A Dynamically Reconfigurable Mixed In-Order/Out-of-Order Issue Queue for Power-Aware Microprocessors

Yu Bai and R. Iris Bahar
Division of Engineering, Brown University, Providence, RI 02912
{Yu_Bai, Iris_Bahar}@Brown.edu

Abstract

In this work we focus on power-aware solutions for the issue queue in an out-of-order superscalar processor. We propose two different schemes. Our first approach partitions the issue queue into FIFOs such that only the instructions at the head of each FIFO may request to issue. We then dynamically monitor the FIFO usage and disable FIFOs that are not being efficiently used. In our second approach we also use a FIFO scheme, but dynamically vary the number and size of each FIFO simultaneously while at the same time keeping the total number of issue queue entries constant. We analyze both approaches and compare them in terms of the performance and power reduction. We find that although the first scheme of completely disabling issue queue entries is more straight-forward to implement, it may not be the best option, particularly for floating point applications. Our best experimental result shows an average power saving of 27.3% in the issue queue with a performance degradation of only 2.7%.

1. Introduction

Although power is of great concern, the main driving force in high-end microprocessor design is still performance. To achieve high performance for the broadest set of applications, many complex architectural features are included in these general-purpose, high-performance microprocessors. While the goal of overall high performance is generally met, it comes at a cost of high power dissipation. Moreover, different applications may vary widely in their degree of instruction-level parallelism (ILP), their branch behavior, and/or their memory access behavior. As a result, the datapath resources required to implement these complex features may not be optimally utilized by all applications; however, some power will be dissipated by these resources regardless of utilization.

To better address power concerns, a good design strategy should be flexible enough to dynamically reconfigure available resources according to the program's needs. In this work, we choose to focus specifically on the "reconfigurability" of the issue queue, since it is a large source of the total power dissipation in out-of-order superscalar processors. As an example, according to [15], the issue logic is responsible for 46% of the total power in the Alpha 21264.

In our proposed design, we partition the issue queue into several sets (or FIFOs). Only the instructions at the head of each FIFO are visible to the request and selection/arbitration logic; therefore, each FIFO issues in-order though overall instruction issue is out-of-order. The FIFO structure is different from a reservation station in that instructions in a FIFO can be issued to any functional unit. However, the strict ordering of instructions in a particular FIFO can reduce complexity and power dissipation of the request and selection logic.

Our main contribution is in showing the power saving potential of a dynamically reconfigurable, mixed in-order and out-of-order issue queue. Using feedback from various hardware performance monitors, we dynamically modify the size of the issue queue and/or the total number of FIFOs in the issue queue in order to adjust the mix of in-order and out-of-order issuing of instructions, thereby saving power whenever possible.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the implementations of the issue queue and all monitoring techniques. Section 4 talks about the power estimations. Section 5 describes our processor model and simulation tools. Results are provided in section 6. Section 7 offers conclusions.

2. Related Work and Motivation

The idea of adjusting available datapath resources to better match program requirements is not new. In [8], Maro *et al.* implemented a hardware mechanism to dynamically monitor processor performance and reconfigure the machine. Using feedback from the performance monitors, part of the integer and/or float point pipelines were disabled during runtime to save power. A similar approach was used in [1] where issue width was varied to allow disabling of a cluster of functional units. Other works proposed dynamically reducing the number of active entries in the instruction window according to processor needs in order to save power [3, 5, 11]. In addition, the work of [12] studied temporally and spatially local algorithms (intra-frame) and their integration with a global algorithm (inter-frame) for real-time multimedia applications. Their goal was to save power by simultaneously varying instruction window size and the

number of active functional units. The shortfall of these approaches is that while dynamically adjusting issue queue size may reduce power in the wake-up and selection logic, doing so narrows the scope of instructions available for exposing ILP. This can be potentially harmful to performance when ILP can only be exposed using a large instruction window. Another limitation of these approaches is that they do not distinguish among valid entries in the issue queue and as such make all of them visible to the wake-up and selection logic. This can be very power inefficient if instructions remain in the issue queue for many cycles before they are ready to "wake-up" and issue.

In [7], Ghiasi *et. al* proposed a technique to use IPC variation to reduce power consumption in microprocessors. In this work, the micro-architecture can be adjusted to meet the desired performance which is indicated by the operating system (OS). They changed the superscalar machine from out-of-order to in-order according to the program's needs. This idea is similar to ours, dynamically mixing in-order and out-of-order issuing; however, our approach is driven by feedback from the hardware, rather than by OS and can therefore adjust the machine at a finer grain.

