Scalable Cache Miss Handling for High Memory-Level Parallelism[*]

James Tuck, **Luis Ceze**, and **Josep Torrellas**
University of Illinois at Urbana-Champaign
{jtuck,luisceze,torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Recently-proposed processor microarchitectures for high Memory Level Parallelism (MLP) promise substantial performance gains. Unfortunately, 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. Yet, designing scalable MHAs is challenging: designs must minimize cache lock-up time and deliver high bandwidth while keeping the area consumption reasonable.

This paper presents a novel scalable MHA design for high-MLP processors. Our design introduces two main innovations. First, it is *hierarchical*, with a small MSHR file per cache bank, and a larger MSHR file shared by all banks. Second, it uses a *Bloom filter* to reduce searches in the larger MSHR file. The result is a high-performance, area-efficient design. Compared to a state-of-the-art MHA on a high-MLP processor, our design speeds-up some SPECint, SPECfp, and multiprogrammed workloads by a geometric mean of 32%, 50%, and 95%, respectively. Moreover, compared to two extrapolations of current MHA designs, namely a large monolithic MSHR file and a large banked MSHR file, all consuming the same area, our design speeds-up the workloads by a geometric mean of 1-18% and 10-21%, respectively. Finally, our design performs very close to an unlimited-size, ideal MHA.

1. Introduction

A flurry of recent proposals for novel superscalar microarchitectures claim to support very high numbers of in-flight instructions and, as a result, substantially boost performance [1, 3, 7, 12, 16, 22, 25]. These microarchitectures typically rely on processor checkpointing, long speculative execution, and sometimes even speculative 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 token in lieu of the missing data. Examples of such microarchitectures include Runahead [16], CPR [1], Out-of-order Commit [7], CAVA [3], CLEAR [12], and CFP [22].

Not surprisingly, these microarchitectures require dramatic increases in Memory Level Parallelism (MLP) — broadly defined as the number of concurrent memory system accesses supported by the memory subsystem [5]. For example, one of these designs assumes support for up to 128 outstanding L1 misses at a time [22]. 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 only a very modest number of outstanding misses at a time. For example, Pentium 4 only supports 8 outstanding L1 misses at a time [2]. Unless the architecture that handles misses (i.e., the *Miss Handling Architecture* (MHA)) is redesigned to support 1-2 orders of magnitude more outstanding misses, there will be little gain to realize from the new microarchitectures.

A brute-force approach to increasing the resources currently devoted to handling misses is not the best route. The key hardware structure in the MHA is the Miss Status Holding Register (MSHR), which holds information on all the outstanding misses for a given cache line [9, 13]. Supporting many, highly-associative MSHRs in a unified structure may end up causing a bandwidth bottleneck. Alternatively, if we try to ensure high bandwidth by extensively banking the structure, we may run out of MSHRs in a bank, causing cache bank lock-up and eventual processor stall. In all cases, since the MHA is located close to the processor core, it is desirable to use the chip area efficiently.

To satisfy the requirements of high bandwidth and low lock-up time in an area-efficient manner, this paper presents a new, scalable MHA design for the L1 cache. More specifically, the paper makes the following three contributions.

First, it shows that state-of-the-art MHAs are unable to leverage the new high-MLP processor microarchitectures.

Second, it proposes a novel, scalable MHA design for these microarchitectures that delivers the highest performance for a given area consumption. The proposed organization, called *Hierarchical*, introduces two main innovations. First, it is a *hierarchical design*, composed of a small per-cache-bank MSHR file, and a larger MSHR file shared by all the cache banks. The per-bank files provide high bandwidth, while the shared one minimizes cache lock-up in the presence of cross-bank access imbalances in an area-efficient manner. The second innovation is a *Bloom filter* that eliminates the large majority of unnecessary accesses to the shared MSHR file, therefore removing a possible bottleneck.

Third, the paper evaluates *Hierarchical* in the context of a high-MLP processor for some SPECint, SPECfp, and multiprogrammed workloads. Compared to a state-of-the-art MHA similar to that of Pentium 4, *Hierarchical* speeds-up the workloads by a geometric mean of 32%, 50%, and 95%, respectively. We also compare *Hierarchical* to two intuitive extrapolations of current MHA designs, namely a large monolithic MSHR file and a large banked MSHR file. For the same area consumption, *Hierarchical* is faster by a geometric mean of 1-18% and 10-21%, respectively. Finally, *Hierarchical* performs very close to an unlimited-size, ideal MHA.

The paper is organized as follows: Section 2 presents a background and motivation; Section 3 assesses the new MHA challenges;

[*]This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Luis Ceze was supported by an IBM Ph.D. Fellowship.

Sections 4 and 5 present our MHA design and its implementation; and Sections 6 and 7 evaluate the design.

2. Background and Motivation

2.1. Miss Handling Architectures (MHAs)

The *Miss Handling Architecture (MHA)* is the logic 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 on a miss) and supported multiple outstanding misses at a time. To support a miss, he introduced a Miss Information/Status Holding Register (MSHR). An MSHR stored the address requested and the request size and type, together with other information. Kroft organized the MSHRs into an MSHR file accessed after the L1 detects a miss (Figure 1(a)). He also described how a store miss buffers its data so that it can be *forwarded* to a subsequent load miss before the full line is obtained from main memory.

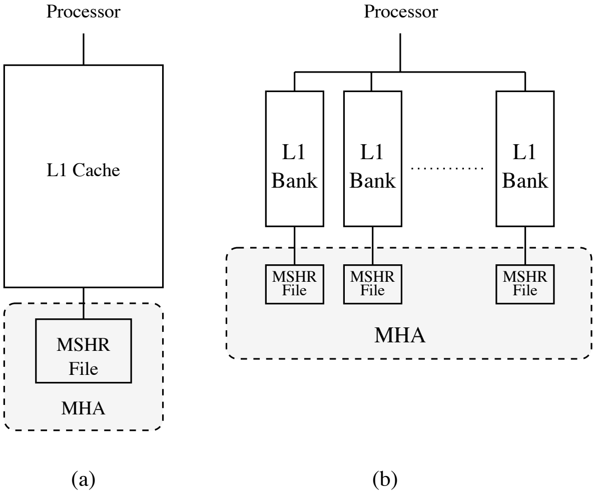


Figure 1. Examples of Miss Handling Architectures (MHAs).

Scheurich and Dubois [19] described an MHA for lock-up free caches in multiprocessors. Later, Sohi and Franklin [21] evaluated the bandwidth advantages of using cache banking in non-blocking caches. They used a design where each cache bank has its own MSHR file (Figure 1(b)), but did not discuss the MSHR itself.

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. A miss is *secondary* if there is already a pending miss on the line. 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. Once an MHA exhausts its MSHRs or subentries, it *locks-up* the cache (or the corresponding cache bank). From then on, the cache or cache bank rejects further requests from the processor. This may eventually lead to a processor stall.

Farkas and Jouppi [9] examined *Implicitly*- and *Explicitly*-addressed MSHR organizations for read misses. In the Implicit one, the MSHR has a pre-allocated subentry for each word in the line. In the Explicit one, any of the (fewer) subentries in the MSHR can be used by any miss on the line. However, a subentry also needs to record the line offset of the miss it handles.

Current MHA designs are limited. For example, Pentium 4's L1 only supports 8 outstanding misses at a time [2] — as communicated to us by one of the designers, this includes both primary and secondary misses.

2.2. Microarchitectures for High MLP

Many proposed techniques to boost superscalar performance significantly increase MLP requirements. Among these techniques, there are traditional ones such as prefetching, multithreading, and other techniques discussed in [5]. Recently, however, there have been many proposals for novel processor microarchitectures that substantially increase the number of in-flight instructions [1, 3, 7, 12, 14, 16, 22, 25]. They typically leverage state checkpointing and, sometimes, retirement of speculative instructions. Unsurprisingly, they also *dramatically* increase MLP requirements (e.g., [22] assumes support for 128 outstanding L1 misses).

One of them, Runahead execution [16], checkpoints the processor and retires a missing load, marking the destination register as invalid. The instruction 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 [25] 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 [3] and CLEAR [12] 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.

