Two-Level Reorder Buffers: Accelerating Memory-Bound Applications on SMT Architectures

Jason Loew and Dmitry Ponomarev
Department of Computer Science
State University of New York at Binghamton
{jloew, dima}@cs.binghamton.edu

Abstract

We propose a low complexity mechanism for accelerating memory-bound threads on SMT processors without adversely impacting the performance of other concurrently running applications. The main idea is to provide a two-level organization of the Reorder Buffer (ROB), where the first level is comprised of small private per-thread ROBs which are used in the normal course of execution in the absence of last level cache misses. The second ROB level is a much larger storage that can be used on demand by threads experiencing last level cache misses. The key feature of our scheme is that the allocation of the second-level ROB partition occurs to a thread experiencing a miss into the last level cache only if the number of instructions dependent on the missing load is below a predetermined threshold. We introduce a novel low-complexity mechanism to count the number of load-dependent instructions and propose two schemes for allocating second level ROB: predictive and reactive. Our results demonstrate about 30% improvement over DCRA resource distribution mechanism in terms of "harmonic mean of weighted IPCs" metric.

1. Introduction

A Simultaneous Multithreading (SMT) is a processor design paradigm that increases the throughput of a superscalar processor in an area-efficient manner by executing multiple threads concurrently using the same out-of-order processing core with minimal datapath resource replication. 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 its own 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 overhead [13,14].

One obvious way to increase the performance of superscalar and SMT processors is to provide sufficient on-chip resources to support large instruction windows for exploiting available Instruction-Level Parallelism (ILP). The key on-chip resources defining the processing capabilities of the out-of-order instruction windows are the issue queue, the register file, the reorder buffers and the load/store queues. Of these resources, the Reorder Buffer (ROB) is perhaps the easiest to physically scale, especially in the architectures where the speculatively produced results are not temporarily kept in the ROB, but are instead directly written into the register file. This approach is exemplified by Intel's Hyperthreading technology, which offers SMT support on top of Pentium 4 microarchitecture. In these designs, the ROB just represents a collection of program counter values and a few service bits to maintain the program order of dispatched instructions and guarantee that the instructions are committed in this exact order. The ROB accesses in the course of normal execution simply amount to the establishment of entries at the time of instruction dispatch and the deallocation of entries at the time of commitment, with some additional manipulations on branch mispredictions, exceptions and interrupts. The ROB structure implemented in this fashion does not involve any associative addressing, and can also enable simplified layouts of the read/write ports, as the read and write accesses performed in a cycle during dispatch and committment activities always occur to the neighboring ROB entries. For these reasons, it is relatively easier to physically scale the ROB to larger sizes compared to the other principal datapath resources (such as the issue queue and the load-store queue) that rely heavily on associative addressing mechanisms.

Despite the relative ease of providing larger ROBs to each thread, previous work had demonstrated that blindly increasing the ROB sizes for all threads beyond a certain limit consistently degrades the SMT processor performance for virtually all multithreaded workloads, as the shared resources can be monopolized by individual threads with large private ROBs thus inhibiting the progress of other threads. This interesting phenomenon was thoroughly described in previous literature [23]. Our experiments also demonstrated that increasing the size of private ROBs beyond a certain threshold negatively impacts the performance, even if advanced resource distribution policies, such as DCRA [3], are used. As explained in [23], the performance degradations 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).

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.24

To address the performance challenges associated with the ROB scalability on SMT processors, the study of [23] proposed to classify the execution intervals into commit-bound and issue-bound phases and only allocate larger ROBs to threads operating in commit-bound phases (hoping that such allocations will not increase the pressure on the shared issue logic). While measurable speedups have been reported in [23] with respect to the state-of-the-art DCRA resource allocation mechanism, the technique is fairly complicated as the phase classification is performed continuously in the course of execution and the ROB entries are allocated and deallocated at the granularity of relatively small partitions. Furthermore, the extent of ROB allocations in [23] is limited to the physical dimension of each thread's ROB, which is not sufficient to cover long memory latencies typical of today's systems.

