A Low-Power In-Order/Out-of-Order Issue Queue

YU BAI and R. IRIS BAHAR
Brown University

To better address power concerns, a good design strategy should be flexible enough to dynamically reconfigure available resources according to the application's needs such that extra power is dissipated only when it is really needed. In this work, we focus on power-aware solutions for the issue queue (IQ) in an out-of-order superscalar processor. We propose two schemes that partition the IQ into FIFOs such that only the instructions at the head of each FIFO may request to issue. We then monitor the processor and dynamically vary the number and/or size of FIFOs in accordance with utilization. Experimenting with two different distributions in power dissipation, we show up to 69% reduction in power dissipation in the wakeup and arbitration loop, while constraining performance degradation to be no more than 5%.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; C.5.3 [**Computer System Implementation**]: Microcomputers—*Microprocessors*; C.4 [**Performance of Systems**]: Design Studies

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: High-performance, instruction issue logic, low power

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 data path 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.

This manuscript is an extension of work published in Bai and Bahar [2003]. In particular, we have included a more in depth analysis of our proposed implementation and provided a better qualitative analysis of power savings.

Author's address: Division of Engineering, Brown University, Providence, RI 02912; email: Iris.Bahar@Brown.edu.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2004 ACM 1544-3566/04/0600-0152 \$5.00

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 (IQ) since it is a large source of the total power dissipation in out-of-order superscalar processors. As an example, according to Wilcox and Manne [1999], the issue logic was expected to be responsible for 46% of the total power in the Alpha 21464 microprocessor.

In our proposed design, we partition the IQ into several sets (or FIFOs). Only the instructions at the head of each FIFO are visible to the arbitration/selection 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 arbitration logic.

Our main contribution is in showing the potential power savings of a dynamically reconfigurable, mixed in-order and out-of-order IQ. Using feedback from various hardware performance monitors, we dynamically modify the size of the IQ and/or the total number of FIFOs in the IQ in order to adjust the mix of in-order and out-of-order issuing of instructions, thereby saving power whenever possible. Experimenting with two different power distributions in the wakeup and arbitration loop, our best results show a savings of up to 69% of power dissipation in the loop, while keeping performance penalty to be no more than 5%.

The rest of the paper is organized as follows. Section 2 discusses related work and additional motivation. Section 3 presents the implementations of the IQ 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 data path resources to better match program requirements is not new. Maro et al. [2000] 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 run-time to save power. A similar approach was used in Bahar and Manne [2001], 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 the processor's needs in order to save power [Buyuktosunoglu et al. 2000; Folegnani and González 2001; Ponomarev et al. 2001]. In addition, Sasanka et al. [2002] 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 IQ size may reduce power in the wakeup and arbitration 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 IQ and as such make all of them visible to the wakeup and arbitration logic. This can be very inefficient if instructions remain in the IQ for many cycles before they are ready to “wake-up” and issue. Lebeck et al. tried to target this inefficiency by adding a large and fast *waiting instruction buffer* (WIB) alongside a small traditional IQ [Lebeck et al. 2002]. Instructions that depend on loads that miss in the first-level cache, are moved to the WIB from the IQ. These instructions become invisible to the arbitration logic once placed in the WIB. When the load data becomes available, all instructions dependent on this load, directly or indirectly, are then moved back to the IQ for issuing. Although the primary focus of Lebeck et al. [2002] was performance and not power, our approach is similar in that and we also prevent instructions dependent on long latency instructions from being visible to the arbitration logic; however, we are not targeting only load-dependent instructions and we do not require moving instructions from one buffer to another. In addition, our approach is dynamically adjustable to satisfy the specific needs of the application.

Ghiasi et al. [2000] proposed a technique to use IPC variation to reduce power dissipation in microprocessors. In this work, the microarchitecture 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 Michaud and Seznec [2001], 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. Another similar technique was proposed in Canal and González [2001]. Those works are different from ours, since they focus on performance instead of power issues. Seng et al. [2001] analyzed the characteristics of critical and noncritical instructions and designed a static mixed in-order and out-of-order IQ to reduce power dissipation in processors. Palacharla et al. [1997] analyzed several specific areas of register renaming, instruction window wakeup and arbitration logic, and operand bypassing. They found that the window wakeup and arbitration logic as well as operand bypass are probably the most critical components for future microprocessors. They then presented an alternative design with a faster clock and simplified wakeup and arbitration logic, which puts chains of dependent instructions into FIFO buffers and issues instructions from multiple buffers in parallel. Another paper indicated that the slower clock rate growth will be one of the biggest obstacles for the rapid improvement of the microprocessor performance in the future [Agarwal et al. 2000]. This further motivates implementation of the microarchitecture presented in Palacharla et al. [1997] to improve microprocessor performance.

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

Fig. 1. Performance comparison using fixed-sized FIFOs.

from only one type of IQ 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 IQ is fixed at 128 entries, but the number of entries in each FIFO is varied from 1 to 128 (e.g., 32, 4-entry FIFOs correspond to 32 FIFOs, each with four entries).¹ In the figure, we show individual performance results in terms of IPC, that is, the total number of committed instructions per cycle, and the average for all benchmarks simulated. We see that some benchmarks such as *art* and *wupwise* are very sensitive to FIFO ordering, whereas others, like *vortex*, *ammp*, and *mesa*, are not. Using a 1-entry FIFO as a base case, 2-entry and 4-entry FIFO configurations have a performance drop of 2% and 9% on average, respectively. The *art* and *wupwise* are main contributors to the performance degradation. As the size of the FIFO increases to 16, the performance loss becomes 29%. When the whole IQ becomes a single FIFO, we lose 65% of the performance.² In our approach, we not only aim to retain some of the power benefits of using FIFOs in the IQ design, as was suggested in Palacharla et al. [1997], but also aim to minimize its effect on performance by allowing the number of FIFOs to be varied dynamically based on a program's issue needs.

It has been shown that performance may vary not only for different applications but also even during run-time within one application's execution [Wall

¹Other processor configuration information can be found in Section 5. Performance is very dependent on how instructions are placed in FIFOs. In Figure 1, we use a dependency-based placement strategy. We will discuss three placement strategies in more detail in Section 3.

²These results are somewhat different from those reported in Palacharla et al. [1997], since our assumptions and baseline simulation model are different. The main difference is that we do not assume one-cycle latency for all functional units or a perfect instruction cache.

1991; Bahar and Manne 2001; Maro et al. 2000]. Our goal is to find an overall more flexible scheme that adjusts better to diverse and changing needs of a program. We have found that schemes based on simply disabling IQ entries such as in Buyuktosunoglu et al. [2000], Folegnani and González [2001], and Ponomarev et al. [2001] do not always provide the needed flexibility of simultaneously extracting maximal ILP and reducing energy requirements. The results and comparisons will be shown in Section 6.4.