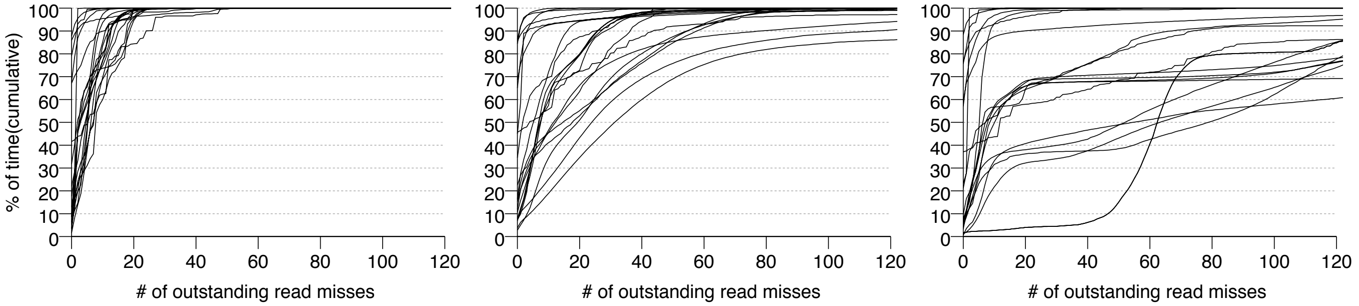
CPR [1] and Out-of-order Commit [7] processors remove scalability bottlenecks from the I-window to substantially increase the number of in-flight instructions. They remove the ROB, relying on checkpointing (e.g., at low-confidence branches) to recover in case of misspeculation. CFP [22] frees the resources 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.

2.3. Vector MHAs

Vector machines have MHAs that differ markedly from the ones that we will consider here. The reason is two fold. First, classical vector machines have memory systems that return misses in FIFO order. As a result, the MSHR file does not need to support associative searches and can be a simple, large FIFO file [6] — e.g., supporting 384 outstanding misses in the Cray SV1. In superscalar machines, we cannot afford such expensive memory systems and, therefore, need associative, more complex MHAs. The second difference is that many vector machines, such as the Cray SV1, have one-word lines, which simplifies the MHA substantially. Even in vector designs that use multi-word lines, such as the Cray X1 [8], the use of vector loads/stores (and vector prefetches) enables a simpler MHA design targeted to relatively few secondary misses. Our MSHR designs have to support many secondary misses, as will be shown later.

2.4. Why Not Reuse the Load/Store Queue State?

The microarchitectures of Section 2.2 generate a large number of concurrent memory system accesses. These accesses need support at two different levels, namely at the load/store queue (LSQ) and at

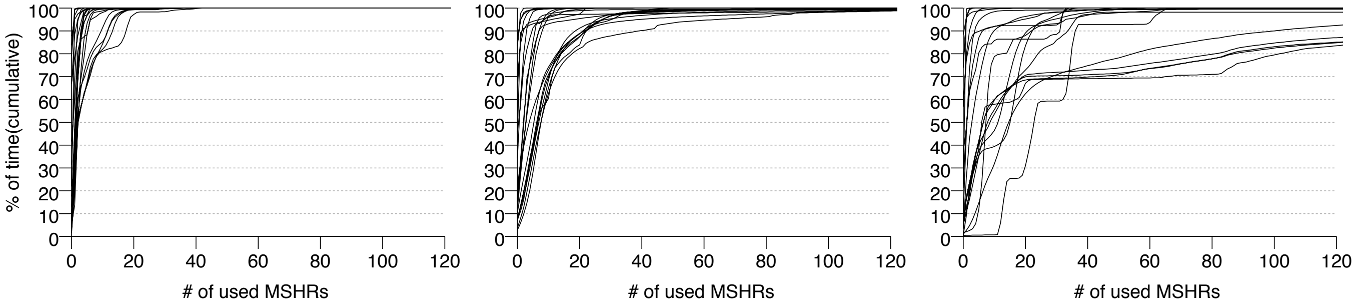


(a) Conventional

(b) Checkpointed

(c) LargeWindow

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



(a) Conventional

(b) Checkpointed

(c) LargeWindow

Figure 3. Number of L1 MSHR entries in use at a time in *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c). Only read misses are considered.

the cache hierarchy level. First, they need an 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 [10, 17, 20], the problem remains unexplored at the MHA level. Our paper addresses this problem.

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

However, we argue that this is not 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 redesigned) would induce a costly global LSQ search. Such search is eliminated if the MHA is enhanced with subentry pointer information.

Second, some of these novel microarchitectures speculatively *retire* instructions and, therefore, deallocate their LSQ entries [3, 12]. Consequently, the MHA cannot rely on information in LSQ entries because, by the time the miss completes, the entries may be gone.

Finally, LSQs are timing-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 keep primary and secondary miss information in the MHA and rely on no specific LSQ design.

3. Requirements for the New Miss Handling Architectures (MHA)

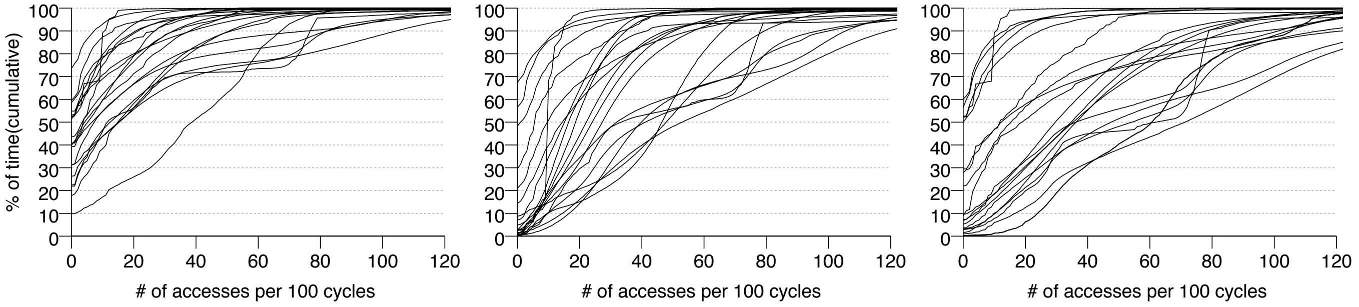
The MLP requirements of the new microarchitectures described put major pressure on the cache hierarchy, especially at the L1 level. To handle it, we can use known techniques, such as banking the L1 and making it set-associative. However, a lesser known yet acute problem remains, namely that the MHA in the L1 is asked to store substantially more information and sustain a higher bandwidth than in conventional designs. This is a new challenge.

In this section, we assess this challenge by comparing three processors: *Conventional*, *Checkpointed*, and *LargeWindow*. *Conventional* is a 5-issue, 2-context SMT processor slightly more aggressive than current ones. *Checkpointed* is *Conventional* enhanced with checkpoint-based value prediction like CAVA [3]. On L2 misses, it checkpoints, predicts the value and continues, retiring speculative instructions. *LargeWindow* is *Conventional* with an unrealistic 512-entry instruction window and 2048-entry ROB. All processors have a 32-Kbyte 2-way L1 organized in 8 banks. The bandwidth of the memory bus is 15GB/s. The rest of the parameters are in Table 4 and will be discussed in Section 6. For this section only, we model an MHA with ideal characteristics: unless otherwise indicated, it has an unlimited number of entries (MSHRs), and an unlimited number of subentries per MSHR. A summary of this section plus additional details are available in [4].

3.1. The New MHAs Need High Capacity

Figure 2 shows the distribution of the number of outstanding L1 read misses at a time. It shows the distributions for the three processors. Each line corresponds to one workload, which can be either one or two concurrent applications from SPECint2000 and SPECfp2000 — the workloads will be discussed in Section 6.2.

