A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors

Magnus Jahre and Lasse Natvig

Norwegian University of Science and Technology
HiPEAC European Network of Excellence
{jahre,lasse}@idi.ntnu.no

Abstract. Chip Multiprocessors (CMPs) mainly base their performance gains on exploiting thread-level parallelism. Consequently, powerful memory systems are needed to support an increasing number of concurrent threads. Conventional CMP memory systems do not account for thread interference which can result in reduced overall system performance. Therefore, conventional high bandwidth Miss Handling Architectures (MHAs) are not well suited to CMPs because they can create severe memory bus congestion. However, high miss bandwidth is desirable when sufficient bus bandwidth is available. This paper presents a novel, CMP-specific technique called the Adaptive Miss Handling Architecture (AMHA). If the memory bus is congested, AMHA improves performance by dynamically reducing the maximum allowed number of concurrent L1 cache misses of a processor core if this creates a significant speedup for the other processors. Compared to a 16-wide conventional MHA, AMHA improves performance by 12% on average for one of the workload collections used in this work.

1 Introduction

Chip multiprocessors (CMPs) are now in widespread use and all major processor vendors currently sell CMPs. CMPs alleviate three important problems associated with modern superscalar microprocessors: diminishing returns from techniques that exploit instruction level parallelism (ILP), high power consumption and large design complexity. However, much of the internal structures in these multi-core processors are reused from single-core designs, and it is unclear if reusing these well-known solutions is the best way to design a CMP.

The performance gap between the processor and main memory has been growing since the early 80s [1]. Caches efficiently circumvent this problem because most programs exhibit spatial and temporal locality. However, adding more processors on one chip increases the demand for data from memory. Furthermore, latency hiding techniques will become more important and these tend to increase bandwidth demand [2].

This work was supported by the Norwegian Metacenter for Computational Science (NOTUR).

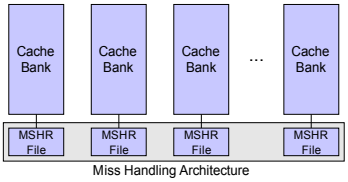


Fig. 1. Miss Handling Architecture (MHA) [4]

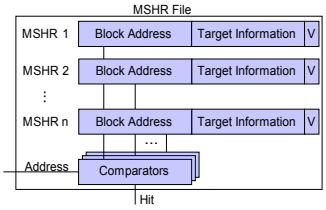


Fig. 2. A Generic MSHR File

A straightforward way of providing more bandwidth is to increase the clock frequency and width of the memory bus. Unfortunately, the number of pins on a chip is subject to economic as well as technological constraints and is expected to grow at a slow rate in the future [3]. In addition, the off-chip clock frequency is limited by the electronic characteristics of the circuit board. The effect of these trends is that off-chip bandwidth is a scarce resource that future CMPs must use efficiently. If the combined bandwidth demand exceeds the off-chip bandwidth capacity, the result is memory bus congestion which increases the average latency of all memory accesses. If the out-of-order processor core logic is not able to fully hide this latency, the result is a reduced instruction commit rate and lower performance.

It is critical for performance that the processor cores are able to continue processing at the same time as a long-latency operation like a memory or L2 cache access is in progress. Consequently, the caches should be able to service requests while misses are processed further down in the memory hierarchy. Caches with this ability are known as *non-blocking* or *lockup-free* and were first introduced by Kroft [5].

Within the cache, the Miss Handling Architecture (MHA) is responsible for keeping track of the outstanding misses. Figure 1 shows an MHA for a cache with multiple banks [4]. The main hardware structure within an MHA is called a *Miss Information/Status Holding Register (MSHR)*. This structure contains the information necessary to successfully return the requested data when the miss completes. If an additional request for a cache block arrives, the information regarding this new request is stored but no request is sent to the next memory hierarchy level. In other words, multiple requests for the same cache block are combined into a single memory access.

