# Dynamic Resizing of Superscalar Datapath Components for Energy Efficiency

Dmitry Ponomarev, *Member*, *IEEE*, Gurhan Kucuk, *Member*, *IEEE*, and Kanad Ghose, *Member*, *IEEE*

**Abstract**—The "one-size-fits-all" philosophy used for permanently allocating datapath resources in today's superscalar CPUs to maximize performance across a wide range of applications results in the overcommitment of resources in general. To reduce power dissipation in the datapath, the resource allocations can be dynamically adjusted based on the demands of applications. We propose a mechanism to dynamically, simultaneously, and independently adjust the sizes of the issue queue (IQ), the reorder buffer (ROB), and the load/store queue (LSQ) based on the periodic sampling of their occupancies to achieve significant power savings with minimal impact on performance. Resource upsizing is done more aggressively (compared to downsizing), using the relative rate of blocked dispatches to limit the performance penalty. Our results are validated by the execution of the SPEC 2000 benchmark suite on a substantially modified version of the Simplescalar simulator, where the IQ, the ROB, the LSQ, and the register files are implemented as separate structures, as is the case with most practical implementations. We also use actual VLSI layouts of the datapath components in a 0.18 micron process to accurately measure the energy dissipations for each type of access. For a 4-way superscalar CPU, an average power savings of about 42 percent within the IQ, 74 percent within the ROB (integrating the register file), and 41 percent within the LSQ can be achieved with an average performance penalty of about 5 percent.

**Index Terms**—Superscalar processor, energy-efficient datapath, power reduction, dynamic instruction scheduling.

## 1 Introduction

Modern superscalar microprocessors are designed following a "one-size-fits-all" philosophy resulting in the permanent allocation of datapath resources to maximize performance across a wide range of applications. Earlier studies have indicated that the overall performance, as measured by the number of instructions committed per cycle (IPC), varies widely across applications [25]. The IPC also changes quite dramatically within a single application since it is a function of the program characteristics (natural instruction-level parallelism—ILP) and that of the datapath and the memory system. As the natural ILP varies in a program, the usage of datapath resources also changes significantly.

It is well-documented that the major power/energy sinks in a modern superscalar datapath are in the dynamic instruction scheduling components (consisting of the issue queue (IQ), the reorder buffer (ROB), the load/store queue (LSQ), and the physical registers). For example, as much as 55 percent of the total power dissipation occurs in the dynamic instruction scheduling logic in the Alpha 21264 microprocessor [26]. It is therefore worthwhile to consider mechanisms for reducing this power without adversely impacting performance. One approach is to resize the processor's resources dynamically by adjusting to the demands of executing applications.

In this paper, we propose exactly such a mechanism. The basic approach is a technology independent solution at the microarchitectural level that divides each of the IQ, the LSQ, and the ROB into incrementally allocable partitions. Such partitioning effectively permits the active size of each of these resources (as determined by the number of currently active partitions) to be varied dynamically to track the actual demands of the application and forms the basis of the power savings technique presented here. We also show how simple circuit-level implementation techniques can be naturally augmented into our multipartitioned resource allocation scheme to achieve substantial power savings without any compromise of the CPU cycle time. Our basic approach for reducing the power dissipation within the IQ, the LSQ, and the ROB is orthogonal to the approach taken by the more traditional techniques that use voltage and frequency scaling; such techniques can be deployed in conjunction with our scheme. The technique proposed here uses sampled estimates of the occupancies of the IQ, the LSQ, and the ROB to turn off unused (i.e., unallocated) partitions within these resources to conserve power. As the resource demands of the application go up, deactivated partitions are turned back on to avoid any undue impact on performance. The proposed approach is, thus, effectively a feedback control system that attempts to closely track the dynamic demands of an application and allocates "just the right amount of resources at the right time" to conserve power.

Recent approaches for power and energy reduction based on the broad notion of feedback control are based on the use of performance metrics like the IPC, commit rates from newly allocated regions of a resource, or sensed temperatures as well as continuous measures of the occupancy of a single resource (namely, the IQ). Rather than use IPCs and other measures of performance such as

- *D. Ponomarev and K. Ghose are with the Department of Computer Science, State University of New York, Binghamton, NY 13902-6000. E-mail: {dima, ghose}@cs.binghamton.edu.*
- *G. Kucuk is with the Department of Computer Engineering, Yeditepe University, Istanbul, Turkey 34755. E-mail: gkucuk@cse.yeditepe.edu.tr.*

*Manuscript received 20 Nov. 2003; revised 14 Feb. 2005; accepted 11 Aug. 2005; published online 21 Dec. 2005.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0222-1103.*

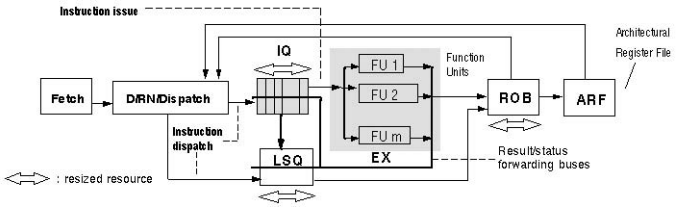


Fig. 1. Superscalar datapath.

cache hit rates, misprediction rates, or physical parameters like sensed temperature to drive dynamic resource allocations and deallocations, we use the immediate history of actual usages of the IQ, the LSQ, and the ROB to control their effective sizes independently. The actual resource usages are not monitored continuously, but sampled periodically, keeping hardware requirements simple. Resources are downsized, if need be, at the end of periodic update intervals by deallocating one or more partitions and turning them off. Additionally, resource allocations are increased before the end of an update period if the resources are fully utilized and cause instruction dispatching to be blocked for a predetermined number of cycles. This relatively aggressive strategy for increasing resource allocation allows us to severely limit the performance loss and yet achieve significant power savings.

The rest of the paper is organized as follows: We describe the superscalar datapath used for this study and trace the sources of energy dissipation in Section 2. Our simulation methodology is presented in Section 3. In Section 4, we describe the requirements for resizing the datapath components. Some observations from the use of multiple datapath resources on the realistic workloads are presented in Section 5. Section 6 describes our dynamic resource allocation strategy. Simulation results are presented in Section 7, followed by the discussion of the related research efforts in Section 8. Our concluding remarks are offered in Section 9.

## 2 Sources of Datapath Energy Dissipation

For this study, we considered a superscalar datapath, where the ROB entry set up for an instruction at the time of dispatch contains a field to hold the result produced by the instruction—this serves as the analog of a physical register (Fig. 1). A dispatched instruction attempts to read operand values either from the Architectural Register File (ARF) directly (if the operand value was committed) or associatively from the ROB (from the most recently established entry for an architectural register), in case the operand value was generated but not committed. Source registers that contain valid data are read out into the IQ entry for the instruction. If a source operand is not available at the time of dispatch in the ARF or the ROB, the address of the physical register (i.e., ROB slot) is saved in the tag field associated with the source register in the IQ entry for the instruction. When a function unit completes, it puts out the result produced, along with the address of the destination ROB slot for this result, on a forwarding bus which runs across the length of the IQ and the LSQ [21]. An associative tag matching process is then used to steer the result to matching entries within the IQ. Since multiple function units complete in a cycle, multiple forwarding buses are used; each input operand field within an IQ entry thus uses a comparator for each forwarding bus. Examples of processors using this datapath style are the Intel Pentium II and Pentium III [20].

For every instruction accessing memory, an entry is also reserved in the LSQ at the time of instruction dispatch. As the address used by a load or a store instruction is calculated, this instruction is removed from the IQ, even if the value to be stored (for store instructions) has not yet been computed at that point. In such situations, this value is forwarded to the appropriate LSQ entry as soon as it is generated by a function unit. All memory accesses are performed from the LSQ in program order with the exception that load instructions may bypass previously dispatched stores if their addresses do not match. If the address of a load instruction matches the address of one of the earlier stores in the LSQ, the required value can be read out directly from the appropriate LSQ entry.

The IQ, the ROB, and the LSQ are essentially implemented as large register files with associative addressing capabilities. Energy dissipation takes place in the issue queue in the course of:

1. establishing the IQ entries for dispatched instructions,
2. forwarding results from the FUs to the matching IQ entries,
3. issuing instructions to the FUs, and
4. flushing the IQ entries for instructions along the mispredicted paths.

Energy dissipations take place within the ROB during reads and writes to the register file that implements the ROB or when associative addressing is used. Specifically, these dissipations occur in the course of:

1. establishing the ROB entries,
2. reading out part of a ROB entry (when memory instructions are moved to the LSQ or when the valid data value for the most recent entry for an architectural register is read out),
3. reading out all of a ROB entry (at the time of committing an instruction),
4. writing results from FUs to the ROB entries, and
5. flushing the ROB entries on mispredictions or interrupts.