Other static techniques have been proposed to reduce overall design complexity and power. In [9], the authors introduced a technique called Data-Flow Prescheduling to reduce the complexity of the issue stage by ordering instructions before they enter the issue buffer. Seng *et. al* analyzed the characteristic of critical and non-critical instructions and designed a static mixed in-order and out-of-order issue queue to reduce power consumption in processors [13]. The work of Palacharla *et. al* analyzed several specific areas of register renaming, instruction window wake-up and selection logic, and operand bypassing [10]. They found that the window wake-up and selection logic as well as operand bypass are probably the most important for future power-saving work. They then presented an alternative design with a faster clock and simplified wake-up and selection logic which puts chains of dependent instructions into FIFO buffers and issues instructions from multiple buffers in parallel.

The drawback of Palacharla's approach, just as with other techniques that use fixed-sized data structures, is that different applications may not all benefit from only one type of issue queue configuration. For instance, if a program has a rather large amount of ILP during portions of its execution, then restricting instruction issue into a fixed order using relatively few FIFOs may greatly restrict performance. Likewise, if instruction execution is limited by long chains of dependent instructions, then using as few as two FIFOs may be enough to meet issue requirements and not hamper performance. This point is emphasized further in Figure 1. The issue queue is fixed at 64 entries, but the number of entries in each FIFO is varied from 1 to 64 (e.g. 32, 2-entry FIFOs correspond to 32 FIFOs each with 2 entries). In the figure, we show individual results for a set of benchmarks which show interesting trends; however, the overall average reflects results for all benchmarks simulated. We see that some benchmarks such as *apsi*, *swim*, and *wave5* are very sensitive to FIFO ordering, whereas others, like *compress*, *gcc*, *li*, and *fpppp*, are not. In fact, for *li* ordering instructions in a FIFO even improves performance since it effectively avoids executing some wrong path instructions. Using a 1-entry FIFO as a base case, 2-entry and 4-entry FIFO configurations have a performance drop of 3% and 13% on average. As the size of the FIFO increases to 8, the performance loss becomes 30%. When the whole issue queue becomes a single FIFO, we lose 84% of the performance.[1] In our approach, we not only aim to retain some of the power benefits of using FIFOs in the issue queue design, as was suggested in [10], but also aim to minimize its affect on performance by allowing the number of FIFOs to be varied according to a program's issue needs.

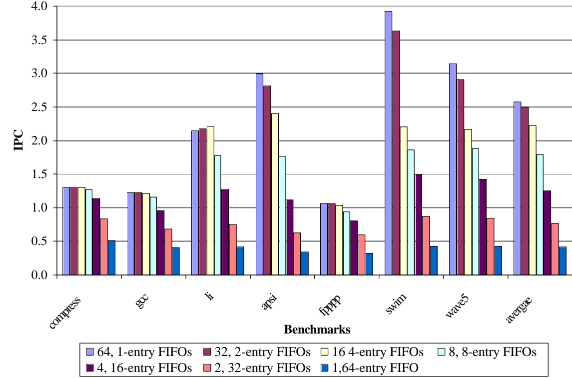

Figure 1. IPC comparison using fixed-sized FIFOs.

3. Implementation

The goal of our approach is to dynamically adjust the active size of the issue queue to more closely match a program's issue needs in order to save power. We implement two schemes:

Scheme #1: We completely disable some underutilized FIFOs in the issue queue according to feedback from performance monitors. In this case, we limit exposure to potential ILP by shrinking the overall size of the issue queue and restricting issue to only instructions at the head of the FIFO.

Scheme #2: We retain the same number of issue queue entries at all times but vary the number and size of the FIFOs simultaneously. In this case, we have more flexibility in exposing potential ILP than in the first scheme, while

[1] Our results are somewhat different from those reported in [10], since our assumptions and baseline simulation model are different; we do not assume 1 cycle latency for all functional units or a perfect instruction cache.

still making it appear to the request and selection/arbitration logic that the queue is actually smaller.

If performance monitors indicate that ILP is increasing and/or performance is suffering, a larger fraction of the issue queue is turned back on or made visible to the request and selection logic. Our issue queue design requires two main components: (1) a reconfigurable issue queue partitioned into a set of smaller queues that each issue in-order, and (2) hardware performance monitors used to determine the optimal configuration for the issue queue over a fixed interval of cycles. We discuss the design of these two components in more detail below.

