Are We Ready for High Memory-Level Parallelism?

Luis Ceze, James Tuck and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {luisceze,jtuck,torrella}@cs.uiuc.edu

***Abstract*— Recently-proposed processor microarchitectures that generate high Memory Level Parallelism (MLP) promise substantial performance gains. However, current cache hierarchies have Miss-Handling Architectures (MHAs) that are too limited to support the required MLP — they need to be redesigned to support 1-2 orders of magnitude more outstanding misses. Designing scalable MHAs can be tricky: they must minimize cache lock-up time and deliver high bandwidth while, at the same time, keeping the area consumption reasonable.**

This paper characterizes how cache misses behave in two latency-tolerant processor architectures: one with checkpointed value prediction; and the other a plain large-window processor. Based on the characterization, we present a set of MHA requirements for latency-tolerant processors. Our experiments use SPECint and SPECfp benchmarks and multiprogramming mixes.

I. Introduction

A flurry of recent proposals for novel superscalar microarchitectures claim to support very high numbers of concurrent, in-flight instructions and, as a result, substantially boost performance [1]–[7]. These microarchitectures typically rely on processor checkpointing with speculative execution and even retirement. They often seek to overlap cache misses by using predicted values for the missing data, by buffering away missing loads and their dependents, or by temporarily using an invalid copy of the missing data. Examples of such microarchitectures include Runahead [5], CPR [1], Out-of-order Commit processors [3], CAVA [2], CLEAR [4], and CFP [6], among others. It is hoped that these microarchitectures will deliver major performance gains over conventional superscalars.

Not surprisingly, these microarchitectures also generate dramatic increases in Memory Level Parallelism (MLP) — broadly defined as the average number of outstanding memory system accesses at a time [8]. For example, one of these designs assumes support for up to 128 outstanding L1 misses at a time [6]. To reap the benefits of these microarchitectures, cache hierarchies have to be designed to support this level of MLP.

Current cache hierarchy designs are woefully unsuited to support this level of demand. Even in designs for high-end processors, the norm is for L1 caches to support little more than a handful of outstanding misses at a time [9]–[11]. For example, Pentium 4 only supports 8 outstanding misses at a time [12]. Unless the architecture that handles misses (i.e., the *Miss Handling Architecture* (MHA)) is re-designed to support 1-2 orders of magnitude more outstanding misses, there will be little gain to realize from the new microarchitectures.

This paper presents extensive data on how L1 cache misses behave under high MLP. We show that state-of-the-art MHAs for L1 caches are unable to leverage new latency-tolerant processor microarchitectures. In addition, we present MHAs requirements suitable for latency-tolerate processors that exploit high degrees of memory-level parallelism. Finally, we examine the effectiveness of one bus prioritization scheme under high MLP.

The paper is organized as follows: Section II presents background information on MHAs; Section III describes the experimental setup we used in our characterization; Section IV characterizes MHAs under high MLP and summarizes its requirements; Section V shows two possible MHA designs and discusses some design constrains; and Section VI concludes.

II. Background

A. Miss Handling Architectures (MHAs)

The *Miss Handling Architecture (MHA)* is the logic and resources needed to support outstanding misses in a cache. Kroft [13] proposed the first MHA that enabled a lock-up free cache (one that does not block the processor on a miss) and supported multiple outstanding misses at a time. To support an outstanding miss, he introduced a Miss Information/Status Holding Register (MSHR). An MSHR stores the address requested, the size and type of the request, and other information. Kroft organized the MSHRs into an MSHR file accessed after the L1 tag detects a miss (Figure 1(a)). He also described how store misses on blocks that are outstanding can buffer their data using the MSHR, enabling *forwarding*. A subsequent load to the same address can read the data, rather than waiting for memory.

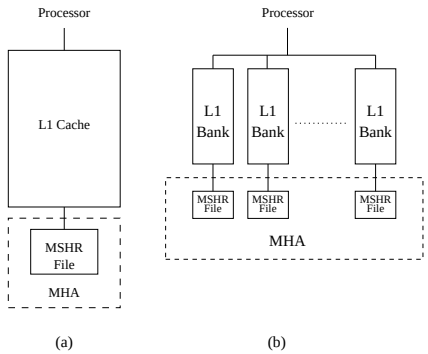


Fig. 1. Examples of Miss Handling Architectures (MHAs).

Scheurich and Dubois [14] described an MHA for lock-up free caches in multiprocessors. They described the organization of an MSHR that keeps information on all the misses outstanding on a given cache line. They showed the tight relationship between the cache coherence protocol and the MHA algorithms. Later, Sohi and Franklin [15] evaluated the bandwidth advantages of using cache banking in non-blocking caches. They showed a design where each cache bank has its own MSHR file (Figure 1(b)) but did not discuss the MSHR itself.