In this work, we investigate the performance impact of non-blocking caches in shared-cache CMPs and introduce a novel Miss Handling Architecture called *Adaptive MHA (AMHA)*. AMHA is based on the observation that the available miss bandwidth should be adjusted according to the utilization of the memory bus at runtime. Memory bus congestion can significantly increase the average memory access latency and result in increased lock-up time in the on-chip caches. If the processor core is not able to hide this increased latency, it directly affects its performance. Memory bus congestion reduces the performance of some pro-

grams more than others since the ability to hide the memory latency varies between programs. AMHA exploits this property by reducing the available miss bandwidth for the latency insensitive threads. Since these programs are good at hiding latency, the reduction in miss bandwidth only slightly reduces their performance. However, the memory latency experienced by the congestion sensitive programs is reduced which results in a large performance improvement. For our *Amplified Congestion Probability Workload* collection, AMHA improves the single program oriented Harmonic Mean of Speedups (HMoS) metric by 12% on average.

The paper has the following outline: First, we discuss previous work in Section 2 before we introduce our multiprogrammed workload collections and discuss system performance metrics in Section 3. Then, our new AMHA technique is presented in Section 4. Section 5 describes our experimental methodology, and Section 6 discusses the results from our evaluation of both conventional and adaptive MHAs. Finally, Section 7 discusses future technology trends and possible extensions of AMHA before Section 8 concludes the paper.

2 Related Work

2.1 Miss Handling Architecture Background

A generic Miss Status/Information Holding Register (MSHR) file is shown in Figure 2. This structure consists of n MSHRs which contain space to store the cache block address of the miss, some target information and a valid bit. The cache can handle as many misses to *different cache block addresses* as there are MSHRs without blocking. Each MSHR has its own comparator and the MSHR file can be described as a small fully associative cache. For each miss, the information required for the cache to answer the processor's request is stored in the *Target Information* field. However, the exact *Target Information* content of an MSHR is implementation dependent. The *Valid (V)* bit is set when the MSHR is in use, and the cache must block when all valid bits are set. A blocked cache cannot service any requests.

Another MHA design option regards the number of misses to the *same cache block address* that can be handled without blocking. We refer to this aspect of the MHA implementation as *target storage*, and this determines the structure of the *Target Information* field in Figure 2. Kroft used *implicit* target storage in the original non-blocking cache proposal [5]. Here, storage is dedicated to each processor word in a cache block. Consequently, additional misses to a given cache block can be handled as long as they go to a *different processor word*. The main advantage of this target storage scheme is its low hardware overhead.

Farkas and Jouppi [6] proposed explicitly addressed MSHRs which improves on the implicit scheme by making it possible for any miss to use any target storage location. Consequently, it is possible to handle multiple misses to *the same processor word*. We refer to the number of misses to the same cache block that can be handled without blocking as the number of targets. This improvement

increases hardware cost as the offset of the requested processor word within the cache block must be stored explicitly. In this paper, we use explicitly addressed MSHRs because they provide low lock-up time for a reasonable hardware cost.

Tuck et al. [4] extended the explicitly addressed MSHR scheme to write-back caches. If the miss is a write, it is helpful to buffer the data until the miss completes which adds to the hardware overhead of the scheme. To reduce this overhead, Tuck et al. evaluated MSHRs where only a subset of the target entries has a write buffer. In addition, they extended the implicitly addressed MSHR scheme by adding a write buffer and a write mask which simplify data forwarding for reads and reduce the area cost. The target storage implementations of Tuck et al. can all be used in our AMHA scheme to provide a more fine-grained area/performance trade-off. In this paper, we opt for the simple option of having a write buffer available to all target storage locations as this is likely to give the best performance.

In addition, Tuck et al. proposed the Hierarchical MHA [4]. This MHA provides a large amount of Memory Level Parallelism (MLP) and is primarily aimed at processors that provide very high numbers of in-flight instructions. In a CMP, providing too much MLP can create congestion in shared resources which may result in reduced performance.

Farkas and Jouppi [6] proposed the inverted MSHR organization which can support as many outstanding requests as there are destinations in the machine. Furthermore, Franklin and Sohi [7] observed that a cache line that is waiting to be filled can be used to store MSHR information. These MHAs are extremes of the area/performance trade-off and we choose to focus on less extreme MHAs. In addition, researchers have looked into which number of MSHRs gives the best performance for conventional architectures [7,8].