3.1. Issue Queue Design

The architecture of the whole pipeline is shown in Figure 2. Note that the pipeline allows for up to 6 instructions to be issued, executed and committed each cycle. In order to implement our techniques, the issue queue is divided into several FIFOs. All new instructions can only be inserted at the tail of a FIFO and the instructions to be issued can only come from the head of the FIFO. All entries in the issue queue except the heads are invisible to the request and selection logic.

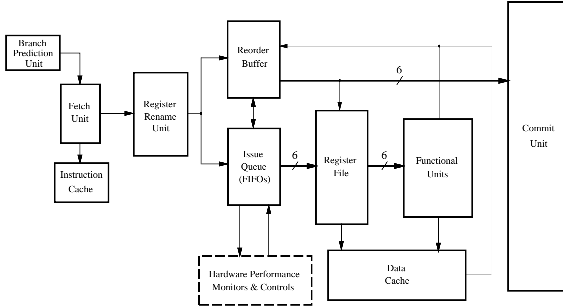

Figure 2. Pipeline organization.

Figure 3 shows the FIFO organization for our first scheme. In this example, the issue queue is statically divided into 4 FIFOs, each with two entries. Note that only half the entries in the issue queue are visible to the request and selection logic. When we detect that the issue queue is underutilizing some FIFOs, we can disable one or more of these FIFOs without degrading performance. We must ensure that a FIFO has been drained of all valid entries before it is disabled. Figure 3 shows how disabling some FIFOs reduces the number of instructions potentially bidding for an issue slot, thus saving power in the wake-up and selection logic. In addition, we also save power by not having to update the ready status of the disabled instruction entries.

Similarly, Figure 4 shows in our second technique how the FIFOs are reorganized in different low-power modes assuming an 8-entry issue queue. In full-power mode (FPM), there are 8 individual FIFOs, each containing a single entry.

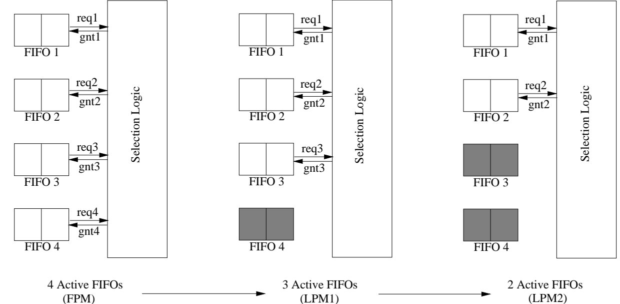

Figure 3. IQ scheme using variable number of FIFOs.

Each entry is at the head of the FIFO, therefore all entries in the issue queue are visible to the request and selection logic. When the number of FIFOs is cut down to 4 (LPM1 mode), each FIFO now holds 2 entries for a total of 8 instructions, but now only the 4 instructions at the heads of the FIFOs can bid for an issue slot, regardless of how many instructions are actually ready. The total number of FIFOs is adjusted using feedback from the performance monitors; however, the total number of issue queue entries must remain fixed at all times, thereby requiring the number of entries in each FIFO to be adjusted simultaneously. We assume that there is no cycle overhead in changing from one FIFO configuration to another since each instruction already has a set of arbiter enable signals indicating its arbiter assignment, and these signals can be disabled according to the instructions position within the FIFO. In this scheme we have more flexibility in how we configure the issue queue compared to the first, but power savings comes only from reduced activity in the request and selection logic.

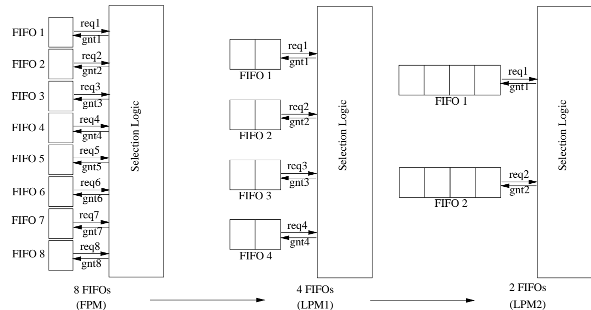

Figure 4. IQ scheme using variable sized FIFOs.

Given that only a fraction of the entries will be visible to the request and selection logic, it is important that instructions be placed in the FIFOs such that most (if not all) of the ready instructions appear at the head of a FIFO. Otherwise, performance is more likely to suffer. We tried several instruction assignment strategies when the instructions are initially put into FIFOs in the dispatch stage. Performance for three different strategies (random, round-robin, and dependency-based) are tested and analyzed.