We see that, for *Conventional*, most workloads have 16 or fewer outstanding load misses 90% of the time. These requirements are



(a) Conventional

(b) Checkpointed

(c) LargeWindow

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

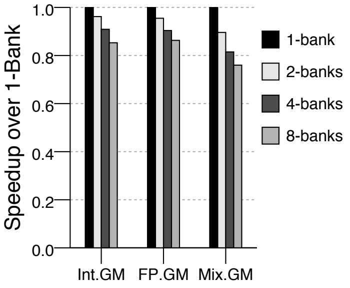
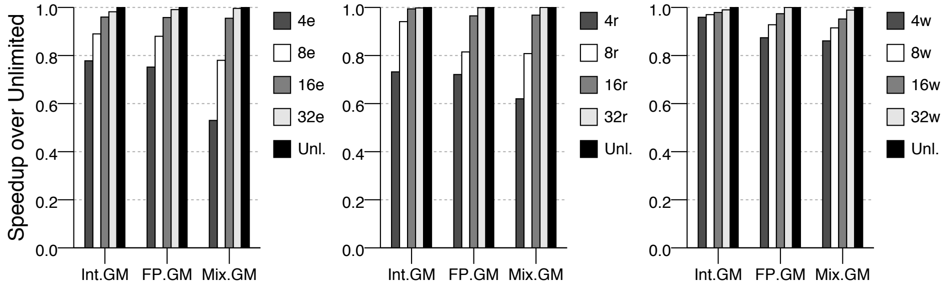


Figure 5. Performance impact of varying the number of MHA banks. Banks have unlimited bandwidth.



(a)

(b)

(c)

Figure 6. Performance impact of varying the number of entries (a), read subentries (b), and write subentries (c). In each case, we vary one parameter and keep the other two unlimited. The MHA has unlimited bandwidth.

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 some workloads sustaining 120 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 95% of the time. However, *Checkpointed* and *LargeWindow* have a greater demand for entries. For *Checkpointed*, we need about 32 MSHRs to have a similar coverage for most workloads. *LargeWindow* needs even more.

3.2. The New MHAs Need High Bandwidth

The MHA is accessed at two different times. First, when an L1 miss is declared, the MHA is read to see if it contains an MSHR for the accessed line. In addition, at this time, if the L1 miss cannot be satisfied by data forwarded from the MHA, the MHA is updated to record the miss. Second, when the requested line arrives from the L2, the MHA is accessed again to pull information from the corresponding MSHR and then clear the MSHR. Based on the assumed width of the MHA ports in this experiment, we need one access per write subentry or per four read subentries.

We compute the number of MHA accesses for both read and write misses during 100-cycle intervals for *Conventional*, *Checkpointed*, and *LargeWindow*. Figure 4 shows the distribution of the number of accesses per interval. For *Conventional*, many workloads have at most 40 accesses per interval about 90% of the time. For *Checkpointed*, the number of accesses to reach 90% of the time is often around 60. For *LargeWindow*, still more accesses are required to reach 90%. Overall, new MHAs need to have higher bandwidth than current ones.

3.3. Banked MHAs May Suffer From Access Imbalance Lock-ups

To increase MHA bandwidth, we can bank it like the L1 cache (Figure 1(b)). However, since the number of MSHRs in the MHA is limited due to area constraints, heavy banking may be counterproductive: 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. Eventually, the processor may stall. This problem is analogous to a cache bank access conflict in a banked L1 [11], except that a "conflict" in a fully-associative MHA bank may last for as long as a memory access time.

To assess this problem, we use an MHA with 16 MSHRs (unlimited number of subentries) and we run experiments grouping them into different numbers of banks: 1 bank of 16 entries, 2 of 8, 4 of 4, or 8 banks of 2 MSHRs. In all cases, a bank is fully associative and has no bandwidth limitations. The L1 has 8 banks.

Figure 5 shows the resulting performance of *Checkpointed*. The figure has three sets of bars, which correspond to the geometric mean of the SPECint2000 applications used (*Int.GM*), SPECfp2000 (*FP.GM*), and multiprogrammed mixes of both (*Mix.GM*). The figure shows that, as we increase the number of MHA banks, the performance decreases. Since a bank has unlimited bandwidth, contention never causes stalls. Stalls occur only if an MHA bank is full. Therefore, we need to be wary of banking the new MHAs too much since, if each MHA bank has modest capacity, access imbalance may cause cache bank lock-up.

3.4. The New MHAs Need Many Entries, Read Subentries, and Write Subentries

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 for reads and for writes. To assess the needs in number of entries, read subentries, and write subentries, we use a single-bank MHA with unlimited bandwidth in *Checkpointed*. We 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 6(a), we vary the number of MSHR entries. Our workloads benefit significantly by going from 8 to 16 MSHRs, and to a lesser extent by going from 16 to 32 MSHRs. In Figure 6(b), we vary the number of read subentries per MSHR. Secondary read misses are frequent, and supporting less than 16-32 read subentries hurts performance. Finally, in Figure 6(c), we vary the number of write subentries per MSHR. Secondary write misses are also important, and we need around 16-32 write subentries.

An additional insight is that individual MSHRs typically need read subentries or write subentries, but less frequently both kinds. This data is shown in Table 1 for *Checkpointed* running with an unlimited MHA. This behavior is due to the spatial locality of read and write misses. We will leverage it in Section 5.5.

Workload	Number of used MSHRs (%)		
	Read Sub Only	Write Sub Only	Read+Write Sub
Int Avg.	67.8	26.5	5.7
FP Avg.	85.3	10.8	3.9
Mix Avg.	85.1	10.9	4.1
Total Avg.	79.4	16.1	4.6

Table 1. Usage of MSHRs in *Checkpointed*.

4. An MHA for High MLP

Given these requirements for the MHA, it follows that current designs such as the one in Pentium 4, which only support 8 outstanding misses (of primary or secondary type) at a time [2], will be insufficient. One solution is to build a large, associative, centralized MSHR file. We call this design *Unified* (Table 2). It has a high capacity and does not cause L1 bank lock-up due to access imbalance (Section 3.3). However, its centralization limits its bandwidth.

Design	Characteristics
Current	8 outstanding L1 misses like Pentium 4
Unified	Large centralized associative MSHR file with many entries and subentries
Banked	Large associative MSHR file with many entries and subentries, banked like L1
Hierarchical	Two-level design with a small per-bank MSHR file (*Dedicated*) and a larger MSHR file shared by all banks (*Shared*)

Table 2. MHA designs considered.

Another solution is a large, associative MSHR file that is banked like the L1 cache. We call this design *Banked*. It has higher bandwidth than *Unified*. However, under program behavior with access imbalance, one of the banks may fill up, causing L1 bank lock-up.

We address these shortcomings with our proposed two-level MHA design. We call it *Hierarchical* (Table 2). It has a small per-bank MSHR file called *Dedicated* and a larger MSHR file shared by all banks called *Shared*. The Dedicated and Shared files have exclusive contents and are accessed sequentially. Entries that overflow from a Dedicated file are collected in the Shared file. Figure 7 shows the design. Table 3 lists our scheme's key innovations, which we consider next.

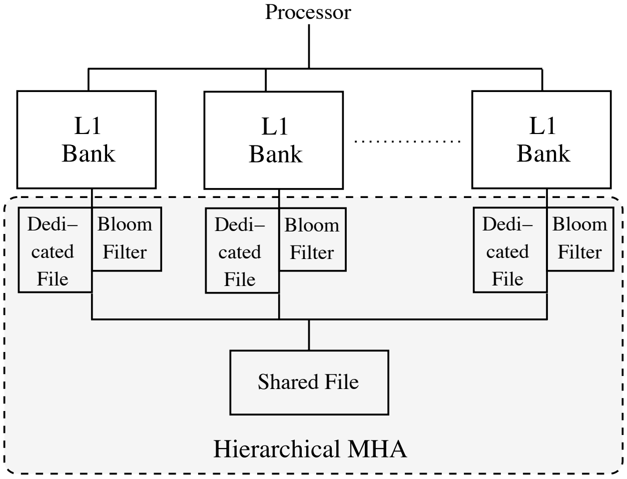


Figure 7. Proposed hierarchical MHA.

4.1. Minimize L1 Lock-up With Area Efficiency

A key goal of any MHA is to minimize the time during which it is out of MSHR file entries or subentries, while using the area efficiently. The latter is important because the MHA uses area close to the processor core. Consequently, we prefer MHA designs that minimize L1 lock-up time per unit area.

Banked does not use area very efficiently, since its capacity is divided into banks. If there is significant access imbalance across banks, one MHA bank can fill up and lock-up the cache bank.

Hierarchical uses area efficiently for two reasons (Table 3). First, it provides shared capacity in the Shared file — for a given need, one shared structure is more efficient than several private ones of the same total combined size, when there is access imbalance across structures. Second, a common reason for lock-up is that a few MSHRs need more subentries than the others and run out of them. In *Hierarchical*, rather than giving a high number of subentries to all the MSHRs, we design the Dedicated files with fewer subentries. This enables area savings. If an entry in a Dedicated file runs out of subentries, it is displaced to the Shared file.