Energy dissipations occur within the LSQ in the course of:

1. establishing a LSQ entry,
2. writing computed effective addresses into a LSQ entry,
3. forwarding the result of a pending store in the LSQ to a later load,

TABLE 1
Architectural Configuration of a Simulated 4-Way Superscalar Processor

| Parameter | Configuration |
|---|---|
| **Machine width** | 4-wide fetch, 4-wide issue, 4-wide commit |
| **Window size** | 32 entry issue queue, 96 entry reorder buffer, 32 entry load/store queue |
| **L1 I-cache (I1-cache)** | 32 KB, 2-way set-associative, 32 byte line, 2 cycles hit time |
| **L1 D-cache (D1-cache)** | 32 KB, 4-way set-associative, 32 byte line, 2 cycles hit time |
| **L2 Cache unified** | 512 KB, 4-way set-associative, 64 byte line, 8 cycles hit time |
| **BTB/Predictor** | 1024 entry, 2-way set-associative, hybrid gshare, bimodal |
| **Memory** | 128 bit wide, 60 cycles first chunk, 2 cycles interchunk |
| **TLB** | 64 entry (I), 128 entry (D), 4-way set-associative, 30 cycles miss latency |
| **FUs and Latency (total/issue)** | 4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24) |

4. forwarding the data in a register to be stored to a matching entry in the LSQ, and
5. initiating D-cache accesses from the LSQ.

All of these datapath components account for a significant amount of the overall chip energy dissipation. For example, according to the results presented in [13], these components contribute more than 50 percent to the overall chip power in a datapath that integrates physical registers within the ROB slots (this excludes the power of the clock distribution network).

## 3 Simulation Methodology

We used the AccuPower toolset [23] to evaluate the impact of the proposed technique on the processor's power and performance. As described above, for this study, we considered the datapath variation, where the physical registers are implemented as the ROB slots. However, the proposed techniques are naturally applicable to other datapath variations as they are not dependent on the particular implementation of out-of-order scheduling nor are they impacted by the location of the result repositories.

The configuration of a 4-way superscalar processor is shown in Table 1. Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. The results from the simulation of the first 1 billion instructions were discarded and the results from the execution of the following 100 million instructions were used.

## 4 Resizing the Datapath Components: The Requirements

In this section, we discuss the hardware facilities needed to support incremental resource allocation and deallocation and the relevant constraints.

### 4.1 Multipartitioned Resources

The organization of the IQ allowing for incremental allocation and deallocation is depicted in Fig. 2. The ROB and the LSQ are partitioned in a similar fashion. The IQ, the ROB, and the LSQ are each implemented as a number of independent partitions. Each partition is a self-standing and independently usable unit, complete with its own precharger, sense amps, and input/output drivers. Using separate prechargers and sense amps for each partition (as opposed to shared prechargers and sense amps for all of the partitions) makes it possible to use smaller, simpler, and more energy-efficient sense amps and prechargers.

A number of partitions can be assembled to implement a larger structure, as shown in Fig. 2. The connection running across the entries within a partition (such as bit-lines, forwarding bus lines, etc.) can be connected to a common through line (shown on the right in Fig. 2) through bypass switches. To add (i.e., allocate) the partition to the IQ and thus extend the effective size of the IQ, the bypass switch for a partition is turned on and the power supply to the partition is enabled. Similarly, the partition can be deallocated by turning off the corresponding bypass switches.

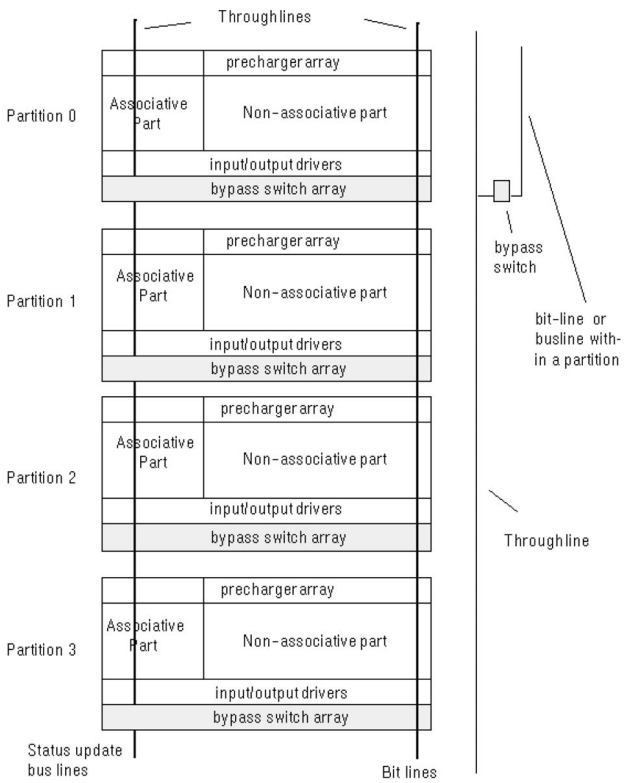


Fig. 2. The partitioned issue queue.

In addition, power to the bitcells in the shut-off partitions can be turned off to avoid leakage dissipation.

Partitions are deallocated in such a way that the powered-on partitions always span consecutive entries within the IQ—this is done for ease of implementation. Entries can be allocated in the IQ—across one or more active partitions in any order; an associative searching mechanism is used to look up free entries at the time of dispatching. The energy overhead for this associative lookup is accounted for in our measurements.

The partition sizes (i.e., the number of entries within each partition) for the IQ, the LSQ, and the ROB have to be chosen carefully. Making the partition sizes smaller allows for finer grain control for resource allocation and deallocation, but small partition sizes can lead to higher partitioning overhead in the form of an increase in the layout area and a decrease in the energy savings. In our studies, we assumed that the IQ and the LSQ partition size is eight entries and the ROB partition size is 16 entries. These choices were found to work out well for a large range of system configurations, particularly the ones that we studied in this effort.

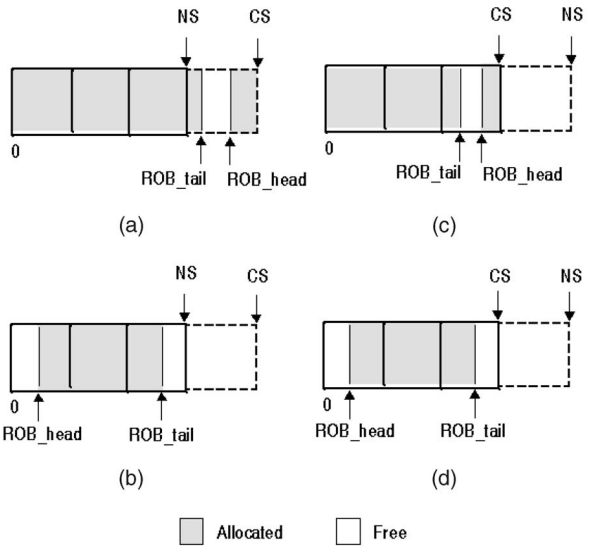


Fig. 3. ROB resizing scenarios. (a) Deallocation. (b) Deallocation. (c) Allocation. (d) Allocation.

### 4.2 Resizing Constraints

In the rest of the discussion, the allocation and deallocation of a resource is sometimes referred to as resource resizing. Deallocation of a partition within a resource is called resource downsizing and allocation of a currently turned-off partition is called resource upsizing. The resources considered in this work differ in the way they are managed. The issue queue is generally a non-FIFO structure, where a free entry for a new incoming instruction is searched associatively and instructions can be issued from any entry in the issue queue. In contrast, the reorder buffer and the load-store queue are FIFO-style circular queues, where entries are established for the new instruction at the tail end of the queues and the instructions commit or start the D-cache access only when they are positioned at the head end of these queues. These differences dictate the resizing constraints, as described below.

The actual downsizing of the issue queue may not always be performed immediately after the downsizing decision is made and is deferred till all instructions are issued from the IQ partition that is to be deactivated. The duration between the time a decision is made to downsize and the time of the actual deallocation of a partition is called a transition period. Instruction dispatch is blocked during the transition period if the IQ entry allocated for the new instruction belongs to the partition that will become inactive. Otherwise, dispatches continue to the active IQ partitions. Allocations of additional IQ partitions can be performed without any delay.

The dynamic resizing of the ROB and the LSQ requires additional considerations because of the circular FIFO nature of these structures. We discuss these particularities below for the ROB; the constraints for the LSQ are similar.

The ROB is a circular FIFO structure with two pointers —the *ROB_tail* and the *ROB_head*. The *ROB_tail* indicates the next free entry in the ROB and is used during instruction dispatching to locate and establish the entries for the dispatched instructions. The *ROB_head* is used during the commit stage to update the architectural registers in program order. Fig. 3 depicts a ROB with four partitions and also shows some possible dispositions of the *ROB_head* and the *ROB_tail* pointers at the time the allocation/deallocation decision is made. The pointer *CS* (Current Size) indicates the current upper bound of the ROB. The pointer *NS* (New Size) specifies the newly established upper bound of the ROB after potential allocation or deallocation of one partition. The partition that is to be allocated/deallocated is shown as a dashed box.

To preserve the logical integrity of the ROB, in some situations, partitions can be allocated and deallocated only when the queue extremities coincide with the partition boundaries. To deallocate a partition, two conditions have to be satisfied after the decision to downsize has been made. First, as in the case of the issue queue, all instructions from the partition to be deallocated must commit. Second, further dispatches must not be made to the partition being deallocated.