The dependency-based scheme is similar to the one presented in [10]. As an instruction is decoded and dispatched

to the issue queue, we attempt to place it in the same FIFO as one or both of its source dependencies. If the instruction is already ready, it is steered to a new empty FIFO. If the instruction has only one pending operand and the producer is the tail of the corresponding FIFO, it is steered to the same FIFO as the producer if possible; otherwise, it is steered to a new empty FIFO. If two operands of the instruction have not been computed, the instruction is steered to the same FIFO as one of its producers. We implement a Last Operand Predictor (LOP) [14] to predict which of two operands will probably become available later. The LOP is implemented by a 2048-entry table of two-bit saturating counters, indexed by program counters[2]. We first try to put the instruction into the same FIFO as its later arrived producer if the producer is the tail of the FIFO and the FIFO still has room. If we fail, we try to put the instruction in the same FIFO as the other producer. If we fail again, the instruction is steered to a new empty FIFO. In all cases, if an empty FIFO is required for dispatching an instruction and one is not available, the dispatch unit is stalled until a FIFO becomes empty. This can only cause a potential performance hit if the instruction is ready at the time of issue or if an instruction behind it in dispatch order is ready.

While the dependency-based scheme provides the best performance of the three strategies from our experiments, it is the most expensive to implement in hardware. Alternatively, the random strategy is simple to implement, but is not acceptable in terms of performance. Although the round-robin scheme is a reasonable alternative to the dependency-based scheme, with an average performance drop of 3.5%, we think it is still worthwhile to implement the dependency scheme. Therefore, all future discussion assumes this scheme.

3.2. Hardware Performance Monitors

We use hardware performance monitors to keep track of various statistics while a program is executing. These statistics are gathered during a fixed-sized sample period (i.e., a cycle *window*). We assume short term past behavior is a good indicator of behavior in the near future. At the end of each sample period, we determine whether to enter lower-power mode, leave it in the same mode, or return to a higher-power mode or full-power mode. In this way, reconfiguration takes place at most once within a single window. We empirically chose our cycle window size to be 1024 cycles such that it is large enough to obtain meaningful statistics over a reasonable snapshot of time, but not too large to remain in an inappropriate configuration. Since the processor may perform differently depending on whether all its resources are enabled or not, we use different combinations of monitoring techniques to determine when to enable or disable low-power mode. Our goal was to limit performance degradation to be no more than 4% of the base case. Following are the hardware monitoring mechanisms we implemented to modify the number of FIFOs. Several have already been used in previous works [1, 3, 5, 8, 11, 12], though none use the same combination of monitors.

[2]Our goal was to implement an instruction steering algorithm that provided the best performance. Without the LOP, performance is degraded by 1% on average compared to the dependency scheme with the LOP. Since this is a relatively simple structure, we assume its contribution to overall power dissipation is negligible.

Monitoring IPC: If the issue IPC is low during the current sample window, this may indicate low ILP in the program. Therefore, not all instructions in the issue queue may need to be visible to the request and selection logic. Similarly, the issue IPC can also be used to decide when to get out of low-power mode.

Detecting variations in IPC: If issue and commit rates vary significantly, this can indicate a high branch misprediction rate. By decreasing the number of FIFOs, we restrict the issue rate and indirectly limit the amount of branch mispredicted instructions issued.

Performance degradation: If the drop in IPC from one sample period to the next exceeds some threshold value, the processor should be restored to the higher-power mode.

Monitoring ready instructions: If a newly decoded instruction is immediately ready for issue, but cannot be dispatched to an empty FIFO (thus causing a stall), this can degrade performance. A high occurrence of these stall events may indicate the need to increase the number of FIFOs; a very low occurrence rate may indicate an opportunity to decrease the number.

Issue queue usage: Low issue queue occupancy rates may indicate an opportunity to reduce the number of FIFOs since the queue is being underutilized.

Non-Critical Instructions: If no instruction is placed behind a ready instruction by the time it is removed from the issue queue (i.e., no instructions depend on), the instruction is non-critical. Delaying such ready instructions won't hurt overall performance; if many non-critical instructions are identified, this may indicate an opportunity to reduce the number of FIFOs.