4.2. High Bandwidth

The other key goal of any MHA is to deliver high bandwidth. *Unified*, due to its centralization, has modest bandwidth.

Hierarchical attains higher bandwidth through three techniques (Table 3). First, it provides a per-bank Dedicated file. Second, when a new MSHR is needed, it is always allocated in the Dedicated file. If there is no free entry in the Dedicated file, we displace one to the Shared file to open up space. We select as victim the entry that was inserted first in the Dedicated file (FIFO policy). Due to the spatial locality of cache misses, a primary L1 cache miss is often quickly followed by a series of secondary misses on the same line. With our policies, these secondary misses are likely to hit in the Dedicated file. The result is higher bandwidth and lower latency.

Finally, each bank also includes a small Bloom filter. It hashes the addresses of all the MSHR entries that were displaced from that bank's Dedicated file to the Shared file. The filter in a bank is ac-

Goal	Proposed Solution
Minimize L1 lock-up while using MHA area efficiently	– *Shared* file – Fewer subentries in *Dedicated* files; more subentries in *Shared* file
High bandwidth	– Per-bank *Dedicated* file – Allocate new entries always in *Dedicated* file: If entry is in MHA, locality typically ensures that it is found in *Dedicated* file – Bloom filter for *Shared* file (no false negatives, few false positives): If entry is not in MHA, filter typically averts access to *Shared* file

Table 3. Innovations in the proposed hierarchical MHA.

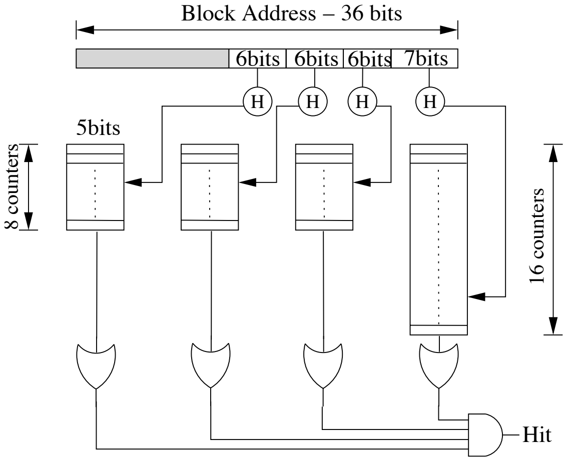


Figure 8. Per-bank Bloom filter in the proposed *Hierarchical* design. In the figure, *H* represents a hash operation.

cessed at the same time as the Dedicated file, and takes the same time to respond. When the Dedicated file misses, the filter indicates whether or not the requested entry may be in the Shared file. If the filter says "no", since a Bloom filter has *no false negatives*, the Shared file is not accessed. This saves a very large number of unneeded accesses to the Shared file, enhancing the MHA bandwidth. If the filter says "yes", the Shared file is accessed, although there may be a small number of false positives.

5. Implementation

This section describes several implementation aspects of our *Hierarchical* proposal: the overall organization and timing, the Bloom filter, the Dedicated file replacement algorithm, the implementation complexity, and the MSHR organizations.

5.1. Overall Organization and Timing

Each Dedicated file is fully pipelined and has a single read/write port. The Dedicated file and the filter in the same bank are accessed in parallel (Figure 7). In most cases, the outcome is either a hit in the Dedicated file or a miss in both the Dedicated file and filter. The first case is a secondary L1 miss intercepted by the Dedicated file. The second case is a primary L1 miss, in which case the Dedicated file allocates an entry and a request is sent to L2.

When a cache miss hits in the Bloom filter and misses in the Dedicated file, the Shared file is accessed. Since this case is less frequent, the Shared file is a single-ported, slow and large structure. It is highly associative and unpipelined. Each entry has many subentries to support many outstanding secondary misses per line.

5.2. Bloom Filter

We use a Bloom filter without false negatives, although some false positives can occur. Some Bloom filter designs require that the filter be periodically purged, so that aliasing does not create too many false positives. However, retraining the filter during operation could lead to false negatives. Consequently, we choose a counter-based Bloom filter design similar to the counter array in [15], which requires no purging. Every time that an entry is displaced from the corresponding Dedicated to the Shared file, a set of counters are incremented to add the address to the filter. The same counters are decremented when the entry is deallocated from the Shared file. The counters to increment or decrement are determined by several bit-fields in the line address. Finally, an access hits in the filter if all the counters corresponding to the address being checked are non-zero.

Figure 8 shows the structure of the filter. In the figure, the bits in the address bit-fields are hashed before using them to index into the arrays of counters.

5.3. Replacement of Entries in the Dedicated File

Since each Dedicated file only has a handful of MSHRs, it is easy to implement a FIFO replacement algorithm. Moreover, once we move the contents of an MSHR to the Shared file, we never move it back to a Dedicated file. This general policy leverages the spatial locality of misses, to capture the active entries (those with frequent secondary misses) in the Dedicated files, and push the inactive entries (typically corresponding to long-latency main memory accesses) to the Shared file.

5.4. Complexity of the Hierarchical Implementation

Hierarchical is simple to implement. To start with, no changes are needed to the cache interface. Compared to other MHAs, any complexity of *Hierarchical* might come from four sources: allocating MSHRs, displacing them into the Shared file, handling replies from memory, and supporting the Bloom filter. Since the filter is a simple counter array that adds little complexity, we focus on the first three concerns.

Before allocation, the MHA uses an *Available* signal to tell the cache bank that it has space to take in a new request. This signal is the logical OR of one signal coming from the Dedicated file in the bank and one from the Shared file. If neither file has space (*Available* is false), the cache bank locks-up. Otherwise, allocation proceeds in the Dedicated file. This step may involve a displacement.

The complexity of a displacement lies in solving three races or problems: (i) two Dedicated files want to displace into the same Shared file entry; (ii) an MSHR is needed while it is in transit from the Dedicated to the Shared file; and (iii) an entry being displaced finds that, despite initial indications to the contrary, there is no space in the Shared file. We avoid these problems using a simple algorithm. Specifically, on a displacement, the Dedicated file retains the MSHR being displaced and the corresponding bank of the L1 is locked-up to incoming requests, until the Shared file reports that it has taken in the data and the filter reports that it has been updated. Moreover, during the actual data transfer, the MSHR being displaced stays in the Ded-

Processors		
All	*Conventional* and *Checkpointed*	*LargeWindow*
Frequency: 5GHz	I-window: 92	I-window: 512
Fetch/issue/comm width: 6/5/5	ROB size: 192	ROB size: 2048
LdSt/Int/FP units: 4/3/3	Int regs: 192	Int regs: 2048
SMT contexts: 2	FP regs: 192	FP regs: 2048
Branch penalty: 13 cyc (min)	Ld/St Q entries: 60/50	Ld/St Q entries: 768/768
RAS: 32 entries		
BTB: 2K entries, 2-way assoc.	*Checkpointed* only:	
Branch pred. (spec. update):	Val. pred. table: 2048 entries	
bimodal size: 16K entries	Max outs. ckps: 1 per context	
gshare-11 size: 16K entries		

Memory System			
	I-L1	D-L1	L2
Size:	32KB	32KB	2MB
Assoc:	2-way	2-way	8-way
Line size:	64B	64B	64B
Round trip:	2 cyc	3 cyc	15 cyc
Ports/Bank:	2	1	1
Banks:	–	8	–
HW prefetcher: 16-stream strided (Between L2 and memory)			
Mem bus bandwidth: 15GB/s			
Round trip to memory: 500 cyc			

Table 4. Processors simulated. In the table, latencies are shown in processor cycles and represent minimum values.

icated file in locked state (inaccessible). If the transfer temporarily fails, the MSHR in the Dedicated file is unlocked, but the L1 bank remains locked-up until the whole process completes.

For replies coming from memory, we reuse the path just described. Replies first check their home Dedicated file and, if they miss, they then check the Shared file. If a reply finds its corresponding MSHR in the locked state, it stalls until the MSHR is unlocked.

Overall, based on this discussion, we feel that *Hierarchical* has a simple implementation.

