RC22192 (W0110-013) October 3, 2001
Computer Science

IBM Research Report

Power Efficient Issue Queue Design

Alper Buyuktosunoglu, David Albonesi
University of Rochester

Stanley E. Schuster, David Brooks, Pradip Bose, Peter W. Cook
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

POWER-EFFICIENT ISSUE QUEUE DESIGN

Alper Buyuktosunoglu, David H. Albonesi
Department of Electrical and Computer Engineering
University of Rochester
buyuktos@ece.rochester.edu

Stanley Schuster, David Brooks, Pradip Bose, Peter Cook
IBM T.J. Watson Research Center

Abstract Increasing levels of power dissipation threaten to limit the performance gains of future high-end, out-of-order issue microprocessors. Therefore, it is imperative that designers devise techniques that significantly reduce the power dissipation of the key hardware structures on the chip without unduly compromising performance. Such a key structure in out-of-order designs is the issue queue. Although crucial in achieving high performance, the issue queues are often a major contributor to the overall power consumption of the chip, potentially affecting both thermal issues related to hot spots and energy issues related to battery life. In this chapter, we present two techniques that significantly reduce issue queue power while maintaining high performance operation. First, we evaluate the power savings achieved by implementing a CAM/RAM structure for the issue queue as an alternative to the more power-hungry latch-based issue queue used in many designs. We then present the microarchitecture and circuit design of an adaptive issue queue that leverages transmission gate insertion to provide dynamic low-cost configurability of size and speed. We compare two different dynamic adaptation algorithms that use issue queue utilization and parallelism metrics in order to size the issue queue on-the-fly during execution. Together, these two techniques provide over a 70% average reduction in issue queue power dissipation for a collection of the SPEC CPU2000 integer benchmarks, with only a 3% overall performance degradation.

1. Introduction

Power dissipation has become a major microprocessor design constraint, so much so that it threatens to limit the amount of hardware that can be included on future microprocessors how fast it can be clocked

year	1999	2002	2005	2008	2011	2014
feature size (nm)	180	130	100	70	50	35
logic trans/cm^2	6.2M	18M	39M	84M	180M	390M
clock (MHz)	1250	2100	3500	6000	10000	16900
chip size (mm^2)	340	430	520	620	750	900
power supply (V)	1.8	1.5	1.2	0.9	0.6	0.5
high-perf power (W)	90	130	160	170	175	183

Table 1. SIA roadmap for power dissipation in current and future microprocessors

[11, 12, 14]. Designers of handheld devices are already quite familiar with the difficulties in meeting ever-increasing performance demands while maintaining reasonable battery life and small product form-factors. However, the increasing packaging, cooling, and power distribution costs, as well as reliability issues, associated with increasing power dissipation in high-end systems also threatens their future viability.

Table 1 summarizes the SIA Roadmap [18] for power dissipation in current and next generation high-performance microprocessors. According to these projections, power dissipation will increase from 90 watts in 1999 to 170 watts in 2008. Clearly, rapid increases in both the clock frequency and chip functionality (complexity) are outpacing circuit and microarchitectural design attempts to maintain reasonable power dissipation limits.

One of the major contributors to the overall power consumption in a modern out-of-order superscalar processor, like the Alpha 21264 and Mips R10000 [13, 16], is the issue queue. The issue queue holds decoded and renamed instructions until they issue to appropriate functional units. The choice of an issue queue size requires striking a careful balance between the ability to extract instruction-level parallelism (ILP) from common programs and achieving high frequency operation. The size of the issue queue represents the *window* of instructions that may be eligible at any given cycle for issue. An instruction residing in the issue queue becomes eligible for issue (or *woken up*) when both of its source operands have been produced and an appropriate functional unit has become available. The *selection logic* determines which instructions (up to maximum issue width of the processor) should issue out of those woken up on a given cycle. Many superscalar processors such as the Alpha 21264 [13] and MipsR10000 [16] use multiple issue queues tailored to the type of instruction (*e.g.*, integer, floating point, and memory). Because the issue queue can be a major contributor to overall power dissipation

(for instance, the integer queue on the Alpha 21264 is the highest power consumer on the chip [15]), new issue queue approaches must be devised that significantly reduce power dissipation without unduly compromising clock speed or instructions per cycle (IPC) performance.

There have been several prior proposals for reducing issue queue power dissipation. *Dynamic adaptation* of the issue queue size to match application demands is proposed in [1, 2] in order to increase performance and reduce power dissipation. The basic idea is to exploit the fact that the issue queue size needed to obtain most of the achievable IPC performance (within reasonable implementation limits) differs from application to application, and therefore the size of the queue can be adjusted (in *increments* of n entries) to fit the application. However, in [1, 2], it was assumed that the best issue queue size for a given application was known *a priori*, no attempt was made to adapt within an individual application, and the circuit-level design issues associated with an adaptive issue queue were not addressed in detail.

Gonzalez et al. [8, 9, 10] propose several schemes that attempt to reduce the issue queue control logic complexity [8] or its power dissipation [9, 10] without significantly impacting IPC performance. In [8], the authors propose and evaluate two different schemes. In the first approach, the complexity of the issue logic is reduced by having a separate *ready queue* which only holds instructions with operands that are determined to be fully available at decode time. Thus, instructions can be issued *in-order* from the ready queue at reduced complexity, without any associative lookup. A separate *first-use table*, indexed by unavailable operand register specifiers, holds those instructions that are the first-time consumers of these pending operands. Instructions which are deeper in the dependence chain simply stall or are handled separately through a separate issue queue. The dependence link information connecting multiple instances of the same instruction in the first-use table is updated after each instruction execution is completed. At the same time, if a given instruction is deemed to be *ready* it is moved to the in-order ready queue. Since none of the new structures require associative lookups or run-time dependence analysis, and yet, instructions are able to migrate to the ready queue as soon as their operands become available, this scheme significantly reduces the complexity of the issue logic.

The second approach relies on static scheduling. Here, the main issue queue holds instructions with pre-determined availability times of their source operands. Because the queue entries are time-ordered (due to known availabilities), the issue logic can use simple, in-order semantics. Instructions with operands which have unknown availability times are

held in a separate *wait queue* and get moved to the main issue queue only when those times become definite. In both approaches described in [8], the emphasis is on reduction of the complexity of the issue control logic. The added (or augmented) support structures in these schemes may actually cause an increase of power, despite the simplicity and elegance of the control logic.

In [9, 10], the main focus is on power reduction. The issue queue is designed to be a circular queue structure, with head and tail pointers, and the effective size is dynamically adapted to fit the ILP content of the workload during different periods of execution. In both [8] and [9, 10], the authors show that the IPC loss is very small with the suggested modifications to the issue queue structure and logic. Also, in [9, 10], the authors use a trace-driven power-performance simulator (based on the model by Cai [7]) to report substantial power savings on dynamic queue sizing. However, a detailed circuit-level design and simulation of the proposed implementations are not reported in [8] or [9, 10]. Without such an analysis, it is difficult to assess whether the cycle time overhead of the extra circuitry required for dynamic adaptation, or its power dissipation overhead, are overridden by the power dissipation benefits of these approaches.

