An Adaptive Issue Queue for Reduced Power at High Performance

Alper Buyuktosunoglu[3], Stanley Schuster[1], David Brooks[1,2], Pradip Bose[1], Peter Cook[1], and David Albonesi[3]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY
[2] Department of Electrical Engineering, Princeton University, NJ
[3] Department of Electrical and Computer Engineering, University of Rochester, NY
buyuktos@ece.rochester.edu

Abstract. Increasing power dissipation has become a major constraint for future performance gains in the design of microprocessors. In this paper, we present the circuit design of an issue queue for a superscalar processor that leverages transmission gate insertion to provide dynamic low-cost configurability of size and speed. A novel circuit structure dynamically gathers statistics of issue queue activity over intervals of instruction execution. These statistics are then used to change the size of an issue queue organization on-the-fly to improve issue queue energy and performance. When applied to a fixed, full-size issue queue structure, the result is up to a 70% reduction in energy dissipation. The complexity of the additional circuitry to achieve this result is almost negligible. Furthermore, self-timed techniques embedded in the adaptive scheme can provide a 56% decrease in cycle time of the CAM array read of the issue queue when we change the adaptive issue queue size from 32 entries (largest possible) to 8 entries (smallest possible in our design).

1 Introduction

The out-of-order issue queue structure is a major contributor to the overall power consumption in a modern superscalar processor, like the Alpha 21264 and Mips R10000 [1,2]. It also requires the use of complex control logic in determining and selecting the ready instructions. Such complexity, besides adding to the overall power consumption, also complicates the verification task. Recent work by Gonzalez et al., [3,4] has addressed these problems, by proposing design schemes that reduce either the control logic complexity [3] or the power [4] without significantly impacting the IPC performance. In [3], 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 this *ready queue* at reduced complexity, without any associative lookup. A separate *first-use* table is used to hold instructions, indexed by unavailable operand register specifiers. Only those instructions that are the first-time consumers of these pending operands are stored

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2000, LNCS 2008, pp. 25–39, 2001.
© Springer-Verlag Berlin Heidelberg 2001

in this table. 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 only holds instructions with pre-determined availability times of their source operands. Since 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 [3], 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, in spite of the simplicity and elegance of the control logic. In [4], 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. The work in [4] leverages previous work [5,6] in dynamically sizing the issue queue. In both [3] and [4], the authors show that the IPC loss is very small with the suggested modifications to the issue queue structure and logic. Also, in [4], 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 [3] or [4]. Without such analysis, it is difficult to gauge the cycle-time impact or the extra power/complexity of the augmented design.

In our work, we propose a new adaptive issue queue organization and we evaluate the power savings and the logic overhead through actual circuit-level implementations and their simulation. This work was done as a part of a research project targeted to explore power-saving opportunities in future, high-end processor development within IBM. Our scheme is simpler than that reported in [3, 4] in that it does not introduce any new data storage or access structure (like the first-use table or the wait queue in [3]). Rather, it proposes to use an existing framework, like the CAM/RAM structure commonly used in the design of issue queues [8]. However, the effective size of the issue queue is dynamically adapted to fit the workload demands. This aspect of the design is conceptually similar to the method proposed in [4] but our control logic is quite different.

2 Power and Performance Characteristics of a Conventional (Non-adaptive) Issue Queue

The purpose of the issue queue is to receive instructions from the dispatch stage and forward *ready instructions* to the execution units. An instruction is ready to issue when the data needed by its source operands and the functional unit are available or will be available by the time the instruction is ready to read the operands, prior to execution.

Many superscalar microprocessors, such as the Alpha 21264 [1] and Mips R10000 [2] use a distributed issue queue structure, which may include separate queues for integer and floating point operations. For instance in the Alpha 21264 [9], the issue queue is implemented as flip-flop latch-based FIFO queues with a *compaction* strategy, i.e., every cycle, the instructions in the queue are shifted to fill up any *holes* created due to prior-cycle issues. This makes efficient use of the queue resource, while also simplifying the wake-up and selection control logic. However, compaction entails shifting instructions around in the queue every cycle and depending on the instruction word width may therefore be a source of considerable power consumption. Studies have shown that overall performance is largely independent of what selection policy is used (oldest first, position based, etc.) [10]. As such, the compaction strategy may not be best suited for low power operation; nor is it critical to achieving good performance. So, in this research project, an initial decision was made to avoid compaction. Even if this means that the select arbitration must be performed over a window size of the entire queue, this is still a small price to pay compared to shifting multiple queue entries each cycle.