The deallocation scenarios are illustrated in Fig. 3a and Fig. 3b. In the situation shown in Fig. 3a, the deallocation is delayed until the *ROB_tail* reaches *NS* and the *ROB_head* becomes zero. Notice that the *ROB_head* wraps around twice before the deallocation can occur. This is, of course, an extreme case as it is rare for the ROB occupancy to be very high immediately prior to the instant of reaching a downsizing decision. This is because the downsizing decision is based on the occupancy estimates (as described later) and it is expected that the ROB occupancy is somewhat low if the deallocation decision is made. Slight variations of this specific case can also be considered, where the *ROB_tail* points to the left of the NS pointer. In that case, the *ROB_head* wraps around only once before the deallocation occurs. Fig. 3b represents the case where the deallocation of the partition marked by the dashed box can be performed immediately.

To allocate a ROB partition, the value of the *ROB_head* pointer must be less than the value of the *ROB_tail* pointer to preserve the correct placement of the newly dispatched instructions into the ROB. Allocation scenarios are illustrated in Fig. 3c and Fig. 3d. Fig. 3c shows the situation where the allocation is deferred until the value of the

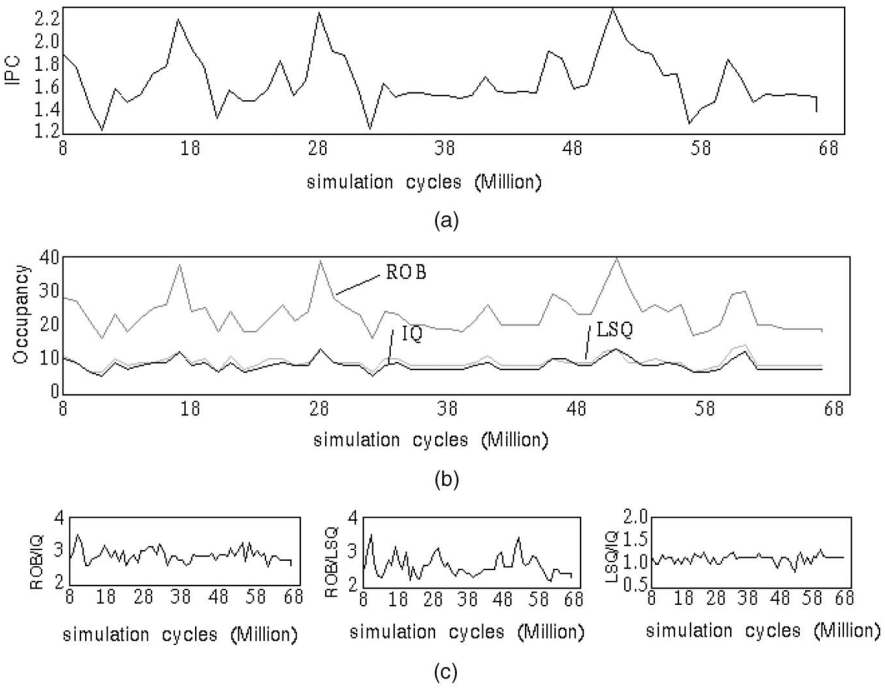


Fig. 4. Dynamic behavior of the **gcc** benchmark for 100 million instructions. (a) Commit IPC. (b) Occupancies of the IQ, the ROB, and the LSQ. (c) Ratios between the occupancies of various datapath resources.

*ROB_head* pointer reaches the value of zero. The bottom part shows the case where the allocation can be performed immediately. In our simulations, we fully accounted for the delays caused by possible misalignment of the ROB pointers and also the delays needed to clear the issue queue partition marked for deallocation.

## 5 Resource Usage in a Superscalar Datapath

To get guidance in the design of dynamic resource allocation strategy, we examined the resource usage patterns using our simulation infrastructure and SPEC 2000 benchmarks. We studied the correlations of program's performance and the occupancies (number of allocated entries) of the IQ, the ROB, and the LSQ. Representative results for some SPEC 2000 integer and floating-point benchmarks are shown in Figs. 4, 5, 6, 7, and 8.

Figs. 4, 5, and 6 show the program's IPCs and the occupancies of the three resources for the *gcc*, *apsi*, and *mcf* SPEC 2000 benchmarks, respectively. Also shown are the ratios of the resource occupancies. Measurements were taken for 100 million committed instructions after skipping the first 1 billion instructions. For each benchmark, we recorded the IPCs and the average occupancies after every 1 million cycles of execution and then graphed the results. The results in each period only reflect the activity within that period—no prior statistic is accounted for. In each of these graphs, M stands for million. As seen from these graphs, the performance (IPC) of all three programs changes considerably in the course of execution. In some situations, the frequency of oscillations is fairly high (*mcf*); in others, we observe a few major phases and, within each phase, the performance does not change significantly (*apsi*).

As the performance changes, so do the occupancies of various datapath queues. In these examples, changes in the resource occupancies are largely synchronized with the variations in IPCs. Where the IPC increases, the resource occupancies also increase, thus maintaining more instructions in the out-of-order scheduling window to extract higher performance. However, as we will see later, such a correlation is not always the case.

Another observation that can be made from these graphs is that the resource occupancies are correlated either positively or negatively. In some situations, as the number of instructions residing in the ROB increases, so does the number of instructions residing in the IQ. This suggests that resizing only one datapath resource, as has been done in some previous studies, is insufficient—in fact, if the sizes of other resources are not dynamically adjusted as well, these resources will remain overcommited most of the time. In other situations, however, the increase in the ROB occupancy is accompanied by the decrease in the IQ occupancy (Fig. 5b, *apsi* benchmark). The occupancy of the LSQ depends on the relative frequency of memory instructions and, therefore, it is correlated with the occupancies of other resources in an unpredictable manner. Even from these representative results, one can see that it is difficult, if not impossible, to adjust the sizes of multiple resources by monitoring one of these resources and rescaling the number of active partitions within other resources proportionately. This is primarily because the ratios of the resource occupancies also change drastically across a program's execution, as shown in Figs. 4c, 5c, and 6c.

For the benchmarks where performance is dictated by the load miss loop (and not the branch misprediction loop), the LSQ is nearly always full. To have the highest performance, the CPU needs to get the loads that miss into the D-cache out to the memory subsystem as quickly as

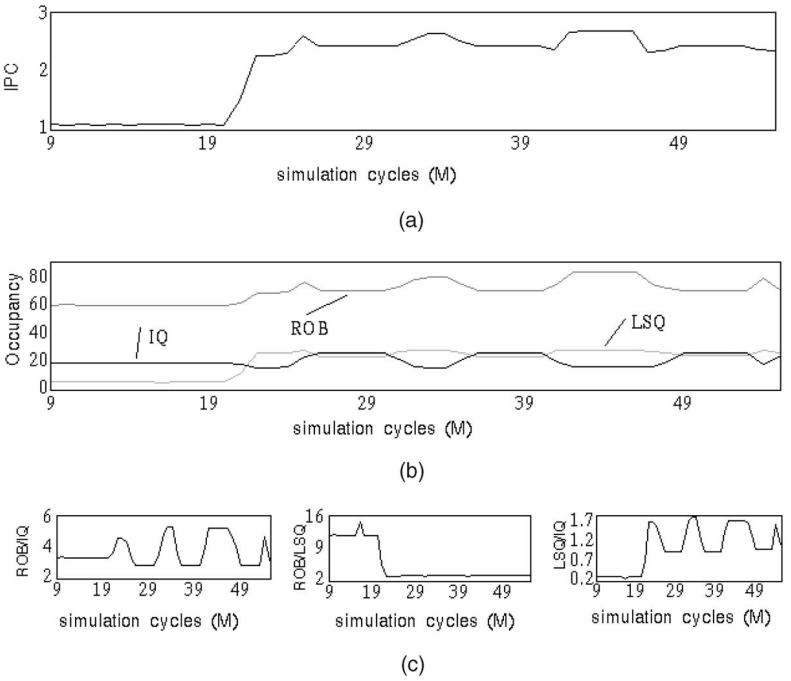


Fig. 5. Dynamic behavior of the **apsi** benchmark for 100 M instructions. (a) Commit IPC. (b) Occupancies of the IQ, the ROB, and the LSQ. (c) Ratios between the occupancies of various datapath resources.