In this chapter, we propose two techniques that achieve significant issue queue power savings while maintaining high speed operation and good IPC performance. First, we propose a CAM/RAM-based issue queue as an alternative to the latch-based design used in many microprocessors. We then propose modifications that divide the queue into four sections, each of which can be individually enabled/disabled, and algorithms that adapt the issue queue size on-the-fly during the running of an individual application. Our scheme is simpler than that reported in [8] in that it does not require any new data storage or access structure (like the first-use table or the wait queue). Rather, we leverage the CAM/RAM structure that we propose as an alternative to a latch-based design. Although our approach of dynamically adapting the issue queue size to match workload demands is conceptually similar to the method proposed in [9, 10], our adaptation is more coarse-grained to limit circuit complexity and our control logic is quite different. Also, unlike [9, 10], we perform a detailed circuit-level evaluation of the power and performance overheads of the adaptive logic in addition to a microarchitecture analysis.

The rest of this chapter is organized as follows. In the next section, we compare the power and performance characteristics of latch-based and CAM/RAM-based issue queues. We then present in Section 3 the adaptive issue queue design in detail, both in terms of circuit-level design

and algorithms for adaptive control. We also perform a circuit and microarchitecture-level analysis of the power benefits, and performance costs, of dynamic adaptivity. Finally, we conclude in Section 4.

2. Latch and CAM/RAM-Based Issue Queues

As previously mentioned, some microprocessors, such as the Alpha 21264 [11, 13, 15], implement latch-based out-of-order issue queues. As shown in Figure 1, each entry in a latch-based queue consists of a set of latches that hold the required instruction information, a pair of comparators for detecting source operand availability (source operand register specifiers are held in the left latch in Figure 1), and selection logic to select instructions to issue from the ready pool. In addition, after an instruction issues, all entries behind the issued one are shifted forward to fill the slot. We refer to this as queue *compaction*. This is the function of the multiplexer shown in Figure 1, which can either hold the current instruction information or pass instruction information forward from the prior entry to fill a "hole" caused by an issuing instruction. Thus, a latch-based issue queue with compaction has the advantage of maintaining the oldest to youngest program order, making it straightforward to give issue priority to the oldest instructions. Furthermore, implementing the issue queue with latches and multiplexers is attractive due to its design simplicity, modularity, and resulting ease of verification.

However, the major disadvantage of latch-based designs is their high power dissipation due to the high power of the latches themselves and the power consumed in compaction. Compaction entails shifting instructions in the queue every cycle and can be a major source of power consumption. Furthermore, studies have shown that overall performance is largely independent of what selection policy is used (oldest first, position based, *etc.* [6]). As such, not only may a compaction strategy be unsuitable for low power operation, it may not be critical to achieving good performance. Note that a non-compacting issue queue can be implemented with a slight modification to the latch-based design shown in Figure 1. The connections between entries (from the output of the latch to the right multiplexer input of the next entry) are removed and instead the input bus (from dispatch) is connected to the right multiplexer input of all entries.

An alternative to a latch-based design is a CAM/RAM-based issue queue (shown in Figure 1) in which the source operand numbers are placed in the CAM structure and the remaining instruction information is placed in the RAM structure. The number of entries in the CAM/RAM structure corresponds to the size of the issue queue. The

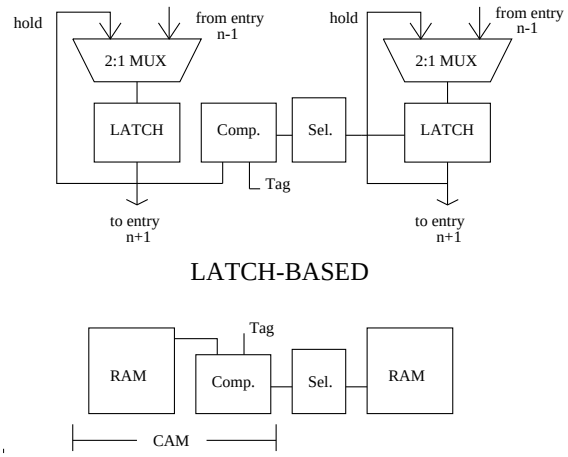

Figure 1. Latch-based (with compaction) and CAM/RAM-based issue queues

CAM/RAM structure is arguably more complex in terms of design and verification time. Also, it does not support compaction, if that feature is deemed necessary for performance. However, because of the lower power density of CAM/RAM logic relative to random logic, the CAM/RAM-based issue queue approach has the potential to eliminate hot spot problems in addition to reducing the average power dissipation of the queue.

While potentially consuming less power than a latch-based solution, a CAM/RAM-based issue queue brings a new set of power-related issues. CAM and RAM structures require precharging and discharging internal high capacitance lines and nodes for every operation. The CAM needs to perform tag matching operations every cycle. This involves driving and clearing high capacitance tag-lines, and also precharging and discharging high capacitance matchline nodes every cycle. Similarly, the RAM also needs to charge and discharge its bitlines for every read operation. Thus, considerable additional effort is required to create a power-efficient CAM/RAM design. Thus, a considerable advantage (speed and/or power) must be demonstrated to justify this additional design and verification effort.

The operation of a CAM/RAM-based issue queue is illustrated in Figure 2. During the first clock phase of issue, new instructions are placed into the queue from the dispatch stage, operand specifiers are compared and ready flags are set, and instruction selection occurs. In parallel, the CAM/RAM read bitlines are precharged. During the second clock phase, the instruction information of issued instructions is read from the

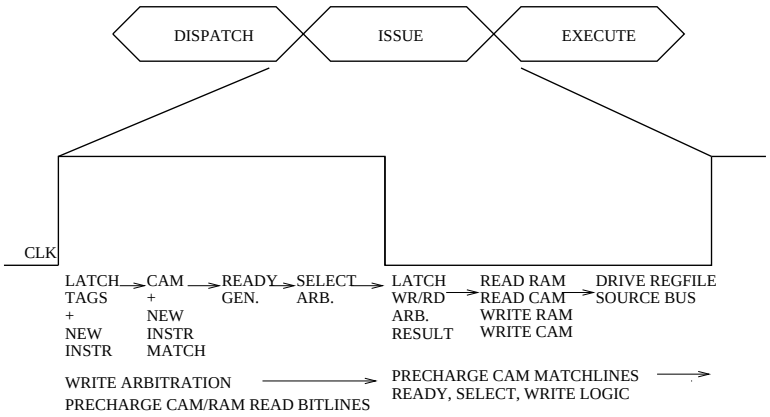

Figure 2. Operations performed by the CAM/RAM-based issue queue

CAM/RAM and the register file source bus is driven. In parallel, the CAM matchlines and ready, select, and write logic are precharged in preparation for phase one.

