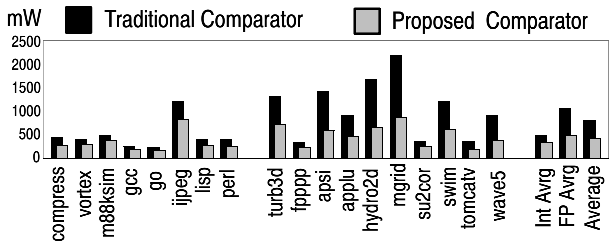

Fig. 3 Power dissipation within the ROB at the time of source operand reads

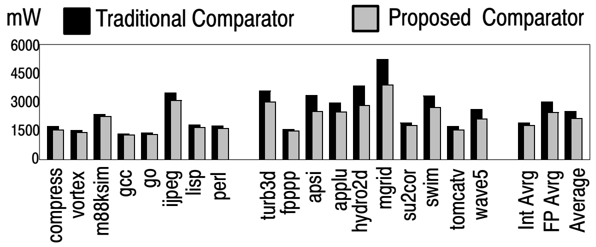

Fig. 4 Total Power dissipation within the ROB with the proposed and the traditional comparators

Dynamic resizing and zero–byte encoding address the power reduction in exactly these two components (writeback and commitment). For the results of dynamic resizing presented below, both *overflow_threshold* and *update_period* were kept at 2K cycles. We selected the value of 2K cycles for the update_period as a reasonable compromise between too large a value (in which case the resizing algorithm will not react quickly enough to the changes in application's behavior) and too small a value (in which case the resizing overhead increases). For comparison, Table 3 summarizes achieved power savings and performance drop for a range of *overflow_threshold* values. Results shown in this table are the averages across all executed benchmarks.

Table 3. ROB power savings and performance drop for various values of *overflow_threshold*

overflow_threshold	128	256	512	1024	2048
power savings(%)	56.1	57.3	59	60.5	62.2
IPC drop (%)	0.06	0.27	0.78	1.87	3.14

Figure 5 summarizes the overall power savings in the ROB realizable using our techniques. Dynamic resizing results in about 60% energy reduction in the ROB on the average. Average power savings attributed to the use of zero–byte encoding are about 17% (this is not shown in the graph). Dynamic resizing and zero–byte encoding in concert reduce the energy by more than 66% on the average. Notice that the total power reduction achieved by these two techniques is not the sum of their individual respective savings. This is because dynamic resizing reduces the lengths of the bitlines which somewhat reduces the savings achieved by zero–byte encoding. The combination of all three techniques – dynamic resizing, zero–byte encoding and the use of fast, power–efficient comparators achieves a remarkable 76% reduction in power dissipated by the reorder buffer on the average with a minimal impact on the performance.Note that

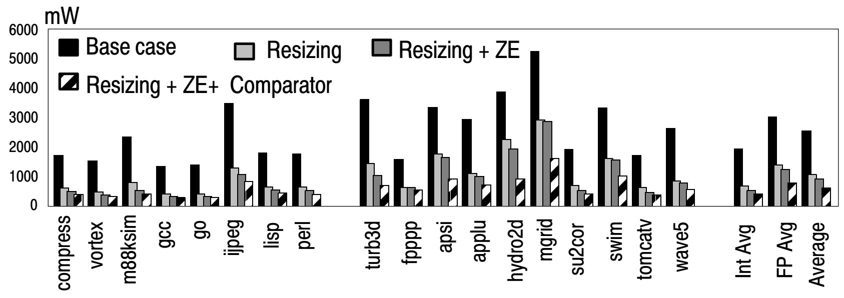

Fig. 5 Total power dissipation within ROB using dynamic resizing, power efficient comparators and zero encoding

in all of the power estimations we assumed that comparators are enabled only for ROB entries that are valid. If spurious comparisons are allowed, the power savings reported here will go up further.