In the context of an MHA, it is useful to distinguish between primary and secondary cache misses. A cache miss on a line is *primary* if there is currently no outstanding miss on the line and, therefore, a new MSHR needs to be allocated. Otherwise, the miss is *secondary*. In this case, the existing MSHR for the line can be augmented to record the new miss, and no request is issued to memory. In this case, the MSHR for a line keeps information for all outstanding misses on the line. For each miss, it contains a *subentry* (in contrast to an *entry*, which is the MSHR itself). Among other information, a subentry for a read miss contains the ID of the register that should receive the data; for a write miss, it contains the data itself or a pointer to a buffer with the data.

MSHRs take significant area. Consequently we cannot have many MSHRs. If the MHA exhausts its MSHRs or subentries, and has no space for future misses, it *locks-up* the cache (or the corresponding cache bank). From then on, the cache or cache bank rejects any further request from the processor. This may eventually lead to a processor stall. The cache remains locked-up until a memory reply opens up space in the MHA.

Farkas and Jouppi [16] investigated *Implicitly*- and *Explicitly*-addressed MSHR organizations for read misses. In the Implicit one, the MSHR has a subentry for each word in the line. On a read miss, the subentry at the corresponding offset is allocated. In the Explicit organization, any of the (fewer) subentries in the MSHR can be used by any miss on the line. However, the subentry needs to save the block offset of the miss. For the simple processor they evaluate, they use a 4-MSHR file, and show that, for a 32-byte cache line rarely more than four subentries are needed.

Finally, there is very limited information on the MHAs used by current processors. It appears that current designs are fairly limited and support only a handful of outstanding L1 cache misses at a time [9]–[11]. For example, Pentium 4 only supports 8 outstanding misses (of primary or secondary type) at a time [12].

B. Microarchitectures for High MLP

Cherry [17] checkpoints the processor and recycles resources (registers and load/store queue entries) that would otherwise be unavailable until instruction commit. Recycling these resources increases the number of instructions in flight.

Runahead execution [5], checkpoints the processor and retires a missing load, marking the destination register as invalid. The I-window is unblocked and execution proceeds, prefetching data into the cache. When the load completes, execution rolls back to the checkpoint. A related scheme by Zhou and Conte [7] uses value prediction on missing loads to continue execution (no checkpoint is made) and re-executes everything on load completion.

Checkpoint-based value prediction schemes like CAVA [2] and CLEAR [4] checkpoint on a long-latency load miss, predict the value that the load will return, and continue execution using the prediction. Speculative instructions are allowed to retire. If the prediction is later shown to be correct, no rollback is necessary.

CPR [1] and Out-of-order Commit [3] processors remove scalability bottlenecks from the I-window to substantially increase the number of in-flight instructions. They remove the ROB, relying on checkpointing (e.g., at low-confidence branches) to recover in case of misspeculation. CFP [6] frees the resources of a missing load and its dependent instructions without executing them. This allows the processor to continue fetching and executing independent instructions. The un-executed instructions are buffered and executed when the data returns from memory.

All these microarchitectures generate a large number of concurrent memory accesses. These accesses need support at two different levels, namely at the load/store queue (LSQ) and at the cache hierarchy level. First, they need a LSQ that provides efficient address disambiguation and forwarding. Second, those that miss somewhere in the cache hierarchy need an MHA that efficiently handles many outstanding misses. While previous work has proposed solutions for scalable LSQs [18]–[20], the problem remains unexplored at the MHA level. Our paper examines this problem.

III. Experimental Setup

We use execution-driven simulations to evaluate MHA configurations for the three processors shown in Table I: *Conventional*, *Checkpointed*, and *LargeWindow*. *Conventional* is a 5-issue, 2-context SMT processor. *Checkpointed* extends *Conventional* with support for checkpoint-based value prediction [2], [4]. Its additional parameters in Table I are the Value Prediction Table and the maximum number of outstanding checkpoints. Each hardware thread has its own checkpoint, and each thread can rollback to its checkpoint without affecting the other thread. *LargeWindow* is *Conventional* with a 512-entry instruction window and 2048-entry ROB.

The three processors have identical memory systems (Table I), including two levels of on-chip caches. The exception is that *Checkpointed* has one bit per line in its L1 cache to mark data updated speculatively.

A. Checkpoint-Assisted Value Prediction

Checkpoint-assisted value prediction hides long-latency misses by providing predicted values for long-latency loads that reach the head of the reorder buffer and stall the retirement of subsequent instructions. The processor state is checkpointed and the missing load is allowed to retire. When the response from memory arrives, the prediction is validated. If the prediction was correct, execution continues normally; otherwise,

TABLE I

PROCESSORS SIMULATED. IN THE TABLE, RAS AND RT STAND FOR RETURN ADDRESS STACK AND MINIMUM ROUND-TRIP LATENCY FROM THE PROCESSOR, RESPECTIVELY. CYCLE COUNTS REFER TO PROCESSOR CYCLES.