2.2 Related Work on Bus Scheduling, Shared Caches and Feedback

Mutlu and Moscibroda [9], Nesbit et al. [10] and Rafique et al. [11] are examples of recent work that use the memory bus scheduler to improve Quality of Service (QoS). These works differ from AMHA in that they issue memory requests in a thread-fair manner while AMHA dynamically changes the bandwidth demand to utilize the shared bus efficiently. Furthermore, memory controller scheduling techniques that improve DRAM throughput are complementary to AMHA (e.g. [12,13]).

Other researchers have focused on techniques that use shared cache partitioning to increase performance (e.g. [14,15]). These techniques optimize for the same goal as AMHA, but are complementary since AMHA's only impact on cache partitioning is due to a reduced cache access frequency for the most frequent bus user.

Recently, a large number of researchers have focused on providing shared cache QoS. Some schemes enforce QoS primarily in hardware (e.g. [16]) while others make the OS scheduler cooperate with hardware resource monitoring and control to achieve QoS (e.g. [17]). It is difficult to compare these techniques to AMHA as improving performance is not their primary aim.

Table 1. Randomly Generated Multiprogrammed Workloads (RW)

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	perlbmk, ammp, parser, mgrid	9	vortex1, apsi, fma3d, sixtrack	17	perlbmk, parser, applu, apsi	25	facerec, parser, applu, gap	33	gzip, galgel, lucas, equake
2	mcf, gcc, lucas, twolf	10	ammp, bzip, parser, equake	18	perlbmk, gzip, mgrid, mgrid	26	mcf, ammp, apsi, twolf	34	facerec, facerec, gcc, apsi
3	facerec, mesa, eon, eon	11	twolf, eon, applu, vpr	19	mcf, gcc, apsi, sixtrack	27	swim, ammp, sixtrack, applu	35	swim, mcf, mesa, sixtrack
4	ammp, vortex1, galgel, equake	12	swim, galgel, mgrid, crafty	20	ammp, gcc, art, mesa	28	swim, fma3d, parser, art	36	mesa, bzip, sixtrack, equake
5	gcc, apsi, galgel, crafty	13	twolf, galgel, fma3d, vpr	21	perlbmk, apsi, lucas, equake	29	twolf, gcc, apsi, vortex1	37	mcf, gcc, vortex1, gap
6	facerec, art, applu, equake	14	bzip, bzip, equake, vpr	22	mcf, crafty, vpr, vpr	30	gzip, apsi, mgrid, equake	38	facerec, mcf, parser, lucas
7	gcc, parser, applu, gap	15	swim, galgel, crafty, vpr	23	gzip, mesa, mgrid, equake	31	mgrid, eon, equake, vpr	39	twolf, mesa, eon, eon
8	swim, twolf, mesa, gap	16	mcf, mesa, mesa, wupwise	24	facerec, fma3d, applu, lucas	32	facerec, twolf, gap, wupwise	40	mcf, apsi, apsi, equake

Unpredictable interactions between processors may result in performance degradation in multiprocessor systems. Feedback control schemes can be used to alleviate such bottlenecks if the reduction is due to inadequate knowledge of the state of shared structures. For instance, Scott and Sohi [18] used feedback to avoid tree saturation in multistage networks. Thottethodi et al. [19] used source throttling to avoid network saturation and controlled their policy by a feedback-based adaptive mechanism. In addition, Martin et al. [20] used feedback to adaptively choose between a directory-based and a snooping-based cache coherence protocol. AMHA further extends the use of feedback control by using memory bus and performance measurements to guide miss bandwidth allocations.