5.5. MSHR Organizations for High MLP

We consider three different MSHR organizations. They extend Farkas and Jouppi's [9] Explicit and Implicit organizations. However, Farkas and Jouppi's MSHRs only record read information, since their caches are write-through and no-allocate. In our case, MSHRs also need to record write information. Kroft's design allocates an empty line in the cache immediately on a miss [13]. As a result, a write on a pending line deposits the update in the empty cache line. In our case, cache misses can take a long time to complete. Therefore, we do not want to allocate an empty line in the cache right away. Instead, we design MSHR organizations with subentries that keep information on many read and write misses on the line.

The first of our three organizations is *Simple* (Figure 9). An MSHR includes an array of *N* explicit subentries, where each one can correspond to a read or a write. A read subentry stores the line offset and the destination register; a write subentry stores the line offset and a pointer into a companion N-entry data buffer where the update is stored. This organization leverages our observations in Section 3.4 that MSHRs need to hold many read and/or write subentries. While this organization is simple, it has two shortcomings. First, to check for forwarding on a read, all the subentries in the MSHR need to be examined, which is time consuming. Second, this design consumes substantial area, since the data buffer needs to be very large in case all subentries are writes.

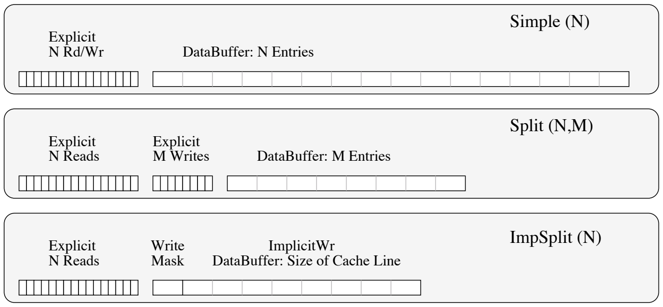


Figure 9. Three different MSHR organizations.

The *Split* organization separates *N* explicit read from *M* explicit write subentries (Figure 9). This design is motivated by the observation in Table 1 that many MSHRs do not need write subentries. Consequently, we reduce the number of supported write subentries to *M*, and only need *M* entries in the data buffer. This design improves area efficiency if *M* is significantly smaller than *N*. However, *Split* has the shortcomings of expensive checks for forwarding (like in *Simple*), and that it causes a stall if an MSHR receives more than *M* writes.

To solve these problems, *ImpSplit* keeps the explicit organization for *N* read subentries, but uses the implicit organization for writes (Figure 9). Each MSHR has a buffer for writes that is as large as a cache line, and a bit-vector mask to indicate which bytes have been written. Writes deposit the update at the correct offset in the buffer and set the corresponding bits in the mask. Multiple stores to the same address use a single buffer entry because they overwrite each other. Forwarding is greatly simplified because it only requires reading from the correct offset in the buffer. Moreover, this organization supports any number of writes at the cost of a buffer equivalent to a cache line; for the numbers of secondary write misses that we see in our experiments, this is area-efficient.

6. Experimental Setup

We use execution-driven simulations to evaluate the MHA designs of Table 2 for the *Conventional*, *Checkpointed*, and *LargeWindow* processors. The architecture of the processors is shown in Table 4. *Conventional* is a 5-issue, 2-context SMT processor. *Checkpointed* extends *Conventional* with support for checkpoint-based value prediction like CAVA [3]. Its additional parameters in Table 4 are the Value Prediction Table and the maximum number of outstanding checkpoints. Each hardware thread has its own checkpoint and can rollback 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 4), including two levels of on-chip caches. The exception is that *Checkpointed* has the few extensions required by the CAVA support [3], namely speculative state in L1 and a table with the value of the predictions made. All designs have a 16-stream strided hardware prefetcher that prefetches into L2 and, therefore, does not use L1 MSHRs.

The evaluation is performed using the cycle-accurate SESC simulator [18]. SESC models in detail the processor microarchitectures and the memory systems.

6.1. Comparing MHAs That Use the Same Area

MHAs offer a large design space from which a designer must choose. In this paper, we compare different designs that use the *same area*. We think that this is a fair constraint. We consider three de-

Area Design Point	MHA Design	MSHR Organization	Number of MSHRs	Assoc.	Tag & Data Size (Bytes)	Approx. Area (% of L1)	Cycle Time (Proc Cyc)	Access Time (Proc Cyc)
Ordinary-Sized: 8% of L1 Area	Unified	ImpSplit(24)	8	Full	1029	8	2	4
	Banked	ImpSplit(8)	2x8banks=16	Full	1550	9	1	3
	Hierarchical	Dedicated: ImpSplit(4)	1x8banks=8	Full	1227	8	1	3
		Shared: ImpSplit(24)	4	Full			2	4
Medium-Sized: 15% of L1 Area	Unified	ImpSplit(32)	32	Full	4620	15	2	4
	Banked	ImpSplit(8)	3x8banks=24	Full	2325	15	1	3
	Hierarchical	Dedicated: ImpSplit(8)	2x8banks=16	Full	2379	15	1	3
		Shared: ImpSplit(24)	8	Full			2	4
Large-Sized: 25% of L1 Area	Unified	ImpSplit(32)	48	Full	6930	24	2	4
	Unified 2-ports	ImpSplit(8)	8	Full	773	25	2	4
	Banked	ImpSplit(12)	4x8banks=32	Full	3352	26	1	3
	Hierarchical	Dedicated: ImpSplit(8)	2x8banks=16	Full	5885	24	1	3
		Shared: ImpSplit(32)	30	Full			2	4
Other MSHR Organizations at 15% of L1 Area	Banked	Simple(10)	3x8banks=24	Full	2505	15	1	3
	Banked	Split(8,8)	3x8banks=24	Full	2514	16	1	3
	Hierarchical	Dedicated: Simple(8)	2x8banks=16	Full	2379	15	1	3
		Shared: ImpSplit(32)	8	Full			2	4
	Hierarchical	Dedicated: Split(8,8)	2x8banks=16	Full	2065	16	1	3
		Shared: ImpSplit(24)	8	Full			2	4

Table 5. Area, cycle time, and access time for the MHA designs and MSHR organizations considered.

sign points: a *Medium-Sized* design, where the MHA uses an area equivalent to 15% of the area of our 8-bank 32-Kbyte L1 cache; a *Large-Sized* design, where it uses the equivalent of 25% of the cache area; and an *Ordinary-Sized* design where, like the Pentium 4 MSHR file, the MHA uses the equivalent of ≈8% of the cache area.

To estimate area, we use the newly available CACTI 4.1 [23]. This version of CACTI is more accurate than the previous 3.2 version, as it has been specifically designed for nanoscale technologies such as the 65nm one considered here. Appendix A gives details on our CACTI runs. As of September 2006, the authors of CACTI acknowledge a bug in the area calculation for a banked cache. For our experiments, we have modified CACTI 4.1 to correct the bug.

We consider combinations of MHA design, MSHR organization, number of MSHRs, number of subentries, and associativity that match each of the three target area points, or come very close. We have an automated script that generates all possible combinations, and computes area, cycle time, and access time. We then use our cycle-accurate processor-memory simulator to evaluate the overall performance of workloads using each design.

Table 5 lists the best designs that we have found. Column 1 shows the four sets of experiments that we perform. The top three compare MHA designs that take the equivalent of 8%, 15%, and 25% of the L1 area, respectively. For each experiment, we use the best *Unified*, *Banked*, and *Hierarchical* designs — although, for the 25% area experiment, we consider two different *Unified* designs, as we will discuss. The third column shows the MSHR organization used, with the number of explicit subentries in parenthesis as in Figure 9. We find that all the best designs use the *ImpSplit* MSHR organization. For completeness, we also perform a fourth experiment (last row of Column 1 of Table 5) comparing designs that use *Simple* and *Split* organizations.

For each MHA design and MSHR organization, Table 5 shows the number of MSHRs used (Column 4), their associativity (Column 5), the size of the tag and data arrays in bytes (Column 6), the area of the tag and data arrays as a fraction of the L1 area (Column 7), the cycle time (Column 8), and the access time (Column 9). All cycle counts are in processor cycles. The size and area of *Hierarchical* are the addition of the contributions of the Dedicated and Shared files. We pipeline all these structures except the Shared file in *Hierarchical*.

The L1 is a 32 Kbyte 2-way cache organized in 8 banks with 1 read/write port per bank. Such a cache in 65nm technology is estimated to be 0.6965 mm^2. We simulate it with a cycle time of 1 cycle and an access time of 3 cycles.

