Adaptive Reorder Buffers for SMT Processors

Joseph Sharkey Deniz Balkan Dmitry Ponomarev
Department of Computer Science,
State University of New York,
Binghamton, NY 13902-6000
{jsharke, dbalkan, dima }@cs.binghamton.edu

ABSTRACT

In SMT processors, the complex interplay between private and shared datapath resources needs to be considered in order to realize the full performance potential. In this paper, we show that blindly increasing the size of the per-thread reorder buffers to provide a larger number of in-flight instructions does not result in the expected performance gains but, quite in contrast, degrades the instruction throughput for virtually all multithreaded workloads. The reason for this performance loss is the excessive pressure on the shared datapath resources, especially the instruction scheduling logic. We propose intelligent mechanisms for dynamically adapting the number of reorder buffer entries allocated to each thread in an effort to avoid such allocations if they detrimentally impact the scheduler. We achieve this goal through categorizing the program execution into issue-bound and commit-bound phases and only performing the buffer allocations to the threads operating in commit-bound phases. Our adaptive technique achieves improvements of 21% in instruction throughput and 10% in the fairness metric compared to the best performing baseline configuration with static ROBs.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Other Architecture Styles –*Pipeline processors.*

General Terms: Performance, Design

Keywords: Simultaneous Multithreading, Reorder Buffer

1. INTRODUCTION

In the last two decades, a plethora of sophisticated microarchitectural techniques, mostly relying on various forms of speculation, have been proposed to extract the instruction-level parallelism (ILP) from single-threaded applications in dynamic out-of-order processors. Unfortunately, the surging complexity and power consumption associated with these mechanisms, as well as diminishing performance returns due to the growing processor-memory gap, make these solutions less and less attractive. Consequently, researchers have been exploring ways to improve the performance/complexity/power trade-offs in processor design by exploiting parallelism across multiple threads of control, or Thread-Level Parallelism (TLP).

Simultaneous Multithreading (SMT) is one processor design paradigm that exploits TLP. In an SMT model, multiple threads share the key datapath resources such as the issue queue (IQ), the pool of physical registers used for renaming, the execution units and the caches. In addition, each thread has a private load/store queue, rename table, program counter and return address stack. It is well established that such an organization provides a significant boost in instruction throughput compared to a superscalar machine with minimal area and complexity overheads [13,14].

One key datapath component that can significantly impact the performance of an SMT processor is the Reorder Buffer (ROB). The ROB is a circular FIFO queue that stores all in-flight instructions in program order and thus facilitates the recovery to a precise state following a branch misprediction, an interrupt or an exception. Instructions are dispatched into the ROB at the tail end (pointed to by the *ROB_tail* pointer) and they are committed from the head end (pointed to by the *ROB_head* pointer). The ROB is essentially implemented as a RAM structure with a number of read and write ports to support instruction dispatching (writes into the ROB) and commitment (reads from the ROB).

While logically each thread has its own private ROB to support the commitment of instructions independently of the progress made by other threads, the physical implementation can either be in the form of a number of private structures (one ROB per thread) or one shared structure with multiple head and tail pointers, one for each thread. Some academic authors [5,12] assume separate per-thread ROBs, while others [9,10] assume one large shared ROB. Even if the shared implementation is assumed, dedicated commit logic needs to be provided for each thread in order to avoid huge commit blockages. If such an organization is used for a *W*-way machine, then up to *W* oldest committable instructions across all threads are committed per cycle, regardless of their actual position in the ROB, noticeably complicating the commitment logic. Industrial SMT designs [16] typically use the shared ROB that is statically partitioned across the threads – that, again, is logically analogous to having separate per-thread ROBs. While the techniques proposed in this work are applicable to any ROB organization, we assume the use of private ROBs throughout the paper, although we also examine the performance implications of shared ROBs in Section 4.

It is well understood and accepted that larger ROBs generally result in higher performance on a single-threaded superscalar machine because a large window of instructions maintained in the out-of-order core allows for the exploitation of more ILP. However, in this paper we show that blindly increasing the sizes of the per-thread ROBs beyond a certain limit *consistently degrades* the performance of an SMT machine across virtually

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

all the multi-threaded workloads that we simulated. This phenomenon is a result of increased pressure on the shared SMT resources, such as the issue queue and the pool of physical registers.

We propose techniques to overcome these performance challenges by dynamically allocating a larger number of ROB entries to some threads only *if and when needed*, without detrimentally impacting the instruction scheduling logic. We achieve this goal through the following key observation. A thread benefits from a larger ROB and the scheduling of instructions from other threads is not impacted if the thread exhibits *commit-bound* behavior, i.e. in an average cycle, the majority of in-flight instructions from that thread have already begun (or completed) the execution. Most instructions from such a thread occupy an ROB entry for a very long time, but only use an IQ entry for a few cycles. In other words, the instructions from such threads issue shortly after being dispatched but then get delayed in the ROB before commitment. These threads require large ROBs, but can tolerate smaller IQs. It is therefore advantageous to provide a larger ROB to such a thread during the phase of execution when such behavior transpires. This occurs, for example, on an L2 cache miss (or some other long-latency event, in general), when the missing load blocks the thread's commitment and many load-independent instructions are also piled up in the ROB. Several recent studies [17, 18] showed that the number of load-dependent instructions is typically very small, much smaller than the number of independent instructions that can fit into the instruction window following an L2 miss.

On the other hand, if a thread exhibits *issue-bound* behavior, i.e. most of the instructions in the ROB are not issued, then allocating more ROB entries to the thread will simply increase the pressure on the shared resources, possibly restricting the availability of these resources to other threads, and thus degrading performance.

The specific contributions and the key results of this paper are:

- We show that the representative intervals of the SPEC 2000 benchmarks, as defined by the Simpoints tool [11], exhibit clear phases of issue-bound or commit-bound behavior, with some benchmarks being issue-bound, other benchmarks being commit-bound, and yet others changing their behavior throughout the execution.
- We propose a logically-partitioned ROB organization, which can be used to adapt the number of ROB entries allocated to each thread by simply controlling the range within which the head and tail pointers can advance, without making any other changes to the ROB structure.
- We investigate dynamic mechanisms to drive the allocation/deallocation decisions of entries within such logically-partitioned ROBs. Our goal is to provide larger ROBs to threads executing in commit-bound phases and limit the ROB allocations to threads executing in issue-bound phases.
- We show that the use of our adaptive techniques on a processor with 128-entry per-thread ROBs increases the throughput IPC by 54% and fairness by 29% compared to the static ROBs of similar size. Compared to the best-performing static configuration (ROBs with 48 entries), our techniques result in 21% improvement in IPC and 10% improvement in fairness. All of this is achieved with minimal complexity.
- We demonstrate that the performance of our techniques very closely approaches that of the machine with the infinite issue queues, effectively addressing the performance challenges associated with the ROB scalability on SMT. This result essentially obviates the need to consider more sophisticated ROB management techniques, such as a completely dynamic sharing of all available ROB partitions across all threads.