3. IMPLEMENTATION

The goal of our approach is to save power in the IQ. We achieve this goal by dynamically adjusting the active size of the IQ to more closely match a program's needs. This effectively reduces the power in the wakeup and arbitration logic of the IQ. The wakeup logic is responsible for updating dependency information for instructions waiting in the IQ, while the arbitration logic is responsible for selecting ready instructions for execution. More details can be found in Section 4. We implemented two schemes.

Scheme 1. The IQ is composed of a number of fixed-sized FIFOs, and the number of available FIFOs is adjusted dynamically according to feedback from performance monitors. While this scheme is relatively straightforward to implement, its drawback is that available ILP may be restricted by reducing the total number of IQ entries. To overcome this limitation, we also proposed Scheme 2.

Scheme 2. We retain the same number of IQ entries at all times but vary the number and size of FIFOs simultaneously. In this case, we have more flexibility in exposing potential ILP than in the first scheme, while still making it appear to the 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 IQ is turned back on or made visible to the arbitration logic. Our IQ design requires two main components: (1) a reconfigurable IQ partitioned into a set of smaller queues that each issues in-order, and (2) hardware performance monitors used to determine the optimal configuration for the IQ 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. In order to implement our techniques, the IQ is divided into several FIFOs. Newly decoded 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 IQ except the heads are invisible to the arbitration logic.

Figure 3 shows the FIFO organization for Scheme 1 assuming an 8-entry IQ. The value shown inside the entry is the instruction index in the physical IQ organization. In this example, the 8-entry IQ is statically divided into four FIFOs, each with two entries. Note that only half the entries in the IQ are visible to the arbitration logic. When we detect that the IQ is underutilizing some FIFOs, we can disable one or more of these FIFOs without

Fig. 2. Pipeline organization.

Fig. 3. IQ scheme using variable number of fixed-sized FIFOs (Scheme 1).

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 that can potentially bid for an issue slot, thus saving power in the arbitration 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 FIFOs are reorganized in different low-power modes assuming an 8-entry IQ. In the full-power mode, there are eight individual FIFOs, each containing a single entry (i.e., a full out-of-order IQ). Each entry is effectively a FIFO head, therefore all entries in the IQ are visible to the arbitration logic. When the number of FIFOs is cut down to 4, each FIFO now holds two entries for a total of eight instructions, but now only four 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 IQ entries must remain fixed at all times, thereby requiring the number of entries in each FIFO to be adjusted simultaneously. In

Fig. 4. IQ scheme using variable sized FIFOs (Scheme 2).

this scheme we have more flexibility in how we reconfigure the IQ compared to Scheme 1, but power savings comes only from reduced activity in the arbitration logic.

There exists cycle overhead in changing from one FIFO configuration to another. For Scheme 1, when we attempt to disable one FIFO, we have to wait until the FIFO becomes empty. However, we assume that there is no overhead when we try to recover previously disabled FIFOs.³ For Scheme 2, doubling the number of FIFOs can be done immediately. However, we must drain the entire IQ before cutting the number of FIFOs to half; otherwise, different dependency chains will be mixed up inside the FIFO, thus causing potential performance and deadlock issues. Obviously, the cycle overhead is smaller for Scheme 1 because only the disabled FIFO must be drained while the entire IQ needs to be cleared for Scheme 2. From our experience, we do not see that this transitional overhead produces any serious performance hit in terms of IPC because switching to the low-power mode is done at carefully selected time periods.

In hardware, an IQ can be implemented as a set of shift registers or circular queues. A shift register consists of a fixed head; only an instruction at this head position can issue. Once it is issued, all instructions following it in the FIFO will be shifted down by one position so the next instruction in line becomes the head. By implementing the FIFO as a shift register, we are simplifying the hardware. That is, only the fixed head needs to include the hardware for request and grant logic (i.e., arbitration logic). Less logic means less area and shorter interconnect which can translate to faster logic and reduced power dissipation. This compensates for extra delay and power needed for shifting.

³To be conservative, we modeled extra overhead when activating previously disabled FIFOs — even with a 5 cycle overhead to reactivate disabled FIFOs, we found performance was unaffected.

A circular queue consists of the movable head and tail. To realize this, each circular queue is provided with registers containing the locations of the head and tail. Instructions are dispatched into the tail and issued from the head. The head register is incremented when an instruction is issued and the tail register is updated when a new instruction is dispatched to the queue. Request and grant logic needs to be included for every entry because all entries can become FIFO heads. However, no shifting is needed once a head is issued. Unlike a shift register, which is essentially a physical FIFO, a circular queue acts as a logical FIFO since no instructions are actually being shifted.

Either implementation may be used to realize Scheme 1. However, only the circular queues can be used to implement Scheme 2. For Scheme 2, in the worst case (i.e., full-power mode) all IQ entries are essentially FIFO heads, so hardware for request and grant logic must exist in all entries. Since all entries include the same logic, there is no advantage to implement this scheme with a set of shift registers. The circular queue implementation for Scheme 2 is somewhat more complicated than that used for Scheme 1, since the FIFO can vary in size. This can be handled by adding a global register to store the size of FIFOs in the current configuration.

To save power in interconnect wiring in Scheme 1, we can use a similar wire buffer method as was proposed in Albonesi [1998]. The idea is to insert buffers along the interconnect wire such that they serve two purposes: improving signal delay/integrity and isolating sections of the interconnect. When unused FIFOs are disabled, interconnect wiring to these entries may also be effectively disabled. To realize this in hardware, we need to organize instructions in the same FIFO to be physically adjacent for either a shift register or a circular queue implementation.

Figure 5(a) shows how the buffers are used with the shift register implementation. Here we have four FIFOs, each with four entries. Notice that we have chosen to place a wire buffer between each FIFO boundary. The optimal number of buffers will depend on the particular design constraints. The buffers provide us with fine-grain configuration capability without introducing a delay penalty. For instance, in the right part of Figure 5(a), when FIFO 4 (entries 13–16) is disabled, the bottom buffer is disabled, effectively isolating the wires from entries 13 to 16. This reduces the wire capacitance on wakeup and arbitration wires routed through the remaining active entries. Similarly, Figure 5(b) shows a circular queue implementation with wire buffers. Again, the IQ consists of four FIFOs with four floating heads (i.e., entries 3, 6, 9, and 16 currently), each with four entries. The corresponding wire buffer is disabled, reducing the effective wire length of the wakeup and arbitration signals. Note that in Scheme 2, we are not disabling entries or FIFOs. Therefore, the wire buffers cannot be used to isolate parts of the interconnect wiring. However, note that although interconnect for the *globally* routed request and grant signals cannot be altered, the *local* request and grant logic found in each nonhead entry can still be disabled by preventing these signals from precharging. This will be considered further in Section 4.

Given that only a fraction of the entries will be visible to the arbitration logic, it is important that instructions be placed in the FIFOs such that most

Fig. 5. The IQ with wire buffers (Scheme 1). (a) A shift register implementation and (b) a circular queue implementation.

Fig. 6. Different instruction assignment strategies.