3 Multiprogrammed Workload Selection and Performance Metrics

To thouroughly evaluate Miss Handling Architectures in a CMP context, we create 40 multiprogrammed workloads consisting of 4 SPEC CPU2000 benchmarks [21] as shown in Table 1. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. We refer to these workloads as *Random Workloads (RW)*. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a different number of clock cycles if the same benchmark is run on more than one core. If there is only one instance of a benchmark in a workload, it is fast-forwarded for 1 billion clock cycles. The

Table 2. Amplified Congestion Probability Workloads (ACPW)

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	mcf, apsi, applu, wupwise	9	galgel, apsi, art, gcc	17	wupwise, vortex1, apsi, gap	25	gzip, mesa, apsi, gcc	33	wupwise, apsi, art, gap
2	gzip, mcf, art, gap	10	mcf, mesa, vortex1, wupwise	18	mcf, mesa, vortex1, gcc	26	galgel, apsi, art, gcc	34	art, apsi, mgrid, gap
3	gzip, mesa, galgel, applu	11	facerec, mcf, gcc, sixtrack	19	mcf, galgel, vortex1, applu	27	facerec, vortex1, art, gap	35	swim, mesa, mgrid, wupwise
4	gzip, galgel, mesa, sixtrack	12	gzip, mcf, mesa, applu	20	mesa, applu, sixtrack, gap	28	vortex1, mcf, mesa, applu	36	facerec, mcf, art, sixtrack
5	facerec, galgel, mgrid, vortex1	13	galgel, apsi, applu, sixtrack	21	swim, mesa, art, sixtrack	29	swim, gcc, vortex1, gap	37	facerec, gzip, gcc, gap
6	gzip, mcf, mesa, art	14	swim, vortex1, apsi, art	22	swim, mcf, gcc, wupwise	30	swim, gzip, galgel, art	38	facerec, mcf, gcc, sixtrack
7	swim, apsi, sixtrack, applu	15	swim, gzip, mesa, applu	23	mesa, apsi, vortex1, sixtrack	31	swim, gzip, galgel, wupwise	39	facerec, swim, vortex1, gzip
8	facerec, swim, art, sixtrack	16	vortex1, galgel, mesa, sixtrack	24	art, galgel, mgrid, gap	32	gzip, mcf, mesa, wupwise	40	facerec, mcf, mgrid, sixtrack

Table 3. System Performance Metrics

Metric	Formula
Harmonic Mean of Speedups (HMoS) [22]	$\frac{1}{\sum_{i=1}^{P} \frac{IPC_i^{\text{baseline}}}{IPC_i^{\text{shared}}}}$
System Throughput (STP) [23]	$\sum_{i=1}^{P} \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{baseline}}}$

second time a benchmark appears in the workload, we increase the number of fast-forward clock cycles for this instance to 1.02 billion. Then, measurements are collected for 100 million clock cycles.

To investigate the performance of AMHA in the situation it is designed for, we create 40 additional workloads where this situation is more likely than in the randomly generated workload. Here, we randomly select two workloads from the 7 SPEC2000 benchmarks that has an average memory queue latency of more than 1000 processor clock cycles when running alone in the CMP. In our simulations, these benchmarks (*mcf*, *gap*, *apsi*, *facerec*, *galgel*, *mesa* and *swim*) have average queue latencies of between 1116 and 3724 clock cycles. The two remaining benchmarks are randomly chosen from the 8 benchmarks that have an average memory queue latency of between 100 and 1000 clock cycles (i.e. *wupwise*, *vortex1*, *sixtrack*, *gcc*, *art*, *gzip*, *mgrid*, *applu*). We also require that a benchmark is only used once in one workload. We refer to these workloads as *Amplified Congestion Probability Workloads (ACPW)* and they are shown in Table 2.