6.2. Workloads

We run our experiments using SPECint2000 codes, SPECfp2000 codes, and workload mixes that combine two applications at a time (Table 6). From SPECint, our infrastructure does not support *eon*, which is written in C++. Moreover, there are 7 SPECint codes that have so few misses that a *perfect* MHA (unlimited number of MSHRs, subentries, and bandwidth) makes not even a 5% performance impact in any of the architectures analyzed. Consequently, we

SPECint2000	SPECfp2000
256.bzip2 (*bzip2*)	188.ammp (*ammp*)
254.gap (*gap*)	173.applu (*applu*)
181.mcf (*mcf*)	179.art (*art*)
253.perlbmk (*perlbmk*)	183.equake (*equake*)
	177.mesa (*mesa*)
	172.mgrid (*mgrid*)
	171.swim (*swim*)
	168.wupwise (*wupwise*)

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

Table 6. Workloads used in our experiments.

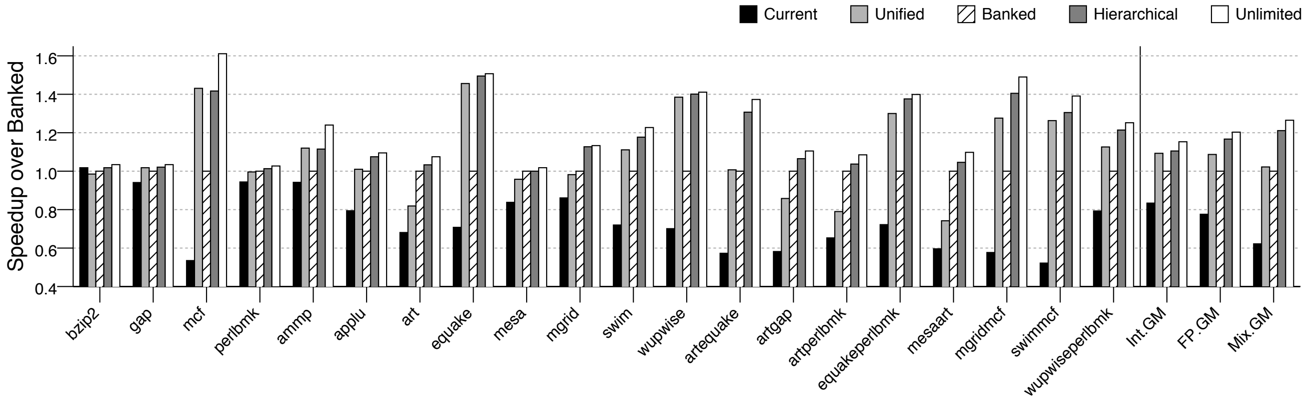


Figure 10. Performance of the different MHA designs at the 15% target area for the *Checkpointed* processor.

only analyze the remaining 4 SPECint codes in the rest of the paper. In the conclusion section, we average out the performance impact of our designs over the 11 SPECint applications that we can simulate.

From SPECfp, we use all the applications except for six that our infrastructure does not support (four that are in Fortran 90 and two that have unsupported system calls). Finally, for the workload mixes, the algorithm followed is to pair one SPECint and one SPECfp such that one has high MSHR needs and the other low. In addition, one mix combines two lows, two combine two highs, and two others combine 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 codes using gcc 3.4 -O3 into MIPS binaries and use the *ref* data set. We evaluate each program for 0.6-1.0 billion committed instructions, after skipping several billion instructions as initialization. To compare performance, we use committed IPC. When comparing the performance of multiprogramming mixes, we use weighted speedups as in [24].

7. Evaluation

In this evaluation, we compare the performance of the different MHA designs at the Medium-Sized area point (15% of the L1 area) and at other area points, characterize *Hierarchical*, and evaluate different MSHR organizations. Unless otherwise indicated, the MHA designs are those shown in the first three rows of Column 1 of Table 5. Also, plots are normalized to the performance of *Banked*.

7.1. Performance of MHA Designs at 15% Area

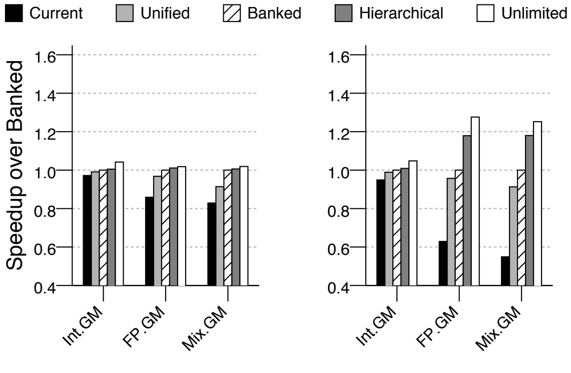
Figure 10 compares the performance of the different MHA designs of Table 5 at the 15% target area for the *Checkpointed* processor. As a reference, the figure also includes *Current* and *Unlimited*. The former is a design like that of Pentium 4 (Table 2); the latter is an infeasible MHA design that supports an unlimited number of outstanding misses with unlimited bandwidth. The rightmost three sets of bars in the figure show the geometric mean of the integer, FP, and mix workloads.

The *Current* design is much worse than the other MHAs for *Checkpointed* processors. Such processors are bottlenecked by *Current* and need aggressive MHA designs. For example, *Hierarchical* speeds-up execution over *Current* by a geometric mean of 32% for SPECint, 50% for SPECfp, and 95% for mixes. These are substantial speedups.

Among the aggressive designs, *Hierarchical* performs the best, and is very close to *Unlimited*. Compared to *Unified*, *Hierarchical* has lower capacity but, thanks to the three techniques of Table 3, it offers higher bandwidth to accesses. As a result, *Hierarchical* is faster than *Unified* by a geometric mean of 1% (SPECint), 7% (SPECfp), and 18% (mixes). The speedups are highest for high-MLP scenarios, such as when these SMT processors run a multiprogrammed load.

On average, *Banked* is worse than *Unified*. This is because the higher bandwidth that it provides is not fully leveraged due to access imbalance (Section 3.3). There are some exceptions, such as the workloads with *art*, which benefit more from higher bandwidth than are hurt by imbalance. On average, *Hierarchical* is faster than *Banked* for the three workload groups by a geometric mean of 10%, 16%, and 21%. Overall, *Hierarchical* delivers significant improvements over the other aggressive designs for a very modest complexity (Section 5.4).

For completeness, we also examine the effect of MHA designs on *Conventional* and *LargeWindow* processors, although we only show the geometric means. They are shown in Figures 11(a) and 11(b), respectively. With *Conventional* processors, the performance difference between *Current* and the aggressive designs is much smaller. This shows that state-of-the-art, relatively low-MLP processors cannot leverage aggressive MHAs as much. Still, *Hierarchical* and *Banked* are consistently the best.



(a) (b)

Figure 11. Performance of the different MHA designs at the 15% target area for the (a) *Conventional* processor and (b) *LargeWindow* processor.

With *LargeWindow* processors (Figure 11(b)), we again see that *Current* bottlenecks the processor, and that *Hierarchical* is significantly faster than the other two aggressive MHA designs — *Hierarchical* is faster than *Unified* by 2%, 23%, and 29% in the three workload groups. In this processor, the profile of speedups is somewhat different than in *Checkpointed* — most notably, *Banked* is better than

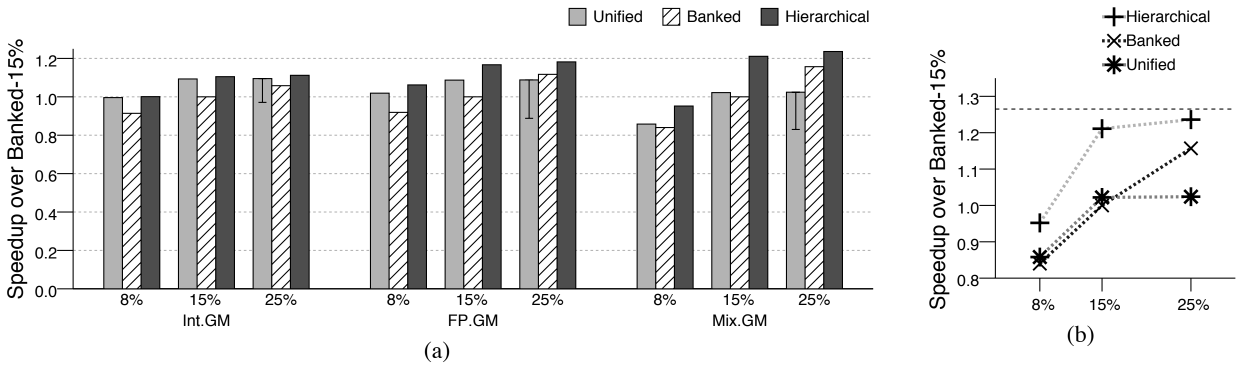


Figure 12. Performance of the *Checkpointed* processor for different MHA designs, and target areas 8%, 15% and 25% (a). The line segments in the bars for 25% *Unified* indicate the reduction in performance if we use the dual-ported *Unified* design. Chart (b) repeats the information in *Mix.GM* of Chart (a) in a different format. The dashed horizontal line shows the performance of *Unlimited*.