In order to understand the power dissipation tradeoffs in latch versus CAM/RAM-based issue queue design, we performed a detailed comparative power analysis using the IBM AS/X circuit simulation tool [17] and next-generation process parameters. Figure 3 presents the relative power dissipation of the latch-based design with compaction, the latch-based design without compaction, and the CAM/RAM-based design. Our first

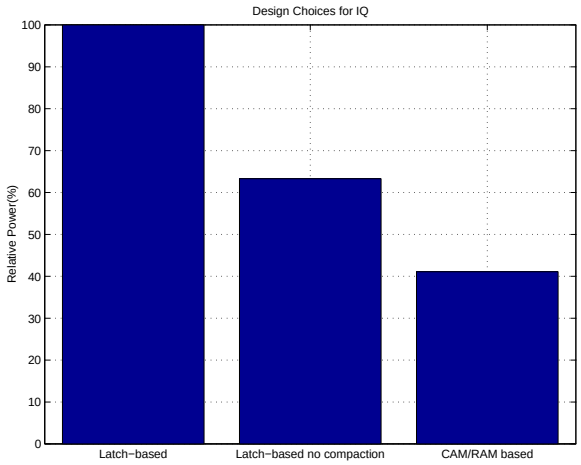

Figure 3. Relative power dissipation of issue queue alternatives

observation is that eliminating compaction has a significant impact on

power dissipation, a reduction of over 35% according to our analysis. Second, the CAM/RAM-based design is significantly more power efficient than the latch-based design without compaction, dissipating about a third less power. Our conclusion is that the added complexity of implementing a CAM/RAM-based issue queue may be justified, especially for power-constrained designs. For this reason, we selected the CAM/RAM design as the baseline for our second power-saving technique: dynamic adaptation of the issue queue size.

3. Dynamic Adaptation of the Issue Queue

Conventional issue queues have fixed-size resources in an attempt to achieve good overall performance over a range of applications. However, an individual application whose requirements are not well-matched to this particular hardware organization may underutilize issue queue resources. Even a single application may exhibit enough variability that causes uneven use of the issue queue resources during different execution phases. Thus, a fixed issue queue wastes power unnecessarily in the entries that are not in use. For example, Figure 4 shows utilization data for one of the queue resources within a high performance proces-

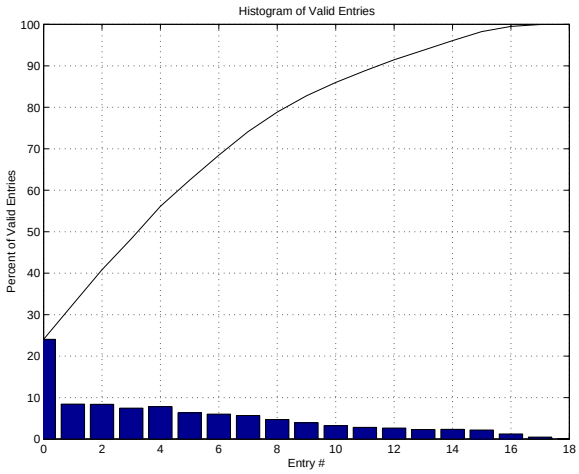

Figure 4. Histogram of valid entries for an integer queue averaged over SPECint95

sor core when simulating the SPEC95 integer benchmarks. From this figure, we see that the upper 9 entries contribute 80% of the valid entry count for the SPEC95 integer suite. One option to save power is to clock-gate each issue queue entry on a cycle by cycle basis. However, in a CAM/RAM-based design, clock gating does not address some of the

largest components of the issue queue power such as the CAM taglines, the CAM/RAM precharge logic, and CAM/RAM bitlines.

Adaptive design ideas (*e.g.*, [1, 2]) exploit workload variability to dynamically adapt the machine resources to match the program characteristics. For the issue queue, shutting down the queue in *chunks* based on application usage addresses those components that are not affected by clock gating in a CAM/RAM design, and thus has the potential to produce significant additional power savings. This idea of dynamic issue queue adaptation forms the basis of the design described in the remainder of this chapter.

3.1 Partitioned CAM/RAM Array Design and Evaluation

The partitioning of the CAM/RAM structure into chunks that can be individually disabled is illustrated in Figure 5. The effective sizes of the individual arrays can be changed at run-time by adjusting the enable inputs that control the transmission gates. For our circuit-level implementation and simulation study, a 32-entry issue queue is assumed which is partitioned into four 8-entry chunks. Thus, the queue can operate with 8, 16, 24, or 32 entries enabled. Note that the chunk at the bottom of Figure 5 is always enabled, and enabled chunks must be adjacent.

Note also that particular attention must be paid to the taglines to avoid a cycle time impact, and therefore we take a different approach than with the bitlines. As shown in Figure 5, a global tag-line is traversed through the CAM array and its local tag-lines are enabled/disabled depending on the control inputs. The sense amplifiers and precharge logic are located at the bottom of both arrays. Another feature of the design is that these CAM and RAM structures are implemented as self-timed blocks. The timing of the structure is performed via an extra dummy bitline, which has the same layout as the real bitlines, within the datapath of the CAM/RAM structures. A logic zero is stored in every dummy cell. Reading the selected cell creates a logical one to zero transition on the dummy bitline that controls the set input of the sense amplifier. (Note that the dummy bitline is precharged each cycle as with the other bitlines.) This work assumes a latching sense amplifier that is able to operate with inputs near Vdd as shown in Figure 5. When the set input is high, a small voltage difference from the memory cell passes through the NMOS pass gates of the sense amplifier. When the set signal goes low, the cross-coupled devices amplify this difference to a full rail signal

as the pass gates turn off to avoid bitline loading from the cross-coupled structure.

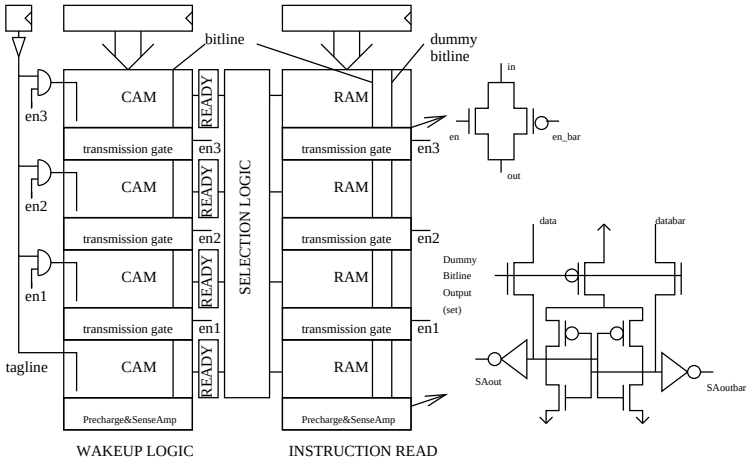

Figure 5. Adaptive CAM/RAM array partitioning

Figure 6 shows data from CAM read AS/X simulations. Here, the issue queue size is successively downsized from 32 entries to 8 entries and the data is correspondingly read. (The third, fourth and fifth signals from the top of Figure 6 correspond to the en3, en2 and en1 signals, respectively, in Figure 5.) The sixth signal waveform from the top of the figure shows the variation in latencies. When the issue queue size is 8, a faster access time is achieved because of the 24 disabled entries. This occurs because the dummy bitline enables the sense amplifiers at the exact time the data becomes available. AS/X simulations demonstrate a 56% decrease in the cycle time of the CAM array read with only 8 entries enabled. However, in this chapter we do not explore options for exploiting the variable cycle time nature of the design, but focus only on its power-saving features.