All	Memory System
Frequency: 6.5GHz at 65nm Fetch/issue/comm width: 6/5/5 LdSt/Int/FP units: 4/3/3 SMT contexts: 2 Branch penalty: 13 cyc (min) RAS: 32 entries BTB: 2K entries, 2-way assoc. Branch predictor (spec. update): bimodal size: 16K entries gshare-11 size: 16K entries	I-L1 / D-L1 / L2 Size: 32KB / 32KB / 2MB Assoc: 2-way / 2-way / 8-way Line size: 64 / 64 / 64 RT: 2 cyc / 3 cyc / 15 cyc Ports/Bank: 2 / 1 / 1 Banks: – / 8 / – HW Pref.: 16-stream strided (bet. L2 and mem.) Mem Bus Bandwidth: 10GB/s Mem RT: 650 cyc
Conventional and *Checkpointed*	*LargeWindow*
I-window/ROB size: 92/192 Int/FP registers: 192/192 Ld/St queue entries: 60/50 *Checkpointed* Only: Val. Pred. Table: 2048 entries Max Outs. Ckps: 1 per context	I-window/ROB size: 512/2048 Int/FP registers: 2048/2048 Ld/St queue entries: 768/768

execution rolls back to the checkpoint. We combine CAVA and SMT to have a scenario of even higher MLP than what CAVA already provides. We extend the CAVA implementation described in [2] to work with SMT processors. This means that each hardware thread is able to hide its long latency misses by employing CAVA techniques. Each hardware thread has its own checkpoint. Rollback to the checkpoint of one hardware thread does not affect normal execution of the other hardware thread. While processor checkpointing in SMT is an interesting design point, it is out of the scope of this paper to discuss its details.

B. Workloads

We run our experiments using SPECint2000 and SPECfp2000 codes, and workload mixes that combine two applications at a time (Table II). From SPECint, we use all the programs for which a *perfect* MHA (unlimited number of MSHRs, subentries, and bandwidth) would make at least a 5% performance impact in any of the architectures analyzed — the remaining SPECints do not have enough misses to make any MHA-enhancing technique worthwhile. From SPECfp, we use all codes but four that are in Fortran 90 and two that have system calls unsupported by our simulator. Finally, for the workload mixes, the algorithm followed is to pair one SPECint and one SPECfp such that one has high MSHR needs and one low. In addition, one mix combines two lows, one combines two highs, and yet another one combines 2 SPECfps. Overall, we cover a range of behaviors in the mixes. In these runs, each application is assigned to one hardware thread and the two applications run concurrently.

We compile the applications using gcc 3.4 -O3 into MIPS binaries and use the *ref* data set. We evaluate each application for 0.6-1.0 billion committed instructions, after skipping several billion instructions as initialization. To compare performance, we use committed IPC. When comparing performance of multiprogramming mixes, we use weighted speedups as in [21].

TABLE II

WORKLOADS USED IN OUR EXPERIMENTS.

SPECint2000	SPECfp2000	Mix
256.bzip2 (*bzip2*)	188.ammp (*ammp*)	179.art, 183.equake (*artequake*)
254.gap (*gap*)	173.applu (*applu*)	179.art, 254.gap (*artgap*)
181.mcf (*mcf*)	179.art (*art*)	179.art, 253.perlbmk (*artperlbmk*)
253.perlbmk (*perlbmk*)	183.equake (*equake*)	183.equake, 253.perlbmk (*equakeperlbmk*)
	177.mesa (*mesa*)	177.mesa, 179.art (*mesaart*)
	172.mgrid (*mgrid*)	172.mgrid, 181.mcf (*mgridmcf*)
	171.swim (*swim*)	171.swim, 181.mcf (*swimmcf*)
	168.wupwise (*wupwise*)	168.wupwise, 253.perlbmk (*wupwiseperlbmk*)

IV. MISS HANDLING UNDER HIGH MLP

The MLP boost by the new microarchitectures described puts major pressure on the cache hierarchy, especially at the L1 level. To handle the pressure on the L1 data cache, we can use known techniques, such as banking the L1 and making it set-associative. However, a lesser known yet acute problem remains, namely that the MHA in the L1 is asked to store substantially more information and sustain a higher bandwidth than in conventional designs. This section characterizes how MHAs affect performance of the processors described in Table I.

A. Occupancy

Figure 2 shows the distribution of the number of outstanding L1 read misses at a time.[1] It shows the distributions for the three processors. Each line corresponds to one workload from Table II.

We see that, for *Conventional*, most workloads have 16 or fewer outstanding load misses 90% of the time. These requirements are roughly on a par with the MHA of state-of-the-art superscalars. On the other hand, *Checkpointed* and *LargeWindow* are a stark contrast, with workloads sustaining more than 128 outstanding load misses for a significant fraction of the time.

The misses in Figure 2 include both primary and secondary misses. Suppose now that a single MSHR holds the state of all the misses to the same L1 line. In Figure 3, we redraw the data showing the number of MSHRs in use at a time. We use an L1 line size of 64 bytes.