Unified, and SPECint codes show smaller speedups. The reason for the first effect is that *LargeWindow*'s outstanding misses require more bandwidth than the *Checkpointed* ones (Figure 4). Consequently, *Banked* works relatively better. The reason for the second effect is that, thanks to value prediction and speculative retirement, *Checkpointed* presents a longer effective window than *LargeWindow* for SPECint codes, which enables higher speedups for these codes. Overall, these results show that processors other than *Checkpointed* can also use aggressive MHA designs. Most likely, any high-MLP architecture will benefit from aggressive MHA designs.

7.2. Performance at Different Area Points

Hierarchical maintains its performance advantage over *Banked* and *Unified* across a wide range of area points. This is seen for *Checkpointed* in Figure 12. Figure 12(a) is organized per workload type. For each workload, as we go from left to right, we increase the target area from 8% (Ordinary-Sized design) to 15% (Medium-Sized design) and 25% (Large-Sized design). In each workload, the bars are normalized to *Banked* with 15%.

At each workload and area point, *Hierarchical* is the fastest design. Moreover, *Hierarchical* at the 15% target area is better than *Unified* or *Banked* at the 25% target area — which use much more area.

Unified is most competitive at the 8% target area, where its better use of area relative to the other designs has the highest impact. As the target area scales up, however, the performance of *Unified* levels out, even though its capacity is the highest (Table 5). The lower bandwidth of *Unified* prevents it from exploiting its higher capacity. To address this problem, we also evaluated a second design for *Unified* at the 25% target area: one with two ports (Table 5). This *Unified* design has higher bandwidth but, to keep the area constant, we have to reduce its number of MSHRs significantly to 8. The performance of this design is shown in Figure 12(a) as the lower end of the line segments in the *Unified* 25% bars. We can see that the performance is always lower than the *Unified* 25% single-ported design. Even though the dual-ported design has higher bandwidth, it is crippled by its low capacity.

If we focus on *Banked*, we see that it is unattractive for the 8% and 15% target areas. This is because it suffers from load imbalance due to its low per-bank capacity. As it gains capacity at the 25% target area, it outperforms *Unified* for *FP* and *Mix* workloads, although it does not match *Hierarchical* yet.

Figure 12(b) repeats the information in *Mix.GM* of Figure 12(a) in a format that emphasizes the scaling trends of each design. In the figure, the performance of *Unlimited* is shown as the dashed horizontal bar. *Hierarchical* offers the highest performance across all area points, obtaining close to *Unlimited* performance already at 15% area. *Unified*'s performance saturates at around the 15% target area due to limited bandwidth. Only by adding a second port at the cost of much higher area can *Unified* achieve better performance. Finally, *Banked* improves its performance as we keep increasing the target area. Eventually, there may be a point where it will match the performance of *Hierarchical*. However, such a design point will be much less area-efficient than the ones presented here.

7.3. Characterization of Hierarchical at 15% Area

Table 7 characterizes *Hierarchical* for *Checkpointed* at the 15% target area. The first group of columns (Columns 2-4) shows how the L1 misses are processed by the different files in *Hierarchical*. Misses can be of three types: primary (Column 2), secondary that hit in a Dedicated file (Column 3), and secondary that hit in the Shared file (Column 4). Primary misses create an entry in a Dedicated file. Of the three types of misses, only the last one involves storing information on the miss in the Shared file. Overall, we see that this happens for only 17-23% of the L1 misses on average.

Column 5 shows the effectiveness of the Bloom filter at saving accesses to the Shared file. The numbers listed are the fraction of primary misses that are prevented from accessing the Shared file by the Bloom filter — in other words, the fraction of *unnecessary* accesses to the Shared file that are eliminated by the Bloom filter. They are unnecessary because they would miss in the Shared file anyway. The numbers shown are averages across all banks. From the table, we see that the Bloom filter eliminates on average 90-96% of useless accesses to the Shared file. It does not remove them all because of false positives. Overall, the Bloom filter ensures that the Shared file does not become a bottleneck.

The third group of columns (Columns 6-7) shows data regarding the displacement of entries from the Dedicated files to the Shared file. Column 6 shows the fraction of such displacements triggered by lack of subentries. This number is on average 18-41%. The other displacements are triggered by lack of entries. Column 7 shows the fraction of all the displacements that are also L2 cache misses. Such fraction is on average 33-42%. Consequently, the Shared file often holds long-latency misses.

The last group of columns shows information about the rest of the memory system: L1 and L2 miss rates, fraction of L1 read misses that get their data forwarded from an MSHR (a significant 15% for SPECints), and bus utilization.

Workload	L1 Miss Breakdown			Accesses	Displacement Stats		L1 Miss	L2 Miss	MSHR Fwd	Bus
	Primary	Dedicated Hit	Shared Hit	Removed	Sub Full	L2 Miss	Rate	Rate	L1 misses	Util
	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
bzip2	57.3	28.6	14.0	76.1	12.1	0.3	3.8	0.0	1.0	0.1
gap	17.9	39.1	43.0	99.0	80.1	47.2	0.8	0.2	49.1	4.4
mcf	39.2	37.9	22.8	95.4	27.3	84.1	15.9	7.6	4.6	45.8
perlbmk	34.2	58.7	7.2	96.6	43.0	35.9	3.3	0.2	6.6	2.7
ammp	40.5	55.5	4.1	56.1	29.7	52.3	18.4	1.7	0.8	4.5
applu	28.1	55.4	16.5	87.6	68.7	20.7	3.0	0.2	12.7	7.0
art	79.4	19.0	1.6	98.6	0.1	18.3	44.2	4.0	0.0	27.1
equake	26.3	30.6	43.1	96.3	59.2	69.2	5.3	1.7	2.0	23.3
mesa	35.2	51.5	13.3	97.6	33.9	21.2	4.8	0.2	7.3	4.8
mgrid	44.3	30.1	25.6	89.9	2.3	7.1	15.0	1.1	0.5	20.2
swim	39.0	32.1	28.9	99.8	24.7	49.9	7.3	2.8	0.7	45.1
wupwise	12.0	35.0	53.0	97.9	76.4	74.8	1.6	1.0	4.8	15.6
artequake	63.7	22.2	14.2	97.6	7.3	25.9	23.4	3.0	0.7	31.5
artgap	65.3	26.8	7.9	98.9	1.6	24.1	27.6	3.2	1.0	31.6
artperlbmk	72.8	22.4	4.8	97.9	1.1	17.3	31.3	2.6	0.4	24.5
equakeperlbmk	29.7	37.1	33.2	96.2	51.0	56.7	6.2	1.4	3.0	23.1
mesaart	67.7	25.0	7.3	98.1	2.5	15.6	28.2	2.2	0.8	25.1
mgridmcf	53.6	25.9	20.5	83.0	3.9	19.6	20.4	3.2	0.9	46.1
swimmcf	44.1	31.5	24.4	98.8	17.8	74.4	11.3	4.7	2.1	51.2
wupwiseperlbmk	26.4	47.9	25.7	98.0	61.8	28.4	4.9	0.3	8.6	8.7
Int.Avg	37.2	41.1	21.8	92.0	40.6	41.9	6.0	2.0	15.3	13.3
FP.Avg	38.1	38.6	23.3	89.9	36.9	39.2	12.5	1.6	3.6	18.5
Mix.Avg	52.9	29.8	17.3	95.9	18.4	32.7	19.2	2.6	2.2	30.2

Table 7. Characterization of the dynamic behavior of *Hierarchical* for *Checkpointed* at the 15% target area.