Figure 7 shows the energy savings (from AS/X simulations) achieved with an adaptive RAM array. (Note that in this figure only positive energy savings numbers are presented.) There are several possible energy/performance tradeoff points depending on the transistor width of the transmission gates. A larger transistor width results in less cycle time impact, although more energy is dissipated. By reducing the transistor width to 0.39um, one can obtain an energy savings of up to 44%. These numbers are inferred from the energy dissipation corresponding to one read operation of a 32-entry conventional RAM array and that of various alternatives of the adaptive RAM array. An interesting feature

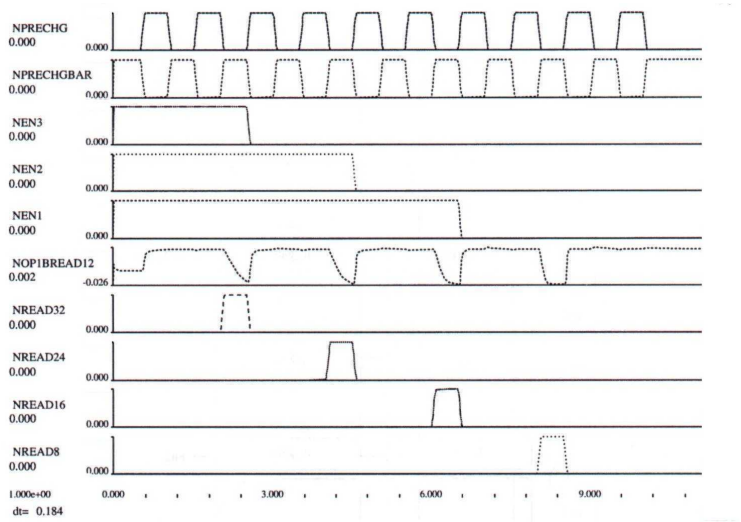

Figure 6. Adaptive CAM array read simulation results

of the adaptive design is that it achieves energy savings even with 32 entries enabled. This is because the transmission gates in the adaptive design reduce the signal swing, resulting in less energy dissipation. The adaptive RAM array delay is illustrated in Figure 8 for various numbers of enabled entries and transmission gate transistor widths. It is important to note that the delay of the additional circuitry did not affect the overall target frequency of the processor across all widths. This was true also for the CAM structure, and thus our goal of maintaining high frequency performance is achieved.

The adaptive CAM array energy and delay values are presented in Figures 9 and 10, respectively, for various numbers of enabled entries and transmission gate transistor widths. These values account for the additional circuitry that generates the final request signal for each entry (input to the arbiter logic). With this structure, a 75% savings in energy dissipation is achieved by downsizing from 32 entries to 8 entries. Furthermore, as previously demonstrated, the cycle time of the CAM array read is reduced by 56%. It should be noted that a 32 entry conventional CAM structure consumes roughly the same amount of energy as the adaptive CAM array with 32 entries. Because the CAM array dissipates ten times more energy than the RAM array (using a 2.34um transmission gate transistor width), a 75% energy savings in the CAM array corresponds to a 70% overall issue queue energy savings. (The

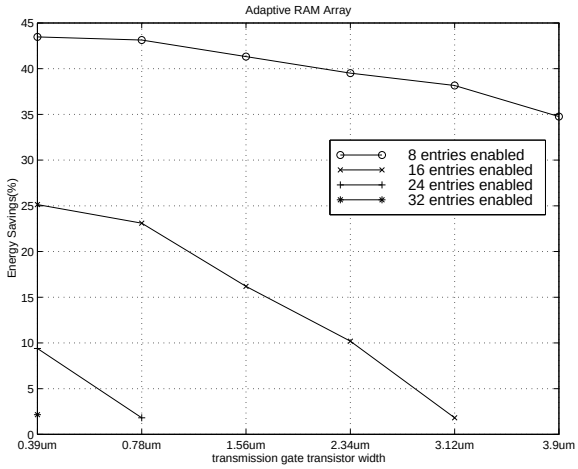

Figure 7. Adaptive RAM array energy savings

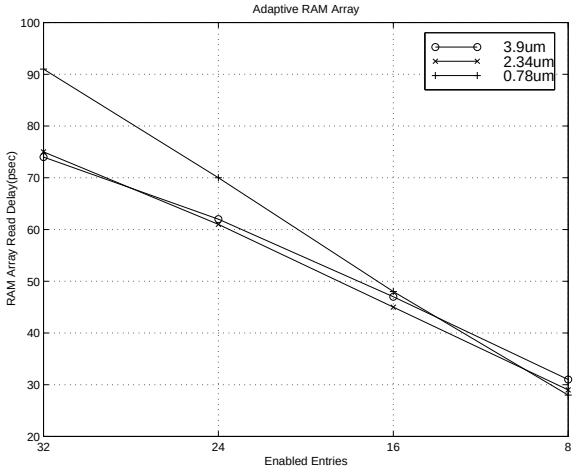

Figure 8. Adaptive RAM array delay values

energy savings takes into account the overhead of the shutdown logic described in Section 3.3.)