Due to the above considerations, a decision was made to use a RAM/CAM based solution [8]. Intuitively, a RAM/CAM would be inherently lower power due to its smaller area and because it naturally supports a *non-compaction* strategy. The RAM/CAM structure forms the core of our issue queue design. The op-code, destination register specifier, and other instruction fields (such as the instruction tag) are stored in the RAM. The source tags are stored in the CAM and are compared to the result tags from the execution stage every cycle. Once all source operands are available, the instruction is ready to issue provided its functional unit is available. The tag comparisons performed by the CAM and the checks to verify that all operands are available constitute the *wakeup* part of the issue unit operation. While potentially consuming less power than a flip-flop based solution, the decision of using a RAM/CAM structure for the issue queue is not without its drawbacks. CAM and RAM structures are in fact inherently power hungry as they need to precharge and discharge 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. Our research on low-power issue queue designs was focused on two aspects: (a) Innovating new circuit structures, which reduce power consumption in the basic CAM/RAM structure; and (b) Dynamic adaptation of the effective

CAM/RAM structure by exploiting workload variability. This paper describes the work done on the second aspect. However, dynamic queue sizing can degrade CPI performance as well. Part of the design challenge faced in this work was to ensure that the overall design choices do not impact performance significantly, while ensuring a substantial power reduction.

Non-adaptive designs (like the R10000 and Alpha 21264) use fixed-size resources and a fixed functionality across all program runs. The choices are made to achieve best overall performance over a range of applications. However, an individual application whose requirements are not well matched to this particular hardware organization may exhibit poor performance. Even a single application run may exhibit enough variability that causes uneven use of the chip resources during different phases. Adaptive design ideas (e.g., [5]) exploit the workload variability to dynamically adapt the machine resources to match the program characteristics. As shown in [5], such ideas can be used to increase overall performance by exploiting reduced access latencies in dynamically resized resources.

Non-adaptive designs are inherently power-inefficient as well. A fixed queue will waste power unnecessarily in the entries that are not in use. Figure 1 shows utilization data for one of the queue resources within a high performance pro-

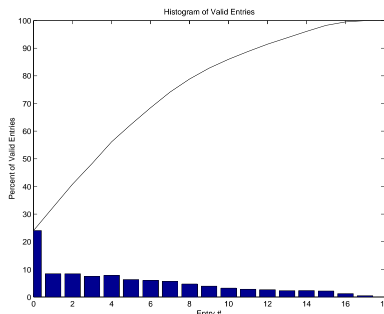

Fig. 1. Histogram of valid entries for an integer queue averaged over SPECint95

cessor core when simulating the SPECint95 benchmarks. From this figure, we see that the upper 9 entries contribute to 80% of the valid entry count. Dynamic queue sizing clearly has the potential of achieving significant power reduction as other research has demonstrated as well [4,6]. One option to save power is to clock-gate each issue queue entry on a cycle by cycle basis. However, clock gating alone does not address some of the largest components of the issue queue power such as the CAM taglines, the RAM/CAM precharge logic, and RAM/CAM bitlines. So a scheme which allows shutting down the queue in *chunks* based on usage reductions to address these other power components can produce significant additional power savings over clock gating. This idea forms the basis of the design described in this paper.

3 Adaptive Issue Queue Design

In this section, we discuss the adaptive issue queue design in detail. First, we describe the high-level structure of the queue. Then, we present partitioning of the CAM/RAM array and the self-timed sense amplifier design. Finally, we discuss the shutdown logic that is employed to configure the adaptive issue queue at run-time.

3.1 High-Level Structure of Adaptive Issue Queue

Our approach to issue queue power savings is to dynamically shut down and re-enable entire blocks of the queue. Shutting down blocks rather than individual entries achieves a more coarse-grained precharge gating. A high-level mechanism monitors the activity of the issue queue over a period of execution called the cycle window and gathers statistics using hardware counters (discussed in section 3.3). At the end of the cycle window, the decision logic enables the appropriate control signals to disable and enable queue blocks. A very simple mechanism for the decision logic in pseudocode is listed below.

```
if (present_IPC < factor * last_IPC)
     increase_size;
else if (counter < threshold_1)
     decrease_size;
else if (counter < threshold_2)
     retain_current_size;
else increase_size;
```

At the end of the cycle window, there are four possible actions. The issue queue size is ramped up to next larger queue size of the current one 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.