Finally, to assess the frequency of cache lock-up, Figure 13 shows the fraction of the time during which *at least one* of the cache banks is locked-up — because of lack of either entries (MSHRs) or read subentries (not due to write subentries because we use *ImpSplit* MSHRs). For each workload, the bars are normalized to *Banked*.

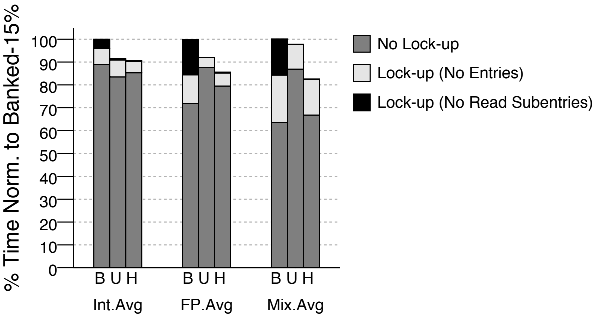


Figure 13. Breakdown of the execution time for *Banked* (B), *Unified* (U), and *Hierarchical* (H) at the 15% target area.

The figure shows that the fraction of time with lock-up tends to increase as we go from *Unified* to *Hierarchical* and to *Banked*. As discussed in Section 3.3, the reason is that load imbalance in banked MHA designs causes some banks to fill up sooner. However, while *Unified* typically has the least lock-up time, it does not have the highest performance; it is hurt by lower bandwidth. The figure also shows that most of the lock-up is due to lack of entries rather than subentries. Overall, *Hierarchical* performs the best because it accomplishes both high bandwidth and modest lock-up time.

7.4. Evaluation of Different MSHR Organizations

We now compare the effect of the different MSHR organizations of Figure 9 in *Banked* and *Hierarchical* for *Checkpointed* at the 15% target area. We compare the *ImpSplit*-based MHA designs used so far to the *Simple*- and *Split*-based MHA designs of the last row of Column 1 of Table 5. Figure 14 shows the performance of the different MHA designs normalized to *Banked-ImpSplit*.

In *Banked*, *Split* performs slightly worse than *ImpSplit* due to a lack of write subentries. *Simple* performs roughly as well as *ImpSplit*; even though it has a few more read subentries, it does not have as much capacity for write misses. In *Hierarchical*, the changes only have a small impact. Part of the reason is that the Shared file minimizes the differences. Overall, we choose the *ImpSplit* organization because it performs as well or better than the other organizations and supports a simpler implementation of read forwarding (Section 5.5).

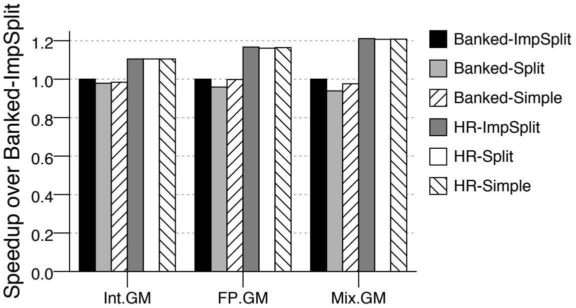


Figure 14. Performance of the *Checkpointed* processor with different MSHR organizations. The target area is 15%. In the figure, *HR* stands for *Hierarchical*.

8. Conclusions

Recently-proposed processor microarchitectures that require substantially higher MLP promise major performance gains. Unfortunately, the MHAs of current high-end systems are not designed to support the necessary level of MLP. This paper focused on designing a high-performance, area-efficient MHA for these high-MLP microarchitectures.

This paper made three contributions. First, it showed that state-of-the-art MHAs for L1 data caches are unable to leverage the new microarchitectures. Second, it proposed a novel, scalable MHA design that supports these microarchitectures. The proposal's key ideas are: (i) a *hierarchical* organization for high bandwidth and minimal cache lock-up time at a reasonable area cost, and (ii) a *Bloom filter* that eliminates most of the unneeded accesses to the large MSHR file.

The third contribution was the evaluation of our MHA design in a high-MLP processor. We focused mostly on a design point where the MHA uses an area equivalent to 15% of that of the L1 data cache. Compared to a state-of-the-art MHA, our design delivers geometric-mean speed-ups of 32% for a subset of SPECint (or a geometric mean of 11% for the 11 SPECint applications that we can simulate, including those with very few misses), 50% for SPECfp, and 95% for multiprogrammed loads. We also compared our design to two extrapolations of current MHA designs, namely *Unified* and *Banked*. For the same area, our design speeds-up the workloads by a geometric mean of 1-18% over *Unified* and 10-21% over *Banked* — all for a very modest complexity. Finally, our design performed very close to an unlimited-size, ideal MHA.

Acknowledgments

We thank the anonymous reviewers and the members of the I-ACOMA group at the University of Illinois for their invaluable comments. Also, we offer a special thanks to David Koufaty and Balaram Sinharoy for helpful discussions about current MSHR file implementations, and to Dennis Abts for feedback on an earlier version of this paper.

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*, November 2003.

[2] 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*, 8(1), February 2004.

[3] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Using Checkpoint-Assisted VAlue Prediction to Hide L2 Misses. *ACM Transactions on Architecture and Code Optimization*, June 2006.

[4] L. Ceze, J. Tuck, and J. Torrellas. Are We Ready for High Memory-Level Parallelism? In *4th Workshop on Memory Performance Issues*, February 2006.

[5] 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.

[6] Cray Computer. U.S. Patent 6,665,774, December 2003.

[7] 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*, February 2004.

[8] T. Dunigan, J. Vetter, J. White, and P. Worley. Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture. In *IEEE Micro Magazine*, January/February 2005.

[9] 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.

[10] 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.

[11] T. Juan, J. J. Navarro, and O. Temam. Data Caches for Superscalar Processors. In *Proceedings of the 11th International Conference on Supercomputing*, July 1997.

[12] 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*, February 2005.

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

[14] 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.

[15] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[16] 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*, February 2003.

[17] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[18] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[19] C. Scheurich and M. Dubois. The Design of a Lockup-free Cache for High-Performance Multiprocessors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, November 1988.

[20] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[21] G. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[22] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[23] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Laboratories Palo Alto, June 2006.

[24] 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.

[25] 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.

Appendix A: Details on the CACTI 4.1 Runs

To estimate areas, we use the newly available CACTI 4.1 [23]. We take each MSHR organization in Figure 9 and break it into its substructures (Table 8). In the table, letters *N* and *M* refer to the number of subentries as in Figure 9. Then, for each substructure and for the L1 Cache, we run CACTI. Table 9 shows some CACTI settings we use and parameters we pass to CACTI. In the table, BITOUT is the number of bits read out and Tag is the number of bits modeled for the tag array. An explicit subentry is assumed to take 2 bytes. For the DataBuffer(N) Substructures, we use the SRAM model, and for the others, we use the detailed cache interface. If we are modeling a banked MHA structure, we calculate the area of a single bank and then multiply by the number of banks. For the L1 cache, we use the (fixed) banking model in CACTI.

MSHR Organization	Substructures
Simple(N)	ExplicitRdWr(N) + DataBuffer(N)
Split(N,M)	ExplicitRd(N) + ExplicitWr(M) + DataBuffer(M)
ImpSplit(N)	ExplicitRd(N) + ImplicitWr + Mask

Table 8. MSHR organizations with their substructures.

Substructure	CACTI Settings	Command Line Parameters
ExplicitRd(N)	BITOUT=32, Tag=36	line=N*2, r/w port=1
ExplicitWr(N)	BITOUT=N*16, Tag=36	line=N*2, r/w port=1
ExplicitRdWr(N)	BITOUT=N*16, Tag=36	line=N*2, r/w port=1
ImplicitWr + Mask	BITOUT=64, Tag=42	line=64+8, r/w port=1
DataBuffer(N)	BITOUT=64, Tag=36	line=N*8, r/w port=1
L1 Cache (32KB)	BITOUT=64, Tag=42	line=64, r/w port=1, 2-way

Table 9. Some CACTI settings and command line parameters.

When a structure is too small for CACTI to directly model, we model larger structures with the same associativity and number of ports and approximate the area and timing using a linear regression. This technique is necessary for any structure smaller than 64 bytes or fully associative structures with four or fewer entries.