3.2 Dynamic Adaptation Algorithms

Thus far, we have explored the potential power savings via dynamic adaptation of the issue queue size. In other words, we have designed a specific, circuit-level solution that allows the possibility of such adaptation, and we have quantified the energy savings potential when the

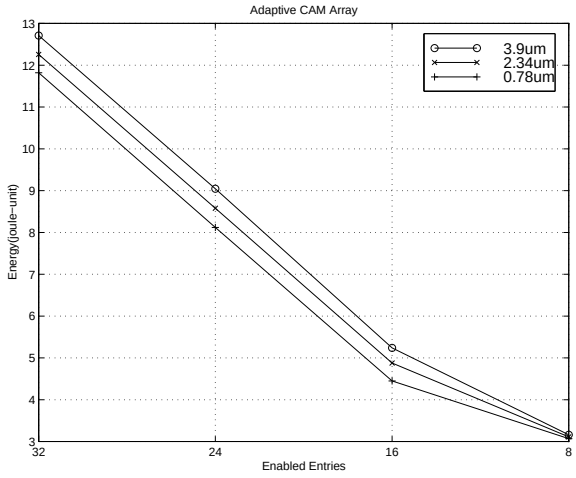

Figure 9. Adaptive CAM array energy values

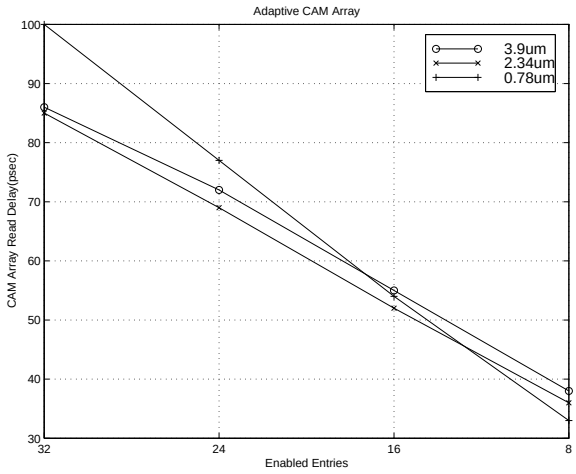

Figure 10. Adaptive CAM array delay values

queue is downsized. In our simulations, we have always factored in the overhead of the extra transistors which result from the run-time resizing hardware. In this section, we explore some of the alternate algorithms (or *decision logic*) one may use in determining what chunks of the queue should be disabled at different points of execution.

The issue unit (in conjunction with the upstream fetch/decode stages) can be thought of as a *producer*. It feeds the subsequent execution unit(s) which act as consumer(s). Assuming, for the moment, a fixed (uninter-

rupted) fetch/decode bandwidth, the issue queue will tend to fill up when the issue logic is unable to sustain a matching issue bandwidth. This could happen because: i) the program dependency characteristics are such that the average number of *ready* instructions detected each cycle is less than the fetch bandwidth seen by the receiving end of the issue queue; or, ii) the execution pipe backend (*the consumer*) experiences frequent stall conditions (unrelated to register data dependencies), causing issue slot *holes.* This latter condition (ii) could happen due to exception conditions (*e.g.*, data normalization factors in floating point execution pipes, or address conflicts of various flavors in load/store processing, *etc.*). On the other hand, the *issue-active* part of the queue will tend to be small (around a value equal to the fetch bandwidth or less) if the consuming issue-execute process is faster than or equal to the producing process. Obviously, this would happen during stretches of execution when the execution pipe stalls are minimal and the issue bandwidth is maximal, as plenty of *ready* instructions are available for issue each cycle. However, one may need a large issue queue window just to ensure that enough *ready* instructions are available to maximize the issue bandwidth. On the other hand, if the stretch of execution involves a long sequence of relatively independent operations, one may not need a large issue queue. So, it should be clear, that even for this trivial case, where we assume an uninterrupted flow of valid instructions into the issue queue, the decision to resize the queue (and in the right direction) can be complicated. This is true even if the consideration is limited only to IPC performance, *i.e.*, if the objective is to always have *just enough* issue queue size to meet the execution needs and dependency characteristics of the variable workload. If the emphasis is more on power reduction, then one can perhaps get by with a naive *utilization-based* algorithm for size adaptation, provided that the average IPC loss across workloads of interest can be kept within acceptable limits.

The pseudocode for one such utilization-based algorithm is listed below:

utilization-based algorithm {
  if ($present\_IPC < factor * last\_IPC$)
      increase_size;
  else if ($counter <= threshold1$)
      issue_queue_size=8;
  else if (($counter > threshold1$)&&($counter <= threshold2$))
      issue_queue_size=16;
  else if (($counter > threshold2$)&&($counter <= threshold3$))
      issue_queue_size=24;

```
    else
        issue_queue_size=32;
}
```

The *counter* is a measure of the number of valid entries in the queue during a fixed period of time (in cycles) called the *cycle window*. At the end of the cycle window, the counter is read and there are four possible actions. The issue queue size is increased to the next highest size if the present IPC is a *factor* lower than the last IPC during the last *cycle window*. This guarding mechanism attempts to limit the performance loss of adaptation. Otherwise, depending on the comparison of counter values with certain threshold values, the decision logic may do the following: i) increase issue queue size by enabling higher order entries, ii) retain the current size, or iii) decrease the size by disabling the highest order entries. Note that a simple NOR of all the active instructions in a chunk ensures that all entries are issued before the chunk is disabled.

An alternative approach to the decision logic, related to that proposed in [10], is to devise a *parallelism-based* metric for issue queue adaptation. Depending on the level of parallelism (distant or close), the issue queue is upsized or downsized. The level of parallelism is estimated by observing where in the Reorder Buffer (ROB) instructions are issued from. Namely, if instructions are frequently issued from deep in the ROB, then the application has distant parallelism (for which a large issue queue may be needed), while the opposite is true if most instructions issued are the oldest (for which a small issue queue may suffice). A counter is associated with each of four sections of a 64-entry portion of the ROB. A particular counter is incremented for every instruction that issues from that ROB section. At the end of the cycle window, the counter values are read, added to form a total count, and acted upon according to the following algorithm:

parallelism-based algorithm {
  if ($present\_IPC < factor * last\_IPC$)
      increase_size;
  else if ($counter\_0\_15 > threshold1 \times total\_count$)
      issue_queue_size=8;
  else if ($counter\_16\_31 > threshold2 \times total\_count$)
      issue_queue_size=16;
  else if ($counter\_32\_47 > threshold3 \times total\_count$)
      issue_queue_size=24;
  else if ($counter\_48\_63 > threshold4 \times total\_count$)
      issue_queue_size=32;
  else retain_current_size;

}

As with the utilization-based algorithm, the parallelism-based approach increases the issue queue size if the IPC of the current interval has degraded by *factor* relative to the IPC of the last interval. If not, the issue queue size is set based on the fraction of instructions that have issued from each section of the 64-entry portion of the ROB. Note that the algorithm favors small-size issue queue configurations over larger ones, in that the *counter* values for the oldest section of the ROB are examined first, and if the condition is met, the size is set irregardless of the *counter* values of other sections. An alternative is to take the opposite approach: walk from the "distant parallelism" section of the ROB towards the "nearby parallelism" section.

Both decision logic schemes require event counts to be gathered within a cycle window. We call the combination of the decision logic and these hardware counters the *shutdown logic*. A primary goal in designing the shutdown logic is to keep the transistor count and energy dissipation overheads to within tolerable limits. To evaluate whether this goal can be achieved, we perform a circuit-level design and characterization of the shutdown logic for the utilization-based approach, and estimate the overhead for the parallelism-based scheme as well.

3.3 Shutdown Logic

Figure 11 illustrates the high-level operation of the shutdown logic. It consists of bias logic at the first stage followed by the statistics process&storage stage. The activity information is first filtered by the bias logic and then it is fed to the process&storage stage where the information is fed to counters. At the end of the *cycle window*, this data passes through the decision logic to generate the corresponding control inputs.