(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 as shown in Figure 6. Here the IQ is composed of 32 fixed-sized FIFOs, each with four entries. No dynamic scheme is applied. We used the same processor configuration as described in Section 5.

The random scheme attempts to insert a newly decoded instruction into a randomly chosen FIFO that is not full. Similarly, the round-robin scheme chooses the next nonfull FIFO. For both schemes, whenever a full FIFO is chosen, the same algorithm is applied until a suitable FIFO is found. Therefore, the system only stalls when the entire IQ is full. The dependency-based scheme is similar to the one presented in Palacharla et al. [1997]. As an instruction is decoded and dispatched to the IQ, we attempt to place it in the same FIFO as one or both of its source dependencies. Based on the number of available operands, the following three cases could happen:

- (1) If the instruction is already ready (i.e., both operands are available), it is steered to a new, empty FIFO.
- (2) If the instruction has only one pending operand,
 - (a) if its producer is the tail of the corresponding FIFO and the FIFO is not full, it is steered to the same FIFO as the producer;
 - (b) otherwise, it is steered to a new, empty FIFO.
- (3) If two operands of the instruction have not been computed, we attempt to steer the instruction to the same FIFO as one of its producers as follows:
 - (a) we first try to assign the instruction to the FIFO corresponding to its left operand based on the same strategy used in the above case (2);
 - (b) if option (a) fails because either the FIFO is full or the left producer is not the tail, we then apply the same method to the right operand;
 - (c) if we fail again, we allocate an empty FIFO for this instruction.

In order to realize a more accurate arbitration strategy, we also implemented a history-based last operand predictor (LOP) [Stark et al. 2000] to predict which of the two operands will most likely be available later. Instructions are preferentially placed into the same FIFO as its producer that is predicted to arrive later. While our goal was to implement an instruction steering algorithm that provided the best performance, our initial experiments showed only a slight performance advantage by using LOP, so we decided not to implement it in our final experiments due to power and complexity considerations.

In all cases for the dependency-based scheme, 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 that comes after it in dispatch order is ready. Our experiments indicate that this case happens around 5% of the total execution time on average. However, it does not necessarily translate to the same amount of performance degradation because the stalled ready instructions might not be on critical paths or they may be on misprediction paths. Our data shown in Section 6 will prove this point.

While the dependency-based scheme provides the best performance of the three strategies from our experiments shown in Figure 6, it is the most expensive to implement in hardware even without the LOP. However, we can use some existing dependency information, which is also needed by some other techniques, like load hit speculation [Moreshet and Bahar 2003], to implement our

dependency-based scheme. Therefore, we believe that the dependency-based scheme does not introduce significant overhead in terms of extra power. Alternatively, the random strategy is simple to implement, but is completely unacceptable in terms of performance except for *vortex*, *ammp*, and *mesa*. Similarly, although the round-robin scheme is a reasonable alternative to the dependency-based scheme for some benchmarks (e.g., *twolf*, *vortex*, *ammp*, *mesa*), we think that it is still worthwhile to implement the dependency-based scheme overall.⁴ 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 that short-term past behavior is a good indicator of behavior in the near future. This assumption is similar to the one used for the PAST scheduler in Weiser et al. [1994]. At the end of each sample period, we determine whether to enter a lower-power mode, leave it in the current 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 modes. Our goal was to limit performance degradation to be no more than 5% of the base case because performance must remain a priority for the high-end, out-of-order processors that we are targeting. We felt that any further reduction would not be an acceptable tradeoff. Following are the hardware monitoring mechanisms we implemented to modify the number of FIFOs. Most of them can be used to both enable and disable low-power modes. Several have already been used in previous works [Bahar and Manne 2001; Buyuktosunoglu et al. 2000; Folegnani and González 2001; Maro et al. 2000; Ponomarev et al. 2001; Sasanka et al. 2002], though none use the same combination of monitors.

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 IQ may need to be visible to the arbitration logic. Similarly, the issue IPC can also be used to decide when to get out of the low-power mode. This monitor requires a resettable counter that is cleared at the beginning of each window and incremented appropriately each cycle. The counter value is then compared with a threshold value at the end of the window. If the measured IPC during the current sample window is below a certain threshold, the machine cuts the number of FIFOs for the next sample window. We can pick some absolute value

⁴Due to its high instruction cache miss rate, *vortex*'s execution time is dominated by the memory access latency. Therefore, it does not make much difference to which FIFOs instructions are steered. The *ammp* behaves similarly due to its high data cache miss rate.

as the IPC threshold. We also want to adjust the IPC threshold to the specific behavior of the program. To do this, we also tried keeping track of the program's overall IPC for the execution of the program so far and picking an IPC threshold that is within some delta value of the overall IPC.⁵ The threshold to enable the low-power mode can also be lowered for successive reductions in the number of FIFOs such that it is increasingly harder to cut the number of FIFOs further. This helps prevent the performance from plummeting. Without this strategy, the machine may continue to restrict the IQ further without recognizing that the IPC is low due to the newly reconfigured IQ and not due to the inherent ILP limitations of the application being run on the processor.

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. This reasoning is similar to the one used in Manne et al. [1998].

Performance degradation. Once we are in the low-power mode, we need to react to local changes in the performance that indicate our overall performance may suffer if we do not return to a higher-power mode. For instance, if the drop in IPC from one sample period to the next one 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 in the dispatch stage), this can prevent a ready instruction from issuing as soon as possible and thus 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.

Dependency counting. True dependencies limit ILP; therefore, configuring the IQ into many FIFOs may not improve performance if we are already ILP limited. If we detect a high number of dependencies, we reduce the number of FIFOs assuming that this will not impact performance.

IQ occupancy. Low IQ occupancy rates may indicate an opportunity to reduce the number of FIFOs since the queue is being underutilized.

Noncritical instructions. A newly decoded instruction that is immediately ready for issue is steered to an empty FIFO. However, if no instruction is placed behind it by the time when it is removed from the IQ (i.e., no instructions depend on), the instruction is noncritical. Delaying such ready instructions will not hurt overall performance; if many noncritical instructions are identified, this may indicate a chance to reduce the number of FIFOs.

4. POWER ESTIMATIONS

The FIFO-based IQ design saves power by turning off underutilized IQ components in the instruction wakeup and arbitration loop. Other components of

⁵Since a real program may run for hours or even days, it is difficult to count the total number of cycles and instructions. To solve this problem, we propose keeping track of the overall IPC during a reasonably long period as the estimation of the overall IPC and update it from time to time.

the processor may show a reduction in power dissipation as well, but in general this will only occur if the total number of instructions issued is less compared to our baseline architecture. This is possible due to a reduction in the number of wrong path instructions observed when more restrictive rules are used to issue ready instructions. To make our power analysis straightforward, we only consider the power-saving estimations in the IQ. We assume that any significant savings in the IQ will transfer to the reasonable savings in the processor since the IQ is a significant contributor to the total power dissipation on the chip [Biro 2003; Wilcox and Manne 1999; Palacharla et al. 1997].