3.2 Partitioning of the RAM/CAM Array and Self-Timed Sense Amplifiers

The proposed adaptive CAM/RAM structure is illustrated in Figure 2. 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. For the taglines, a separate scheme is employed in order to avoid a cycle time impact. 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

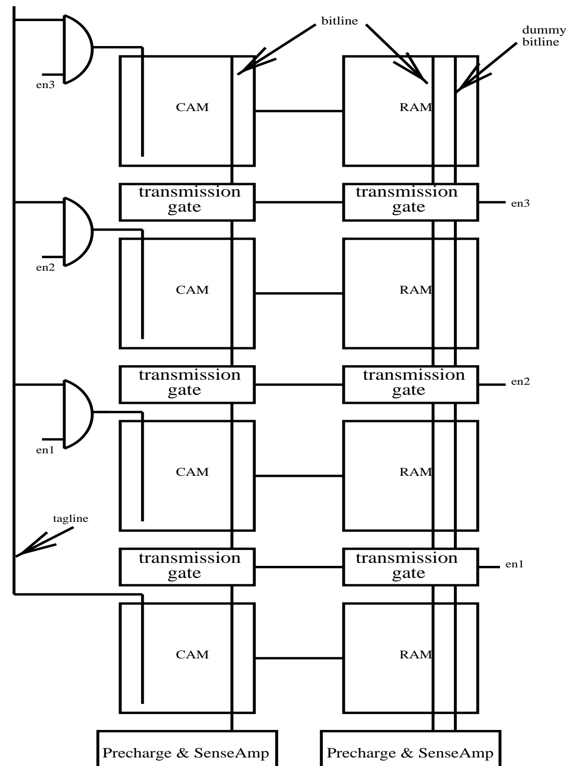

Fig. 2. Adaptive CAM/RAM structure

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 within the datapath of CAM/RAM structures, which has the same layout as the real bitlines. A logic zero is stored in every dummy cell. A reading operation of 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. 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 the cross-coupled structure from bitlines load. When the issue queue size is 8, a faster access time is achieved because of the 24 disabled entries. The self-timed sense amplifier structure takes advantage of this feature by employing the dummy bitline to allow faster operation, i.e., the dummy bitline enables the sense amplifiers at the exact time the data becomes available. Simulations show that one may achieve up to a 56% decrease in the cycle time of the CAM array read by this

method. Therefore, downsizing to a smaller number of entries results in a faster issue queue cycle time and saves energy, similar to prior work related to adaptive cache designs [11,12,13]. However, in this paper we do not explore options for exploiting the variable cycle time nature of the design, but focus only on its power-saving features.

3.3 Shutdown Logic

A primary goal in designing the shutdown logic is not to add too much overhead to the conventional design in terms of transistor count and energy dissipation. Table 1 shows the complexity of the shutdown logic in terms of transistor count.

Table 1. Complexity of shutdown logic in terms of transistor count

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%

From this table it is clear that the extra logic adds only a small amount of complexity to the overall issue queue. AS/X [14] simulations show that this extra circuitry dissipates 3% of the energy dissipated by the whole CAM/RAM structure on average. Figure 3 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 is to statically choose either active or not active in this particular case. Another approach is to dynamically change this choice by making use of an extra logic signal variable. (See Adaptive Bias Logic in Figure 3.)