Compared to the previous figure, we see that the distributions move to the upper left corner. The requirements of *Conventional* are few. For most workloads, 8 MSHRs are enough for 98% of the time. However, *Checkpointed* and *LargeWindow* have a much greater demand for entries. For *Checkpointed*, many workloads need 16-64 MSHRs for a large fraction of the time. Therefore, the new MHAs need higher capacity than current designs.

B. Bandwidth

The MHA is accessed at three different times. First, every L1 miss reads the MHA to see if it contains an MSHR for the accessed line. In addition, every L1 miss that is not satisfied by a data forward from the MHA also updates the MHA to record the miss. Finally, every L1 primary miss induces

[1] Although we use a write-back L1 cache, we only show read misses so that the data is also relevant to write-through L1 caches.

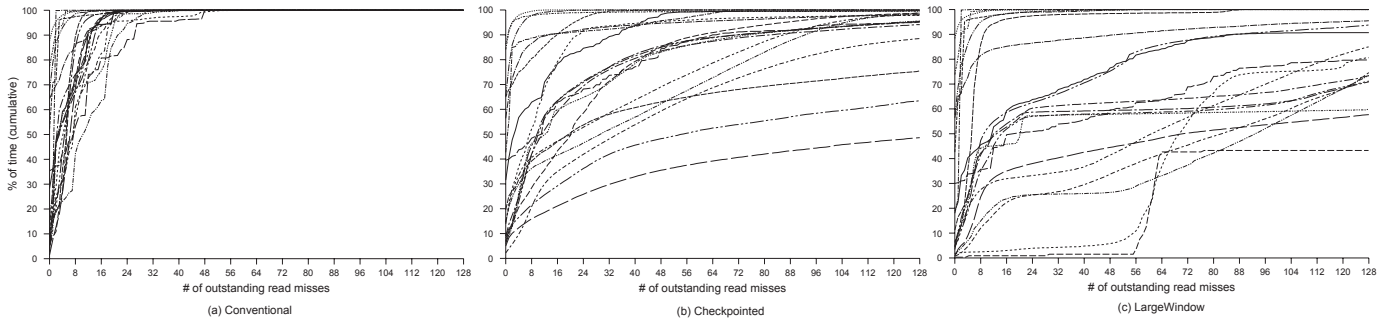


Fig. 2. Number of outstanding L1 read misses at a time in *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c).

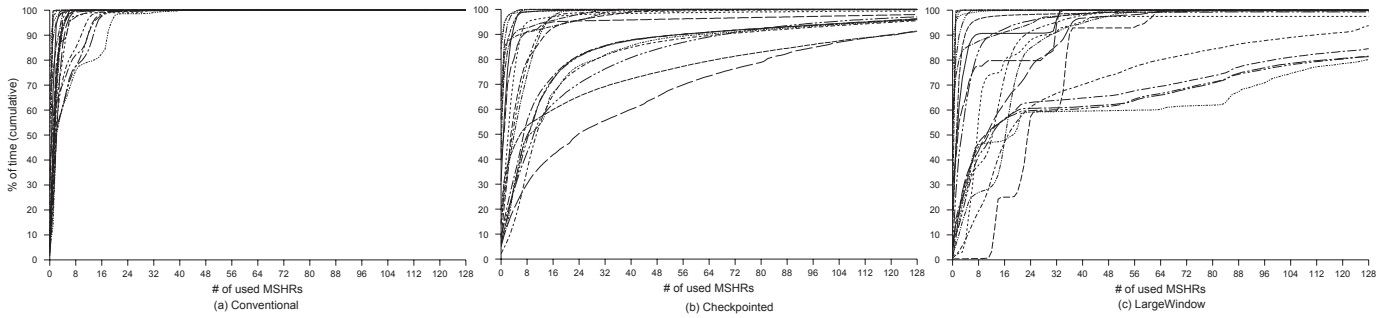


Fig. 3. Number of L1 MSHR entries in use at a time in *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c).

another update to the MHA to clean up its MSHR when the line arrives from the L2.

We compute the number of MHA accesses during 100-cycle intervals for *Conventional*, *Checkpointed*, and *LargeWindow*. Figure 4 shows the distribution of the number of accesses per interval. For *Conventional*, most workloads have at most 30-40 accesses per interval about 90% of the time. For *Checkpointed*, the number of accesses to reach 90% of the time is about twice that. For *LargeWindow*, still more accesses are required to reach 90%. Overall, new MHAs need higher bandwidth than conventional ones.

C. Capacity

L1 misses can be either primary or secondary, and be caused by reads or writes. For each case, the MHA needs different support. For primary misses, it needs MSHR entries; for secondary ones, it needs subentries. The latter typically need different designs depending on whether they are for reads or writes. Given the many outstanding misses, new MHAs have to be designed to support many misses of every type.

We want to assess the needs in number of entries, read subentries, and write subentries. For this, we use a single-bank MHA and vary one parameter while keeping the other two unlimited. If the varying dimension runs out of space, the L1 refuses further processor requests until space opens up.