To estimate the power savings in the IQ when operating in a low-power mode, we extrapolated from Alpha 21264 and 21464 power estimations [Biro 2003; Wilcox and Manne 1999]. As with our design, the 21464 was designed to use a unified, noncollapsible, out-of-order IQ, capable of 8-wide issue. In addition, designers of this processor were available to us for consultation. We are not using architectural-level power estimation tools, such as Wattch [Brooks et al. 2000], since we have found that they tend to assume rather simple arbitration logic which therefore leads to an underestimation of the IQ power [Brooks 2003].⁶

We will focus our power analysis on the wakeup and arbitration loop, which consists of the generation of data ready, instruction request, and instruction grant signals. Below, we first discuss how the IQ works, specifically in the wakeup and arbitration loop, and why the loop is so important in terms of power dissipation. Then we show how our schemes simplify the logic and how power is reduced.

The complexity of designing and implementing any out-of-order IQ is a function of N , the issue width of the processor and M , the number of entries in the IQ [Bahar and Manne 2001; Palacharla et al. 1997]. With increasing values of M and N across successive generations of processors, the IQ has become a significant contributor to overall complexity and processor power dissipation. The IQ is responsible for three main tasks: tracking data dependencies among all the instructions in the IQ, determining when instructions are ready for issue (i.e., the wakeup logic), and arbitrating among all the ready instructions and granting a subset of these instructions permission to issue (i.e., the arbitration logic). Figure 7 shows the simplified design for one of the IQ entries and the general design for the whole IQ. Each IQ entry holds information for a single instruction. This includes a valid bit, two source tags (each with its own ready flag), and one destination tag.

The wakeup logic, shown in Figure 7(a), includes a set of comparators and result tag buses running the length of the IQ. When an instruction is about to complete execution, its destination tag is broadcast to all entries in the IQ. Each instruction then compares this tag against its own two source tags. If the destination tag matches either source tag, the corresponding ready flag is set. In each cycle, at most N instructions can drive their destination tags, so $2N$

⁶Our experiments with Wattch indicate that less than 2% of the IQ power is dissipated in the arbitration logic, which is quite different from estimates that we obtained on power dissipation in the 21464 (which also used a single unified buffer).

(a) A single issue queue entry.

(b) Issue queue entries.

Fig. 7. The wakeup and arbitration loop in the IQ. (a) A single IQ entry and (b) IQ entries.

comparators are required for each of the M IQ entries for comparing against its source tags. This wakeup logic needs to be very fast so it tends to be quite complex and takes up a lot of area. Depending on different power or performance constraints, the logic may be implemented differently in hardware. For instance, we may choose to precharge the ready line every cycle and discharge it when the corresponding source is not ready (i.e., keep it precharged only when it becomes ready). This circuit design is fast since it can be accomplished using precharged logic, for example, domino logic. Nevertheless, the ready line is kept switching between precharged and discharged states before the operand becomes ready, making this implementation power inefficient. Instead, if the implementation is such that the line only gets discharged when the operand is ready (i.e., remains precharged if it is not ready), power dissipation is reduced greatly due to the switching reduction, but with a sacrifice in performance in terms of delay and circuit complexity [Biro 2003]. In summary, tradeoffs

between power and performance always need to be considered, even in high-performance processors. Therefore, the amount of power that is saved with our dynamically reconfigurable FIFO schemes will depend on the underlying implementation.

The arbitration logic, shown on the left-hand side in Figure 7(b), is composed of N arbiters and M pairs of request/grant signals along the entire IQ. When an instruction has both its ready flags set, it requests to issue. All of the M IQ instructions that are ready bid to issue. Arbiters prioritize these bidding instructions to determine which of them will be granted permission to issue according to the selection policy. The grant signals are broadcast from the arbiters across the length of the IQ in order to reach all M entries. This allows granted instructions to update their request status and drive operand information to the register file and execution units. In addition, these grant signals are latched by their corresponding instructions in the queue and used to drive the destination tags in the following cycle or later for multicycled operations. Various selection policies are proposed and employed in processor designs. Palacharla et al. [1997] used a *location-based policy* and formed a tree-structure arbitration logic. Alternatively, we assume an *oldest first policy*, in which the requested instruction that is the earliest in program order gets the permission to issue.

Due to its complicated hardware implementation and control logic, the wakeup and arbitration loop is on the critical path of the IQ, which is responsible for significant portion of the IQ power dissipation [Biro 2003]. As mentioned earlier, depending on different needs and implementation strategies, the power distribution within the loop may vary. For example, if a power-efficient wakeup logic is implemented (e.g., a ready signal switches only when an instruction “wakes up”), half of the loop power may be attributed to the wakeup logic and the other half to the arbitration logic [Biro 2003]. We call this distribution “DISTR1.” If a faster, simpler, but more power-hungry wakeup design is employed, the power distribution could be 70% in the wakeup logic and the remaining 30% in the arbitration logic. This is referred to as “DISTR2.” We will show power savings in the loop based on both distributions in Section 6.

In Scheme 1, we deactivate FIFOs if they are found to be underutilized. We do not need to update any information associated with disabled entries. Thus, we can save power in the wakeup logic according to the fraction of FIFOs disabled. Power dissipation in the arbitration logic is also reduced due to the reduced activity on the corresponding request and grant lines. In addition, interconnect wire capacitance can be reduced in the remaining active request and grant lines by isolating off the portion of the wires routed through the disabled entries (see Figure 5). To conclude, for Scheme 1, power in the wakeup and arbitration loop is saved according to the fraction of FIFOs disabled regardless of how the power is distributed within the loop. However, note that our first scheme starts with a multiple-entry FIFO configuration rather than a monolithic IQ. Suppose that we start with L , T -entry FIFOs, where $L * T = M$. So in the worst case (i.e., the initial state), the arbitration logic only needs to pick among L instructions instead of M instructions because there are only L FIFOs and thus L heads. Therefore, if compared to a traditional IQ, that is, M , 1-entry FIFOs, the initial

state in our first scheme has already saved $50\% * 0\% + 50\% * (1 - \frac{L}{M}) = \frac{M-L}{2M}$ of the total power in the wakeup and arbitration loop for DISTR1. Similarly, we save $70\% * 0\% + 30\% * (1 - \frac{L}{M}) = \frac{3(M-L)}{10M}$ for DISTR2. Notice that the wakeup power is not reduced in the initial state for both distributions as the wakeup power depends only on the total number of enabled entries in the IQ. This will be discussed in more detail in Section 6.