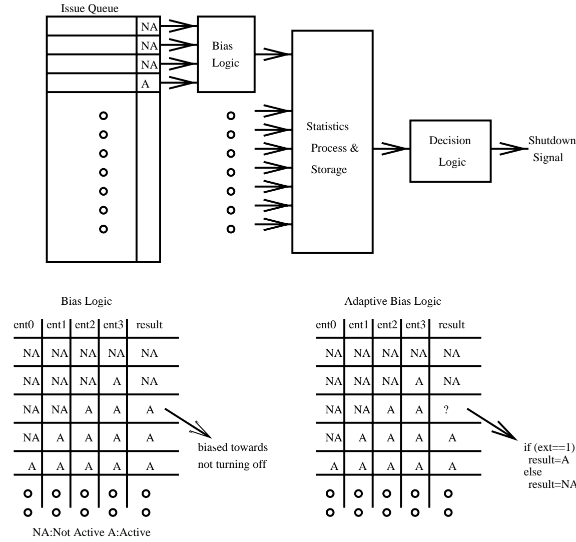

Fig. 3. High-level structure of shutdown logic and logic table for bias logic

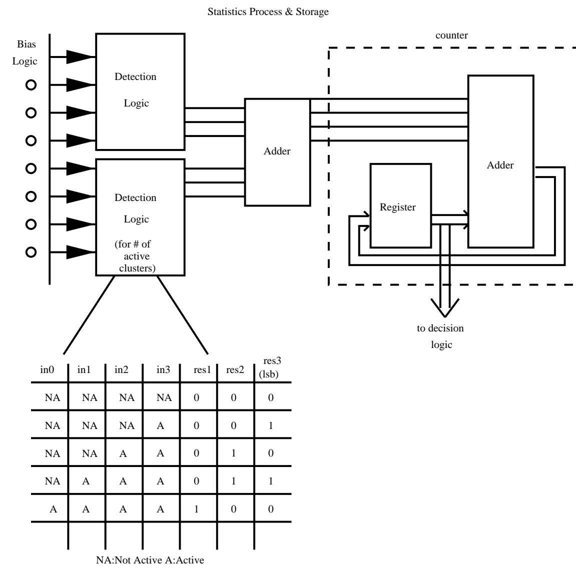

Fig. 4. Statistics process and storage stage for shutdown logic

The statistics process&storage stage, which is shown in Figure 4, 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 8-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 (8 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.

4 Simulation Based Results

In this section, we first present circuit-level data and simulation results. Later, we discuss microarchitecture-level simulation results that demonstrate the workload variability in terms of issue queue usage.

4.1 Circuit-Level Data

Figure 5 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. The cycle time impact of the additional circuitry did not affect the overall target frequency of the processor across all cases. (This was true also for the CAM structure.) By going down to 0.39um transistor width, 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. (The size of the queue is varied over the value points: 8, 16, 24 and 32.) An interesting feature 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 therefore resulting in less energy dissipation.

The adaptive CAM array energy and delay values are presented in Figure 6 and Figure 7, 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, 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 2.34um transmission gate transistor width) a 75% energy savings in the CAM array corresponds to a 70% overall issue queue energy savings (shutdown logic overhead is included).

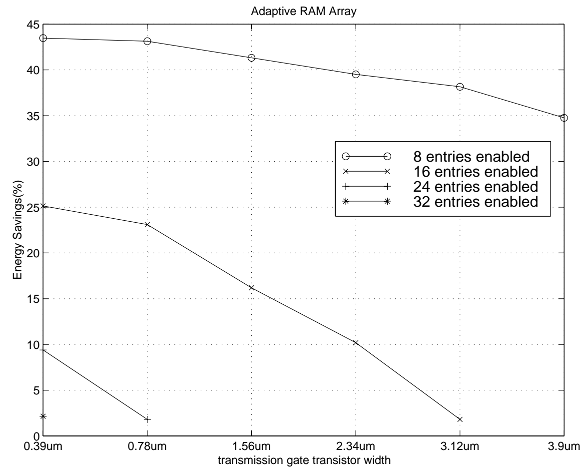

Fig. 5. Adaptive RAM array energy savings

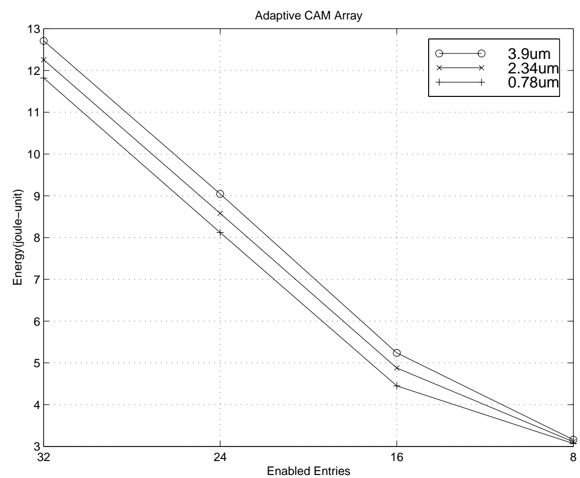

Fig. 6. Adaptive CAM array energy values

4.2 Microarchitecture-Level Simulation and Results

The work reported thus far in this paper demonstrates 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, through simulation, the energy savings potential when the queue is sized downwards. In our simulations, we have always factored in the overhead of the extra transistors, which result from the run-time resizing hardware.