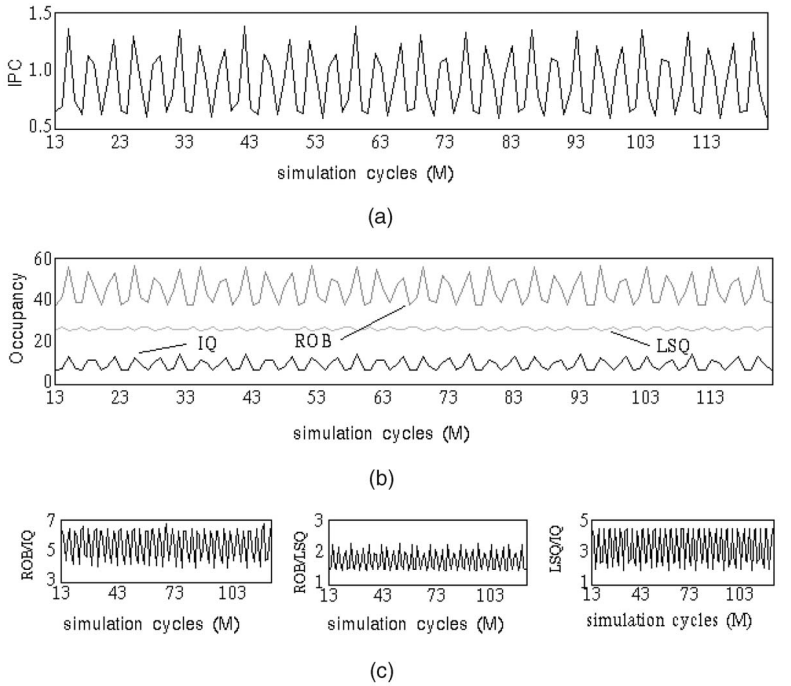


Fig. 6. Dynamic behavior of the **mcf** benchmark for 100 M instructions. (a) Commit IPC. (b) Occupancies of the IQ, the ROB, and the LSQ. (c) Ratios between the occupancies of various datapath resources.

possible. Therefore, the goal of the ROB and the IQ is to not limit the visibility of the loads to memory. For MCF, there are lots of independent loads that occur often enough to not require a large ROB or IQ. However, this is not the case with APSI. APSI needs the large ROB and IQ because it probably needs the larger windows to extract ILP.

We also performed experiments analyzing the program's behavior during more fine-grain timing intervals. Figs. 7 and 8 show the IPCs and the occupancies of the datapath queues for two SPEC 2000 benchmarks—*gcc* and *art*—executed for 5 million instructions after skipping the first 1 billion instructions. Here, we measured the relevant statistics after every 10,000 cycles. The graphs show that the behavior of programs varies significantly across the smaller time intervals, as expected. An interesting phenomenon can be observed by analyzing the graphs in Figs. 7 and 8. While, for the *gcc* benchmark, the changes in the ROB occupancy largely correlate with the IPC changes (with the exception

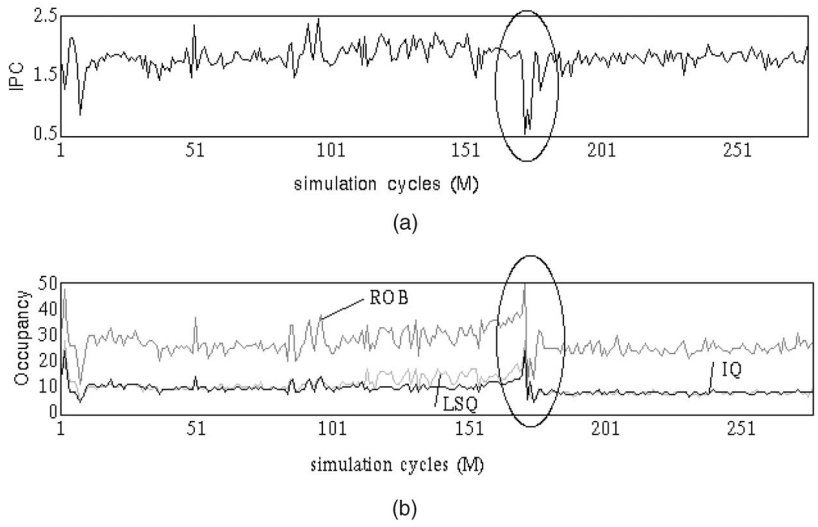


Fig. 7. Dynamic behavior of the **gcc** benchmark for 5M instructions. (a) Commit IPC. (b) Occupancies of the IQ, the ROB, and the LSQ.

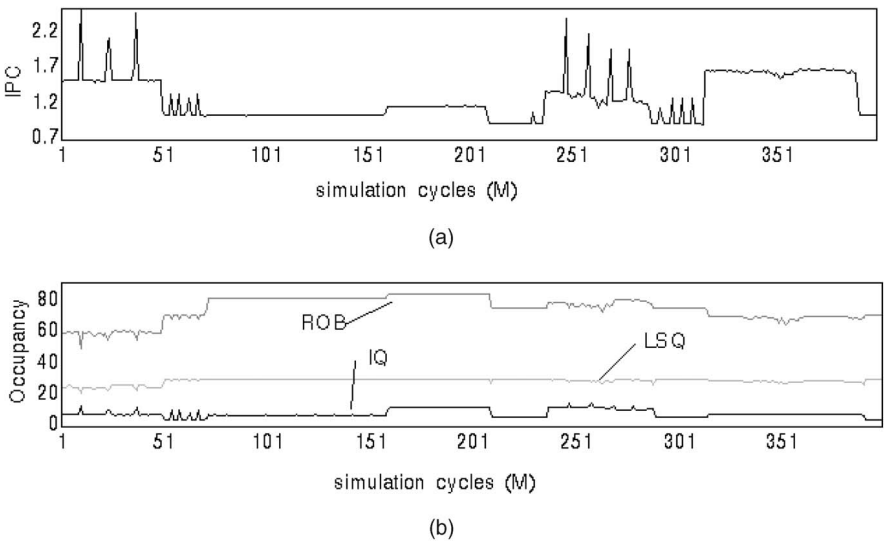


Fig. 8. Dynamic behavior of the **art** benchmark for 5M instructions. (a) Commit IPC. (b) Occupancies of the IQ, the ROB, and the LSQ.

of the behavior in the circled region), the situation is quite the opposite for the *art* benchmark (Fig. 8). Here, the decrease in IPC is accompanied by higher ROB utilization and vice versa. The same phenomenon can be seen in the circled region in Fig. 7. One possible explanation for this is the following: When the IPC is high, there is a large amount of ILP in the code and the bulk of the instruction stream is formed by the low-latency operations, many of them independent. These instructions move through the pipeline quickly and the ROB is not fully utilized because the ability to buffer a large number of instructions is not needed to extract ILP. The IPC drop is usually caused by the presence of long latency operations or long dependence chains. In either of these cases, the instructions will now spend more time in the pipeline, causing the ROB to gradually fill up and the ROB occupancy to increase. As a result, it is not always possible to correlate the IPC measures with the actual resource demands of the applications and, therefore, the absolute IPC values or the IPC drops may not be adequate indicators for driving the resource allocations. On the other hand, the actual resource occupancy is an accurate indicator of the application's demands for this resource.

A number of approaches to resizing multiple datapath resources can be used. At first glance, it seems that, to reduce the overhead of the control logic, one can consider taking advantage of the potential correlations among the resource utilizations and only monitor the use of one of the resources to drive the allocations within a group of resources whose usage is correlated with the usage of the resource being monitored. However, as our experiments demonstrate, this is not easy to do in practice. Instead, we rely on independent monitoring of the individual occupancies of the IQ, the ROB, and the LSQ and make the resizing decision for each resource independently of the others, only based on the local information. We now explain the details of our resizing strategy.

## 6 Resource Allocation Strategies

The resizing algorithm operates in two phases—one is used for downsizing a resource (turning off a partition) and the

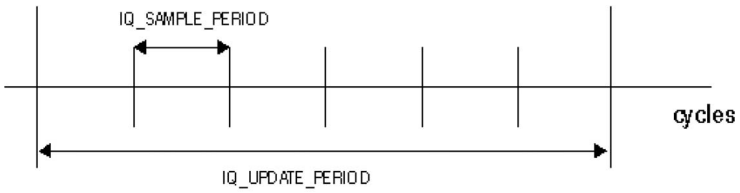


Fig. 9. Sample period and update period used for the IQ downsizing.

other is used for upsizing the resource (adding a partition). We describe these phases in the example of an issue queue, the mechanisms used for the ROB and the LSQ are similar, provided that adequate reservations are made for the circular nature of the ROB and the LSQ, as discussed in Section 4.2.

### 6.1 The Downsizing Phase

The downsizing of the IQ is considered periodically—at the end of every IQ update period (*IQ_update_period*). During this period, the IQ occupancy, as measured by the number of allocated entries within the active partitions of the IQ, is sampled several times at periodic intervals. The value of the sampling interval (*IQ_sample_period*) is an integer divisor of *IQ_update_period*. The average of these occupancy samples is taken as the active IQ size (maintained in the variable *active_IQ_size*) in the current update period. Both *IQ_update_period* and *IQ_sample_period* were chosen as powers of two to let the integer part and the fractional part of the computed IQ occupancy be isolated easily in the occupancy counter. This avoids the use of a full-fledged division logic. Fig. 9 depicts the relationship between *IQ_update_period* and *IQ_sample_period*.