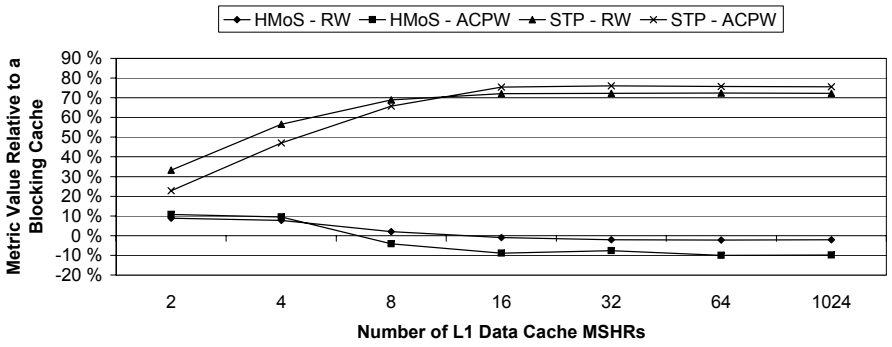


Fig. 3. Average MHA Throughput (Aggregate IPC)

Eyerman and Eeckhout [24] recently showed that the *System Throughput (STP)* and *Harmonic Mean of Speedups (HMoS)* metrics are able to represent workload performance at the system level. The STP metric is a system-oriented performance metric, and the HMoS metric is a user-oriented performance metric. Both metrics require a performance baseline where all programs receive equal access to shared resources. In this work, we give each process exactly a $\frac{1}{P}$ share of the shared cache and at least a $\frac{1}{P}$ share of the available memory bus bandwidth where P is the number of processors. To divide memory bandwidth fairly between threads, we use Rafique et al.'s Network Fair Queueing technique [11] with a starvation prevention threshold of 1. Consequently, access to the memory bus is allocated in a round-robin fashion if all processors have at least one waiting request. The formulae used to compute the HMoS and STP metrics are shown in Table 3. The HMoS metric was originally proposed by Luo et al. [22] and the STP metric is the same as the weighted speedup metric originally proposed by Snavely and Tullsen [23].

4 The Adaptive Miss Handling Architecture (AMHA)

4.1 Motivation

Our Adaptive MHA technique is based on the observation that it is possible to drastically improve the performance of certain programs by carefully distributing miss bandwidth between threads when the memory bus is congested. This differs from earlier research on MHAs where the aim has been to provide as much miss bandwidth as possible in an area-efficient manner. Unfortunately, our results show that following this strategy can create severe congestion in the memory bus which heavily reduces the performance of some benchmarks while hardly affecting others. Figure 3 shows the performance of a conventional MHA in a 4-core CMP plotted relative to the throughput with a blocking cache. To reduce the search space, we only modify the number of MSHRs in the L1 data cache.

The 1024 MSHR architecture is very expensive and is used to estimate the performance of a very large MHA.

Figure 3 shows that a large conventional MHA is able to provide high throughput as measured by the STP metric. Furthermore, throughput increases with more MSHRs up to 8 MSHRs for the RW collection and up to 16 MSHRs with the ACPW collection. The reason for this difference is that there are more memory intensive benchmarks in the ACPW collection which perform better when more miss parallelism is available. Consequently, we can conclude that throughput is improved by adding more MSHRs up to 16.

The trend with the HMoS metric in Figure 3 is very different. Here, the best values are achieved with 2 or 4 MSHRs while adding 16 or more MSHRs reduces the HMoS value below that of a blocking cache for both workload collections. The reason for this trend is that memory bus congestion does not affect all programs equally. For a memory intensive program, an increase in latency due to congestion will not create a large performance degradation. The reason is that these programs already spend a lot of their time waiting for memory. However, less memory intensive programs can hide the most of the access latency and make good progress as long as the memory latencies are reasonably low. When the memory bus is congested, the memory latencies become to large to be hidden which result in a considerable performance degradation. By carefully reallocating the available miss bandwidth, AMHA improves the performance of these latency sensitive benchmarks by reducing the available miss parallelism of the latency insensitive ones.

Figure 3 also offers some insights into why choosing a small number of MSHRs to avoid congestion results in a considerable throughput loss. When both metrics are taken into account, the MHA with 4 MSHRs seems like the best choice. However, this results in an average throughput loss of 9% for the RW collection and 16% for the ACPW collection. In other words, simply reducing the likelyhood of congestion carries with it a significant throughput cost and an adaptive approach is needed.