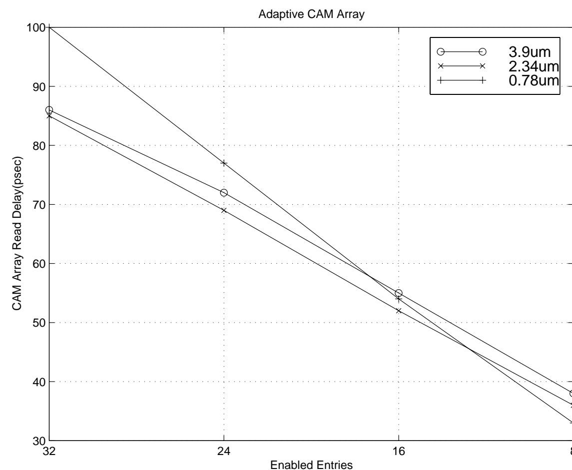

Fig. 7. Adaptive CAM array delay values

In this section, we begin to address the following issues: (a) what are some of the alternate algorithms one may use in implementing the *decision logic* referred to earlier (see section 3.1)? That is, how (and at what cycle windows) does one decide whether to size up or down? (b) What are the scenarios under which one scheme may win over another? (c) How does a simple naive resizing algorithm perform from a performance and energy perspective, in the context of a given workload?

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 (uninterrupted) 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: (a) 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, (b) the execution pipe backend (*the consumer*) experiences frequent stall conditions (unrelated to register data dependencies), causing issue slot *holes.* This latter condition (b) 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: up or down) can be complicated. This is true even if the consideration is limited only to CPI 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 heuristic for size adaptation, provided the simulations validate that the average IPC loss across workloads of interest is within acceptable limits.

To illustrate the basic tradeoff issues, first, we provide data that shows the variation of CPI with integer issue queue size across several SPEC2000 integer benchmarks (see Figure 8). We used SimpleScalar-3.0 [15] to simulate an ag-

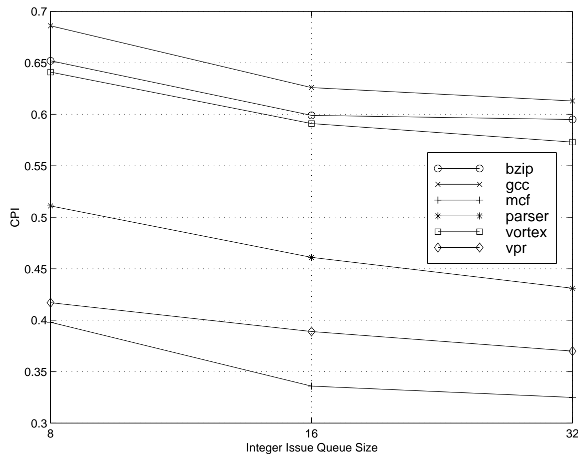

Fig. 8. CPI sensitivity to issue queue size

gressive 8-way superscalar out-of-order processor. The simulator uses separate integer and floating point queues. The simulation parameters are summarized in Table 2. The data in Figure 8 shows that for most of the benchmarks simulated, there is considerable variation in CPI as integer issue queue size varies between 8 and 32. In order to gain insight into the potential of our adaptive issue queue, we implemented the algorithm discussed in Section 3.1 in SimpleScalar. We chose a cycle window size of 8K cycles, as this provided the best energy performance tradeoff compared with the other cycle windows that we analyzed. We ran each benchmark for the first 400 million instructions.

Our dynamic algorithm picks the appropriate size for the next cycle window by estimating the usage in the last cycle window, and comparing this value with certain threshold values. The algorithm also compares the IPC of the last interval with the present interval IPC. For this purpose, we also analyzed the

Table 2. Simplescalar simulator parameters

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

configurations with different *factor* values. Threshold values are adjusted such that, if the issue queue utilization for a certain size is at the border value of its maximum size (e.g., for an issue queue size of 8 entries, the border is 7 entries) then the issue queue size is ramped up to the next larger size. Figure 9 shows what

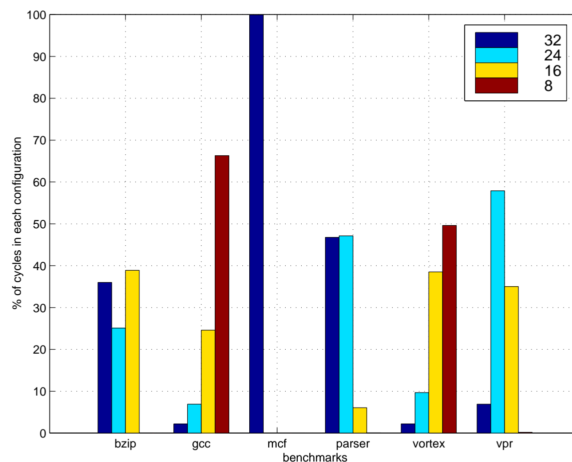

Fig. 9. Percentage of utilization for each queue size with the dynamic adaptation