In Figure 5, we vary the number of MSHR entries. Our workloads benefit significantly by going from 8 to 16, and less so from 16 to 32, both in *Checkpointed* and *LargeWindow*. Note that this effect is much less pronounced in *Conventional*, justifying why current processors support only a handful of outstanding misses.

In Figure 6(a), we vary the number of read subentries per MSHR for *Checkpointed*. Secondary read misses are frequent, and supporting less than 16-32 read subentries hurts performance. In Figure 6(b), we vary the number of write subentries per MSHR for *Checkpointed*. Secondary write misses are also important, and we need around 16 write subentries. An additional insight is that individual MSHRs typically need read subentries or write subentries, but rarely both kinds. This data is shown in Figure 6(c) running with an unlimited MHA. This behavior is due to read miss and write miss locality.

D. Associativity

MHA capacity is provided by increasing the number of MSHRs. Intuitively, it would be ideal to have fully associative MHAs, but this is expensive in hardware. This is addressed by decreasing associativity. However, by decreasing associativity, the chances of lockup increase, because if any set fills up, the MHA stops accepting new requests. Figure 7 shows the performance of a 32-entry MHA varying associativity. The plots show that for *Checkpointed*, 8-way attains almost the same performance as a fully associative structure. Also, note that there is a significant drop in performance from 8-way to 4-way. Finally, we observed that *LargeWindow* is less sensitive to associativity than *Checkpointed*, especially for SPECints.

E. Summary of Requirements

The characterization numbers show that latency-tolerant processors such as *Checkpointed* and *LargeWindow* both need

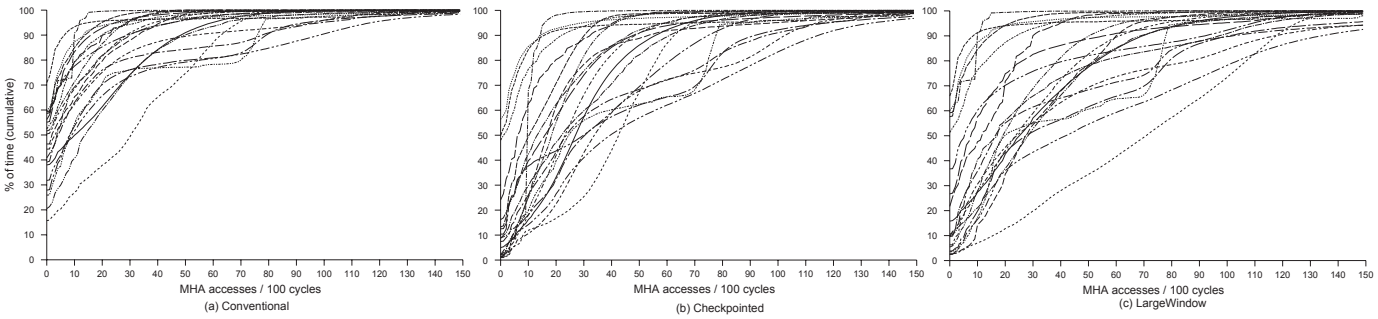
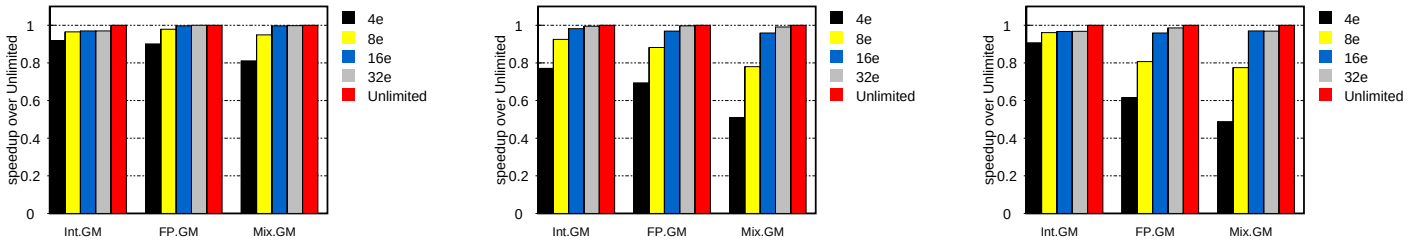
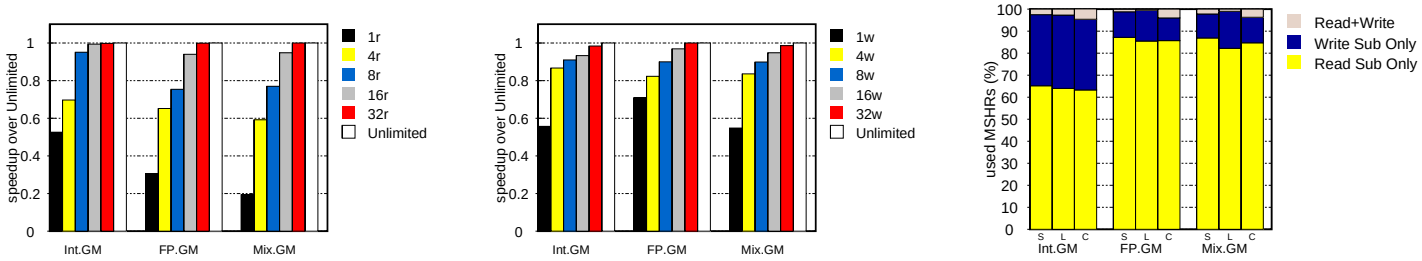


Fig. 4. Bandwidth required from the MHA for *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c).