In Scheme 2, we reconfigure the IQ by dynamically modifying the number and size of FIFOs while keeping each entry in the IQ active at all times. In this case, we always start with the base case, that is, M , 1-entry FIFOs. Even if an instruction in the IQ is not made visible to the arbitration logic since it is not a head, the IQ entry 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 wakeup logic remains the same regardless of the FIFO configuration. However, power dissipation in the arbitration logic is saved due to the reduced activity on the request and grant lines by selectively inhibiting them from precharging. We only need to precharge the request and grant lines for instructions that are at the head of a FIFO. So the power dissipation in the arbitration logic is directly proportional to the number of active FIFOs. If the total number of FIFO heads in the IQ is cut in half, this should reduce the switching and therefore the power on the request and grant lines and associated logic by approximately half. In summary, if Scheme 2 cuts the total number of FIFOs in half, power dissipation in the wakeup and arbitration loop is saved by $50\% * 0\% + 50\% * 50\% = 25\%$ if DISTR1 is assumed or $70\% * 0\% + 30\% * 50\% = 15\%$ if DISTR2 is assumed.

In addition, for both schemes, if there are ever less than N FIFOs active, this effectively reduces the issue width of the machine, so we can save more power by disabling all the request and grant lines associated with the unused issue slots.

5. EXPERIMENTAL METHODOLOGY

The simulator that we used in this study is derived from the SIMPLESCALAR tool suite [Burger and Austin 1999], which executes PISA (portable ISA) binaries. 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 reorder buffer. In our work, the RUU is split into the reorder buffer (ROB) and the 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 IQ 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 IQ's FIFOs according to their input dependencies.

Table I shows the complete configuration of the processor model. Note that the ALU resources listed in the table may incur different latency and occupancy values depending on the type of operation that is being performed by the unit.

Table I. Processor Resources

Parameter	Configuration
Inst. window	256-entry LSQ, 512-entry ROB 128-entry IQ
Machine width	8-wide fetch, issue, commit
Fetch queue	16
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	8 KB 2-way; 32 B line; 1 cycle
L1 Dcache	8 KB 2-way; 32 B line; 1 cycle
L2 Cache	256 KB 8-way; 64 B line; 12 cycle
Memory	128 bit-wide; 150 cycles on hit
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.

The pipeline allows for up to eight instructions to be issued, executed, and committed each cycle. In addition, the base case assumes a unified 128-entry out-of-order IQ. All comparisons in Section 6 are made to this case.

Our simulations are executed on a subset of the SPEC2000 integer and floating-point benchmarks [Henning 2000].⁷ They were compiled using a retargeted version of the GNU *gcc* compiler with full optimization. All benchmarks are fast-forwarded to avoid startup effects using SimPoint [Sherwood et al. 2002]. The benchmarks are then executed for 100 million committed instructions.

6. EXPERIMENTAL RESULTS

As mentioned in Section 5, we chose a 128-entry IQ for our baseline experiments. Figure 8 shows the relative performance with various sizes of the IQ and the average for all benchmarks simulated.⁸ We found that performance is insensitive to the IQ size for some benchmarks, such as *vortex*, *ammp*, *equake*, and *mesa*, when the IQ size exceeds 64. Nevertheless, we notice that as the IQ is made smaller than 64 entries, the IPC starts to degrade significantly for most benchmarks. On average, the performance penalty is 12% for a 32-entry IQ compared to our baseline. Although, for some benchmarks, the performance is almost the same for the 32-entry, 64-entry, 128-entry, and 256-entry IQ, we also notice that many floating-point benchmarks, such as *art* and *wupwise*, benefit from the larger queue. For these reasons, we chose to use a 128-entry IQ as the base case for our experiments.

For our first scheme, we start with 32 FIFOs, each containing four entries, and then adjust the number of FIFOs according to feedback from hardware

⁷They are all the SPEC2000 PISA binaries available to us compiled for our system. Others need additional modifications to compile successfully.

⁸In all figures and tables in this paper, “average” means an arithmetic mean.

Fig. 8. Performance as a function of the IQ size (normalized to the 128-entry IQ, i.e., the fourth column).

Table II. Starting States for Four Schemes

Schemes	Number of FIFOs	Size of FIFOs
1	32	4
2	128	1
Hybrid	128	1
No FIFO	128	1

performance monitors. For our second scheme, we start with 128 FIFOs, each containing a single entry (i.e., a traditional IQ), and then modify both the number and size of FIFOs dynamically. Next, we combine two IQ reconfiguring schemes to show the advantages of using both together. Finally, we compare our schemes with a dynamic IQ without any FIFO implementation included. To clarify the starting state for each scheme, we show the number and size of FIFOs for each scheme in Table II.

6.1 Results for Scheme 1

We experimented with many combinations of different performance monitors to determine when to enable and disable FIFOs. Table III shows the best combination of monitors we found. We list the specific monitors in order of relative importance. We give priority to enable FIFOs since we try to be conservative without sacrificing much performance. Using more monitors helps preserve performance; however, from our experience, it is reasonable to use only 1 or 2 different monitors and still obtain significant power savings though performance may drop by an additional 1–2%.

Using the power estimations and distributions in the wakeup and arbitration loop that we made in Section 4, we can estimate the total power savings in the

Table III. Hardware Performance Monitors for Scheme 1

Disable one FIFO when at least one of the following conditions holds:
(1) Less than $\frac{1}{4}$ of newly decoded ready instructions are prevented from dispatching to the IQ due to the lack of an available FIFO
(2) Less than $\frac{2}{3}$ of the FIFOs are actually used on average
(3) More than 15% of dispatched instructions are noncritical
(4) current IQ occupancy rate is less than $\frac{1}{4}$ of the average occupancy rate
Enable one FIFO when at least one of the following conditions holds:
(1) Current issue rate (IPC_{issue}) drops by more than 20% compared to the last window executed in the initiate-power mode (i.e., 32, 4-entry FIFOs)
(2) Current IPC_{issue} drops by more than 20% compared to the previous window
(3) More than $\frac{1}{3}$ of newly decoded ready instructions are prevented from dispatching to the IQ due to the lack of an available FIFO

Table IV. Results for Scheme 1

Benchmarks	Average Number of FIFOs	128-entry IQ				
		32, 4-entry FIFOs		128, 1-entry FIFOs		
		Loop Power Savings	Δ IPC	Loop Power Savings		Δ IPC
				DISTR1	DISTR2	
bzip2	29.4	8.0%	0.8%	19.5%	14.9%	7.8%
gzip	19.9	37.9%	4.1%	45.7%	42.6%	6.3%
mcf	29.9	6.7%	1.6%	18.4%	13.7%	7.9%
parser	24.4	23.7%	4.3%	33.3%	29.4%	11.3%
twolf	29.4	8.1%	0.5%	19.6%	15.0%	6.4%
vortex	18.4	42.5%	4.3%	49.7%	46.9%	5.7%
vpr	26.1	18.4%	2.1%	28.6%	24.5%	7.8%
ammp	29.6	7.5%	0.2%	19.0%	14.4%	0.2%
art	30.0	6.2%	6.8%	17.9%	13.2%	44.9%
equake	28.0	12.6%	-0.6%	23.5%	19.2%	1.9%
mesa	28.7	10.2%	0.2%	21.4%	16.9%	1.2%
wupwise	30.0	6.2%	3.7%	18.0%	13.3%	26.8%
Average	27.0	15.7%	2.3%	26.2%	22.0%	10.7%

loop. Table IV shows the average number of FIFOs used (in column 2), the total power savings in the wakeup and arbitration loop, and the performance degradation for all benchmarks. We show power savings and performance degradation compared to both the starting state (the full-power mode for Scheme 1), in which the IQ is composed of 32, 4-entry FIFOs (in columns 3 and 4),⁹ and the 1-entry FIFO state, which is the same as a traditional non-FIFO scheme (in columns 5–7). Note that power savings is given for two distributions when compared to a traditionally unified IQ. These two distributions are defined in Section 4. However, we only show one power savings when compared to the Scheme 1 starting state since power reduction is not dependent on the distribution as discussed in Section 4.