4. Power Estimations

To estimate the total power savings of our processor when in low-power mode, we extrapolated from available Alpha 21264 power estimates [15]. Although the issue queue design of the Alpha 21264 differs from our model in some ways (e.g. use of a single cluster design with a unified issue queue compared to a 2-cluster design with 3 separate queues), both use an out-of-order issuing scheme.

Figure 5. Request logic for one row of the scoreboard with modifications shown in gray. Taken from [4].

We assume that the issue logic includes register files, register mapping, and the issue queue. The issue queue itself includes the register scoreboard, request logic, and arbiters. We assume that half of the issue logic power results from the issue queue based on relative power numbers for the Alpha 21264 processor. Using estimates from the Alpha 21264 [15, 1], we will estimate that 60% of the power in the issue queue is consumed by the arbitration logic and 15% is consumed by the request logic. Register scoreboard and all other logic in the issue queue occupy the remaining power.

In the issue queue, we save power disabling some amount of the queue logic. When in low-power mode we pick a maximum of N out of X instructions where N is the issue width of the machine, and X represents the total number of heads in the issue queue. For instance, for the second approach, $X = M/2^i$ where M is the total number of issue queue entries, and i represents the number of times the total number of FIFOs has been cut in half. When FIFOs are disabled or the number of FIFOs is cut in half, we reduce the number of instructions visible to the arbitration logic as well. Thus, we reduce the activity on the arbiter enable signals, and the request logic and signals.

An instruction that is not at the head of a FIFO is not allowed to issue during that cycle so its request lines can be disabled. Assuming a circuit implementation similar to the Alpha 21264, we can accomplish this by only allowing the request lines for an instruction to be precharged if it is at the head of a FIFO. That is, we conditionally clock the precharge line with a *FIFO_head* signal as shown in Figure 5. The final request signal *REQ_L* can only be asserted during cycles when this instruction is at the head of a FIFO. We assume the head is updated by the end of the previous cycle, so this precharge line should not be on the critical path. Configuring the issue queue using multiple FIFOs can save significant power on the request lines since a conventional out-of-order issue queue would precharge these lines every cycle, but only the request lines for the ready instructions would remain high (on average, this may be a small fraction of the total entries in the issue queue so this eliminates a lot of "wasted" precharging and discharging).

As shown in Figure 5, the request logic also includes input for execution assignment information (i.e. logic for generating *state_cond* and *EX_cond* signals) that needs to be updated regardless of whether an entry is at the head of a FIFO or not. Only in the case that this entry is completed disabled (as is possible with the first scheme) will power associated with this extra logic be saved. We estimate that for our first scheme we can save 80% of the request logic power for a disabled entry. The remaining 20% of the power is associated with the power consumption on long wires along the whole request logic. For our second scheme, we save 40% of the power for that entry if it is not at the head of the FIFO.

Power dissipation in the arbitration logic is reduced due to the reduced activity on the request lines and by selectively inhibiting grant lines from precharging. We only need to precharge the grant lines for instructions that are at the head of the FIFO, in a similar manner as was done for the request logic shown in Figure 5. If the total number of FIFO heads in the issue queue is cut in half, this should reduce the switching (and therefore the power) on the grant lines and associated logic by approximately half as well since at most only N instructions will be allowed to issue regardless of how many instructions are ready in a given cycle. That is, the grant lines from approximately the same number of instructions would have been precharged and subsequently discharged anyway. By this reasoning, we assume that the power dissipation of the arbitration logic is directly proportional to the number of active FIFOs. In addition, if there are ever less than N FIFOs enabled, this effectively reduces the issue width of the machine, so we can save more power by disabling all the grant lines associated with the unused issue slots.

The register scoreboard logic tracks data dependencies among the instructions in the issue queue. Therefore, even if an instruction in the issue queue is not made visible to the arbitration logic, the register scoreboard still needs to update the dependency information and readiness of instruction operands. By this reasoning, we assume that with the second scheme, power dissipation in the register scoreboard remains the same regardless of the FIFO configuration. On the other hand, if we completely disable all entries in a particular FIFO, it is unnecessary for the register scoreboard to update any information with respect to these entries. In this case we save power in the register scoreboard logic according to the fraction of entries disabled.