At the end of every IQ update period, we compute the difference, *diff = current_IQ_size-active_IQ_size*, where *current_IQ_size* is the size of the issue queue at that instant (in number of allocated entries). If *diff* is less than the size of one IQ partition (*IQ_p_size*), no resizing is needed. If, however, diff is greater than (or equal to) the *IQ_p_size*, two scenarios are possible, depending on whether an aggressive or a conservative downsizing strategy is implemented. In a conservative scheme, at most one IQ partition can be deactivated at a time. In an aggressive downsizing scheme, the maximum allowable number of partitions, *max_p*, as obtained directly from *diff* (*max_p = floor(diff/IQ_p_size)*), are deallocated.

The variable *active_IQ_size* provides a reasonable approximation of the average IQ occupancy within the most recent IQ update period. The difference between the current size of the IQ (*current_IQ_size*) and *active_IQ_size* in the update period indicates the degree of overcommited (that is, underused or unused) IQ partitions. The IQ is scaled down in size only if the sampling of the IQ occupancy indicates that the IQ partitions are overcommited. By thus downsizing resources only when resource partitions are overcommited, we minimize the penalty on performance.

At the end of every IQ sampling period, we record the IQ occupancy by using bit-vector counting logic for each active partition and then adding up these counts.

### 6.2 Upsizing Phase

The second phase of the resizing algorithm is used to scale the IQ size back up once the application begins to demand more resources. This is necessary because the deallocation of several partitions can cause significant performance loss if the application enters a new phase which requires the use of a full instruction window to extract all available parallelism. Consequently, we need to somehow identify when the insufficient size of the issue queue has a negative impact on performance. One way of doing this is to monitor the actual commit IPCs (as the ultimate performance measure) and increase the issue queue size if the IPC drops below a certain threshold with regard to a previously recorded IPC value. The fundamental problem with this approach is that the reason for the IPC drop is not known and it is assumed that the performance loss is always due to the limited size of the scheduling window. However, the loss may actually be attributed to the increased cache miss rates or increased branch misprediction rates. In those situations, increasing the issue queue size does not elevate the performance, but simply wastes power saving opportunities.

Instead of relying on the absolute IPC loss to drive the issue queue upsizing, we looked for the gauges, indicating that the performance drop is definitely due to the limited size of the issue queue. One such simple gauge is the rate at which instruction dispatching blocks due to the nonavailability of entries within the IQ. We use the IQ overflow counter to count the number of cycles for which dispatch blocks because an entry cannot be established in the issue queue for a new instruction. For a W-way superscalar machine, we consider that a dispatch block occurs in a cycle if at least one of the W instructions cannot be injected into the pipeline because the issue queue is full. The counter is initialized to zero at the beginning of every IQ update period and also immediately after making additional resource allocations within the IQ. The reason for resetting the counter in the latter instance is described below. Once the value of this counter exceeds a predetermined IQ overflow threshold, (maintained in the variable *IQ_overflow_threshold*), one of the currently deactivated IQ partitions, if any, is turned on, effectively upsizing the IQ. Note that additional resource allocations can take place well before the expiration of an IQ update period to react quickly to the increased demands of the program. The upsizing phase is thus more aggressive than the downsizing phase. While the downsizing decisions are synchronized with respect to the update period, the upsizing decisions are asynchronous and depend only on the dispatch rate and the current size of the scheduling window.

The main reason for taking multiple samples within a single update period is to get a better estimate of the average resource usage within this update period. An upsizing occurring within an update period can distort this average quite drastically and result in an erroneous downsizing at the end of this period. Such a situation causes instability in the control mechanism used for our scheme. It is for this reason that we abandon the current update period

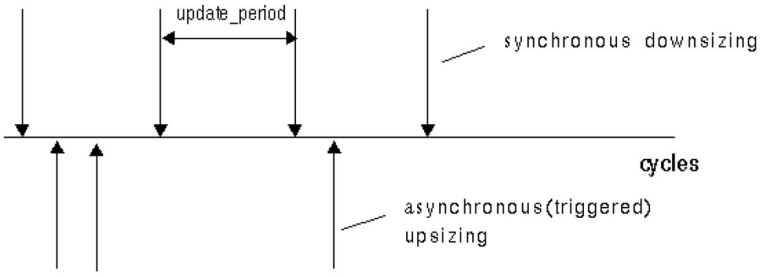


Fig. 10. Synchronous downsizing versus asynchronous upsizing.

TABLE 2
Percentage of Turned-Off Entries within the IQ, the ROB, and the LSQ and Performance Degradation for Various Overflow Thresholds

| OT | IPC drop (%) | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| 32K | 0.83 | 12.36 | 19.22 | 5.14 |
| 64K | 2.01 | 16.61 | 24.46 | 9.22 |
| 128K | 5.22 | 27.78 | 34.88 | 20 |
| 256K | 11.05 | 39.77 | 46.01 | 33.78 |

and start a new update cycle immediately after resizing up. For similar reasons, during the transition periods immediately prior to an actual downsizing, no statistics are collected (all counters controlling resizing are frozen at 0).

By varying the values of *IQ_overflow_threshold* and *IQ_update_period*, either of the upsizing or the downsizing phases can be made more aggressive depending on the design goals as dictated by power (downsizing)/performance (upsizing) trade-offs.

### 6.3 Summary of the Control Strategy

The control strategy for dynamic resource allocation, as described above, is very simple in that only three parameters are used to control the allocations for each resource and each resource is controlled separately and independently. The ratio of *update_period* to *overflow_threshold* determines the aggressiveness of resizing. The higher ratio means that the resource upsizing is triggered more often as fewer dispatch blockings can be tolerated. The *sample_period* determines the frequency of occupancy sampling within the update period. Fig. 10 summarizes the control strategy by showing the synchronous downsizing and asynchronous upsizing. As seen in the picture, more than one upsizing can be triggered during one update period. It is also possible that two downsizings can occur in a row, with no upsizing act in an update period. In the next section, we evaluate this design in terms of power-performance trade-offs and study the sensitivity of the proposed strategy to each of the parameters.

## 7 Results and Discussions

Table 2 shows performance degradation caused by the dynamic resizing of multiple datapath resources. Also shown in this table is the percentage of entries for each resource that are turned off in an average cycle. Separate statistics are shown for the IQ, the ROB, and the LSQ. Presented results are the averages over all simulated SPEC 2000 benchmarks. For these experiments, we varied the overflow threshold and kept the value of the update period fixed at 512K cycles for each resource in all experiments, thus modeling a very coarse-grain resizing. This value has to be chosen carefully—making it too large would cause variations in the resource usages to go unnoticed, while making it too small would result in a prohibitive resizing overhead. We studied the effects of various update periods on our resizing scheme and results are presented later in this section. The value of the *sample_period* was chosen as 32 cycles, allowing for the acquisition of 16K occupancy samples in one update period.

To understand the results of Table 2, it is useful to think about the meaning of the resizing parameters. For example, the OT of 32K cycles for the UP of 512K cycles means that the act of the resource upsizing is triggered when instruction dispatching blocks for 32K cycles in the course of one update period. This implies that, on average, at least one dispatch blocking must occur in 16 cycles for the upsizing to be triggered. Such a low target on the number of blocked dispatches is likely to be reached well before the expiration of the update period. Consequently, any undercommitment of a resource, caused by premature deallocation, can be detected and corrected very fast, thus avoiding performance degradation. Of course, the opportunity for power savings will then be grossly reduced.

As seen from the data in Table 2, lower values of *overflow_threshold* result in virtually no performance loss, but the average number of turned-on entries and, thus, power savings potential, are limited. For larger values of *overflow_threshold*, the potential power savings can be more significant, but this comes at a cost of noticeable performance drop—as high as 11 percent on the average for the OT of 256K cycles. The OT of 256K cycles essentially tolerates dispatch blocking if the blocking rate is less than one in two cycles. Across the individual benchmarks, the largest performance drop is in *perl* (almost 25 percent), *gcc* (17 percent), and *equake* (16 percent).

We did not perform any experiments for the values of *overflow_threshold* less than 32K because the average performance loss was already well below 1 percent in this

TABLE 3
Performance Drop and Percentage of Turned-Off Entries (OT = 128K)

| | IPC drop (%) | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| Int avg | 5.71 | 33.18 | 46.40 | 23.47 |
| FP avg | 4.61 | 21.02 | 20.48 | 15.66 |
| Average | 5.22 | 27.78 | 34.88 | 20.00 |

TABLE 4
Performance Drop and Percentage of Turned-Off Entries (OT = 64K)

| | IPC drop (%) | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| Int avg | 1.79 | 22.22 | 36.71 | 11.38 |
| FP avg | 2.28 | 9.61 | 9.14 | 6.52 |
| Average | 2.01 | 16.61 | 24.46 | 9.22 |

case. Decreasing the value of *overflow_threshold* further would have an adverse effect on energy dissipation (more partitions will be on) with no further performance improvement. The maximum possible value of *overflow_threshold* for a resource is equal to the value of *update_period* used for that resource (512K in these experiments). In this case, the resource upsizing is triggered if, during the update period, resource overflows are encountered every cycle —this will virtually never happen because, in some cycles, no new instructions may be fetched and decoded. We did not simulate this configuration because, even with the OT of 256K cycles, the performance drop was already very significant and not acceptable for the high-performance designs. Thus, we effectively exhausted the space of all reasonable values of OT for a given update period (considering, of course, that these values are powers of two for implementation reasons, as discussed above).