The 32-entry issue queue is partitioned into 8-entry chunks that are separately monitored for activity. The bias logic block monitors the activity of the issue queue in 4-entry chunks. This scheme is employed to decrease the fan-in of the bias logic. The bias logic simply gathers the activity information over four entries and averages them over each cycle. The activity state of each instruction may be inferred from the *ready flag* of that particular queue entry. One particular state of interest is when exactly half of the entries in the monitored chunk are active. One alternative (the Bias Logic table in Figure 11) is to statically choose either active or not active in this particular case. Another approach (the Adaptive Bias Logic in Figure 11) is to dynamically change this choice by making use of an extra logic signal variable. We chose the former approach due to its simplicity.

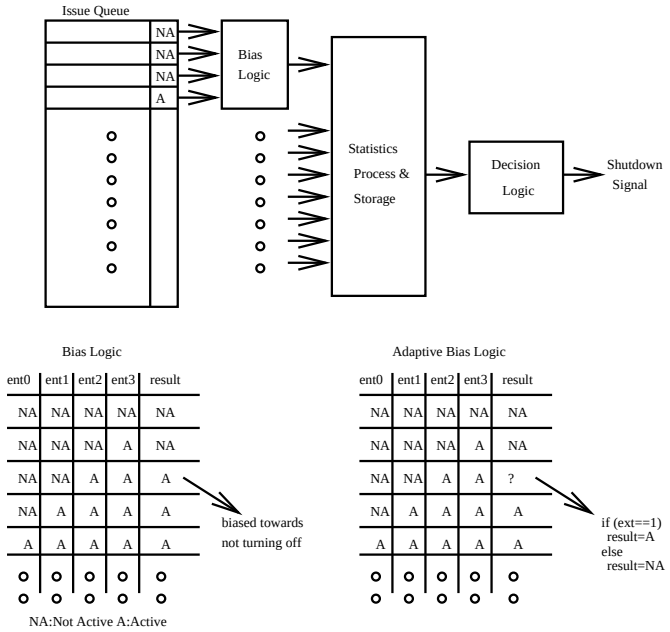

Figure 11. High-level structure of shutdown logic and logic table for bias logic

The statistics process&storage stage, which is shown in Figure 12, is comprised of two different parts. The detection logic provides the value that will be added to the final counter. It gathers the number of active chunks from the bias logic outputs and then generates a certain value (*e.g.*, if there are two active 4-entry chunks, the detection logic will generate binary two to add to the final counter). The second part, which is the most power hungry, is the flip-flop and adder pair (forming the counter). Each cycle, this counter is incremented by the number of active clusters (4 entry chunks). In this figure one can also see the function of the detection logic. The zeros in the inputs correspond to the non-active clusters and the ones to active clusters. The result section shows which value in binary should be added. For 32 entries, two of these detection circuits and a small three-bit adder are required to produce the counter input. One of the detection logic units covers the upper 16 entries and the other one covers the bottom 16 entries.

Table 2 shows the complexity of the shutdown logic in terms of transistor count. From this table it is clear that the extra logic adds only a small amount of complexity to the overall issue queue. AS/X simulations show that this extra circuitry dissipates on average 3% of the power dissipated by the whole CAM/RAM structure.

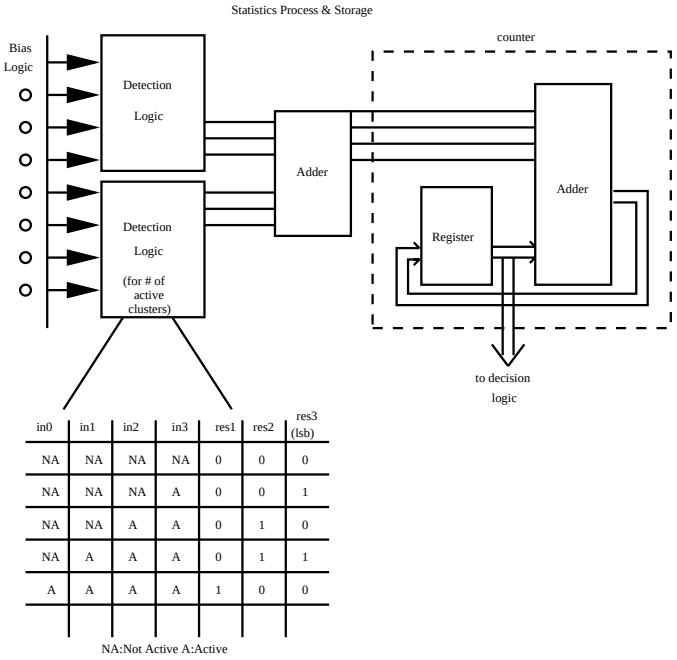

Figure 12. Shutdown logic statistics process and storage stage

Issue Queue Number of Entries	Transistor Counts Issue Queue	Transistor Counts Shutdown Logic	Complexity of Shutdown Logic
16	28820	802	2.8%
32	57108	1054	1.8%
64	113716	1736	1.5%
128	227092	2530	1.1%

Table 2. Complexity of shutdown logic in terms of transistor count

We have also estimated the transistor count overhead for the parallelism-based algorithm. This approach uses four counters, each of which counts the number of active entries in a 16-entry section of the ROB, and adds them together to form a total count. The transistor count overhead is almost 8%, or four times that of the utilization-based algorithm.

These overheads can be tolerated assuming that significant power savings can be realized through dynamic adaptation without significant performance degradation. In the next section, we present a microarchitectural analysis to evaluate this premise.

Branch predictor	comb. of bimodal and 2-level Gag
Fetch and Decode Width	16 instructions
Issue Width	8
Integer ALU/Multiplier	4/4
Floating Point ALU/Multiplier	2/2
Memory Ports	4
L1 Icache, Dcache	64KB 2-way
L2 unified cache	2MB 4-way

Table 3. Simplescalar simulator parameters

3.4 Microarchitecture Simulation-Based Results

We used SimpleScalar-3.0 [5] to simulate an aggressive 8-way superscalar out-of-order processor. The simulator has been modified to model separate integer and floating point queues. We chose a workload of six of the SPEC2000 integer benchmarks (each of which is run for 400 million instructions), and characterize the performance and power dissipation of conventional and adaptive 32-entry integer queues. The simulation parameters are summarized in Table 3. We first analyze the utilization-based algorithm with different *factor* and *cycle window* values to find the values that work best over all of the benchmarks. We also experimented with different *threshold* values, eventually settling on values of 7, 15, and 23 for $threshold1$, $threshold2$, and $threshold3$, respectively. Figures 13 and 14 show the power savings and performance degradation for each benchmark as well as the overall average with different *factor* and *cycle window* values. To estimate the power savings, we assumed a power variation profile which is essentially linear in the number of entries, based on the circuit-level simulation data reported earlier. We also take into account the shutdown logic overhead. The performance degradation and power savings are both relative to a fixed 32-entry integer issue queue.