4.2 AMHA Implementation

AMHA exploits the observation that throughput can be improved by adapting the available miss parallelism to the current memory bus utilization. Figure 4 shows a 4-core CMP which uses AMHA. Implementing AMHA requires only small changes to the existing CMP design. First, an *Adaptive MHA Engine* is added which monitors the memory bus traffic. At regular intervals, the AMHA Engine uses run time measurements to modify the number of available MSHRs in the L1 data cache of each core. Furthermore, the MHAs of the L1 data caches are modified such that the number of available MSHRs can be changed at runtime.

The AMHA Engine Figure 5 shows the internals of the AMHA Engine. It consists of a control unit and a set of registers called *Performance Registers*. These registers are used to measure the performance impact of an AMHA decision on

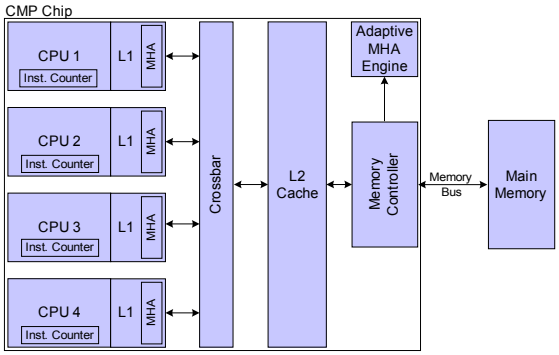


Fig. 4. General Architecture with Adaptive MHA

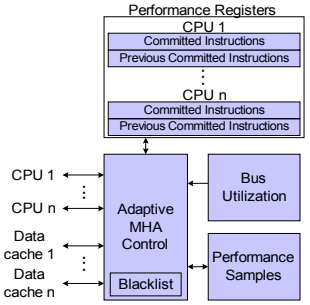


Fig. 5. Adaptive MHA Engine

all threads. In addition, the AMHA Engine stores the average bus utilization during the last sample. Here, the memory controller increments a counter each time it schedules a memory operation. The value of this counter is proportional to the actual bus utilization because each memory request occupies the data bus for a fixed number of cycles and the time between each AMHA evaluation is constant. For clarity, we will refer to this quantity as bus utilization even if AMHA uses the counter value internally. Lastly, the AMHA Engine uses a bit vector called the *blacklist* and a set of registers called *performance samples*. The blacklist is used to mark configurations that should not be tested again, and the performance samples are used to store the performance measurements for certain MHA configurations.

The performance registers store the number of committed instructions in the current and previous MHA samples. Since the number of clock cycles between each AMHA decision is constant, this quantity is proportional to IPC. These values are collected from performance counters inside each processor core when the current MHA is evaluated. By comparing these values, it is possible to estimate the performance impact of a given AMHA decision. This is necessary because it is difficult to determine the consequences of an MHA change from locally measurable variables like average queueing delays or bus utilization. The reason is that the performance with a given MHA is a result of a complex trade-off between an application's ability to hide memory latencies and its congestion sensitivity.

Evaluating the Current MHA Every 500000 clock cycles, the current MHA is evaluated using the information stored in the *Performance Registers*. This evaluation is carried out by the control unit and follows the pseudocode outlined in Algorithm 1. We refer to the time between each evaluation as one MHA *sample*.

To adapt to phase changes, AMHA returns all data caches to their maximum number of MSHRs at regular intervals. We refer to the time between two such

Algorithm 1 Adaptive MHA Main Algorithm

```
1: procedure EvaluateMHA
2:     if RunCount == PERIODSIZE then
3:         Reset all MHAs to their original configuration, set phase = 1 and useAMHA = true
4:         return
5:     end if
6:     if First time in a period and no congestion then
7:         Disable AMHA in this period (useAMHA = false)
8:     end if
9:     Retrieve the current number of committed instruction from the performance counters
10:    if phase == 1 and useAMHA then                                   ▷ Search Phase 1
11:        if Symmetric MHAs remaining then
12:            Reduce the MSHRs of all L1 data caches to the nearest power of 2
13:        else
14:            Choose the best performing symmetric MHA and enter Phase 2
15:        end if
16:    else if phase == 2 and useAMHA then                              ▷ Search Phase 2
17:        if Performance improvement of last AMHA decision not acceptable and useAMHA then
18:            Roll back previous decision and add processor to the blacklist
19:        end if
20:        Find the processor with the largest MHA performance impact that is not blacklisted
21:        if Processor found then
22:            Reduce or increase the number of MSHR to the nearest power of 2
23:        else
24:            All processors are blacklisted, keep current configuration for the rest of this period
25:        end if
26:    end if
27:    Increment RunCount
28:    Move current committed instructions to previous committed instructions
29: end procedure
```