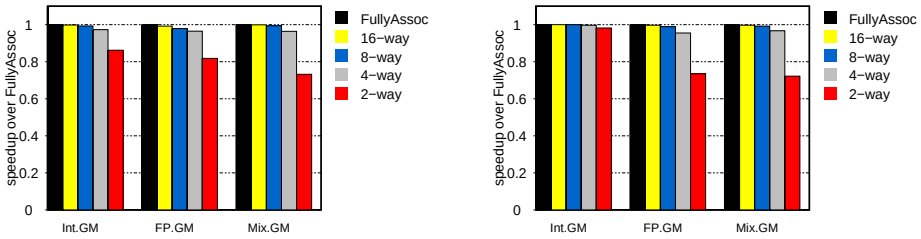
(a) *Conventional* (b) *Checkpointed* (c) *LargeWindow*

Fig. 5. Effect of varying the number of MHA entries.



(a) Read subentries. (b) Write subentries. (c) Distribution.

Fig. 6. Sub-entry usage characterization. (a) and (b) are for a *Checkpointed* processor. (c) was obtained with unlimited read/write subentries. In (c), *S,L,C* refers to *Conventional*, *LargeWindow* and *Checkpointed*, respectively.



(a) *Checkpointed* (b) *LargeWindow*

Fig. 7. Effect of varying the associativity of a 32-entry MHA.

large capacity with high associativity and support for many read and write secondary misses. Namely, in terms of capacity, a 32-entry, 8-way set-associate MHA with support 32 read subentries and 32 write subentries will offer similar performance to an ideal, unlimited MHA. Unlimited write subentries can be provided by using an *Implicit* organization [16]. However unlimited read subentries are very complex to provide.

We showed that *Checkpointed* and *LargeWindow* both perform significantly more accesses to the MHA per unit of time than *Conventional*. That means that an MHA for a high MLP

processors will have to provide sufficient access bandwidth. To increase MHA bandwidth, we can bank it like the L1 cache (Figure 1(b)). However, the number of MSHRs in the MHA is fairly limited due to area constraints. With few MSHRs, heavy banking may be counter-productive: if the use of the different MHA banks is imbalanced, one of them may fill up. If this happens, the corresponding L1 bank locks-up; it rejects any further requests from the processor to the L1 bank, potentially stalling the processor. This problem is analogous to cache conflicts in a banked L1 [22], except that a "conflict" in a fully-associative MHA bank lasts for as long as all its entries are in use.

Finally, MSHRs are large structures and also regarded as fairly intricate structures. For that reason, a requirement for MHA is to meet a reasonable area and complexity budgets to keep design costs under control.

V. Two Possible High MLP Designs

We look at two designs similar to prior proposals and scale them to meet the requirements presented in the previous section. First, consider *Unified*, a single MSHR file sized large enough to meet the capacity demands under high MLP; second, consider *Banked*, a set of eight MSHR files with one per cache bank, which may be better suited to meet high bandwidth demands. For a fair comparison between *Unified* and *Banked*, we select designs of equivalent area (25% of the L1 cache) as modeled in CACTI [23]. The characteristics of each design are shown in Table III.

TABLE III
Two conventional designs sized at 25% the size of the L1 cache.

Name	MSHRs	Assoc.	Rd Sub.	Wr Sub.
Unified	16	8	32 Exp.	8 Imp.
Banked	2x8	Full	32 Exp.	8 Imp.

We evaluate the performance of both *Unified* and *Banked* in the context of *Checkpointed*, *LargeWindow*, and *Conventional* processor architectures. We also investigate bus contention of *Unified* and *Banked* on *Checkpointed*.

A. Performance

The performance of *Unified* and *Banked* are shown in Figure 8 in comparison to *Current*, a conventional design with 8 MSHRs like the Pentium 4, and to *Unlimited* running on *Checkpointed*, *LargeWindow*, and *Conventional*. As shown in Figure 8(a), *Unified* and *Banked* outperform *Current* for almost all benchmarks. Overall, *Unified* does better than *Banked* because partitioning the MSHR file limits its effective capacity. However, there are a few cases like *art*, *applu*, and few mixes which include *art* for which *Banked* does better. In these cases, the higher bandwidth afforded by *Banked* is important. However, though *Unified* and *Banked* are much better than *Current*, there is still a significant performance gap when compared to *Unlimited*, especially for Mix.