The different *factor* and *cycle window* values present different power-performance tradeoff points. As expected, a decrease in *factor* increases the power savings but also the performance degradation. The effect of varying the *cycle window* provides less consistent results. However, note the negative power savings results with *mcf* using the larger *cycle windows* of 8K and 16K. This occurs because at this coarse level of dynamic adaptation, the 32-entry configuration is always selected. The use of the smaller *cycle windows* allows the dynamic adaptation algorithm to capture the finer-grain phase-change behavior of *mcf*, resulting in smaller configurations being selected. Note also that over all of these

benchmarks, the use of smaller *cycle windows* results in a higher power savings and a lower performance degradation than when larger *cycle windows* are used.

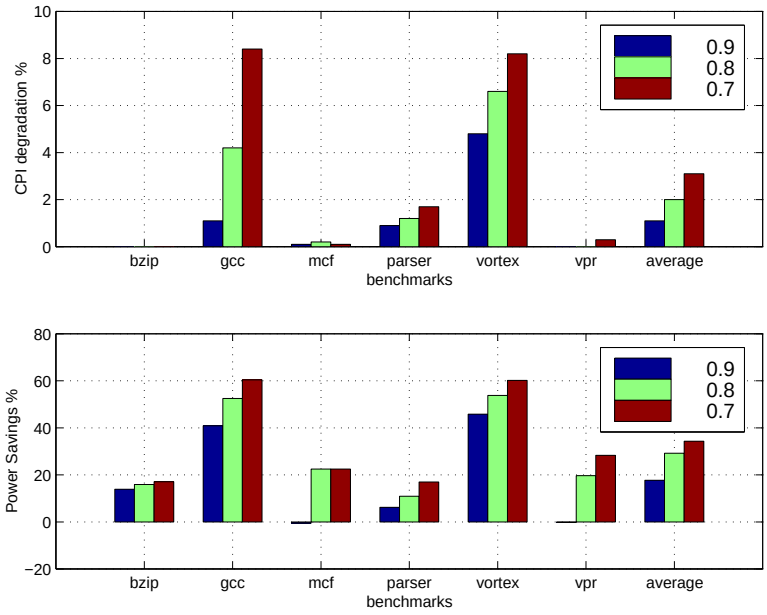

Figure 13. CPI degradation and power savings are plotted over all the benchmarks. *Cycle window*=4K. *Factor* varies as 0.9, 0.8, 0.7

Figure 15 shows the percentage of time each queue size is used with the dynamic algorithm. These results demonstrate a broad range of workload variability. For *mcf*, the 24-entry issue queue configuration is used throughout almost its entire execution whereas, for *vortex* and *gcc*, only 8 and 16 entries are largely used. The remaining benchmarks as well display characteristics that differ from these three benchmarks.

Figure 16 illustrates the variation in queue usage during execution of an Unmanned Airborne Vehicle (UAV) application for a conventional queue, and the queue size chosen by the utilization-based adaptive algorithm. This graph demonstrates how the algorithm is able to identify phase changes and adapts the issue queue size accordingly during the execution of this application.

We also experimented with a number of threshold values corresponding to the parallelism-based algorithm, ultimately finding that values of 0.9, 0.3, 0.1, and 0.05 for the oldest to youngest ROB instruction sections worked best over all of the benchmarks. Figure 17 shows a comparison of the utilization and parallelism-based algorithms in terms of

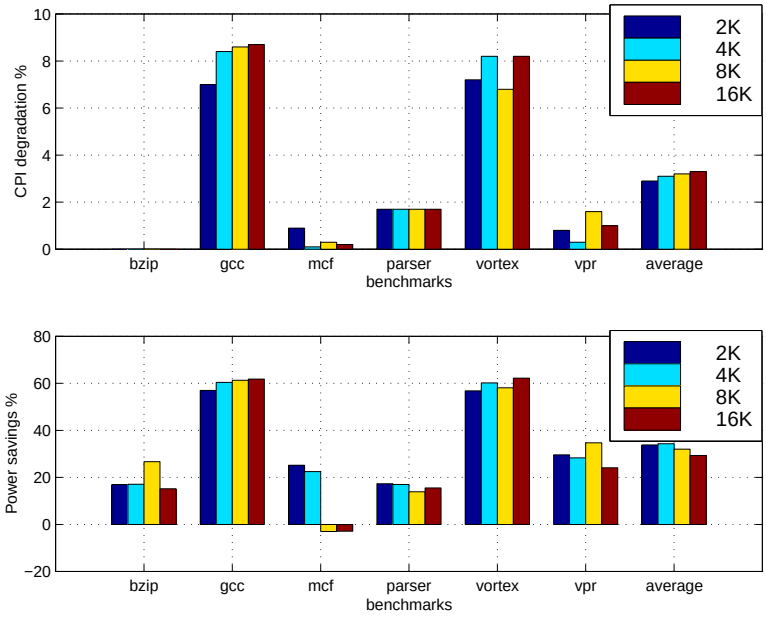

Figure 14. CPI degradation and power savings are plotted over all the benchmarks. *Factor*=0.7. *Cycle window* varies as 2K, 4K, 8K, 16K

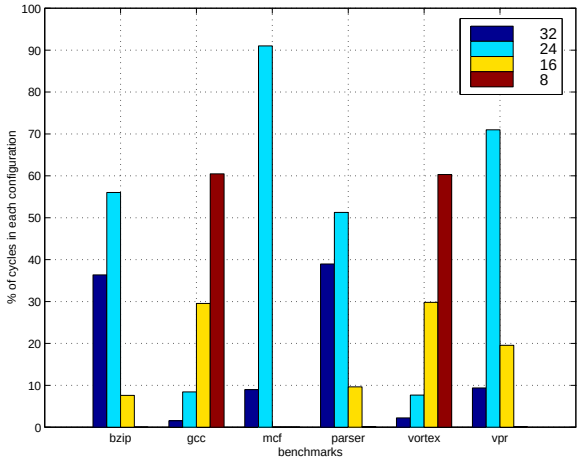

Figure 15. Percentage of utilization for each queue size with the dynamic adaptation (*factor*=0.7, *cycle window*=4K)

power savings and performance degradation for each benchmark as well as the overall averages. We make several observations from these results. First, the results vary significantly among benchmarks, with some (like