⁹As discussed in Section 4, the starting configuration of Scheme 1 has already produced some power reduction if compared with a conventional, monolithic IQ.

As we expected, the first scheme tends to work more reasonably for the integer benchmarks because most floating-point benchmarks usually need the entire IQ to extract ILP. Moreover, according to Figure 1, the starting state, that is, 32, 4-entry FIFOs, is completely unreasonable for some floating-point benchmarks because the starting state itself introduces a large loss in performance. For instance, 41% performance degradation is introduced by starting with a 4-entry FIFO state for *art*. So the first scheme is not suitable for these floating-point benchmarks, since we have a fairly rigid starting state. However, for comparison, we still include the results for the floating-point benchmarks in Table IV. Notice that *equake* improves performance slightly when using our first scheme compared to the starting 4-entry FIFO state. This is mainly due to the avoidance of executing some wrong path instructions if fewer FIFOs are available in the low-power mode.

For the integer benchmarks, we do a reasonable job of dynamically disabling FIFOs from this starting state, that is, we retain performance compared to the starting state of always using 32, 4-entry FIFOs, but overall performance cost may still be too high compared to the non-FIFO scheme. However, even with this limitation, *gzip* and *vortex* are good examples to show how effectively our scheme can work; they can reduce more than 38% and 42.5% of the loop power with a performance drop of only about 4% compared to the starting state, respectively.

On average, Scheme 1 can save 16% of the wakeup and arbitration loop power with a performance degradation of around 2% compared to the starting state for all benchmarks. If compared to the non-FIFO scheme, our first scheme can save 26% and 22% of the loop power for DISTR1 and DISTR2, respectively, but it comes at a performance drop of 11% on average and 45% for *art*, which is an unacceptable performance hit.

6.2 Results for Scheme 2

Similar to the first scheme, we also tried a number of combinations of policies for enabling and disabling low-power modes for Scheme 2. The combination that produced overall best results in terms of both performance and power is shown in Table V.¹⁰ Again, we order them according to relative importance and we give priority to double the number of FIFOs.

Figure 9 shows the percentage of time that the processor is in each mode. For several floating-point benchmarks, such as *art* and *wupwise*, our FIFO technique cannot reduce the total number of FIFOs significantly, since these benchmarks need more flexibility in reordering instructions to maintain performance. They are very sensitive to FIFO ordering as Figure 1 shows. 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 *vortex*, the system spends only 10% of time in the 1-entry FIFO state, but 14% of time in an 8-entry FIFO state. In addition, for the floating-point benchmark *mesa*, we are still able to obtain nice distributions. It reduces the number of FIFOs to 64, 32, 16, 8, 4, and 2,

¹⁰In the table, $\text{AVG_IPC}_{\text{issue}}$ is the overall issue IPC over a long period.

Table V. Hardware Performance Monitors for Scheme 2

<p>Cut the number of FIFOs in half (and double the number of entries in each FIFO) when at least one of the following conditions holds:</p> <ol style="list-style-type: none"> (1) $(IPC_{issue} - IPC_{commit}) > 1.0$ (2) less than 3% of newly decoded ready instructions are prevented from dispatching to the IQ due to the lack of an available FIFO (3) $IPC_{issue} < 2.7$ (threshold lowered by 0.2 for each successive reduction in the 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 the number of FIFOs.)
<p>Double the number of FIFOs (and halve the number of entries in each FIFO) when at least one of the following conditions holds:</p> <ol style="list-style-type: none"> (1) current IPC_{issue} drops by more than 7% compared to the previous window (2) current IPC_{issue} drops by more than 15% compared to the last window executed in the full-power mode (i.e., 128, 1-entry FIFOs) (3) more than 15% of newly decoded ready instructions are prevented from dispatching to the IQ due to the lack of an available FIFO

Fig. 9. FIFO usage for Scheme 2. 128, 64, 32, 16, 8, 4, 2, and 1 FIFOs mean each FIFO has 1, 2, 4, 8, 16, 32, 64, and 128 entries, respectively; thereby always retaining the same number of IQ entries.

for about 15%, 17%, 18%, 18%, 16%, and 8% of the execution time, respectively. Overall, we can spend an average of 60% of the time in some reduced power mode.

Using results from Figure 9, and power estimations and distributions made in Section 4, we can estimate the total power savings in the wakeup and arbitration loop for both distributions. Table VI shows the total power reduction in the loop for each benchmark compared to a monolithic 128-entry IQ, assuming both power distributions. In columns 2 and 3, we report percentage of power

Table VI. Results for Scheme 2

Benchmarks	128-entry IQ		
	Loop Power Savings		Δ IPC
	DISTR1	DISTR2	
bzip2	12.4%	7.6%	1.3%
gzip	30.6%	19.6%	1.8%
mcf	7.6%	4.6%	4.4%
parser	21.2%	13.4%	2.9%
twolf	17.8%	11.1%	3.7%
vortex	43.7%	29.0%	4.1%
vpr	23.1%	14.5%	2.8%
ammp	19.9%	12.1%	0.3%
art	7.8%	4.7%	5.0%
equake	21.6%	13.4%	2.6%
mesa	52.7%	36.5%	3.4%
wupwise	8.1%	4.9%	4.3%
Average	22.2%	14.3%	3.1%

saved in the wakeup and arbitration loop for DISTR1 and DISTR2, respectively. In column 4, we report overall performance degradation.

As can be seen in the table, for a few benchmarks, we manage to save more than $\frac{1}{3}$ of the loop power for either distribution. This will translate to saving a reasonable portion of the IQ power, with the savings coming from the arbitration logic. Overall, it is easier to cut the number of FIFOs for integer benchmarks and hence save more power.