Table 3 shows similar results for OT of 128K cycles. In terms of performance, only three integer benchmarks (*gcc*, *perl*, and *vortex*) have a degradation of more than 10 percent. For many of the programs, the drop is confined within low single digits. Higher power savings can be achieved for integer benchmarks because they are generally less resource-hungry than floating-point benchmarks and present more opportunities for downsizing. For the IQ, the lowest percentage of turned-off entries are for *mgrid*, *swim*, *wupwise*, and *bzip2* benchmarks—these use full IQ most of the time. Most of the time two or three IQ partitions are on. For the ROB, a larger number of partitions can usually be turned off, for the integer benchmarks almost half of the ROB can be deactivated, while, for floating-point benchmarks, this number is only about 20 percent. This is a consequence of the fact that floating-point programs typically have higher parallelism than the integer codes and the ROB occupancy is higher. For *mgrid*, all ROB partitions are always on in the course of execution. Also, all LSQ partitions are always turned on for the *mcf*, *art*, and *mgrid* benchmarks.

Table 4 shows similar results for the OT of 64K cycles. The important advantage here is that the performance degradation for *all* executed benchmarks is below 3 percent. Yet, one IQ partition can be turned off most of the time for most of the benchmarks and a quarter of the ROB entries can also be turned off on the average. For many benchmarks, the LSQ must be kept entirely on. However, for the *gcc* benchmark, we can turn off half of the LSQ with little impact on performance.

Figs. 11, 12, and 13 show that the actual number of allocated entries within each resource is typically higher than the resource occupancy by about the size of one to one and a half partitions. Results in these figures are presented for the OT of 128K cycles. For some benchmarks, the difference is higher than for the others. The difference is due to the relatively large number of cycles that it takes for the downsizing phase to react to the occupancy changes. The downsizing decision only takes place at the end of a fairly long update period, thus, if the resource occupancy drops significantly in the beginning of an update period, then the difference between the actual number of allocated entries and the number of occupied entries can be significant. As the update period decreases, the difference becomes closer to the size of one partition as the occupancy changes are quickly reflected in resource downsizing. In any case, the number of allocated entries tracks the actual resource occupancies, as expected.

We also performed some experiments to study the sensitivity of the allocation strategy to the variations in update periods. Some of these results, averaged across all benchmarks, are summarized in Table 5. Table 5 shows the results, averaged over all benchmarks, where the UP to OT ratio was kept at 4. The sample period of 32 cycles was assumed in all these simulations.

As expected, larger update periods result in somewhat higher performance loss because of the slower reaction of the upsizing phase to the changes in the application's demands. This is, of course, the consequence of higher overflow thresholds, which have to increase commensurately with the update periods to maintain the trade-off between power and performance. The advantage of larger update periods is the reduction of control logic overhead because resource monitoring and resizing decisions are made at a coarser granularity. The differences in terms of performance and the percentages of resource entries that can be turned off are not significant across various update periods. There is only a 0.3 percent difference in performance and about half a percentage point difference in terms of the number of entries that can be deallocated between UP of 512K cycles and UP of 2,048 cycles. Therefore, to keep the

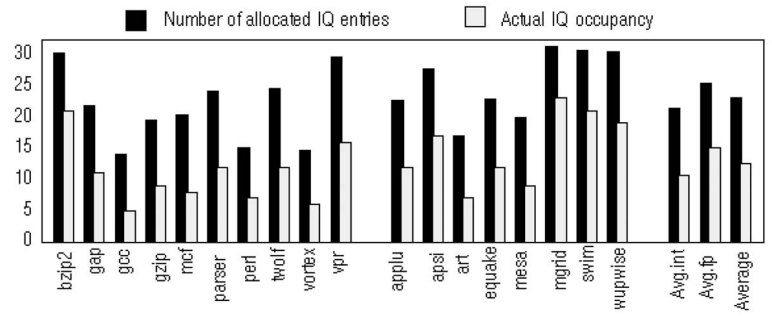


Fig. 11. Number of allocated entries within IQ versus IQ occupancy.

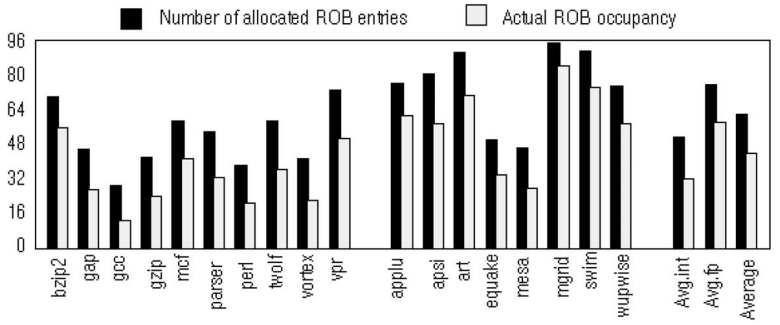


Fig. 12. Number of allocated entries within ROB versus ROB occupancy.

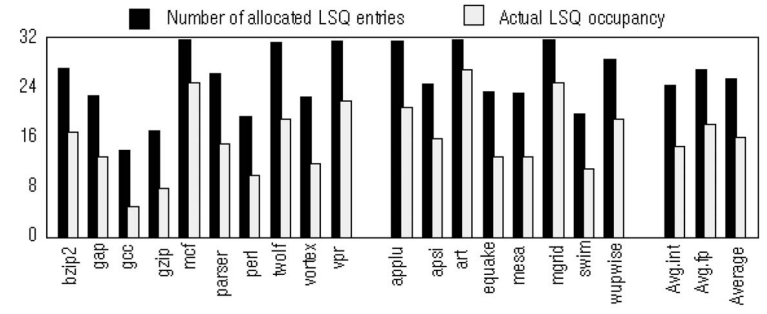


Fig. 13. Number of allocated entries within LSQ versus LSQ occupancy.

TABLE 5
Statistics for Various Update Periods where UP/OT = 4

| UP, OT | IPC drop % | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| 512K, 128K | 5.2 | 27.66 | 34.89 | 20.02 |
| 32K, 8K | 4.94 | 27.27 | 34.44 | 19.91 |
| 2048, 512 | 4.91 | 27.17 | 34.29 | 19.46 |

monitoring overhead low, course-grain update periods can be effectively used.

We also studied the impact of sampling periods on the performance of the proposed algorithm. Table 6 summarizes some of the results for UP of 512K cycles and OT of 128K cycles. Results are averaged over all executed SPEC 2000 benchmarks.

As seen from the table, the results are practically identical for various sampling periods. We observed the same behavior for other update periods and overflow thresholds. For smaller update periods, the performance loss somewhat increases only if the number of samples is very low. For example, Table 7 represents the results for the UP of 2,048 cycles and the OT of 512 cycles. As we see, in the extreme scenario when only one occupancy sample is used to trigger the downsizing, the performance loss increases by almost 2 percent compared to the situation when sampling occurs every 32 cycles. This erratic sampling also allows for a higher percentage of entries to be turned off within each resource, but the trade-off is not favorable because only about 4 percent more of each resource is deactivated at the expense of an almost 2 percent decrease in performance. Thus, if sampling occurs with a reasonable frequency (allowing for the acquisition of at least several tens of samples in an update period), then the results are virtually identical to those obtained by performing cycle-by-cycle occupancy sampling.

TABLE 6
Impact of the Sample Period on the Resizing Scheme (UP = 512K Cycles)

| sample period (cycles) | IPC drop % | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| 2 | 5.21 | 27.67 | 34.81 | 19.97 |
| 32 | 5.20 | 27.66 | 34.89 | 20.02 |
| 2048 | 5.22 | 27.78 | 34.88 | 20.00 |

TABLE 7
Impact of the Sample Period on the Resizing Scheme (UP = 2K Cycles)

| sample period (cycles) | IPC drop % | % of turned-off IQ entries | % of turned-off ROB entries | % of turned-off LSQ entries |
|---|---|---|---|---|
| 2 | 4.82 | 27.69 | 34.34 | 19.46 |
| 32 | 5.01 | 27.47 | 34.29 | 19.46 |
| 2048 | 6.66 | 30.49 | 38.43 | 23.91 |

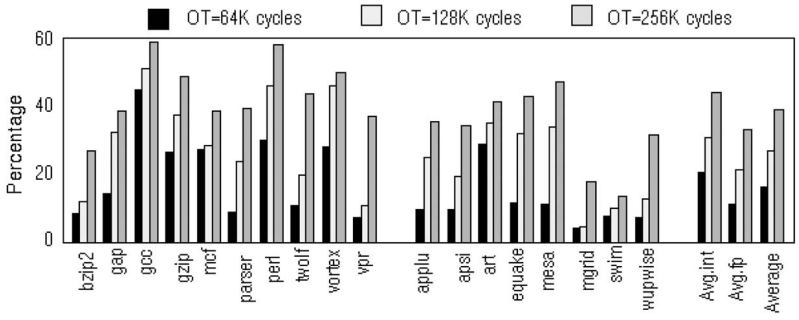


Fig. 14. Savings in energy per cycle within the issue queue.