Figure 8(b) shows the performance on *LargeWindow*. Here, *Banked* is better than *Unified* indicating that bandwidth is more important for this processor design. However, *Banked* and *Unified* still fall short of *Unlimited* by a significant fraction. For *Conventional*, there are small gains using *Unified* and *Banked* due to the low MLP provided by this processor.

To close the performance gap to *Unlimited* for *Checkpointed* and *LargeWindow*, we can build larger structures for either *Unified* or *Banked*. However, the size of these structures may quickly become prohibitive. For example, if we double the number of MSHRs in either *Unified* or *Banked*, the MHA will be roughly half the size of the cache. Hence, instead of naively increasing the size of either *Unified* or *Banked*, we are investigating the design of new MHAs at 25% of the cache that nearly match the performance of *Unlimited*.

B. Bus Concerns in High MLP

Increasing the maximum number of outstanding misses obviously increases the pressure on the memory bus. Figure 9 shows the bus contention for *Unified* and *Banked* on *Checkpointed* normalized to *Current*. Note that the bus contention is significantly increased, on average, for each class of benchmarks.

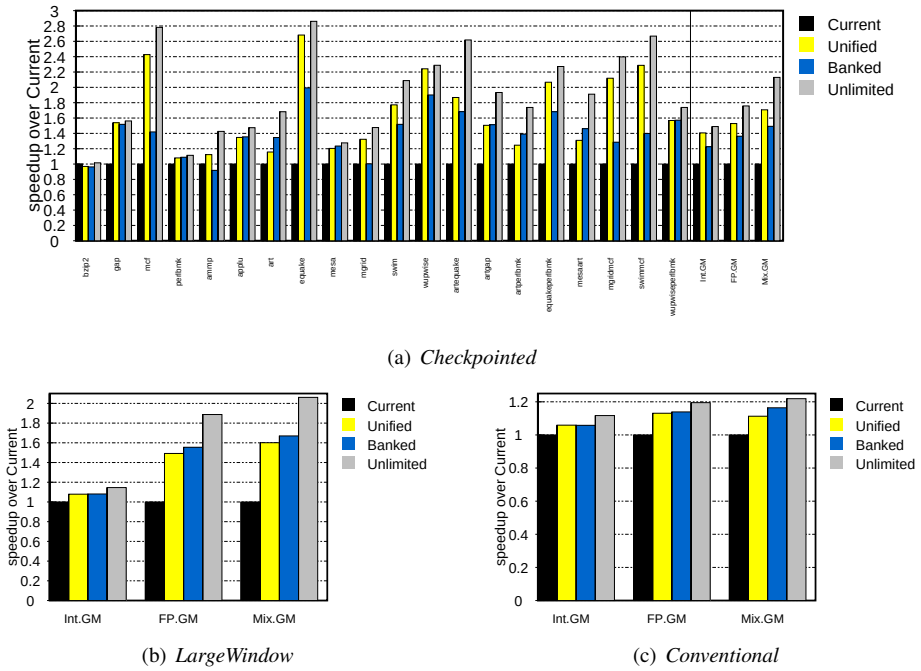
The primary reason for the high bus contention is the increase in speculative memory requests made by the processor. In case speculation is successful, these requests have paid off and improved performance. However, in some cases, pressure and contention on the bus are so high that demand misses are put off in favor of earlier less critical requests that have possibly been squashed in a misspeculation. This situation gets even worse in chip multiprocessors, where the memory bus is shared among several cores.

For that reason, we ran some experiments with a memory bus that supports requests with two priorities, high and low. We gave high priority to non-speculative read requests and speculative read requests whose associated predicted value sent to the processor was of low confidence. All other requests were given low priority.

Figure 10 shows the results of these experiments. It shows the speedups of prioritization over no prioritization. Some applications show little to no difference in performance when prioritization is used, while others show speedups of as high as 20%. The numbers on top of the bars show the percentage of requests that were assigned high priority. This shows that prioritizing requests might become necessary as the memory bus contention goes higher.

C. Reusing Load/Store Queue State for Miss Handling

An important concern in the design of an MHA is its relationship with the Load/Store Queue. It is possible to conceive a design where the MHA is kept to a minimum by leveraging the LSQ state. Specifically, we can allocate a simple MSHR on a primary miss and keep no additional state on secondary misses — the LSQ entries corresponding to the secondary misses can keep a pointer to the corresponding MSHR. When the data arrives from memory, we can search



(a) *Checkpointed*

(b) *LargeWindow*

(c) *Conventional*

Fig. 8. Performance of *Banked* and *Unified* compared normalized to *Current*. *Unlimited* is not a feasible design, it is included for comparison.