The rest of the paper is organized as follows. We review the related work in Section 2. Our simulation methodology is described in Section 3. Section 4 examines the reasons for the performance losses with larger ROB sizes on SMT. We categorize the execution of the SPEC benchmarks into commit-bound and issue-bound phases in Section 5. Section 6 presents our mechanisms for dynamic ROB adaptation. We present and discuss the results in Section 7, and offer our concluding remarks in Section 8.

2. RELATED WORK

The use of shared as well as partitioned resources in an SMT processor can be indirectly controlled by instruction fetching mechanisms. Various fetching policies have been proposed in the literature to provide the best supply of instruction mixes from multiple threads for building the most efficient execution schedules. The I-Count fetching policy [13] gives fetching priority to threads with fewer instructions in decode, rename and the IQ. The goal is to avoid clogging of the IQ with the instructions from one thread. Several optimizations of I-Count have also been proposed in an effort to avoid fetching the instructions that are likely to be stalled in the IQ for a large number of cycles. STALL [12] prevents the thread from fetching further instructions if it experienced an L2 cache miss. FLUSH [12] extends STALL by squashing the already dispatched instructions from such a thread, thus making the shared IQ resources available for the instructions from other threads. FLUSH++ [4] combines the benefits of STALL and FLUSH and uses the cache behavior of threads to dynamically switch between these two mechanisms. The Data Gating technique of [5] avoids fetching from threads that experience an L1 data miss.

In [3], a novel resource allocation policy (called DCRA) exercising a more fine grained dynamic control over shared SMT resources (such as the IQ and the register file) was proposed. DCRA first classifies the threads according to their demands for the resources and based on this classification determines how the resources should be distributed among the threads. In contrast to the previous methods that stall or flush threads which have cache misses, the technique of [3] actually attempts to help these threads by providing more resources to them (if such resources are available) to increase the memory-level parallelism by overlapping multiple cache misses. While providing benefits compared to the previously proposed fetching schemes, the technique of [3] requires a few additional counters and the logic to implement the resource sharing model. It was shown in [3], and corroborated by our analysis, that the DCRA method is generally superior to all previously proposed fetching policies. In this work, we use the DCRA mechanism as our baseline case for comparison and show that our techniques provide significant additional benefits on top of DCRA.

The effects of various resource partitioning schemes on the performance of SMT processor were examined in several works. In [10], a partitioned version of the oldest-first issue policy is proposed, where separate issue queues are used to buffer the instructions from different threads. In [9], the effects of partitioning the datapath resources, including the issue queues and reorder buffers, across multiple threads, were discussed. The authors of [9] compared the use of private ROBs with a structure that is shared by all threads, but that still allows the commitment of W oldest committable instructions (for a W-way machine), possibly belonging to different threads, to be performed in the same cycle. The main conclusion of [9] is that the statically-partitioned ROB results in performance advantages compared to the fully shared design for smaller ROB sizes (as sharing can easily monopolize the ROB by the instructions from one thread in this case). At larger ROB sizes, the performance of architectures with shared and private ROBs was found to be almost identical. Our studies showed similar trends, and some results are presented in Section 4.

The work of [15] explored dynamic resource allocation on SMT processors that preserve, as much as possible, the performance of a single "foreground" thread while still permitting other, "background" threads to share the resources. These low-priority transparent threads are suitable for performing non-critical computations and can be used, for example, as helper threads for prefetching. Our work, in contrast, focuses on the resource allocations among treads with equal priority levels.

3. METHODOLOGY

For estimating the performance impact of the schemes described in this paper, we used M-Sim [22] - a significantly modified version of the Simplescalar 3.0d simulator [1] that separately models pipeline structures such as the issue queue, re-order buffer, and physical register file, both for superscalar and SMT machines [13,14]. For the SMT machine, we assume the shared register file, execution units, caches, and issue queue [9]. As shown previously [9], and also observed in our simulations, a shared issue queue provides better performance than a private issue queue (by 8% for the 64-entry per-thread ROBs in our simulations) and we therefore assume this model through the paper. The simulator also supports speculative instruction scheduling [19] and models the "squash" recovery following a load-latency misprediction, as implemented in Alpha 21264. In the squash recovery model, all instructions that have issued but not yet begun execution at the time that a load-latency misprediction is detected are replayed. The details of the studied processor configuration are shown in Table 1. To fetch instructions from multiple threads, we use the DCRA policy [3], which was discussed in more detail in Section 2. In some cases, we also examine the I-Count [13] policy.

Table 1: Simulated processor configuration.

Parameter	Configuration
Machine width	8-wide fetch, 8-wide issue, 8-wide commit
Window size	64 entry IQ, 48 entry per-thread load/store queue, ROB as specified
Pipeline Depth	5 cycles fetch to dispatch, 3 cycles issue to execute
Function Units and Lat (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 4 Load/Store (2/1), 8 FP Add (2), 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Phys. Registers	300 integer + 300 floating point (including architectural registers)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 1 cycles hit time
L1 D–cache	64 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache unified	512 KB, 8–way set–associative, 128 byte line, 12 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Pred. Per Thread	4K entry gShare, 10-bit global history
Load-latency Predictor	4K entry bimodal predictor
Memory latency	300 cycles
TLB	64 entry (I), 128 entry (D), fully associative

We simulated the full set of 26 SPEC 2000 integer and floating point benchmarks [6], using the precompiled Alpha binaries available from the Simplescalar website [1]. We skipped the initialization part of each benchmark using the procedure prescribed by the Simpoints tool [11] and then simulated the execution of the following 100 million instructions. In SMT mode, we stopped the simulations when 100 million instructions from at least one of the threads committed.

Our multithreaded workloads contain a subset of all possible combinations of the simulated benchmarks. In selecting the multithreaded workloads, we first simulated all benchmarks in the single-threaded superscalar environment and used these results to classify them as low, medium, and high ILP, where the low ILP benchmarks are memory bound and the high ILP benchmarks are execution bound.

Table 2: Simulated multi-threaded workloads

Classification	Mix Name	Benchmarks
4 LOW ILP	Mix 1	*mcf, equake, art, lucas*
	Mix 2	*twolf, vpr, swim, parser*
4 MED ILP	Mix 3	*applu, ammp, mgrid, galgel*
	Mix 4	*gcc, bzip2, eon, apsi*
4 HIGH ILP	Mix 5	*facerec, crafty, perlbmk, gap*
	Mix 6	*wuwpise, gzip, vortex, mesa*
2 LOW ILP + 2 HI ILP	Mix 7	*mcf, equake, mesa, vortex*
	Mix 8	*parser, swim, crafty, perlbmk*
2 LOW ILP + 2 MED ILP	Mix 9	*art, lucas, galgel, gcc*
	Mix 10	*parser, swim, gcc, bzip2*
2 MED ILP + 2 HI ILP	Mix 11	*gzip, wupwise, fma3d, apsi*
	Mix 12	*vortex, mesa, mgrid, eon*

In total, we simulated 12 4-threaded workloads, two from each of the following six categories: 1) 4 low-ILP programs; 2) 4 medium-ILP programs; 3) 4 high-ILP programs; 4) 2 low-ILP and 2 high-ILP programs; 5) 2 medium-ILP and 2 high-ILP programs; 6) 2 medium-ILP and 2 low-ILP programs. All workloads are described in detail in Table 2.