It is clear that Scheme 2 works more effectively than Scheme 1 on average if power reduction and performance penalty are both considered (see Tables IV and VI). Although several floating-point benchmarks need 128-entry FIFOs for a large percentage of the running time, we can still find several examples, like *ammp*, *equake*, and *mesa*, where our scheme can still be applied with significant power savings. According to Figure 1, *mesa* can stand from 2-entry to 16-entry FIFO states, but performance degrades by 12% if we continue to cut the number of FIFOs further. However, our technique can effectively find the appropriate periods to cut the number of FIFOs while still retaining performance. Using our second scheme, *mesa* can stay at 32-entry and 64-entry FIFO states for about 24% of the execution time with only a performance degradation of 3%. Another good example is *vortex*. From Figure 1, we can see that *vortex* performs well with 16, 8-entry fixed-sized FIFOs; but performance suffers if fewer FIFOs are used. In comparison, using our dynamic technique, *vortex* can even enter the low-power mode with 32 entries in each FIFO. After studying *mesa* and *vortex* more carefully, we found that their instruction cache miss rates are higher than other benchmarks. For an 8 KB 2-way instruction cache, *mesa*'s miss rate is 10.5% and *vortex*'s miss rate is 10.9%, while *vpr* has the next highest miss rate, which is only 4.7%. Improving miss rates will change the FIFO size distribution. For instance, *mesa* with a 16 KB 2-way instruction cache has only a 6.6% miss rate and only spends 9% of the time in 32-entry and 64-entry FIFO states, while with an 8 KB 2-way cache, it can stay in 32-entry and 64-entry FIFO states for around 24% of the execution time as shown above. We found the same trend

using different instruction cache sizes for *vortex*. With the higher instruction cache miss rate, we have more opportunities to find appropriate periods to cut the number of FIFOs deeply due to the lack of ready instructions in the queue. While the high instruction cache miss rate plays a significant role in being able to exploit the FIFO scheme, we would like to emphasize that many other factors, like more long latency operations and very little parallelism with long chains of dependencies instructions, make *mesa* and *vortex* excellent benchmarks to show the efficacy of our second scheme.

We also did experiments to see how sensitive Scheme 2 is to the memory latency by decreasing the memory latency from 150 to 100 cycles. We found that for the similar performance impact, the system spends shorter time in the power-saving states with the shorter memory latency. With the longer latency, the system has to stall a longer time to wait for the access to the main memory, during which we can reconfigure the IQ to save more power without compromising performance. In comparison, we have less chance to reconfigure the IQ with the shorter latency. Similarly, we found this tendency in Scheme 1, as well.

On average, the best results produced by Scheme 2 can save 22% (for DISTR1) or 14% (for DISTR2) of the total power in the wakeup and arbitration loop, with a performance degradation of only 3%.

6.3 Results for Hybrid Scheme

While Scheme 1 is able to save more power but with a larger performance drop, Scheme 2 has more flexibility in exposing potential ILP and can retain performance but with less power savings. Combining two schemes comes into our mind naturally. If we can take advantage of both schemes and compensate for their weaknesses, we expect to see even better results.

In this combined scheme, we always start in the same state as Scheme 2, that is, the conventional IQ with 128, 1-entry FIFOs. Once we find that the IQ is underutilized, we reconfigure the IQ in the way that Scheme 2 does, that is, modifying both the number and size of FIFOs while keeping the entire IQ active. When the FIFO size reaches some predefined threshold, named MAX_FIFO_SIZE, Scheme 1 is applied to reconfigure the IQ, that is, completely disabling FIFOs when needed. We experimented with different values for the threshold MAX_FIFO_SIZE. Tables VII and VIII show the power savings in the loop for each distribution and the performance effects when MAX_FIFO_SIZE is equal to 4 and 2, respectively. We used the same combinations of policies for enabling and disabling low-power modes as Scheme 2 used. As we expected, a small MAX_FIFO_SIZE will provide with more opportunities to reduce power dissipation, but sacrifice slightly more performance. Overall, we save 3–5% more power with almost the same performance drop if using the smaller threshold. Compared to Scheme 1, we lose much less performance especially for the floating-point benchmarks (all less than 5% now). On the other hand, if compared to Scheme 2, we lose almost the same performance, but win more power reduction. For some benchmarks, such as *art* and *wupwise*, there is not much benefit in applying hybrid scheme since those benchmarks need the entire IQ active to exploit potential ILP for significant portions of

Table VII. Results for Hybrid Scheme
(MAX_FIFO_SIZE = 4)

Benchmarks	128-entry IQ		
	Loop Power Savings		Δ IPC
	DISTR1	DISTR2	
bzip2	12.4%	7.6%	1.3%
gzip	33.4%	24.7%	2.0%
mcf	7.6%	4.6%	4.4%
parser	22.4%	15.5%	2.9%
twolf	18.4%	12.1%	3.7%
vortex	51.0%	42.4%	4.4%
vpr	23.8%	15.8%	2.8%
ammp	19.7%	12.2%	0.3%
art	7.8%	4.7%	5.0%
equake	21.6%	13.7%	2.3%
mesa	62.4%	55.6%	3.4%
wupwise	8.2%	4.9%	4.2%
Average	24.1%	17.8%	3.1%

Table VIII. Results for Hybrid Scheme
(MAX_FIFO_SIZE = 2)

Benchmarks	128-entry IQ		
	Loop Power Savings		Δ IPC
	DISTR1	DISTR2	
bzip2	14.0%	10.0%	1.3%
gzip	37.2%	32.0%	1.8%
mcf	7.9%	5.0%	4.4%
parser	29.2%	24.7%	3.1%
twolf	21.7%	17.1%	3.7%
vortex	57.7%	53.5%	4.4%
vpr	28.5%	23.2%	2.8%
ammp	21.3%	14.0%	0.3%
art	8.1%	5.2%	5.0%
equake	25.5%	19.9%	2.6%
mesa	68.5%	65.4%	3.8%
wupwise	8.2%	4.9%	4.3%
Average	27.3%	22.9%	3.1%

the running time. However, for some benchmarks, like *mesa* and *vortex*, hybrid scheme saves up to 30% more power dissipation with a negligible performance difference.

On average, the best results produced by our combined scheme can save 27% (for DISTR1) or 23% (for DISTR2) of the loop power dissipation, with a performance degradation of only 3%.

6.4 Comparison with Previous Schemes

In Section 2, we mentioned that previous works proposed a dynamic IQ whose size can be modified according to the program's needs [Buyuktosunoglu et al. 2000; Folegnani and González 2001; Ponomarev et al. 2001]. We implemented

Table IX. Results for a Dynamic IQ (no FIFO Implementation)