The impact of resizing on the power dissipation depends on the underlying assumption about the usage of multi-partitioned resources. There are two variations. First, as a natural consequence of partitioning, the well-known technique called bitline segmentation can be used within each resource to reduce power. With the use of bitline segmentation, only the partition that contains the currently addressed entry is activated in the course of directly addressed reads or writes, thus saving power. In this case, the number of turned-off partitions has almost no impact on the power dissipated in the course of reads and writes—a small advantage of turning off partitions is the opportunity to deactivate the second-level decoders associated with the turned-off partitions. The main advantage of dynamic resizing compared to the bitline segmentation is the reduction of power consumption in the course of associative comparisons and instruction selection. Here, the comparators associated with the inactive partitions can be turned off and, also, the capacitance of the result/forwarding buses can be reduced by disconnecting them from inactive partitions. If, however, the bitline segmentation is not used, then the amount of dynamic power dissipated in the course of directly addressed reads and writes will also strongly depend on the number of activated partitions.

Fig. 14 shows the savings in energy per cycle achievable within the issue queue if dynamic resource allocation mechanism is used. Here, we assume that bit line segmentation is not used and the bitlines are driven across all active partitions. All results are for the UP = 512K cycles and sampling period of 32 cycles. On the average, the power reduction is 16.5 percent for the OT of 64K cycles, 26.8 percent for the OT of 128K cycles, and 39.4 percent for the OT of 256K cycles. These percentages are essentially identical to the percentages of deallocated IQ entries presented in Table 2. However, if the bitline segmentation is also used within the IQ to only activate the partition that is being accessed, then the situation changes. Power savings possible in the IQ with the incorporation of bitline segmentation are shown in Fig. 15. The power reduction is 36 percent, 42 percent, and 50 percent on the average for the OT values of 64K, 128K, and 256K cycles. As expected, the power savings are higher with bitline segmentation, but the difference from the results presented in Fig. 14 is relatively small, especially for the larger values of OT, where a significant percentage of the IQ entries is turned off using our technique. On the flip side, bitline segmentation can increase the cycle time since the predecoded address bits specifying the accessed segment must be driven to the inputs of the complementary switches at every cycle. If the bitline segmentation is also implemented in the baseline machine, then the additional power savings in the issue queue due to the dynamic resizing.

Fig. 16 shows the percentage of power savings achievable within the reorder buffer. Again, we first consider the situation, where bitline segmentation is not used. Twenty percent, 32 percent, and 45 percent of the ROB power can be saved by the use of our technique for the OT of 64K, 128K, and 256K cycles, respectively. Again, these percentages are very close to the numbers reported in Table 2. Notice that, for some benchmarks (*mgrid, swim*) where almost no deallocation of the ROB can be performed, the technique actually results in some small power increase due to the additional power dissipated accessing the bypass switches. Bitline segmentation provides significant

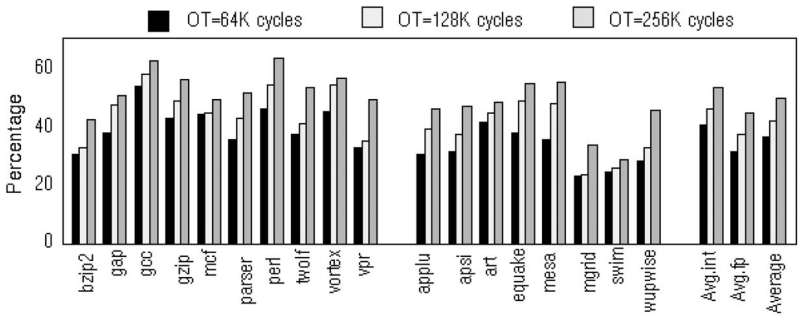


Fig. 15. Savings in energy per cycle within the issue queue with the use of dynamic resizing and bitline segmentation.

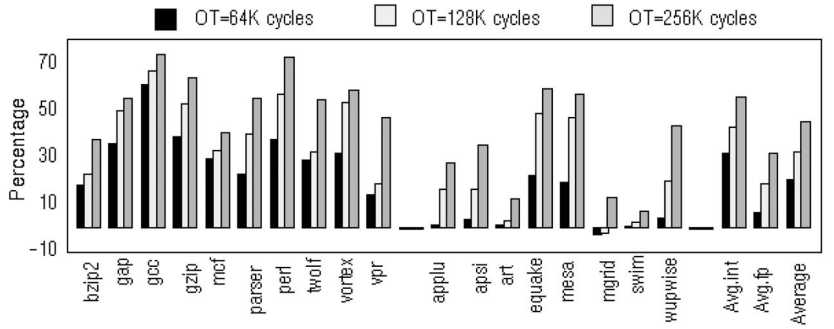


Fig. 16. Savings in energy per cycle within the reorder buffer.

additional benefits because all accesses to the ROB are directly addressed, thus resulting only in the activation of one 16-entry partition and six bypass switches. Due to the lack of any associative addressing or forwarding logic, the savings achievable within the ROB are essentially independent of the number of deactivated partitions. The small difference is due to the deactivation of second-level address decoders associated with inactive ROB partitions. The resulting savings are 73 to 74 percent if bitline segmentation is used within the ROB—a significant increase compared to the results of Fig. 16.

Some power reduction is possible within the load-store queue. Here, especially for the lower values of OT, many of the benchmarks actually have a power increase because very few LSQ entries can be deallocated, as discussed earlier. On the average, there is a power increase of about 1 percent for OT of 64K cycles, a power reduction of about 14 percent for the OT of 128K cycles, and a power reduction of about 31 percent for the OT of 256K cycles. Another reason for this slightly different power situation in the LSQ is the relatively higher contribution of associative addressing in the form of tag and address matching.

If the bitline segmentation is assumed to be in place in the baseline design, then, from the dynamic power standpoint, there is no added advantage due to dynamic resizing for the ROB. For the issue queue, the additional power savings compared to the basecase with bitline segmentation are 10 percent, 16 percent, and 24 percent for the OT of 64K, 128K, and 256K, respectively. For the LSQ, the additional savings are also noticeable as a significant amount of LSQ accesses is in the fore of various associative searches—the savings are 0.5 percent, 10 percent, and 22 percent. Furthermore, dynamic resizing can potentially reduce the leakage energy within the on-chip storage components and the savings will be proportional to the number of deactivated partitions.

## 8 Related Work

The technique presented in this paper for reducing power dissipation in the instruction scheduling logic hinges on the estimation of resource occupancies. The resource usages and the dynamic behavior of the SPEC 95 benchmarks were reported in [24] using the well-used Simplescalar simulator [8], where the reorder buffer, the physical registers, and the issue queue are integrated into an unified structure, called the Register Update Unit (RUU). The correlations among the IPC, RUU occupancy, cache misses, branch prediction, value prediction, and address prediction were also documented in [24]. We extend the study of [24] to architectures where the issue queue and the reorder buffer are implemented as distinct resources, as in most practical implementations of superscalar datapaths. We also study the correlations among the usage of these resources. Our studies show why a distributed, dynamic allocation of these resources is needed for reducing the power/energy dissipation.

Dynamic resource allocations within a *single* datapath component (the IQ) for conserving power was studied in [9], [10], [13]. Specifically, in [9], [10], the authors explored the design of an adaptive issue queue, where the queue entries were grouped into independent modules. The number of modules allocated was varied dynamically to track the ILP; power savings was achieved by turning off unused modules. In [9], [10], the activity of an issue queue entry was indicated by its ready bit (which indicates that the corresponding instruction is ready for issue to a function unit). The number of active issue queue entries, measured on a cycle-by-cycle basis, was used as a direct indication of the resource demands of the application.

In [13], Folegnani and Gonzalez introduced a FIFO issue queue that permitted out-of-order issue but avoided the compaction of vacated entries within the valid region of the queue to save power. The queue was divided into regions and the number of instructions committed from the most recently allocated issue queue region in FIFO order (called the "youngest region") was used to determine the number of regions within the circular buffer that were allocated for the actual extent of the issue queue. To avoid a performance hit, the number of regions allocated was incremented by one periodically; in-between, also at periodic intervals, a region was deactivated to save energy/power if the number of commits from the current youngest region was below a threshold. In [19], we have introduced an issue queue design that achieves significant energy savings using a variety of techniques, including the use of zero-byte encoding, comparators that dissipate energy on a match, and a form of dynamic activation of the accessed regions of the issue queue using bit-line segmentation. Many of the techniques used in [19] can be employed in conjunction with the scheme proposed in this paper to achieve a higher level of power savings.