In this paper, we propose a simpler and arguably more efficient mechanism for addressing this problem. Specifically, we argue that in the absence of last level cache misses (L2 cache misses in our configuration), phase classification may be an overkill and a set of fairly small private ROBs allocated to each thread is sufficient for sustaining high overall SMT throughput. At the same time, it is beneficial for performance to allocate extremely large ROBs to threads experiencing L2 cache misses, but only if the number of instructions dependent on the missing LOAD is small. In this case, additional instructions dispatched in the shadow of an L2 cache miss will not exercise significant additional pressure on the shared issue logic (i.e. they will be allocated an issue queue slot, will be rapidly issued, and then wait for commitment in the ROB). Consequently, *memory-bound applications experiencing frequent last level cache misses can be accelerated without adversely impacting the scheduling of other concurrently running threads* and without denying scarce issue queue resources to those threads. Of course, the register entries and load-store queue entries will still be occupied by a thread missing in the last level cache even of the number of dependent instructions is small, but this has smaller performance impact for the following reasons. Only the load and store instructions are allocated entries in the load/store queue, and therefore the size of the load and store queue is expected to be a lesser bottleneck than the size of the issue queue. On the register file side, as no associative addressing/searching is involved in accessing the register file, it is easier to implement larger register files, possibly at the expense of pipelining the access over several cycles. Furthermore, several alternative techniques have been proposed to increase the efficiency of using registers on superscalar and SMT processors [24]. One of these techniques – L2-miss-driven early register deallocation for SMT [24] – can be easily synergized with the mechanisms proposed in this paper. In any case, without loss of generality in this paper we are interested in limiting the negative impact on the issue logic, and additional research (which is beyond the scope of this paper) is needed to alleviate the bottlenecks associated with other resources.

To implement the functionality described in the previous paragraph, we propose a 2-level ROB architecture, where the ROB storage is organized in two levels. The first level is comprised of small private per-thread ROBs, which are used in the normal course of execution in the absence of cache misses. The second ROB level is a much larger storage that can be used by the threads experiencing last level cache misses on demand, but only by one thread at a time. Physically, the second level can be either a separately located ROB partition, or it can be comprised of the sub-portions of individual threads' traditional ROBs. We propose two schemes for allocating second-level ROB partition – reactive and predictive. In the reactive technique, the number of load-dependent instructions is actually counted after the miss is detected, while in the predictive technique the number of dependents is predicted and later verified. Both schemes rely on a novel light-weight mechanism for counting the dependent instructions, which does not involve propagating the register tags.

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 describes the details of our proposed techniques. We present and discuss the results in Section 5. Finally, our concluding remarks are offered in Section 6.

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 ICOUNT fetching policy [13] gives fetching priority to threads with fewer instructions in decode, rename and the issue queue. The goal is to avoid clogging of the IQ with instructions from a single thread. Several optimizations of ICOUNT 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.

Another approach to optimizing instruction fetching on SMT processors is to exploit Memory-Level Parallelism (MLP) and make fetch policies themselves MLP-aware. One recently proposed technique [25] predicts the degree of MLP for each long-latency load, stops further fetches, and distributes shared resources to other threads unless overlapped loads are predicted to occur. Just like the scheme of [23] described previously in Section 1, this technique cannot proceed past the boundary of a single ROB, and therefore is limited in its MLP exploitation capabilities. The ROB adaptation mechanism proposed in this paper, on the other hand, can provide much larger scope for MLP exploitation and has a potential to significantly improve

performance. In this sense, it can even work synergistically with the technique of [25], although the question of what needs to be addressed first – degree of MLP or degree of dependence – needs further attention and is beyond the scope of this paper.

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 resource demands and based on this classification determines how the resources should be distributed amongst 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 processors 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.

3. Simulation Methodology

In order to evaluate the impact of the proposed schemes, 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]. 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 ICOUNT [13] policy. Using precompiled Alpha binaries (available from the Simplescalar website [1]), we simulated the full set of SPEC 2000 [6] integer and floating point benchmarks. Execution is simulated for the 100 million instructions as defined by the Simpoints tool. With multithreaded workloads, simulations were stopped after 100 millions instructions from any thread had committed.