We used several metrics for evaluating the performance of the multithreaded workloads throughout this paper. The first metric is the total instruction throughput in terms of commit IPC rate. However, this metric is biased towards the architectures that favor threads with high IPC at the expense of possibly hindering threads with low IPC [7]. Therefore, we also present the "fairness" metric of the harmonic mean of weighted IPCs [7], which takes into account individual per-thread performance. Throughout the rest of the paper, we present our performance results in terms of both throughput and fairness (often using separate graphs).

4. ROB SCALING: WHY CAN LARGER ROBS DECREASE THE PERFORMANCE OF SMT?

Figure 1 shows how the performance scales with the increase of the ROB size if only one thread is executed at a time. Specifically, when the ROB size is increased from 32 to 128 entries the performance increases by 39% and the IPC increases *monotonically* as a function of the ROB size.

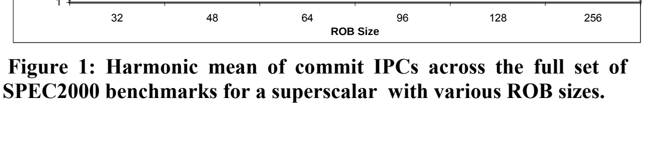

Figure 1: Harmonic mean of commit IPCs across the full set of SPEC2000 benchmarks for a superscalar with various ROB sizes.

Quite in contrast to the results presented in Figure 1, the simultaneous increase in the ROB size of all threads on a multithreaded machine results in some unexpected performance challenges due to the side-effects on shared SMT resources. These trends are depicted in Figure 2.

The bottom line on the graph in Figure 2 (labeled "Baseline") shows how the throughput of a 4-threaded SMT machine changes as the per-thread ROB size is increased from 32 entries to 256 entries. For these results, we assume private per-thread ROBs and the DCRA fetch policy proposed in [3]. Across the simulated multithreaded mixes of SPEC 2000 benchmarks, there is a 6% harmonic mean IPC loss as the per-thread ROB in enlarged from 32 to 64 entries and further 25% IPC loss as the ROB increases to 128 entries.

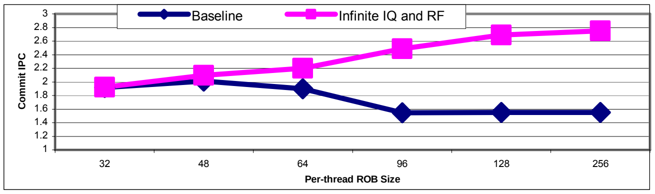

Figure 2: Commit IPCs for the baseline SMT machine (as described in Table 1) and the SMT machine with the infinite IQ and RF both using the DCRA fetch policy [3].

The performance degradations shown in Figure 2 stem from the fact that a simultaneous, across-the-board increase in the number of in-flight instructions from all threads results in elevated pressure on the shared datapath resources, such as the issue queue (IQ) and the register file (RF). For example, if a thread operates in a phase where instructions spend most of their lifetime waiting to be executed, then the increase of that thread's ROB will simply result in the placement of a larger number of instructions in the IQ for longer periods of time, thus denying the scarce issue resources to other threads and decreasing the overall issue efficiency and, consequently, performance. At the same time, had the IQ and the RF contention not been a problem, the larger per-thread ROBs would have provided a significant performance boost, as shown in the top line of Figure 2 (labeled "Infinite IQ and RF"), where the infinite IQ and RF were simulated. Notice that the sharpest performance drop in the baseline configuration comes with the per-thread ROB increase from 64 to 96 entries. This is because, at this point, the main performance bottleneck shifts from the ROB to the IQ.

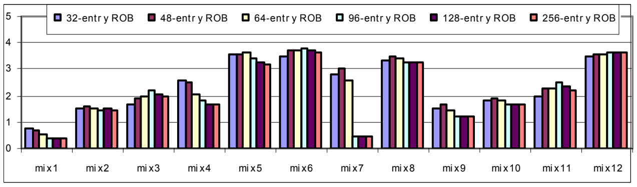

Figure 3: Per-benchmark commit IPCs for the baseline SMT machine with various sizes of the per-thread ROBs

Of course, the phenomenon depicted in Figure 2 can be avoided *at these ROB sizes* by implementing larger IQs and RFs. However, both of these components generally lie on the critical timing path [20, 21] and the schedulers are not easily pipelined without significant performance loss [20]. As the enlargements of these structures are likely to prolong the processor cycle time, it is important to investigate techniques that support large ROBs without relying on the increase of the IQ and the RF sizes to mitigate the upsetting trends demonstrated by Figure 2. Also, even if larger IQ and RF are used, the trends exhibited by Figure 2 will still manifest themselves at larger ROB sizes.

The behavior presented in Figure 2 is not unique to a few specific workloads, but is common to almost all simulated multithreaded mixes. Figure 3 presents the per-mix results pertaining to the commit IPCs as a function of the number of entries in the per-thread ROBs. While *mix 7* exhibits an especially significant drop in performance as the ROBs are increased beyond 64 entries, some performance degradations are experienced by almost all of the other workloads as the ROB size increases from 64 to 128 entries and sometimes even from 48 to 64 entries. For most workloads, there are still some performance gains when the ROB is increased from 32 to 48 entries, as the ROBs, but not the issue queue, represent a performance bottleneck in those configurations.

We now examine the impact of both the fetching policy and the use of shared versus private ROB organizations on the performance trends depicted in Figures 2 and 3. These comparisons are presented in Figure 4. The X-axis is labeled in terms of the total number of ROB entries. For the private ROBs, the size of the per-thread ROBs is determined by dividing this number by the number of threads (for example, 256 total ROB entries = 64-entry ROBs per thread * 4 threads).

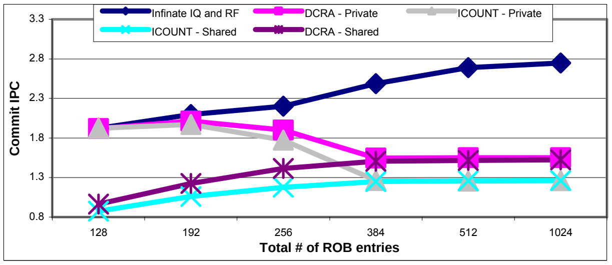

Figure 4: Commit IPCs for the baseline SMT machine with various ROB organizations (private versus shared) and various fetching policies.

The top line of the graph again depicts the performance of the machine with private ROBs and the infinite issue queue. The next two lines depict the performance of the private ROB organization using the DCRA [3] and the ICOUNT [14] fetching policies, respectively. As seen from the graph, the DCRA fetch policy outperforms the ICOUNT policy for all sizes of the private ROBs, by as much as 24% for 96-entry per-thread ROBs. The next two lines in the graph show the performance of the shared ROB organization using the DCRA and ICOUNT fetching policies. Once again, the DCRA fetch policy provides superior performance to ICOUNT – by as much as 21% and 20% for the shared ROB sizes of 256-entries and 384-entries, respectively. Notice that the performance increases monotonically as the size of the shared ROB is increased from 128 to 1024 entries, but that it remains significantly lower than that of the private ROBs; especially for sizes up to 384 entries, at which point the performance of the shared and private ROBs converges. Similar trends were presented in [9]. The best performance is obtained from the private per-thread ROBs with the DCRA fetch policy. Therefore, due to the space constraints and without loss of generality, we focus on this organization as the basis for our study. However, the techniques and statistics presented in the rest of the paper are certainly applicable to both ROB organizations and the various fetching policies.