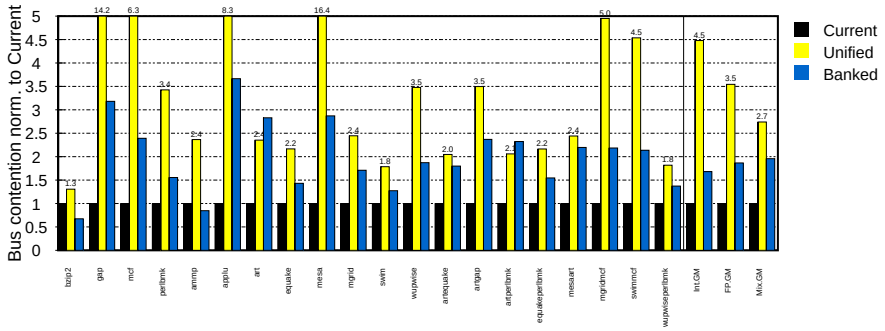


Fig. 9. Bus contention normalized to *Current* in the *Checkpointed* processor.

the LSQ with the MSHR ID and satisfy all the relevant LSQ entries.

We argue that this might not be a good design for the advanced microarchitectures described. First, it induces global searches in the large LSQ. Recall that scalable LSQ proposals provide efficient search from the *processor-side*. The processor uses the word address to search. In the approach discussed, LSQs would also have to be searched from the *cache-side*, when a miss completes. This would involve a search using the MSHR ID or the line address, which (unless the LSQ is re-designed) would induce a costly global search. Such search is eliminated if the MHA is enhanced with subentry pointer information.

Second, some of these novel microarchitectures speculatively retire instructions and, with them, deallocate their LSQ entries [2], [4]. Consequently, the MHA cannot rely on information in the LSQ because, by the time the miss completes, it may be gone.

Finally, LSQs are time-critical structures. It is best not to augment them with additional pointer information or support for additional types of searches. In fact, we claim it is best to *avoid restricting* their design at all.

Consequently, we believe the best choice is keeping primary and secondary miss information in the MHA and not relying on specific LSQ designs.

VI. Conclusions

Recently-proposed processor microarchitectures that substantially increase MLP promise major performance gains. Unfortunately, the MHAs of current high-end systems are not designed to support the resulting level of MLP. This paper provided a quantitative analysis of how MHA design

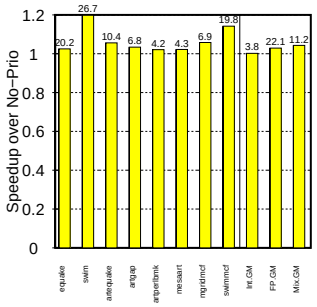


Fig. 10. Performance impact of memory request prioritization in the *Checkpointed* processor. The numbers on top of the bars refer to the percentage of bus requests that were assigned high priority.

parameters affect the performance of high MLP processors. Based on that, we compiled a set of requirements for new MHAs to meet. We scaled two previously proposed MHA designs to meet some of the requirements and showed that there is still quite a bit of room for improvement compared to *Unlimited*. Finally, we showed that together with significantly more outstanding misses comes a significantly higher bus contention.

VII. Acknowledgments

We would like to thank the members of the I-ACOMA group at the University of Illinois for their helpful comments. We especially thank Karin Strauss for valuable insights and patient proofreading of this paper. Luis Ceze is supported by an IBM PhD Fellowship.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Proceedings of the 36th International Symposium on Microarchitecture*, Nov. 2003.

[2] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Hiding L2 Misses with Checkpoint Assisted VAlue Prediction," *Computer Architecture Letters*, December 2004.

[3] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, Feb. 2004.

[4] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed Early Load Retirement," in *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, Feb. 2005.

[5] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, Feb. 2003.

[6] S. T. Srinivasan, R. Rajwar, H. Akkary, and A. G. amd Mike Upton, "Continual Flow Pipelines," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[7] H. Zhou and T. Conte, "Enhancing Memory Level Parallelism via Recovery-Free Value Prediction," in *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[8] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Memory-Level Parallelism," in *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[9] D. Koufaty, "Personal communication," Intel Corp.

[10] T. Shanley, *The Unabridged Pentium 4: IA32 Processor Genealogy*. Addison Wesley, 2005.

[11] B. Sinharoy, "Personal communication," IBM Corp.

[12] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman, "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, vol. 8, no. 1, February 2004.

[13] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981.

[14] C. Scheurich and M. Dubois, "The Design of a Lockup-free Cache for High-Performance Multiprocessors," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 352–359.

[15] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," in *Proceedings of theInternational Symposium on Computer Architecture*, 1991.

[16] K. I. Farkas and N. P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[17] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002.

[18] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, "Scalable load and store processing in latency tolerant processors," in *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[19] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue," in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 411.

[20] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 399.

[21] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[22] T. Juan, J. J. Navarro, and O. Temam, "Data Caches for Superscalar Processors," in *ICS '97: Proceedings of the 11th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1997, pp. 60–67.

[23] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache, Timing, Power, and Area Model," Compaq Western Research Laboratory, Tech. Rep., February 2001.