Table 1: Configuration of the Simulation Environment

Parameter	Configuration
Machine width	8-wide fetch, 8-wide issue, 8-wide commit
Window size	**Per Thread**: 32 entry 1st level ROB, 48 entry LSQ**; Shared**: 64 entry IQ
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)
Physical Registers	224 integer + 224 floating-point physical registers
L1 I–cache	64 KB, 2–way set–associative, 64 byte line, 1 cycle hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 1 cycle hit time
L2 Cache unified	Unified: 2 MB, 8–way set–associative, 128 byte line, 10 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Predictor	2K entry gShare with 10-bit global history per thread
Load-Hit Predictor	2-bit bimodal: 1k entries, 8-bit global history per thread
Fetch Policy	ICOUNT 2.8, DCRA
Memory	64 bit wide, 500 cycle first chunk access, 2 cycle interchunk access

Table 2: Simulated Benchmark Mixes

Classification	Mix Name	Benchmarks
4 Low IPC	Mix 1	*ammp, art, mgrid, apsi*
	Mix 2	*art, mgrid, apsi, parser*
	Mix 3	*ammp, mgrid, apsi, parser*
3 Low IPC + 1 Mid IPC	Mix 4	*art, mgrid, apsi, vortex*
	Mix 5	*ammp, apsi, parser, crafty*
	Mix 6	*art, apsi, parser, gap*
2 Low IPC + 2 Mid IPC	Mix 7	*ammp, apsi, vortex, eon*
	Mix 8	*art, parser, vpr, gzip*
	Mix 9	*mgrid, parser, perlbmk, mcf*
4 High IPC	Mix 10	*lucas, twolf, bzip2, wupwise*
	Mix 11	*equake, mesa, swim, twolf*

The simulated workloads are derived from the subsets 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. Combining the benchmarks with various characteristics is different ways, we generated 11 4-threaded workloads. Since the goal of our work is to optimize the memory-bound applications, most of the simulated workloads included several low-ILP mixes. The workloads are shown in Table 1.

As performance metrics, we used the traditional overall throughput as well as the "fair throughput" (FT) metric, originally proposed in [7], which is calculated as the harmonic mean of weighted IPCs of the individual threads constituting a multithreaded workload. Weighted IPC of a thread is its relative slowdown in the multithreaded environment compared to its execution in a single-threaded situation. Unlike the traditional throughput IPC metric, FT metric is NOT biased towards the architectures that favor threads with high IPC at the expense of possibly hindering threads with low IPC [7]. Thus, it represents a fair measure of the total throughput, by combining performance and fairness considerations. (Note that in the original work of [7] the FT metric was simply called "fairness").

4. Two-level ROB Architecture

The key idea behind the two-level ROB architecture is to organize the ROB as a 2-level structure. The first level is comprised of relatively small traditional per-thread ROBs sufficient to sustain high overall throughput in the absence of long-latency cache misses. In our evaluations, we assume that each thread has a first-level ROB of 32 entries. In addition, a much larger second level of the ROB is also maintained, which is shared among all threads. Furthermore, the ROB entries comprising the second level can only be allocated as a unit to one thread at a time. Unless this storage is relinquished by a thread it was allocated to, no other thread is allowed to make use of it.

While the specific implementation details of a 2-level ROB structure are not central to the ideas presented in this paper, several physical realizations are possible. The second level of the ROB storage can either be implemented as a separate centralized structure, or it can be composed of the portions of larger private per-thread ROBs, viewed as an aggregate unit. For example, the physical implementation of the two-level ROB can be in the form of large private per-thread ROBs, say 128 entries per thread. Of these, the first 32 entries are used as private first-level ROB and the remaining 96 entries are used as second level ROB. In this case, the second ROB level will have a total of 384 (96*4) entries, allocated as an atomic unit.