To understand the source of the performance losses shown in Figures 2 and 3, we present some additional statistics in Figures 5, 6 and 7. Figure 5 depicts the average number of cycles that instructions spend in the IQ. On average, as the ROB size is increased from 32 to 256 entries, the time spent by the instructions in the IQ almost triples, because the uncontrolled increase in the number of allocated ROB entries creates situations where chains of instructions from one thread that depend on a long-latency event are placed in the IQ and reside there for a large number of cycles, denying other threads the opportunity to use these IQ entries. Had the ROB size not been increased, this situation would have been avoided because these problematic instructions would not have been allowed to enter the scheduling window due to the lack of space in that thread's ROB. Figure 6 presents similar results for the register file. As seen from the graph, the average time that a physical register remains allocated also increases significantly with the increase of the ROB size.

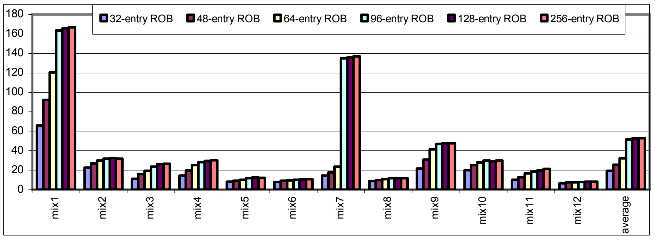

Figure 5: Average number of cycles spent in the issue queue by instructions for various sizes of the per-thread ROBs.

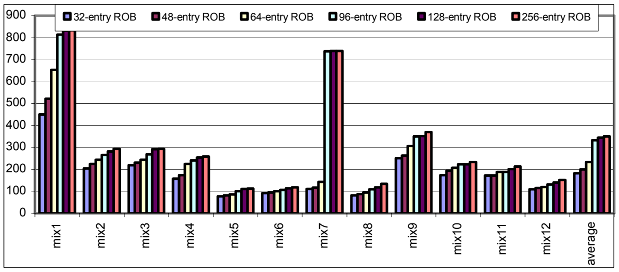

Figure 6: Average number of cycles for which a physical register remains allocated for various sizes of the per-thread ROBs.

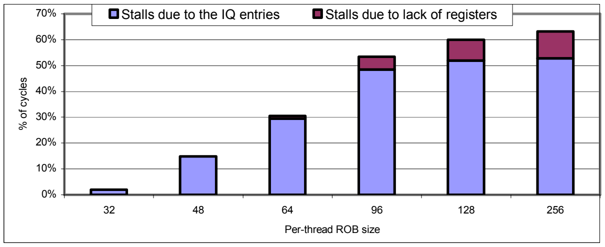

Figure 7: Average percentage of cycles where dispatch is stalled due to the lack of issue queue entry or a free physical register

As a consequence of the behavior presented in Figures 5 and 6, the percentage of cycles that instruction dispatch is stalled due to the lack of the IQ entries or free physical registers increases. Figure 7 shows how this percentage of stalled cycles changes with the increase of the ROB size. The bottom portion of the stacked bars shows the percentage of stalled cycles due to the absence of a free IQ entry, and the top portion of the stacked bars shows the *additional* percentage due to the absence of a free physical register. Overall, the percentage of stalled cycles increases from about 2% for 32-entry ROBs to more than 60% for 256-entry ROBs. This is the main reason behind the performance degradations.

The detailed results presented in Figures 5, 6 and 7 clearly show that it makes little sense to allocate more ROB entries to threads in an SMT machine, if such allocations increase the pressure on the shared resources. However, some threads, for which increasing the ROB size does not commensurately elevate the pressure on the shared resources, can benefit from such allocations. It is therefore important to consider techniques that adaptively allocate ROB entries to threads without creating contention for the use of shared resources.

5. CATEGORIZATION OF PROGRAM PHASES

In an effort to design such dynamic allocation algorithms, we first categorize the behavior of the full set of benchmarks from the SPEC 2000 suite, when running in single-threaded mode, into commit-bound and issue-bound phases. Figures 8, 9 and 10 depict some representative results, showing the varying behavior of different benchmarks.

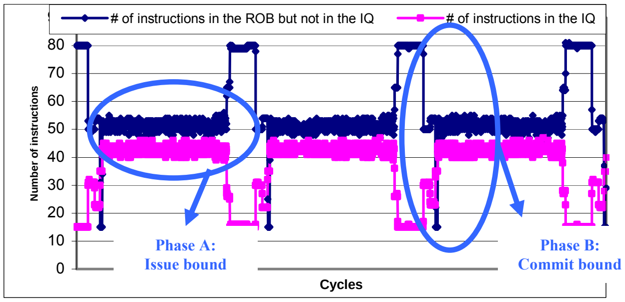

Figure 8: Phase behavior of the *equake* benchmark

Figure 8 shows the execution profile of the *equake* benchmark. Two lines are shown on this graph. The top line (shown in the darker shade) depicts the average number of instructions which have been issued but not yet committed, i.e. the instructions which are located in the ROB, but not in the issue queue. The bottom (lighter) line shows similar results for the average number of dispatched instructions that have not yet begun their execution, i.e. the instructions that reside in both the ROB and the IQ. Each point on the graph represents the average value of these metrics sampled every 100,000 cycles – the results are presented for the full 100 million instruction execution period as determined by Simpoints. For *equake*, there are distinct phases of execution, such that in some phases (for example, phase B circled in the figure) the number of issued instructions is much larger than the number of non-issued instructions, and in other phases (such as phase A) these numbers are close to each other. We refer to the phases that exhibit the behavior similar to that of phase B as *commit-bound* phases (the throughput is limited by commitment) and we refer to the phases that behave similar to phase A as *issue-bound* phases. In issue-bound phases, a significant percentage of in-flight instructions are present in the IQ, therefore increasing the ROB size in such situations is likely to result in additional pressure on the IQ.

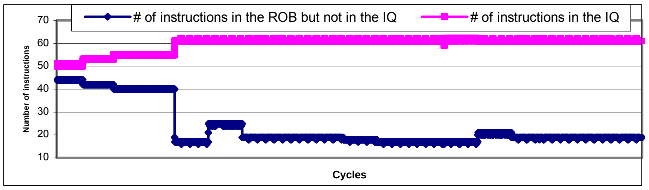

Figure 9: Phase behavior of the *lucas* benchmark.

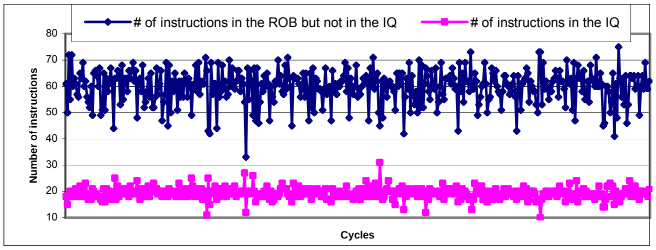

Figure 10: Phase behavior of the *vortex* benchmark.