5. Experimental Methodology

The simulator we used in this study is derived from the SIMPLESCALAR tool suite [2]. However, we added several modifications to SIMPLESCALAR to better model our reconfigurable processor. Specifically, in the original SIMPLESCALAR, the register update unit (RUU) is a combined instruction window, array of reservation stations, and re-

order buffer. In our work, the RUU is split into the reorder buffer (ROB) and the issue queue (IQ). This modification to the SIMPLESCALAR design allows us to more accurately model current and next generation processors that have separate issue and commit queues. In addition, it allows us to model an issue queue with multiple FIFOs where instructions are no longer placed in program order, thus requiring a separate reorder buffer structure when instructions retire. In the dispatch stage, new instructions are put into the reorder buffer in program order, but are steered into the issue queue's FIFOs according to their input dependencies.

Table 1 shows the complete configuration of the processor model. Note that the base case assumes an issue width of 6 and a unified 64-entry out-of-order issue queue. All comparisons in Section 6 are made to this case.

Table 1. Processor resources

Parameter	Configuration
Inst. Window	256-entry LSQ, 512-entry ROB 64-entry IQ
Machine Width	6-wide fetch, issue, commit
Fetch Queue	8
FUs & Latency	8 Int add (1), 2 Int mult/div (3/20) 4 FP add (2), 2 FP mult/div/sqrt (4/12/24) 4 Load/Store (1)
L1 Icache	32KB 2-way; 32B line; 1 cycle
L1 Dcache	32KB 2-way; 32B line; 1 cycle
L2 Cache	256KB 4-way; 64B line; 6 cycle
Memory	128 bit-wide; 20 cycles on hit, 50 cycles on page miss
Branch Pred.	4k 2lev + 4k bimodal + 4k meta 6 cycle mispred. penalty
BTB	1K entry 4-way set assoc.
RAS	32 entry queue
ITLB	64 entry fully assoc.
DTLB	64 entry fully assoc.

Our simulations are executed on a subset of the SPEC95 integer and floating point benchmarks [6]. They were compiled using a re-targeted version of the GNU *gcc* compiler with full optimization. All benchmarks are fast-forwarded for 50 million instructions to avoid startup effects. The benchmarks are then executed for 100 million committed instructions, or until they complete, whichever comes first. All inputs come from the reference set.

6. Experimental Results

As mentioned in the previous section, we chose a 64-entry issue queue for our baseline experiments. A 32-entry issue queue is probably large enough for many benchmarks, especially for integer benchmarks; however, we also noticed that many floating point benchmarks benefited from the larger queue. For these reasons, we chose to use a 64-entry issue queue. For our first scheme, we start with 16 FIFOs, each containing 4 entries, and then adjust the number of FIFOs according to feedback from the hardware monitors. For our second scheme, we start with 64 FIFOs, each containing a single entry, and then modify both number and size of FIFOs dynamically.

6.1. Results for Scheme #1

We experimented with many combinations of different performance monitors to determine when to enable and disable FIFOs. We show the best combination we found below. We list the specific monitors in order of relative importance. Using more monitors helps preserve performance (from our experiments by 1–2%); however, it may still be reasonable to use only 1 or 2 different monitors and still obtain acceptable performance and power results.

Disable one FIFO when one following condition holds:

1. less than $\frac{1}{4}$ of ready instructions are stalled;
2. less than $\frac{2}{3}$ of the FIFOs are actually used on average;
3. more than 15% of dispatched instructions are non-critical;
4. current IQ occupancy rate is less than $\frac{1}{4}$ of the average occupancy rate.

Enable one FIFO when one following condition holds:

1. current issue rate (IPC_{issue}) drops by more than 10% compared to the last window executed in full-power mode (i.e., 16 FIFOs);
2. current IPC_{issue} drops by more than 15% compared to the previous window;
3. more than $\frac{1}{3}$ of ready instructions are stalled.

Using the power estimations and the distribution of power in the issue queue we made in Section 4, we can estimate the total power saving for different logic in the issue queue and the total power saving in the entire issue queue. Table 2 shows the total power savings in the issue queue, the average number of FIFOs used, and the performance degradation for integer benchmarks. We show power saving and performance degradation compared to both the starting state, in which the issue queue is composed of 16 4-entry FIFOs, and the single-entry FIFO state, which is the same as the non-FIFO scheme. We only apply the first scheme to the integer benchmarks since the floating point benchmarks usually need the entire issue queue to extract ILP. In addition, according to Figure 1, the starting state is completely unreasonable for most floating point benchmarks because the starting state itself introduces a large loss in performance. So our first technique is not suitable for floating point benchmarks, since we have a fairly rigid starting state. For some benchmarks, it is reasonable; however, for others,

Table 2. Results for Scheme #1.