As discussed previously, providing large ROBs to all threads at all times can be detrimental to the SMT performance, because of the increased pressure on the shared resources such as the issue queue and the register file. Instead, in the course of normal CPU-bound operations, the two-level ROB mechanism furnishes each thread with a small first level ROB. The second ROB level is only allocated to threads that experience L2 cache misses to cover long memory latencies, and such allocations are only performed in cases when the number of instructions dependent on the missing load is below a predetermined threshold. In the rest of the paper we refer to the number of instructions dependent on a LOAD as the LOAD's *Degree of Dependence* (DoD). If the LOAD instruction results in a miss in the L2 cache and its DoD metric is small (say, smaller than the quarter of the issue queue size for a 4-way SMT machine), then allocating the second-level ROB to this thread will result in sustained instruction execution while the cache miss is being serviced *without* monopolizing the shared issue queue nor detrimentally impacting the performance of other simultaneously running threads. The identification and exploitation of such performance-boosting opportunities is the key contribution of this paper.

Assuming that the allocation of large ROB storage to a thread experiencing an L2 cache miss does not adversely impact the use of shared resources by injecting a significant number of load-dependent instructions into the pipeline, such allocation results in two key performance advantages: 1) after the load miss is serviced, a high commit rate from this thread can be sustained because most of the instructions that were dispatched after the load have already executed and are ready to commit, and 2) subsequent cache misses can be discovered earlier resulting in an overlapping of multiple cache misses and exploitation of Memory-Level Parallelism (MLP). The key to the success of this mechanism is to ensure that such allocations do not adversely impact other threads. In this paper, we propose a new mechanism for eliminating the negative side-effects of large ROB allocations, which relies on tracking or predicting the DoD metric for each load instruction missing into the cache and making the ROB allocations based on this information. Specifically, we propose two general approaches – reactive and predictive - for controlling second level ROB allocations. We describe these approaches, along with the design trade-offs, in the rest of this section.

4.1 Reactive DoD-based Allocation of Second Level ROB

Our first approach to allocating second level ROB partition is purely reactive in nature. In this scheme, after a long latency load miss is detected, the actual number of load-dependent instructions that are already in the pipeline is counted and if this number is below a predetermined threshold, then a second-level ROB allocation to this thread is performed, provided that this second level partition is free. Two main issues arise in this scheme: how are the load-dependent instructions counted and when exactly such counting occurs.

A brute force approach to counting dependent instructions is to propagate the dependence information through a series of additional register tag broadcasts and matching operations, either on demand after the miss occurs or proactively in the course of register renaming. In either case, such additional tag propagation would be expensive and complicated. Instead, we use a different approach, which provides near-perfect dependent counts at extremely low complexity. Specifically, we simply count the number of instructions in the (first-level) ROB following the missing LOAD that have not yet executed. We assume that each ROB entry has an additional bit indicating the result validity and that this bit is set when the corresponding instruction completes execution. Typical ROBs already have such functionality to support precise interrupts, recover from branch misspeculations and identify instruction's eligibility for commitment [24]. A

simple 5-bit counter (for 32-entry first-level ROB) is maintained and it is incremented every time that an ROB entry holding a "not-yet-executed" instruction is encountered. The counter's value is checked after the examination of all "result valid" bits in the ROB is completed and if the value of the counter is below a predetermined threshold, then the allocation of a second level ROB is initiated. This dependence counting mechanism assumes that every unexecuted instruction is dependent on the load, so it may not represent the exact count, but only serves as an approximation. The accuracy of this counting mechanism depends on the timing of counting initiation with respect to the timing of the cache miss detection.

If the counting is initiated shortly after the miss is detected, then the likelihood of having unexecuted instructions which are load-independent increases, as the dependency information is propagated through other dependency chains as instructions are executed. Assuming that the delays due to the dependencies, through load-unrelated dependency chains, are much shorter than the delays due to the dependency on the L2 cache miss, if sufficient time is allowed between the detection of the miss and the initiation of the counting, it is expected that all unexecuted instructions will be load-dependent and the counting will be accurate. However, longer delays between the miss detection and the counting initiation may limit the efficiency of the 2-Level ROB technique as the larger ROB will be exploited for a smaller portion of the miss service duration. This is a fundamental trade-off of the reactive scheme. In the results section, we explore various counting initiation timings and analyze the trade-offs between the counting accuracy and performance potential. Another tweak is to initiate the counting only after the missing load becomes the oldest instruction in the ROB from that thread, or whenever the predetermined number of cycles since the miss discovery has elapsed *and* the missing load is the oldest instruction in the ROB.