Table 3: Phase classification of SPEC2K benchmarks

Classification	Benchmarks
Commit-bound	*ammp, applu, art, fma3d, galgel, gap, swim, vortex, wupwise*
Issue-bound	*crafty, eon, gzip, lucas, mcf, perlbmk, sixtrack, twolf*
Varying Behavior	*apsi, bzip2, equake, facerec, gcc, mesa, mgrid, parser, vpr*

Note that in the case of *equake*, the issue-bound and commit-bound phases interchange throughout the execution, although the duration of each such phase is quite long. For some other benchmarks, the entire simulated execution interval exhibits only one type of behavior, either issue-bound or commit-bound. For example, the entire execution of *lucas* (Figure 9), is characterized as an issue-bound phase. In fact, for *lucas* we observe that the number of instructions in the issue queue is actually significantly larger than the number of instructions that are in the ROB but not in the IQ. This is illustrated in the graph by the fact that the lighter-shaded line is above the darker-shaded line throughout the entire execution. In contrast, the *vortex* benchmark (Figure 10) exhibits commit-bound behavior throughout its execution, where the number of instructions in the issue queue represents just a small percentage of the total number of instructions in the ROB. In the picture, this is manifested by the fact that the darker line is always above the lighter line.

The trends demonstrated by Figures 8, 9 and 10 are quite representative of the behavior of the SPEC 2000 suite at large. Table 3 shows the classification of all the SPEC 2000 benchmarks, executed in a single-threaded mode, into three groups: commit-bound, issue-bound and the ones with varying behavior. Formally, a phase of execution was determined to be issue-bound if the number of non-issued instructions (those presented in both the ROB and the IQ) was greater than one third of the total number of instructions in the ROB. While the resource occupancies obviously change in an SMT mode, this characterization shows that the relative demands for the IQ and the ROB resources are generally very different both across and within the applications.

The existing fetch and resource allocation policies do not exploit the commit-bound or issue-bound behavior directly and therefore experience performance degradations with larger ROB sizes, as shown in Section 4. For example, DCRA distinguishes fast and slow threads indirectly, based solely on the presence of outstanding L1 cache misses: a thread with such a miss is considered as “slow”. However, not all L1 misses are created equal – some can be seamlessly hidden by the out-of-order execution mechanisms, others can result in significant pressure on the shared resources, and yet others can even miss into the lower levels of the memory hierarchy. In contrast, the adaptive ROB mechanism proposed in this paper directly classifies threads as “commit-bound” or “issue-bound”, providing a more comprehensive view of the resource needs of the individual threads. Our classification effectively encompasses all possible sources for slow or fast execution such as cache misses as well as long dependency chains of long-latency instructions. Furthermore, while DCRA only controls the use of *shared* resources, our technique effectively adds another dimension to SMT resource allocation by also dynamically adjusting the sizes of the *private* ROBs. In a way, adaptive ROBs provide additional level of control to correct the imbalances in the resource distribution created by DCRA, perhaps as a result of erroneous thread classification.

6. STRUCTURES AND ALGORITHMS FOR DYNAMIC ROB ADAPTATION

In this section, we describe the physical structures and algorithms for dynamic ROB adaptation that exploit the phase characteristics presented in the previous section to provide larger ROBs to threads in commit-bound phases and limit the number of ROB entries available to threads in issue-bound phases.

6.1 A Logically Partitioned ROB Organization

To support the dynamic allocation of ROB entries, we propose a *logically-partitioned* ROB organization, which simply limits the extent to which the *ROB_tail* and the *ROB_head* pointers can advance, *without making any other changes to the ROB structure*. While logically the ROB is divided into a number of partitions (each with multiple entries) and the reconfiguration decisions are made at the partition granularity, the physical structure of the ROB does not change at all. In contrast to the partitioning schemes that target power reduction by turning off the power supply to the deactivated partitions [8] (which requires a fairly substantial amount of additional circuitry), the partitioning that we propose is purely logical and its only goal is to control performance by limiting the advance of the ROB pointers. The advancement of the ROB pointers can simply be controlled by a single bit associated with an ROB entry at a partition boundary.

Figure 11 shows an example of using the logically-partitioned ROBs for a 3-threaded SMT processor. Each of the 3 threads has its own ROB which is divided into 4 partitions. The last entry of each partition has a *next_partition_allocated* bit associated with it, which simply indicates whether the next partition is in use or not. For an ROB with N partitions, N-1 such bits are used because the first partition is always allocated. If this bit is set, then the next ROB partition is also allocated for use by this thread. Whenever the *ROB_head* or *ROB_tail* pointer reaches the border of a partition where the *next_partition_allocated* bit is set, it just moves to the next partition when it advances. If this bit is not set, the next partition is not allocated for use, and the ROB pointers wrap to the beginning of the ROB (entry number 0) instead of advancing to the next partition. The figure shows the situation where the first thread is using all of its available ROB partitions. The second thread has only the first partition allocated for its use, so the rest of its three partitions are not allocated and they are idle. The third thread has three of its ROB partitions allocated to hold the in-flight instructions and the last partition is not allocated. Note that the allocated partitions are not necessarily holding useful instructions at the moment but they are available for use by the corresponding thread. The *next_partition_allocated* bits do not have to be stored within the ROB itself, but instead can be incorporated into the logic that controls the advancement of the ROB pointers. In the architectures which support the "walk-back" examination of ROB entries to handle branch mispredictions, such a traversal can be performed by storing the largest index of an allocated entry to ensure that consecutive entries can be examined when the ROB pointers wrap around.

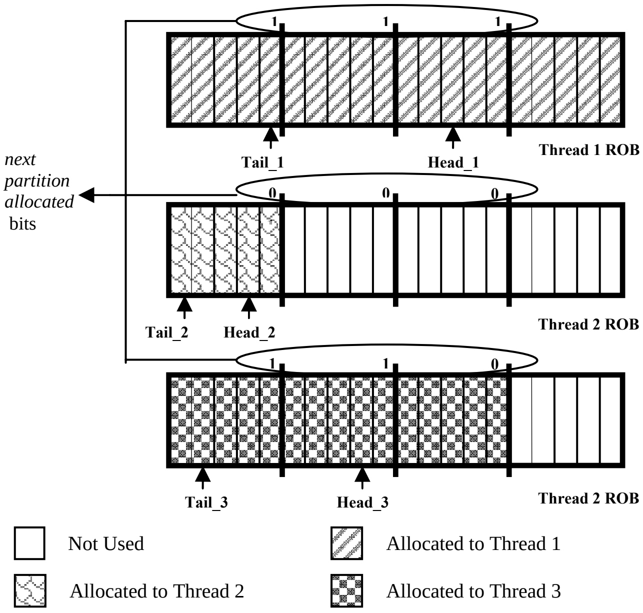

Figure 11: Logically partitioned ROBs.

In summary, almost no additional complexities are incurred as a result of the logical partitioning (except for the addition of a few bits). The unique feature of our dynamic adaptation framework is that it is sometimes advantageous to leave the partitions unused for performance reasons!