percentage of the time each queue size was used with the dynamic algorithm with *factor* set to be 0.7. Table 3 shows the energy savings and CPI degradation for each benchmark as well as the overall average with different factor values. These different factor values represent different energy/performance tradeoff points. To estimate the energy savings, we assumed an energy variation profile which is essentially linear in the number of entries, based on the circuit-level simulation data reported earlier in Figure 6. We also take into account the shutdown logic

Table 3. Energy savings and CPI degradation with different factor values

factor		bzip	gcc	mcf	parser	vortex	vpr	average
0.9	CPI degradation %	0.0	1.6	0.3	0.9	3.3	0.0	1.0
	Energy Savings %	14.0	43.3	-3.0	6.4	43.6	5.1	18.2
0.8	CPI degradation %	0.0	4.3	0.3	1.0	4.8	0.0	1.7
	Energy Savings %	15.9	53.9	-3.0	8.9	51.2	14.8	23.6
0.7	CPI degradation %	0.0	8.6	0.3	1.7	6.8	1.4	3.1
	Energy Savings %	26.6	61.3	-3.0	13.7	58.1	33.6	31.7

overhead. CPI degradation and energy savings are both relative to a fixed 32-entry integer issue queue.

The results from Figure 9 demonstrate the broad range of workload variability. For mcf, the full 32 entry queue is used throughout its entire execution whereas for vortex and gcc, only 8 and 16 entries are largely used. For bzip, the algorithm almost equally chooses issue queue sizes of 32, 24, and 16 entries. For parser, the 32 and 24 entry issue queue configurations dominate whereas for vpr, 24 or 16 entries are largely used. On average, this very naive algorithm provides a 32% decrease in the issue queue energy (61% maximum) with a CPI degradation of just over 3% with factor set to be 0.7.

5 Conclusion

We examine the power saving potential in the design of an adaptive, out-of-order issue queue structure. We propose an implementation that divides the issue queue into separate chunks, connected via transmission gates. These gates are controlled by signals which determine whether a particular chunk is to be disabled to reduce the effective queue size. The queue size control signals are derived from counters that keep track of the *active state* of each queue entry on a cycle-by-cycle basis. After a (programmable) cycle window, the decision to resize the queue can be made based on the activity profile monitored. The major contribution of this work is a detailed, circuit-level implementation backed by (AS/X) simulation-based analysis to quantify the net power savings that can be achieved by various levels of queue size reduction. We also simulated a dynamic adaptation algorithm to illustrate the scenarios where the resizing logic would size the queue up or down, depending on the particular priorities of performance and energy.

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

Acknowledgements. We wish to thank John Wellman, Prabhakar Kudva, Victor Zyuban and Hans Jacobson for many interesting discussions and helpful hints.

References

1. R. Kessler. The Alpha 21264 microprocessor. IEEE Micro, 19(2): 24-36, March/April 1999.
2. K.Yeager. The Mips R10000 superscalar microprocessor. IEEE Micro, 16(2): 28-41, April 1996.
3. R. Canal and A. Gonzalez. A low-complexity issue logic. Proc. ACM Int'l. Conference on Supercomputing (ICS), pp. 327-335, Santa Fe, N.M., June 2000.
4. D. Folegnani and A. Gonzalez. Reducing the power consumption of the issue logic. Proc. ISCA Workshop on Complexity-Effective Design, June 2000.
5. D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. Proc. ISCA-25, pp. 282-292, June/July 1998.
6. D. H. Albonesi. The Inherent Energy Efficiency of Complexity-Adaptive Processors. Proc. ISCA Workshop on Power-Driven Microarchitecture, June 1998.
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. S. Palacharla, N. P. Jouppi and J. E. Smith. Complexity-effective superscalar processors. Proc. ISCA-97, pp. 206-218, June 1997.
9. 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.
10. M. Butler and Y.N Patt. An investigation of the performance of various dynamic scheduling techniques. Proc. ISCA-92, pp. 1-9.
11. 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.
12. 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, December 2000.
13. M. D. Powell, S.H. Yang, B. Falsafi, K. Roy, T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2000.
14. AS/X User's Guide. IBM Corporation, New York, 1996.
15. D. Burger and T. Austin. The Simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.