4.2 Predictive Allocation of Second Level ROB

One potential limitation of reactive allocation scheme described in Section 4.1 is the delay between the discovery of a cache miss and the actual ROB allocation – this can reduce the opportunity for performance improvement. To perform additional ROB allocations faster, we also explore a scheme that relies on predicting the DoD metric for each load instruction and performing the allocation of the second level ROB partition if the DoD is predicted to be below a threshold.

The effectiveness of this technique obviously depends on the prediction accuracy. Fortunately, for the same control flow path the number of load-dependent instructions does not change for all dynamic instances of the LOAD. Therefore, the only variation in the number of load-dependent instructions is due to the different behavior of branches that follow the LOAD. However, since the behavior of most branch instructions is heavily biased, it reasonable to expect that in most situations the same control flow path will be taken following the dynamic instances of the same static LOAD instruction. Consequently, we can expect reasonably high DoD prediction accuracy.

The simplest implementation of the DoD predictor is in the form of last-value predictor, where the number of dependents of a LOAD instruction is predicted to be the same as the number of dependents observed during the last dynamic instantiation of the same static LOAD instruction. This predictor can be implemented either through a separate PC-indexed table, or the prediction information can be incorporated into the I-cache. In our experiments we assume a table-based implementation. Of course, the actual counting of the load-dependant instruction needs to be performed to verify the predictions and possibly update the predictor with the correct information. In this case, the counting can be performed several cycles prior to the completion of the miss service to obtain the most accurate estimates. If the DoD threshold is predetermined, then prediction information can amount to just a single bit, that would signify if the number of load-dependent instructions exceeds the threshold. Alternatively, the actual count of dependents can be stored, opening up opportunities for more flexible and dynamic adjustments of threshold. Further increases in DoD prediction accuracy can be realized if the gshare-style prediction is implemented, where different predictions are generated for different control flow paths. In this case, aside from possible interferences due to the finite size of the prediction table, predictions will always be accurate.

5. Results and Discussion

In this section, we compare the fair throughput performance of our schemes to that of the baseline processor. In addition, we report various statistics on the number of load dependents observed – both as a justification of our DoD threshold choices and as an indicator of increased MLP performance.

5.1 Number of Load-dependent Instructions

Figure 1 evaluates the number of instructions dependent on long-latency load (directly or indirectly) instructions observed within the ROB at the load miss service time. As seen from the graph, a typical number of load-dependent instructions is fairly small for all simulated mixes, thus justifying the motivation for the proposed design. Similar conclusions were reached by some previous work [18]

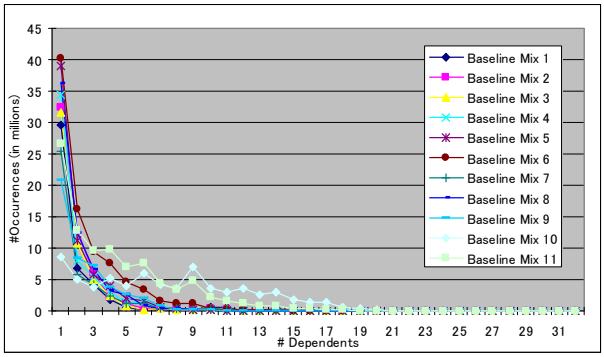

Figure 1: Number of Instructions Dependent on a Long-Latency Load in a 4-way SMT

5.2 Reactive 2-Level ROB Schemes

The first reactive 2-Level ROB scheme (2-Level R-ROB) that we examined allocates the second level of the ROB to a thread only if the following three conditions are met: 1) the LOAD instruction missing into the L2 cache is the oldest in the ROB; 2) the first-level ROB is full; and 3) the number of detected load dependents is less than an established DoD threshold. These conditions are checked the first cycle the L2 miss is detected and rechecked every 10 cycles if no allocation decision was made. We implemented this scheme with various DoD thresholds, ranging from 1 to 16 instructions. Further increases in the threshold value permits disproportionate IQ use resulting in issue queue clog and resulting in lower performance.