Benchmarks	Avg. # of FIFOs	64-entry IQ			
		16, 4-entry FIFOs		64, 1-entry FIFOs	
		Power Saving	Δ IPC	Power Saving	Δ IPC
compress	7.5	51.4%	3.6%	75.9%	3.6%
gcc	11.1	29.8%	3.5%	65.2%	3.9%
go	11.7	25.8%	3.7%	63.2%	4.5%
ijpeg	13.4	15.8%	2.4%	58.3%	5.1%
li	12.2	22.9%	5.8%	61.8%	2.7%
perl	12.8	19.6%	3.3%	60.1%	8.3%
average	11.5	27.6%	3.7%	64.1%	4.7%

it is not. We do a reasonable job of dynamically disabling FIFOs from this starting state, i.e., we retain performance compared to the starting state of always using 16 4-entry FIFOs, but overall performance cost may still be too high compared to the non-FIFO scheme.[3] However, even with this limitation, *compress* is still a good example to show how effectively our scheme can work; we save more than 75% of the power with a performance drop of only 3.6%. On average, the best results produced by our first scheme can save 27.6% of the issue queue power with a performance degradation of 3.7% compared to the starting state. If compared to the non-FIFO scheme, our first scheme can save 64.1% of the issue queue power, but it comes at a performance drop of 4.7% on average and 8.3% for *perl*, which we found to be not acceptable.

6.2. Results for Scheme #2

Similar to the first scheme, we also tried a number of combinations of policies for enabling and disabling the low-power modes. Following is the combination that produced overall best results in terms of both performance and power. Again, we order them according to relative importance.

Cut the number of FIFOs in half (and double the number of entries in each FIFO) when one following condition holds:

1. $(IPC_{issue} - IPC_{commit}) > 1.0$;
2. less than 3% of ready instructions are stalled;
3. $IPC_{issue} < 2.7$; (threshold lowered by 0.2 for each successive reduction in number of FIFOs.);
4. current IQ occupancy rate is less than 20% of the average occupancy rate;
5. $(AVG_IPC_{issue} - IPC_{issue}) > 0.15$ (threshold increased by 0.15 for each successive reduction in number of FIFOs.).

Double the number of FIFOs (and halve the number of entries in each FIFO) when one following condition holds:

1. current IPC_{issue} drops by more than 8% compared to the previous window;
2. current IPC_{issue} drops by more than 6% compared to the last window executed in full-power mode.
3. more than 15% of ready instructions are stalled.

Figure 6 shows the percentage of time the processor is in each mode. For several floating point benchmarks such as *applu*, *apsi*, *mgrid*, and *swim*, our FIFO technique can't reduce the total number of FIFOs since these benchmarks need more flexibility in reordering instructions to maintain performance. However, for most integer benchmarks, we can cut the FIFOs at least in half for a significant portion of the running time. Some benchmarks can even stay in an 8-FIFO state for a significant period of time. For *compress*, the system spends virtually no time in the single-entry FIFO state. In addition, for the floating point benchmark *fpppp*, we are still able to reduce the number of FIFOs to 2 for about 34% of the execution time. Overall, we can spend an average of 61% of the time in some reduced power mode.

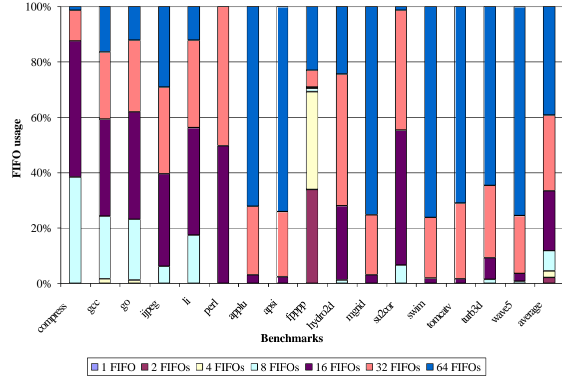

Figure 6. FIFO usage for Scheme #2. Note we always retain the number of IQ entries to 64.

Using the results from Figure 6 and the power estimations we made in Section 4, we can estimate the total power saving for the request and arbitration logic. Table 3 shows the total power reduction in the issue queue for each benchmark compared to the single-entry FIFO state, using the same power distribution assumptions as was made for the first scheme. In columns 2–3 we report percentage of power saved in the request and arbitration logic, respectively. In column 4 we report total power saved for the entire issue queue and in column 5 we show overall performance degradation.

Table 3. Results for Scheme #2.