In reality, the usage of *multiple* resources within a datapath is highly correlated, as seen from the results presented in this paper (and in some of the work cited below). The allocation of multiple datapath resources to conserve power/energy was first studied in the context of a multiclustered datapath (with nonreplicated register files) in [27], where dispatch rates and their relationship to the number of active clusters were well-documented. A similar but more explicit study was recently reported in [5], for the multiclustered Compaq 21264 processor with replicated register files. The dispatch rate was varied between 4, 6, and 8 to allow an unused cluster of function units to be shut off completely. The dispatch rate changes were triggered by the crossing of thresholds associated with the floating-point and overall IPC, requiring dispatch monitoring on a cycle-by-cycle basis. Fixed threshold values as used in [5] were chosen from the empirical data that was generated experimentally. Significant power savings within the dynamic scheduling components were achieved with a minimum reduction of the IPC.

In [17], the dispatch/issue/commit rate ("effective pipeline width") and the number of reorder buffer entries (the unified RUU size of the Simplescalar simulator) were dynamically adjusted to minimize the power/energy characteristics of hot spots in programs. This was done by dynamically profiling the power consumption of each program hot spot on all viable ROB size and pipeline width combinations. The configuration that resulted in the lowest power with minimum performance loss was chosen for subsequent executions of the same hotspot.

In [14], resource usages are controlled indirectly through pipeline gating and dispatch mode variations by letting the Operating System dictate the IPC requirements of the application. An industry standard, Advanced Configuration and Power Interface (ACPI) defining an open interface used by the OS to control power/energy consumption is also emerging [1]. In [4], a dynamic thermal management scheme was investigated to throttle power dissipation in the system by using several response mechanisms during the periods of thermal trauma. These included voltage and frequency scaling, decode throttling, speculation control, and I-cache toggling.

The dynamic allocation technique presented in this paper was first introduced in [22]. Since then, the scheme was extended by Dropsho et al. in [12], where, instead of using the average occupancy during the update period, the authors relied on the use of the limited histogramming. Their position is that the upper tail of the occupancy distribution is a better metric than the average occupancy. Based on the presented results, it is difficult to see any obvious gains in terms of power/performance trade-offs between our approach and the technique of [12]. It is certain, however, that the use of limited histogramming considerably complicates the control logic. In addition, the work of [12] includes the resizing of register files and caches in combination with the issue queue.

The positional approach to the processor's adaptation (in contrast to the temporal approaches discussed earlier) was proposed in [16], where Huang et al. exploited the similarity between different invocations of the same code section at the granularity of subroutines to save energy. To determine the best configuration for a code section, different configurations are tested on different executions of the same code section. Once the best configuration is determined, it is applied on future executions of the same code section. This approach is based on the intuition that a program's behavior at a given time is mostly related to the code that is being executed.

## 9 Concluding Remarks

The "one-size-fits-all" philosophy in allocating datapath resources results in resource overcommitment. Our approach to minimizing the power requirements of the datapath is to use a dynamic resource allocation strategy that tries to closely track the actual dynamic resource demand of the executing program. We primarily focused on using simple techniques to resize the IQ, the ROB (integrating physical registers), and the LSQ independently. In particular, these components are partitioned and the number of active (i.e., powered up) partitions are chosen dynamically to closely track the actual demands of the program. The IQ, the LSQ, and the ROB are controlled independently. Downsizing is driven by directly using sampled estimates of their individual occupancies. Upsizing, on the other hand, is done more aggressively, based effectively on the rate at which dispatches block due to the lack of these individual resources.

Our results show that the error arising from the computations of the average occupancy by sampling the actual occupancies at discrete points is tolerable since significant energy savings are achieved using our approach with very little impact on performance.

The power/performance trade-offs achieved by the use of the proposed mechanism are highly sensitive to the ratio of the update period to the threshold on the tolerable number of dispatch blockings. As long as this ratio stays unchanged, the results are largely independent of the absolute value of update period or of the number of occupancy samples taken during the update period.

## ACKNOWLEDGMENTS

This work was supported in part by the US Defense Advanced Research Projects Agency through contract number FC 306020020525 under the PAC-C program and the US National Science Foundation through award numbers MIP 9504767 and EIA 9911099 and IEEC at SUNY-Binghamton. This is an extended version of the paper that appears in the *Proceedings of MICRO '01* ([22]).

## REFERENCES

[1] Advanced Configuration and Power Interface Specification (Intel, Microsoft, Toshiba), 1999.
[2] D. Albonesi, "Selective Cache ways: On-Demand Cache Resource Allocation," *Proc. Int'l Symp. Microarchitecture,* 1999.
[3] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture,* 2000.
[4] D. Brooks and M. Martonosi, "Dynamic Thermal Management for High-Performance Microprocessors," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* 2001.
[5] I. Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing," *Proc. Int'l Symp. Computer Architecture,* pp. 218-229, 2001.
[6] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. 34th Int'l Symp. Microarchitecture,* 2000.
[7] D. Bhandarkar, *Alpha Implementations and Architecture Complete Reference and Guide.* Digital Press, 1996.
[8] D. Burger and T.M. Austin, "The SimpleScalar Tool Set: Version 2.0," technical report, Dept. of Computer Science, Univ. of Wisconsin-Madison, June 1997, and documentation for all SimpleScalar releases.
[9] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. Great Lakes Symp. VLSI Design,* 2001.
[10] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," *Proc. Workshop Power-Aware Computer Systems,* Nov. 2000.
[11] G. Cai, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation," *Proc. Cool-Chips Tutorial: An Industrial Perspective on Low Power Processor Design,* 1999.
[12] S. Dropsho et al., "Integrating Adaptive On-Chip Structures for Reduced Dynamic Power," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2002.
[13] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," *Proc. Int'l Symp. Computer Architecture,* pp. 230-239, 2001.
[14] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption," *Proc. Workshop Complexity-Effective Design,* June 2000.
[15] M. Huang, J. Renau, S.-M. Yoo, and J. Torellas, "A Framework for Dynamic Energy Efficiency and Temperature Management," *Proc. 33rd Int'l Symp. Microarchitecture,* 2000.
[16] M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," *Proc. 30th Int'l Symp. Computer Architecture (ISCA),* 2003.
[17] A. Iyer and D. Marculescu, "Run-Time Scaling of Microarchitecture Resources in a Processor for Energy Savings," *Proc. Kool Chips Workshop,* Dec. 2000.
[18] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. Int'l Symp. Computer Architecture,* pp. 240-251, 2001.
[19] G. Kucuk, K. Ghose, D. Ponomarev, and P. Kogge, "Energy Efficient Instruction Dispatch Buffer Design for Superscalar Processors," *Proc. Int'l Symp. Low-Power Electronics and Design,* pp. 237-242, 2001.
[20] *Microprocessor Report,* various issues, 1996-1999.
[21] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Quantifying the Complexity of Superscalar Processors," Technical report CS-TR-96-1308, Dept. of Computer Science, Univ. of Wisconsin, 1996.
[22] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources," *Proc. 34th Int'l Symp. Microarchitecture,* pp. 90-101, 2001.
[23] D. Ponomarev, G. Kucuk, and K. Ghose, "AccuPower: An Accurate Power Estimation Tool for Superscalar Microprocessors," *Proc. Fifth Design and Test in Europe Conf. (DATE '02),* pp. 124-129, 2002.
[24] T. Sherwood and B. Calder, "Time Varying Behavior of Programs," Technical Report No. CS99-630, Dept. of Computer Science and Eng., Univ. of California San Diego, Aug. 1999.
[25] D.W. Wall, "Limits on Instruction Level Parallelism," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Nov. 1991.
[26] K. Wilcox and S. Manne, "Alpha Processors: A History of Power Issues and a Look to the Future," *Cool-Chips Tutorial,* Nov. 1999.
[27] V. Zyuban and P. Kogge, "Optimization of High-Performance Superscalar Architectures for Energy Efficiency," *Proc. Int'l Symp. Low-Power Electronics and Design,* pp. 84-89, 2000.

**Dmitry Ponomarev** received the systems engineering degree from the Moscow State Institute of Electronics and Mathematics, Russia, in 1996 and the MS degree in computer and information science from the State University of New York (SUNY), Institute of Technology at Utica/Rome, in 1995, and the PhD degree in computer science from SUNY, Binghamton in 2003. He is currently an assistant professor in the Department of Computer Science, SUNY Binghamton. His research interests are in computer architecture, particularly in the optimizations of high-end microprocessors for energy efficiency. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

**Gurhan Kucuk** received the BS degree in computer engineering from Marmara University, Istanbul, Turkey, in 1995, the graduate degree in computer engineering from Yeditepe University, Istanbul, Turkey, in 1999, and the PhD degree in computer science from the State Uuniversity of New York, Binghamton in 2004. In August 2004, he joined the Department of Computer Engineering at Yeditepe University. His research interests include energy-efficient and high-performance microprocessor design, parallel computing and wireless sensor networks. He is a member of the IEEE and the IEEE Computer Society.

**Kanad Ghose** received the MS and PhD degrees in computer science from Iowa State University, Ames, in 1986 and 1988, respectively. In 1987, he joined the Department of Computer Science, State University of New York at Binghamton, where he is currently a professor and chair. His current research interests include microarchitectural and circuit-level techniques for power reduction, high-performance networking, and systems for handling and visualizing large data sets. He is a member of the IEEE, the IEEE Computer Society, the ACM, and Phi Kappa Phi.