Figure 2 shows the FT (Fair Throughput) metric of the most appropriate DoD thresholds (labeled 2-Level R-ROB#, where # stands for the specific threshold value). The results are compared to two baseline machine configurations. The first baseline configuration (called Baseline_32) has only one level of ROB for each thread with 32 entries each. The second baseline configuration (referred to as Baseline_128) corresponds to the baseline architecture where each private ROB has 128 entries. The significance of the second baseline is that it provides the same total number of ROB entries as the proposed 2-Level ROB design. As seen in Figure 2, the Baseline_128 configuration significantly underperforms the Baseline_32 configuration due to the increased pressure on the shared resources – as explained in Section 1. The 2-Level R-ROB scheme is shown for the DoD threshold of 16 which was determined to be the best configuration from our simulations. On average across all simulated mixes, 2-Level R-ROB16 result in a 30.53% FT performance improvement compared to the Baseline_32 machine and 59.5% improvement compared to the Baseline_ 128 machine.

Providing a larger instruction window as a result of deploying a second ROB level naturally increases the number of load-dependent instructions that can be in-flight due to the deeper exploitation of dependency chains. Figure 3 shows an increase of long-latency load dependents - which are captured by the ROB following the L2 miss - by 56% compared to the results of Figure 1.

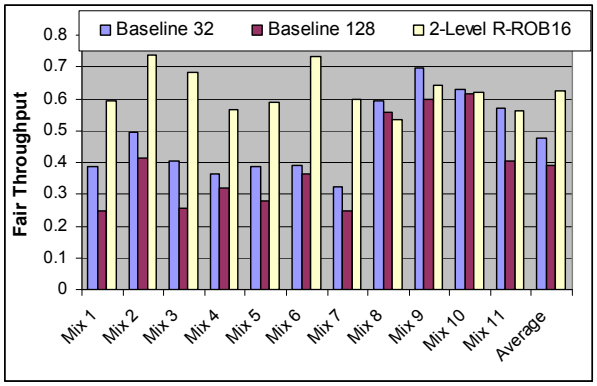

Figure 2: FT Performance with 2-Level R-ROB

Consequently, the performance improvements of the 2-Level ROB scheme come from 2 sources: 1) deeper exploitation of parallelism in the shadow of a missing long-latency load and/or 2) overlapping the servicing of multiple L2 misses.

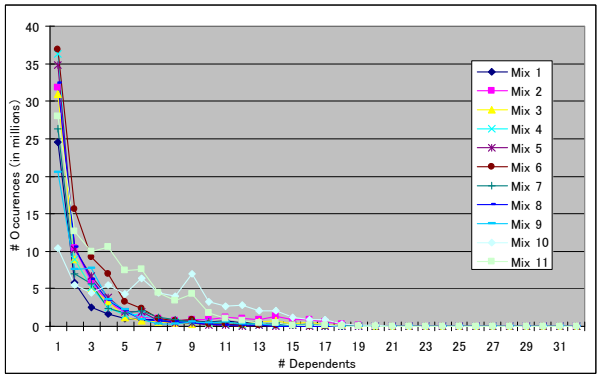

Figure 3: Number of Instructions Dependent on a Long-Latency Load with a 2-Level R-ROB

Since this scheme is reactive in nature, larger performance improvements can be achieved if the decision to provide the second level ROB resources is made as quickly as possible. To this end, we also measured the performance of a 2-Level R-ROB in a less restrictive context. By relaxing the requirement that the first level ROB has to be full before the allocation of the second level, the delay between L2-cache miss detection and second level ROB allocation can be reduced.

Figure 4 presents the results of this variation of 2-Level R-ROB (we call it 2-level Relaxed R-ROB) in terms of fair throughput. On the average across all benchmarks, fair throughout increases by 28.9% compared to the baseline machine. This is slightly lower than the performance gained when the decision is delayed until the first level ROB is full (as in the prior scheme). The lower performance is due to the fact that the dependency count is sometimes taken when the first level ROB is only partially full – this results in an artificially lower dependency count. Therefore, sometimes 2-Level Relaxed R-ROB scheme will provide second-level ROB resources to a thread that would not have received them with 2-Level R-ROB alone, which in some cases can be detrimental to performance.