6.2. Mechanisms for Dynamic ROB Allocation

We now describe the algorithms to dynamically adapt the size of the ROB for each thread to reap the benefits of large instruction windows without clogging the shared issue queue. As explained earlier, our goal is only to make more ROB entries available to a thread if this will not commensurately increase the pressure on the shared resources such as the issue queue. To this end, we propose algorithms to dynamically allocate and deallocate the ROB partitions to each thread based on whether the thread executes in a commit-bound or issue-bound phase. Our technique works in two independent and cooperative phases – one for controlling allocations and one for controlling deallocations. These decisions are made simultaneously and independently for all threads. We describe these two mechanisms separately.

6.2.1 Controlling ROB Allocations

For each thread, we maintain two counters. The first counter – called *not_issued_count*, keeps the count of the number of instructions in the issue queue. The second counter – called *total_count* - keeps the count of the overall number of instructions residing in the ROB. The *not_issued_count* is incremented at the time of dispatching and is decremented when instructions issue or are flushed from the pipeline following a branch misprediction. The *total_count* is incremented when an instruction gets dispatched, and it is decremented when the instruction is committed or is flushed from the ROB as a result of a branch misprediction. The values stored in these counters are adjusted on a cycle-by-cycle basis.

The decision of whether to perform additional allocations of the ROB partitions to threads is made periodically. The duration between the two consecutive allocation decisions is called the *Evaluation_Period* (EP in the rest of the paper). At the end of every EP, the following actions take place for each thread to make a decision for allocating new ROB partitions:

1) The *issued_count* is computed as *(total_count - not_issued_count*). The *issued_count* refers to the number of instructions that reside in the ROB but not in the IQ. While it is possible to accumulate this information directly (without performing the subtraction), the logic needed to update this counter would be more complicated than simply maintaining the total number of in-flight instructions.

2) The average per-cycle values of *not_issued_count* and *issued_count* are computed by shifting the values stored in the corresponding counters by N positions to the right, where $N=log_2EP$. To make sure that such a shifting provides accurate average values across the EP, we limit the EP to be a power of 2.

3) The difference computed as (*not_issued_count – issued_count*) is then compared against the *allocation_threshold,* and if it is determined that (*not_issued_count – issued_count*) < *allocation_threshold* then a new free ROB partition is allocated to this thread. If this inequality does not hold, then no ROB allocations are performed and the thread continues to execute using its current ROB. The allocation threshold is empirically determined and we present our results across the range of various threshold values in the results section.

The intuition behind this algorithm is that by directly comparing the number of issued instructions with the number of non-issued instructions in a manner presented in this section, we can distinguish the issue-bound and commit-bound phases of execution. Specifically, if the *issued_count* is significantly larger than the *not_issued_count*, then the program executes in a commit-bound phase and additional allocations of the ROB partitions to this thread can be performed, as they are unlikely to impact the scheduling efficiency of other threads. However, if the *not_issued_count* is larger or about the same as the *issued_count*, then the program executes in an issue-bound phase and no more allocations to the ROB of this thread will be performed.

6.2.2 Controlling ROB Deallocations

We now describe the second component of our reconfiguration algorithm – the logic that controls the ROB deallocation decisions. As with allocations, deallocation decisions are made periodically, at the end of every EP. The specific actions involved in the deallocation algorithm are as follows:

1) At the end of every EP, the average per-cycle value of the *not_issued_count* is computed by shifting the counter value by N positions to the right, where $N=log_2EP$.

2) The shifted value of the *not_issued_count* is compared against the *deallocation_threshold*. If the counter value exceeds the threshold (*not_issued_count* > *deallocation_threshold*), then one of the ROB partitions belonging to this thread is deallocated.

We experimented with the various parameters to be used as deallocation thresholds. The intuition behind the deallocation approach is that when a thread monopolizes a disproportionate amount of the issue queue entries (perhaps as a result of some erroneous decisions made by the allocation phase), the deallocation logic corrects the situation by scaling down that thread's ROB. We present the evaluations across the range of the deallocation thresholds in the results section.

At the end of an EP, it may be the case that the values of the *issued_count* and *not_issued_count* are such that the allocation and deallocation conditions are *both* met. In this situation, the deallocation decision takes precedence and the allocation request is ignored. Note that with the appropriate values of the *allocation_threshold* and the *deallocation_threshold*, the situations where both conditions are met simultaneously can be minimized.

After the allocation/deallocation decisions are made, the actual allocations or deallocations happen after a number of cycles, when the ROB pointers are aligned accordingly. We refer the readers to [8] where the causes of possible delays are described in detail. However, since our evaluation periods are generally large (as shown in the results section), such delays have a negligible performance impact, but were nevertheless accounted for in the simulations. Finally, it is important to recall that the actions needed to accomplish the actual allocation/deallocation of the partitions only amount to the setting or resetting of one *next_partition_allocated* bit once the head and tail pointers are aligned appropriately.

7. RESULTS AND DISCUSSIONS

We begin by presenting the sensitivity analysis of our adaptation techniques to the ROB partition size. Figure 12 presents the performance results as the size of the ROB partition varies from 4-entries to 48-entries. We used the 96-entry per-thread ROBs for these experiments. As seen from the graph, the best performance was obtained when the ROB partition size was set at 8 entries. Notice that, from the implementation standpoint, the partition size is not much of a concern because the number of partitions merely determines the number of *next_partition_allocated* bits necessary to control the advancement of the ROB pointers. From the performance standpoint, if the partition size is too small, then the allocation/deallocation decisions have a relatively small impact because only a few entries are allocated/deallocated at a time. On the other hand, if the partition size is too large then the optimal point that provides a sufficient number of ROB entries to threads and at the same time avoids the scheduler starvation is more difficult to find. We experimented with various ROB sizes and different thresholds in our configuration algorithms and found that the 8-entry partitions represented the optimal design point in all cases. Therefore, in the rest of this section, the 8-entry partition size is used.

Figures 13 and 14 show the impact of our dynamic ROB adaptation techniques on the performance of a 4-threaded SMT machine in terms of the throughput IPC (Figure 13) and fairness metrics (Figure 14). The performance trends change significantly compared to the situation where the same static number of ROB entries is used by each thread (in the baseline case). The dynamic adaptation scheme provides measurable performance improvements as the ROBs increase from 48 to 64, to 96, and then to 128 entries. After that point, the performance flattens out as the other resources become the bottlenecks at these ROB sizes. Note that the trend of monotonically increasing performance at larger ROB sizes is re-established by the dynamic ROB adaptation techniques.

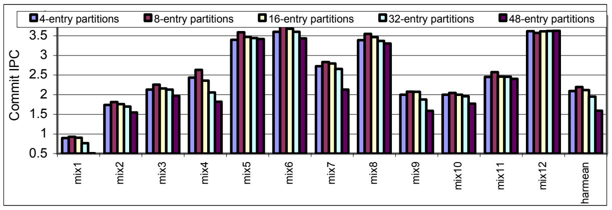

Figure 12: Performance of adaptive ROBs with 96-entry ROBs per thread for various partition sizes.