To summarize, the dynamic ROB resizing provides the highest power savings within the ROB. Fundamentally, this is because the ROB resizing immediately exploits its overcommitments by turning off the complete partitions, effectively turning off many comparators and reducing the capacitances of all bitlines and address lines which results in very significant power savings. In the future, as processor designers tend to increase the issue widths and employ larger ROBs, the effect of dynamic resizing will become even more dramatic. The drawbacks of the dynamic resizing are in the form of some performance loss and in its requirement of a fair amount of modifications to the datapath, although we tried to keep the latter to the minimum in our design. The use of energy–efficient comparators and zero–byte encoding can also achieve substantial *additional* energy savings, as seen from our results.

9. Concluding Remarks

We studied three relatively independent techniques to reduce the energy dissipation in the reorder buffer used in modern superscalar processors. First, we used a technique to dynamically resize the ROB based on its actual occupancy. Second, we used fast comparators in the associative search logic that dissipate the energy mainly on a full match of the architectural register addresses. For the technology studied, the associative lookup of the most recent value of an architectural register using these comparators is more energy–efficient than using a rename table lookup. Third, we considered the use of zero–byte encoding to reduce the number of bitlines that have to be driven during instruction dispatch, writeback and commitment. Combined, these three mechanisms reduce the power dissipated by the reorder buffer by more than 76% on the average across all SPEC 95 benchmarks with the average penalty on the IPC below 3%.

The ROB power reductions are achieved without compromising the cycle time and only through a modest growth in the area of the ROB (about 12%, including the new comparators, ZE logic and dynamic resizing logic). Our ongoing studies also show that the use of all of the techniques that reduce the ROB power can also be used to achieve reductions of a similar scale in other datapath artifacts that use associative addressing (such as issue queues and load/store queues). As the power dissipated in instruction dispatching, issuing, forwarding and retirement can often be as much as half of the total chip power dissipation, the use of dynamic resizing, the energy–efficient comparators and zero–byte encoding offers substantial promise in reducing the overall power requirements of contemporary superscalar processors.

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] Buyuktosunoglu, A., Schuster, S., Brooks, D., Bose, P., Cook, P. and Albonesi, D., “An Adaptive Issue Queue for Reduced Power at High Performance”, in PACS Workshop, November 2000.

[3] Canal R., Gonzalez A., and Smith J., “Very Low Power Pipelines using Significance Compression”, in Proceedings of International Symposium on Microarchitecture, 2000.

[4] Folegnani, D., Gonzalez, A., “Energy–Effective Issue Logic”, in Proc. of ISCA, July 2001.

[5] Kucuk, G., Ghose, K., Ponomarev, D., Kogge, P., “Energy–Efficient Instruction Dispatch Buffer Design for Superscalar Processors”, in Proc. of ISLPED, 2001.

[6] Palacharla, S., Jouppi, N. P. and Smith, J.E., “Quantifying the Complexity of Superscalar Processors”, Technical report CS–TR–96–1308, Dept. of CS, Univ. of Wisconsin, 1996.

[7] Ponomarev, D., Kucuk, G., Ghose, K., “Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources”, in Proc. of MICRO–34, 2001.

[8] Ponomarev, D., Kucuk, G., Ghose, K., “AccuPower: an Accurate Power Estimation Tool for Superscalar Microprocessors”, in Proc. of 5th Design, Automation and Test in Europe Conference, 2002.

[9] Villa, L., Zhang, M. and Asanovic, K., “Dynamic Zero Compression for Cache Energy Reduction”, in Proceedings of International Symposium on Microarchitecture, 2000.

[10] Wall, D.W., “Limits on Instruction Level Parallelism”. In Proceedings of ASPLOS, November, 1991.

[11] Brooks, D. and Martonosi, M., “Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance”, Proc. HPCA, 1999.

[12] Kucuk, G., Ponomarev, D., Ghose, K., “Low–Complexity Reorder Buffer Architecture“, in Proc. of the 16th International Conference on Supercomputing, 2002.