Benchmarks	128-entry IQ	
	Loop Power Savings	Δ IPC
bzip2	5.5%	0.3%
gzip	7.5%	0.2%
mcf	35.2%	1.8%
parser	36.3%	5.0%
twolf	5.6%	0.8%
vortex	2.5%	0.1%
vpr	13.6%	0.9%
ammp	0.4%	0.0%
art	10.2%	2.1%
equake	2.5%	0.1%
mesa	4.0%	0.7%
wupwise	1.5%	1.4%
Average	10.4%	1.1%

a similar IQ in order to better compare with our schemes and then show our schemes' efficacy in exposing inherent ILP and reducing power dissipation. The results are shown in Table IX. The IQ is also initialized to have 128 entries. In terms of the program's requirements, we can adjust the IQ size by disabling and enabling IQ entries back and forth. No FIFO implementation is added in this set of experiments. Hardware performance monitors to find an optimal IQ configuration are the same as those used in our schemes. For brevity, we do not show them again. In Table IX, the second column shows percentage of power saved in the wakeup and arbitration loop. As in Scheme 1, power reduction here is not a function of the power distribution in the wakeup and arbitration loop. Column 3 reports overall performance degradation. The power-saving computation is based on the same assumptions as used in Scheme 1. We notice that power savings shown in Table IX may not be as significant as those reported in previous works since our underlying power estimations are different. We can see that although this technique works efficiently for some benchmarks, like *mcf*,¹¹ *parser*, and *art*, the total power saved in the wakeup and arbitration loop is small for most integer and floating-point benchmarks. We tried to continue to reconfigure the IQ more aggressively to save more power since the average performance degradation is only 1%, but we found that the performance drop of some benchmarks, such as *art* and *parser*, will soon reach 10%, which is unacceptable. If we compare Table IX with Table VIII, we find that our combined scheme can save much more power (27% versus 10%) with a larger average performance drop (3% versus 1%), but the maximal performance penalty is the same for two approaches, which is 5%. This indicates that previous dynamic IQ design might be good enough to extract potential ILP and reduce power for some applications, but overall our scheme has more flexibility in adjusting the IQ to

¹¹*mcf* can withstand disabling part of the IQ; according to Table IX, it only loses 1.8% performance while more than 35% of the power dissipation is saved in the loop. However, it cannot withstand the actual instruction ordering required of the FIFO scheme as shown in Figure 9 and Tables VI, VII, and VIII. So our proposed approaches are not suitable for this benchmark.

Table X. Overall Results Produced by Our Schemes and the Previous Scheme

Schemes	Loop Power Savings		Δ IPC
	DISTR1	DISTR2	
1	26.2%	22.0%	10.7%
2	22.2%	14.3%	3.1%
Hybrid	27.3%	22.9%	3.1%
No FIFO	10.4%		1.1%

better satisfy diverse and changing needs for all applications that we experimented with in this work. Our scheme realizes this flexibility by partitioning the IQ into FIFOs, limiting the number of entries visible to the arbitration logic, and also adjusting this number during run-time.

7. CONCLUSION

In this paper, we exploit the fact that programs vary in their ILP. Our approach is to selectively reconfigure parts of the IQ during low ILP periods to save power without hampering performance. We designed the IQ into variable numbers of FIFOs to facilitate this reconfiguration. By decreasing the number of entries in the IQ or restricting the number of instructions visible to the arbitration logic, we save significant power in the IQ 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 parts of the IQ may only further hamper performance. However, by restricting which instructions are visible to the arbitration logic while retaining the same sized IQ will still allow us to save power. Table X summarizes all experimental results by showing the average power savings in the wakeup and arbitration loop within the IQ and the average performance degradation for our newly proposed schemes and a dynamic IQ design without the FIFO implementation. The hybrid scheme, which presents the best compromise between power savings and performance effects, shows an average power reduction of 27% (for DISTR1) or 23% (for DISTR2) in the wakeup and arbitration loop within the IQ with a performance degradation of only 3% on average.

ACKNOWLEDGMENTS

We wish to thank Larry Biro and Peter Soderquist from Intel for their helpful discussions regarding the Alpha IQ designs. We would also like to thank the anonymous reviewers for their valuable comments.

REFERENCES

- AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Vancouver, British Columbia, Canada.
- ALBONESI, D. 1998. Dynamic IPC/clock rate optimization. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Barcelona, Spain.

- BAHAR, R. I. AND MANNE, S. 2001. Power and energy reduction via pipeline balancing. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Göteborg, Sweden.
- BAI, Y. AND BAHAR, R. I. 2003. A dynamically reconfigurable mixed in-order/out-of-order issue queue for power-aware microprocessors. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*. IEEE, Tampa, FL.
- BIRO, L. 2003. Private communication.
- BROOKS, D. 2003. Private communication.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Vancouver, British Columbia, Canada.
- BURGER, D. AND AUSTIN, T. 1999. The simplescalar tool set, Version 3.0. Tech. rep., University of Wisconsin, Madison.
- BUYUKTOSUNOGLU, A., SCHUSTER, S., BROOKS, D., BOSE, P., COOK, P., AND ALBONESI, D. 2000. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*. Springer, Cambridge, MA.
- CANAL, R. AND GONZÁLEZ, A. 2001. Reducing the complexity of the issue logic. In *Proceedings of International Conference on Supercomputing*. ACM/SIGARCH, Sorrento, Italy.
- FOLEGNANI, D. AND GONZÁLEZ, A. 2001. Energy-effective issue logic. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Göteborg, Sweden.
- GHIASI, S., CASMIRA, J., AND GRUNWALD, D. 2000. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*. ACM/IEEE, Vancouver, British Columbia, Canada.
- HENNING, J. L. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Comput.* 33, 7 (July), 28–35.
- LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Anchorage, AK.
- MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: Speculation control for energy reduction. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Barcelona, Spain.
- MARO, R., BAI, Y., AND BAHAR, R. I. 2000. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Workshop on Power-Aware Computer Systems*. Springer, Cambridge, MA.
- MICHAUD, P. AND SEZNEC, A. 2001. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of International Symposium on High-Performance Computer Architecture*. IEEE, Nuevo Leone, Mexico.
- MORESHET, T. AND BAHAR, R. I. 2003. Power-aware issue queue design for speculative instructions. In *Proceedings of Design Automation Conference*. IEEE/SIGDA/EDA, Anaheim, CA.
- PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. 1997. Complexity-effective superscalar processors. In *Proceedings of International Symposium on Computer Architecture*. ACM/IEEE, Denver, CO.
- PONOMAREV, D., KUCUK, G., AND GHOSE, K. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of International Symposium on Microarchitecture*. ACM/IEEE, Austin, TX.
- SASANKA, R., HUGHES, C. J., AND ADVE, S. V. 2002. Joint local and global hardware adaptations for energy. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose, CA.
- SENG, J. S., TUNE, E. S., AND TULLSEN, D. M. 2001. Reducing power with dynamic critical path information. In *Proceedings of International Symposium on Microarchitecture*. ACM/IEEE, Austin, TX.
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose, CA.
- STARK, J., BROWN, M. D., AND PATT, Y. N. 2000. On pipelining dynamic instruction scheduling logic. In *Proceedings of International Symposium on Microarchitecture*. ACM/IEEE, Monterey, CA.

- WALL, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Boston, MA.
- WEISER, M., WELCH, B., DEMERS, A. J., AND SHENKER, S. 1994. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*. ACM, Monterey, CA.
- WILCOX, K. AND MANNE, S. 1999. Alpha processors: A history of power issues and a look to the future. In *Proceedings of the Cool-Chips Tutorial*. ACM/IEEE, Haifa, Israel. Held in conjunction with the *International Symposium on Microarchitecture*.

Received May 2003; revised December 2003; accepted April 2004