Compared to the baseline machine where a fixed number of ROB entries are statically assigned to each thread throughout its execution, our dynamic mechanism increases the IPC by 5% for 48-entry ROBs, 16% for 64-entry ROBs, 47% for 96-entry ROBs, and 53% for 128-entry ROBs. In terms of the fairness metric (harmonic mean of weighted IPCs [7]), these percentages are 2%, 7%, 25% and 29% (Figure 14). Compared to the best performing static ROB configuration on the baseline machine, which is the one with 48-entry ROBs per thread, our dynamically adaptable ROBs with 128-entries per thread increase the IPCs by 21% and fairness by 10% with respect to the harmonic mean across all simulated mixes. Recall that the increase of the per-thread ROB sizes from 48 to 128 entries in the baseline machine leads to a 25% IPC reduction and 15% reduction in fairness.

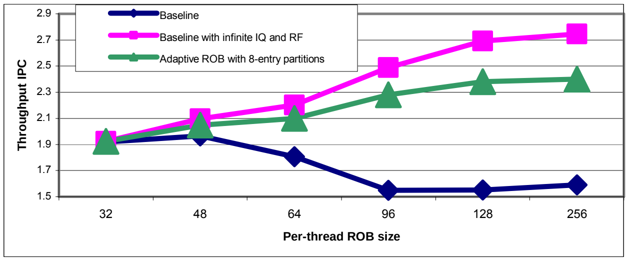

Figure 13: Harmonic mean throughput IPCs for the adaptive ROBs with various sizes of the per-thread ROBs for the configurations with 8-entry partitions.

Our experiments showed that the EP has little impact on the performance. Specifically, the performance is insensitive for the evaluation periods up to at least 128K cycles. This result correlates well with the data presented in Figures 8, 9, and 10, which show that the issue-bound and commit-bound phases of applications usually last for several 100,000 cycles at a time.

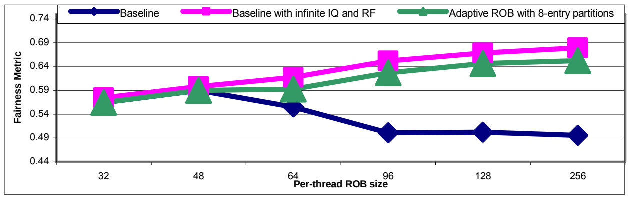

Figure 14: Harmonic mean results for the adaptive ROBs with various sizes of the per-thread ROBs for the configurations with 8-entry partitions presented in the form of the fairness metric (of harmonic mean of weighted IPCs).

We now examine the impact of various values of the *allocation_threshold* and the *deallocation_threshold* on the performance. We experimented with various mechanisms to set these thresholds, and our best results were achieved when the thresholds were expressed as a percentage of the ROB partition size. Figure 15 presents the sensitivity of our reconfiguration algorithm to the *allocation_threshold*. If we denote the number of entries in an ROB partition as λ, the threshold values considered as *allocation_thresholds* were in the range between λ/2 and –λ/2. The larger values of the *allocation_threshold* (towards the left side of the graph) result in more aggressive allocations, while lower values (towards the right side of the graph) result in less aggressive allocations.

For deallocations, the higher thresholds are more aggressive while lower thresholds are less aggressive. In the figure, the results are presented for two values of deallocation thresholds, which provided the best results. We also experimented with other values of deallocation thresholds and found them to be worst than the best results presented here. As seen from the graph, the scheme using 2*λ as the *deallocation_threshold* outperforms the datapath with 3*λ as the threshold in general for nearly all values of the *allocation_threshold* examined.

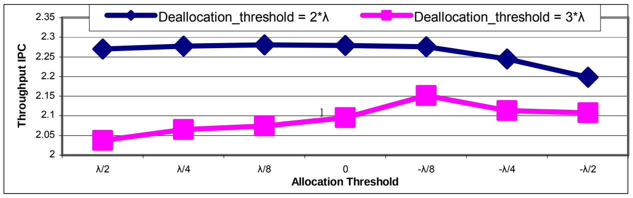

Figure 15: Performance Sensitivity to Allocation and Deallocation Thresholds Presented as Harmonic Mean of Throughput IPC. λ is the ROB partition size.

In general, low values of the *allocation_theshold* (including the negative values) provide lower performance because they are not aggressive enough to allocate a sufficient amount of ROB entries to threads that could make use of them. On the other hand, a very large *allocation_threshold* is too aggressive and thus allows issue-bound threads to further clog the issue queue (although the effects of this are somewhat mitigated by an appropriately selected *deallocation_threshold*). The optimal configuration lies somewhere between these extremes and, in our framework, is obtained with the *deallocation_threshold* of 2*λ and the *allocation_threshold* of λ/8.

Having determined the optimal thresholds, we now show some performance aspects of our reconfiguration algorithms. Table 4 presents the detailed statistics for two of the simulated multithreaded mixes – *mix 1* and *mix 10*. The details for other mixes are not presented due to the space constraints, but similar trends were observed for those workloads. For each thread, the table shows the average number of allocated ROB partitions, the distribution of the evaluation periods according to the type of reconfiguration decision made, the average number of instructions in the IQ, the average number of instructions which are in the ROB but not in the IQ, and the classification of the thread. The results generally demonstrate the effectiveness of our adaptive techniques. For example, consider mix 1. The *mcf* benchmark (classified as heavily issue-bound) has only 2 ROB partitions allocated to it on the average, while *art* (classified as heavily commit-bound) has its full ROB allocated throughout the run (12*8 = 96). In fact, *art* is so consistently commit-bound, that our algorithm quickly allocates all 12 available partitions to it and never deallocates them again.

The behavior of *equake*, on the other hand, varies between commit-bound and issue-bound phases, and on the average 9 ROB partitions are allocated. *Lucas* is issue-bound, but the degree of boundedness is less than in the case of *mcf*, so a larger number of ROB partitions (6 on the average) are allocated to *lucas*. Overall, each benchmark in the mix gets a different number of ROB entries according to its characteristics. We can also see that for some benchmarks (*art*, *swim*), the number of allocated ROB partitions is stable, while for others (*lucas*, *parser*) relatively frequent allocation/deallocation decisions are made.

While the goal of this paper is to improve the performance scalability of the ROB, similar mechanisms can be used to reduce dynamic and leakage power. If a more complicated ROB partitioning circuitry can be deployed [8], then the unused partitions can be temporarily put into a stand-by mode, thus reducing the leakage dissipations. Furthermore, the dynamic dissipations can be reduced by using bitline segmentation [8]. As demonstrated in Table 4, many ROB partitions are in the deallocated state at any given time and remain so for a large number of cycles (as the commit-bound / issue-bound phases last several 100,000 cycles as shown in Figures 8, 9 and 10). While we do not present a full detailed power evaluation of such power reduction techniques, a first-order approximation of power-saving potential can be determined based on the number of deallocated (inactive) ROB partitions and assuming that the extent of power savings is proportional to the fraction of partitions that are deallocated. During an average cycle, 32%, 25%, and 11% of the ROB partitions are in the deallocated state for the per-thread ROB sizes of 128-entries, 96-entries, and 48-entries, respectively. Finally note that if the optimization process is performed in the power/performance domain, then the optimal thresholds used by the adaptive ROB algorithms presented in this paper are likely to change.