Benchmarks	64-entry IQ			
	Power Saving			Δ IPC
	Request	Arbitration	Total	
compress	38.6%	75.9%	51.4%	0.7%
gcc	29.5%	59.9%	40.4%	2.4%
go	30.5%	62.4%	42.0%	3.5%
ijpeg	21.3%	46.0%	30.8%	2.2%
li	28.8%	60.1%	40.4%	2.6%
perl	27.9%	62.4%	41.6%	2.5%
applu	6.0%	14.6%	9.7%	1.3%
apsi	5.6%	13.6%	9.0%	2.2%
fpppp	45.6%	73.2%	50.8%	3.0%
hydro2d	19.7%	44.9%	29.9%	4.2%
mgrid	5.4%	13.1%	8.7%	1.4%
su2cor	29.4%	64.0%	42.8%	4.7%
swim	5.0%	12.3%	8.1%	4.0%
tomcatv	6.1%	14.9%	9.9%	1.9%
turb3d	8.7%	20.1%	13.4%	3.2%
wave5	5.6%	13.2%	8.8%	2.9%
average	19.6%	40.7%	27.3%	2.7%

As can be seen in the table, for several benchmarks, we manage to save more than 60% of the power from the arbitration logic and more than 30% of the request logic power.

[3] As noted earlier, *li* improves in performance when using 16, 4-entry FIFOs compared to 64, single-entry FIFOs, so this explains the performance improvement in the last column of Table 2.

This translates to saving as much as 51% of the issue queue power, with most of the savings coming from the reduced activity in the arbitration logic. Obviously, it is much easier to cut the the number of FIFOs for integer benchmarks and hence save power. We were able to save at least 30% of the issue queue power on all integer benchmarks.

It is clear that our second scheme works more efficiently than our first scheme (see Table 2 and Table 3). Although most floating benchmarks need 64 FIFOs for a large percentage of the running time, we can still find several examples, like *fpppp*, *hydro2d*, and *su2cor*, where our scheme can still be applied with small performance degradation. According to Figure 1, *fpppp* can stand both 2-entry and 4-entry FIFO states, but performance degrades by 11.8% if we continue to cut the number of FIFOs further. However, our techniques can effectively find the appropriate periods to cut the number of FIFOs while still retaining performance. Using our second scheme, *fpppp* can stay at 16-entry and 32-entry FIFO states for about 70% of the execution time with only a performance degradation of 3.0%. On average, the best results produced by our second scheme can save 27.3% of the issue queue power, with a performance degradation of only 2.7%.

7 Conclusion

In this paper, we exploit the fact that programs vary in their ILP. Our approach is to selectively reconfiguring parts of the issue queue during low ILP periods to save power without hampering performance. We designed our issue queue into variable numbers of FIFOs to facilitate this reconfiguration. By decreasing the number of entries in the issue queue or restricting the number of instructions visible to the request and selection logic, we save significant power in the issue queue while maintaining performance. Our second scheme may be particularly effective for programs where ILP is limited by long chains of dependent instructions for portions of their execution. Trying to save power by simply disabling part of the issue queue may only further hamper performance. However, by restricting which instructions are visible to the request and selection logic while retaining the same sized queue will still allow us to save power. Our results show an average power saving of 27.3% in the issue queue with a performance degradation of only 2.7%.

References

[1] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[2] D. Burger and T.Austin. The simplescalar tool set. Technical report, University of Wisconsin, Madison, 1999. Version 3.0.

[3] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power and high performance. In *Workshop on Power-Aware Computer Systems*, November 2000. Held in conjunction with the *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] J. A. Farrell and T. C. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, May 1998.

[5] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[6] J. Gee, M. Hill, D. Pnevmatikatos, and A. J. Smith. Cache performance of the spec benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[7] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*, June 2000. Held in conjunction with the *International Symposium on Computer Architecture*.

[8] R. Maro, Y. Bai, and R. I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Workshop on Power-Aware Computer Systems*, November 2000. Held in conjunction with the *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *7th International Symposium on High-Performance Computer Architecture*, January 2001.

[10] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[11] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *34th International Symposium on Microarchitecture*, December 2001.

[12] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[13] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *34th International Symposium on Microarchitecture*, December 2001.

[14] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *33rd International Symposium on Microarchitecture*, December 2000.

[15] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look to the future. In *Cool-Chips Tutorial*, November 1999. Held in conjunction with the *32nd International Symposium on Microarchitecture*.