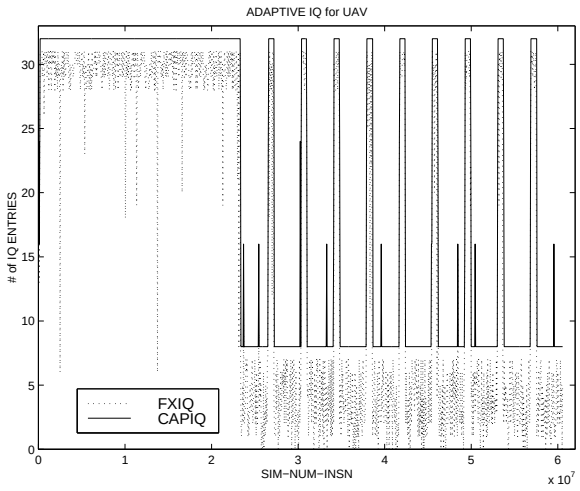

Figure 16. Conventional issue queue utilization and adaptive queue size decisions for the UAV application

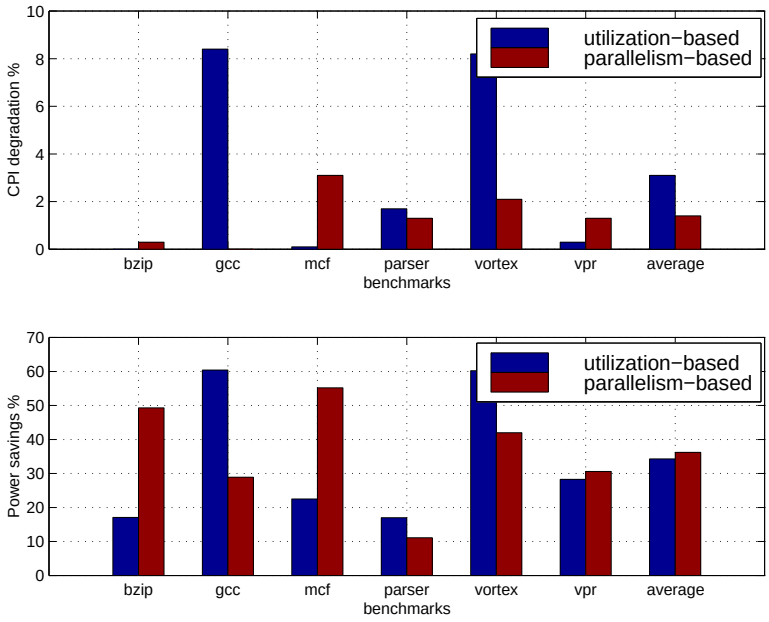

Figure 17. Comparison of the utilization and parallelism-based algorithms (*factor*=0.7, *cycle window*=4K)

gcc) exhibiting greater power savings and performance degradation with the utilization-based algorithm, while others (like *mcf*) experiencing the

opposite effect. We expect that fine-tuning the *factor*, *cycle window*, and *threshold* values to each benchmark, potentially via profiling and feedback optimization, would yield significantly more consistent results. Second, the parallelism-based algorithm exhibits slightly better power savings with half of the performance degradation of the utilization-based approach. However, recall that the parallelism-based approach has a four-fold greater transistor count overhead than the utilization-based approach.

Overall, the utilization-based approach achieves a 35% savings in issue queue energy with only a 3% average performance degradation as compared with a standard CAM/RAM design. Compared with the latch-based design with compaction, the CAM/RAM design with dynamic adaptation achieves over a 70% reduction in issue queue power dissipation with only a 3% average performance degradation. Such a dramatic power reduction has the potential to greatly reduce overall chip power in designs where the issue queue power dominates the overall power consumption as well as reduce thermal hot spot problems, leading to significant power-related cost savings.

4. Conclusions

We present two techniques for increasing the power efficiency of the issue queues in out-of-order microprocessors. The first is the replacement of the latch-based issue queue used in many processors with a CAM/RAM-based structure. We find that a CAM/RAM-based design affords a significant power savings over the latch-based design, which may justify its added design and verification complexity in power-sensitive environments.

The second approach is on-the-fly dynamic adaptation of the issue queue size to match application characteristics. Through a detailed circuit-level analysis, we determine that for a 32-entry adaptive issue queue that can be divided into four equal-size chunks, we conclude that the delay associated with the extra circuitry required to partition the queue is small enough so as not to unduly impact processor cycle time. Furthermore, we find that the transistor count and energy dissipation overheads of the adaptive control algorithm logic was small compared to the significant power savings that were realized. When combined, these techniques achieve over a 70% reduction in issue queue power dissipation with only a 3% average performance degradation.

Future work includes exploring alternate hardware algorithms for queue-size adaptation, pursuing improvements at the circuit level that provide better configuration flexibility, investigating methods for exploiting the

self-timed issue queue capability, and exploring feedback-based optimizations that tailor the adaptive control algorithms to the application.

5. Acknowledgements

We wish to thank John Wellman, Prabhakar Kudva, Victor Zyuban and Hans Jacobson for many interesting discussions and helpful hints. We also wish to thank Rajeev Balasubramonian for his help in building up the microarchitectural simulation infrastructure.

References

[1] D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. Proc. ISCA-25, pp. 282-292, June/July 1998.

[2] D. H. Albonesi. The Inherent Energy Efficiency of Complexity-Adaptive Processors. Proc. ISCA Workshop on Power-Driven Microarchitecture, June 1998.

[3] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. 33rd International Symposium on Microarchitecture, pp. 245-257, December 2000.

[4] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Dynamic Memory Hierarchy Performance Optimization. Proc. ISCA Workshop on Solving the Memory Wall Problem, June 2000.

[5] D. Burger and T. Austin. The Simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[6] M. Butler and Y.N Patt. An investigation of the performance of various dynamic scheduling techniques. Proc. ISCA-92, pp. 1-9, May 1992.

[7] G. Cai. Architectural level power/performance optimization and dynamic power estimation. Proc. of the Cool Chips Tutorial, in conjunction with Micro-32, 1999.

[8] R. Canal and A. Gonzalez. A low-complexity issue logic. Proc. ACM Int'l. Conference on Supercomputing (ICS), pp. 327-335, June 2000.

[9] D. Folegnani and A. Gonzalez. Reducing the power consumption of the issue logic. Proc. ISCA Workshop on Complexity-Effective Design, June 2000.

[10] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. Proc. ISCA-01, June 2001.

[11] M. K. Gowan, L. L. Biro, D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. Design Automation Conference, June 1998.

[12] L. Gwennap. Power issues may limit future CPUs. Microprocessor Report, 10(10), August 1996.

[13] R. Kessler. The Alpha 21264 microprocessor. IEEE Micro, 19(2): 24-36, March/April 1999.

[14] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, F. Baez. Reducing power in high-performance microprocessors. Design Automation Conference, June 1998.

[15] K. Wilcox and S. Manne. Alpha Processors: A history of power issues and a look to the future. Proc. of the Cool Chips Tutorial, in conjunction with Micro-32, 1999.

[16] K.Yeager. The Mips R10000 superscalar microprocessor. IEEE Micro, 16(2): 28-41, April 1996.

[17] AS/X User's Guide. IBM Corporation, 1996.

[18] The National Technology Roadmap for Semiconductors, Semiconductor Industry Association, 1999.