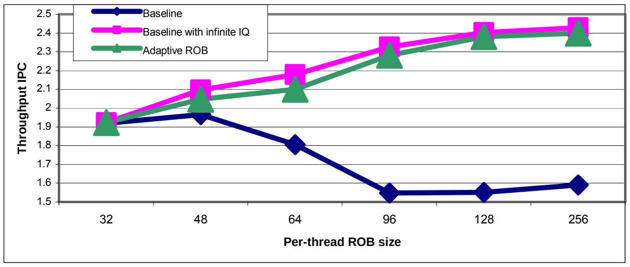

Figure 16: Performance Comparison of Various Configurations Presented as Harmonic Mean of Throughput IPC.

Finally, Figure 16 compares the performance of the dynamic ROB allocation schemes presented in this paper against the performance of the static per-thread ROB (baseline machine), the baseline machine with an infinite IQ but limited number of physical registers (as per Table 1). As seen from the graph, our mechanisms essentially provide the same performance as that of the baseline machine with the infinite number of issue queue entries. This shows that the use of our techniques completely alleviates the problem of clogging in the shared issue queue and avoids the performance degradations incurred by increasing the ROB sizes. This is because the case where the shared resources are idealized servers as an upper bound for the performance that can be achieved by dynamic ROB adaptation. Indeed, with infinite shared resources, dynamic ROB adaptation can only degrade performance compared to statically sized ROBs.

Table 4: Detailed performance statistics for representitive mixes.

		Avg # of partitions allocated	% of evaluation periods where			Avg # of instructions in the IQ	Avg # of instructions not in the IQ	Categorization from Table 3
			allocations are made	deallocations are made	no changes are made			
mix1	*mcf*	2	5%	5%	90%	9	7	*issue-bound*
	equake	9	15%	15%	70%	23	41	*varying behavior*
	art	12	0%	0%	100%	12	61	*commit-bound*
	lucas	6	33%	33%	34%	15	14	*issue-bound*
mix10	*parser*	9	23%	23%	54%	19	32	*varying behavior*
	swim	12	0%	0%	100%	3	69	*commit-bound*
	gcc	11	17%	17%	66%	22	34	*varying behavior*
	bzip2	5	21%	21%	58%	13	13	*varying behavior*

Another implication of these results is that a more complicated ROB partitioning and allocation scheme, where the partitions can be organized in a global shared pool to be assigned across all threads dynamically is not needed, as there is no further performance benefit that can be reaped from such a scheme. Similar conclusions can be arrived at if the fairness metric is considered (fairness results are not presented due to space constraints).

8. CONCLUDING REMARKS

Complex interactions between the shared and private per-thread resources need to be considered in order to understand the nuances of SMT architectures and realize the full performance potential of multithreading. One interesting and somewhat unexpected phenomenon is that across-the-board increase in the size of per-thread reorder buffers often decreases the instruction throughput on SMT due to the excessive pressure on the shared SMT resources such as the issue queue and the register file. In this paper, we proposed mechanisms and the underlying ROB organization to dynamically adapt the number of ROB entries allocated to threads only when such adaptations do not result in increased pressure on the shared datapath resources. To this end, we characterized the execution phases of the SPEC 2000 benchmarks into commit-bound and issue-bound and designed the algorithms to allocate more ROB entries to threads executing in commit-bound phases and limit the ROB allocations to threads in issue-bound phases.

Our results indicate that such dynamic adaptation of the ROBs results in significant increases on top of the DCRA resource allocation policy in terms of both throughput (54% compared to similarly-sized static ROBs and 21% compared to the best-performing static configuration) and fairness (29% and 10% respectively). These gains are achieved with very little hardware overhead and simple reconfiguration algorithms. We also demonstrated that the performance of adaptive ROBs approaches that of the datapath with an infinite issue queue, thus completely eliminating the size-effects of ROB scaling on the shared issue queue and obviating the need for more complex ROB management mechanisms.

9. REFERENCES

[1] D. Burger, T. Austin. "The SimpleScalar tool set: Version 2.0." Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[2] A. Buyuktosunoglu, et al. "A Circuit-Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors." in Proc of Great Lakes Symposium on VLIS, 2001.

[3] F. Cazorla, et al. "Dynamically Controlled Resource Allocation in SMT Processors." in Proc Int'l Symp. on Microarchitecture, 2004.

[4] F. Cazorla, et al. "Improving Memory Latency Aware Fetch Policies for SMT Processors." in Proc International Symposium on High Performance Computing, 2003.

[5] A. El-Moursy, D.Albonesi. "Front-End Policies for Improved Issue Efficiency in SMT Processors." in Proc. HPCA, 2003.

[6] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", IEEE Computer, 33(7):28-35, July 2000.

[7] K. Luo, et al. "Balancing Throughput and Fairness in SMT Processors." in Proc ISPASS, 2001.

[8] D. Ponomarev, G.Kucuk, K.Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources." in Proc. International Symposium on Microarchitecute (MICRO), 2001.

[9] S. Raasch, S. Reinhardt, "The Impact of Resource Partitioning on SMT Processors." in Proc. PACT, 2003.

[10] B. Robatmili et al. "Thread-Sensitive Instruction Issue for SMT Processors." Computer Architecture News, 2004.

[11] T. Sherwood, et al. "Automatically Characterizing Large Scale Program Behavior." Proc. ASPLOS, 2002.

[12] D. Tullsen, et al. "Handling Long-Latency Loads in a Simultaneous Multi-threaded Processor." in Proc of International Symposium on Microarchtiecture, 2001.

[13] D. Tullsen, et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." in Proc International Symposium on Computer Architecture, 1996.

[14] D. Tullsen, et al. "Simultaneous Multithreading: Maximizing on-chip Parallelism." , Int'l Symp. on Computer Architecture, 1995.

[15] G. Dorai, et al., "Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance", Int'l Conference on Parallel Architectures and Compilation Techniques, 2002.

[16] D. Marr, et al, "Hyperthreading Technology Architecture and Microarchitecture", Intel Tech. Journal, vol. 6, No.1, Feb. 2002.

[17] S. Srinivasan et al, "Continual Flow Pipelines", in Proceedings of ASPLOS, 2004.

[18] S. Sarangi, et al, "Re-Slice: Selective Re-execution of Long-Retired Misspeculated Instructions Using Forward Slicing", in 38th International Symposium on Microarchitecture, 2005.

[19] I. Kim, M. Lipasti, "Understanding Scheduling Replay Schemes", Int'l Symp. High Perf. Computer Architecture, 2004.

[20] J. Stark, et al., "On Pipelining Dynamic Instruction Scheduling Logic", in Proc. of MICRO, 2000.

[21] S. Palacharla, et al., "Complexity-Effective Superscalar Processors", in Proc. of the Int'l Symp. On Computer Architecture (ISCA), 1997.

[22] J. Sharkey, "M-Sim: A Flexible, Multi-threaded Simulation Environment." Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005. http://www.cs.binghamton.edu/~jsharke/m-sim