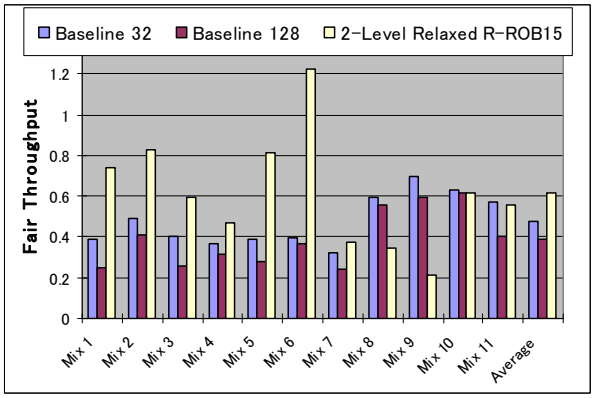

Figure 4: FT Performance with 2-Level Relaxed R-ROB15

Alternatively, these schemes could be modified to be more proactive in nature. The DoD threshold is pivotal in preventing the issue queue clog generated by large ROBs; therefore, only the first two constraints (a full first level ROB and the L2-cache missing LOAD being the oldest instruction in the ROB) can be relaxed. Counting dependents immediately after an L2-cache miss is detected does not yield significant benefits for two reasons: a) the DCRA fetch scheme distributes instructions relatively evenly and b) the ROB may have unrelated instructions prior to the missing LOAD. In order to get a reasonable idea of the number of dependents, an adequate number of cycles must pass which allow the first level ROB to be filled with instructions.

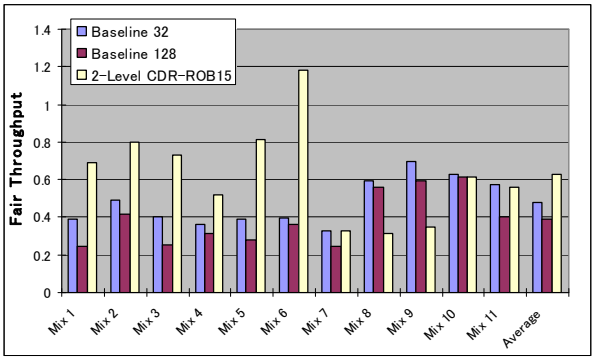

Figure 5: FT Performance with 2-Level CDR-ROB15

The results in Figure 5 depict the FT performance results of 2-Level R-ROB using a 32-cycle delay between the L2-cache miss detection and the act of taking the snapshot of the dependence count (we call this scheme 2-level CDR-ROB). FT improvements of 31.5% are observed in this scheme on average across all the benchmarks.

5.3 Predictive 2-Level ROB (2-Level P-ROB)

In contrast to the reactive nature of the previous schemes, 2-Level P-ROB variation relies on a dependency predictor in order to make decisions about second level ROB allocations at the time of an L2-cache miss detection. This predictor is populated by the dependency data from loads that miss into the L2 cache. Shortly before servicing the offending load, the number of dependents is counted and stored in the predictor table to be used in subsequent instances of this load.

Figure 6 reveals an important distinction regarding the behavior of the Predictive scheme compared to the Reactive scheme. FT performance with 2-Level P-ROB is best when a low DoD threshold (rather than a high DoD threshold as in the case with 2-Level R-ROB) is used. 2-Level P-ROB3 and 2-Level P-ROB5 variations achieve 19.71% and 20.72% FT performance improvement respectively.

The difference in DoD thresholds is attributed to how the threads behave based on second level ROB allocation time. When allocation of a second-level partition is performed quickly, as in the Predictive scheme, performance is improved as a result of overlapping several L2-cache misses and servicing them in parallel – this is indicated by an increase in the average number of L2 dependents – a larger increase than the Reactive scheme. Figure 7 shows an increase of long-latency load dependents by 120.31% using the Predictive scheme.

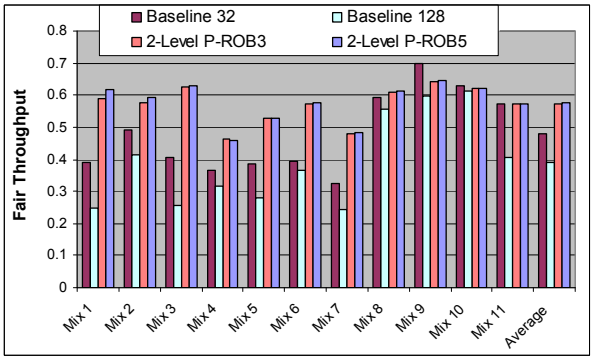

Figure 6: FT Performance with 2-Level P-ROB

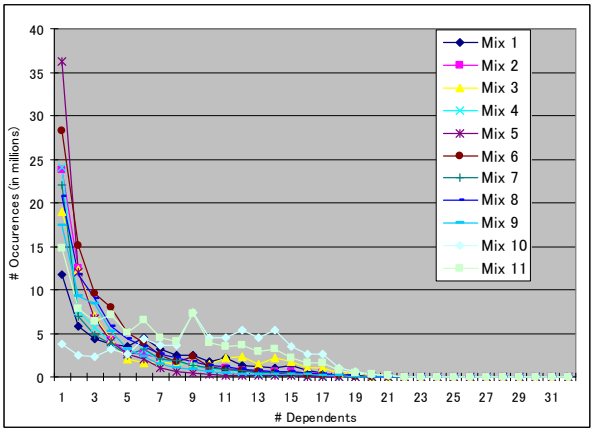

Figure 7: Number of Instructions Dependent on a Long-Latency Load with a 2-Level P-ROB

6. Concluding Remarks

We presented a scheme for accelerating the performance of memory-bound applications on SMT processors without negatively impacting the performance of other concurrently running applications. This is achieved by arranging the ROB storage in the form of two levels and allocating large second level of ROB to threads experiencing L2 cache misses only if the number of instructions dependent on the missing load does not exceed the predetermined threshold.

Our solution effectively addresses the challenges associated with ROB scalability on SMT processors and allows to utilize the larger ROBs in a manner that achieves performance improvement in memory-bound programs as well as the overall SMT performance. We evaluated reactive as proactive variants of 2-Level architecture an established that the reactive schemes generally perform better. On the average across simulated mixes of SPEC 2000 benchmarks, the best of the proposed designs achieves about 30% improvement in fair throughput compared to the baseline machine. In contrast, if the ROB is simply scaled in the

baseline machine without using the proposed 2-level organization, then not only it is impossible to achieve performance improvements, but the slowdowns are often observed due to the extra pressure on the shared datapath resources.

The proposed 2-Level ROB design is unique in its capability of explicitly targeting the performance improvements of slow memory-bound threads without slowing down other concurrently running applications. As a result, significant improvements in the overall fair throughput SMT metric can be realizable as demonstrated by our results.

7. Acknowledgements

This work was supported in part by NSF through awards CNS 0454298 and CNS 0720811, and by Intel. Jason Loew is partially supported by the Clark Fellowship.

8. 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 VLSI, 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. International Symposium on High-Performance Computer Architecture (HPCA), 2003.

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

[7] K. Luo, et al. "Balancing Throughput and Fairness in SMT Processors." in Proc International Symposium on Performance Analysis of Systems and Software, 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." in Proc of Int'l Symp. on Computer Architecture, 1995.

[15] G. Dorai, et al., "Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance", in Proceedings of 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, February 2002.

[17] S. Srinivasan et al, "Continual Flow Pipelines", in Proceedings of the International Conference on Architectural Support for Prog. Languages and Operating Systems", 2004.

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

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

[20] J. Stark, et al., "On Pipelining Dynamic Instruction Scheduling Logic", in Proc. of the 33rd Int'l Symposium on Microarchitecture (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.

[23] J. Sharkey, D. Balkan, D. Ponomarev, "Adaptive Reorder Buffers for SMT Processors", in Proceedings of 15th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006.

[24] J. Sharkey, D. Ponomarev, "An L2-Miss-Driven Early Register Deallocation for SMT Processors", in Proceedings of the International Conference on Supercomputing (ICS), June, 2007.

[25] S. Eyerman, L. Eeckhout, "A Memory-Level-Parallelism Aware Fetch Policy for SMT Processors", Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA), February 2007.