resets as an *AMHA period*. After a reset, we run all data caches with their maximum number of MSHRs in one sample to gather performance statistics. If the bus utilization is lower than a configurable threshold in this sample, AMHA decides that the memory bus is not congested and turns itself off in this AMHA period. We refer to this threshold as the *Congestion Threshold*. The AMHA search procedure has a small performance impact, so we want to be reasonably certain that it is possible to find a better MHA for it to be invoked.

AMHA has now established that the memory bus is most likely congested, and it starts to search for an MHA with better performance. This search consists of two phases. In the first phase, AMHA looks for the best performing *symmetric* MHA. A symmetric MHA has the same number of MSHRs in all L1 data caches. Here, AMHA starts with the largest possible MHA and then tries all symmetric MHAs where the number of MSHRs is a power of two. At the end of each sample, AMHA stores the performance with this MHA in a *Performance Samples* register and tries the next symmetric MHA. When AMHA has tried all symmetric MHAs, the *Performance Samples* registers are analyzed and the best performing MHA is chosen. Since the performance measurements might not be representable for the whole period, we require that a smaller MHA must outperform the largest MHA by a certain percentage called the *Acceptance Threshold*. For each symmetric configuration, we also store the number of committed instructions for each processor. This information is used in search phase 2.

In search phase 2, AMHA attempts to improve performance by searching for an *asymmetric* MHA. Here, we adjust the MHA of one processor each time the MHA is evaluated. Since a new MHA might have been chosen in phase 1, the bus may or may not be congested. Therefore, we need to choose between increasing or decreasing the number of MSHRs in this phase. If the bus utilization is larger than the *Congestion Threshold*, AMHA assumes that the bus is congested and decreases the number of MSHRs to the nearest power of two. If not, the number of MSHRs is increased to the nearest power of two. At the end of the sample, the performance impact is computed and the MHA is either kept or rolled back. If the MHA is not accepted, the processor is blacklisted and phase 2 finishes when all processors have been added to the blacklist. To maximize the performance benefit, we start with the processor where the symmetric MHA had the largest performance impact and process them in descending order.

We use a heuristic to accept or reject an MHA change in search phase 2. If the last operation was a decrease, we sum the speedups of all processors that did not have their MSHRs reduced and compare this to the degradation experienced by the reduced processor. If the difference between the sum of speedups and the degradation is larger than the configurable *Acceptance Threshold*, the new MHA is kept. For simplicity, we use the same acceptance threshold in both search phases. If the memory bus is severely congested, reducing the number of MSHRs of a processor can actually increase its performance. In this case, we set the degradation to 0. In addition, we reject any performance degradations of processors that have not had its number of MSHRs reduced as measurement errors. If the last operation increased the number of MSHRs, we sum the performance degradations of the other processors and weigh this against the performance improvement of the processor that got its number of MSHRs increased. Again, the difference must be larger than the *Acceptance Threshold* to keep the new MHA.

For each AMHA evaluation, we need to carry out P divisions in phase 1 and P divisions in phase two where P is the number of processors. The reason is that AMHA's decisions are based on relative performance improvements or degradations and not the number of committed instructions. Since there are no hard limits to when the AMHA decision needs to be ready, it can be feasible to use a single division unit for this purpose. For simplicity, we assume that the AMHA Engine analysis can be carried out within 1 clock cycle in this work. Since we need relatively large samples for the performance measurements to be accurate, it is unlikely that this assumption will influence the results. We leave investigating area-efficient AMHA Engine implementations and refining the experiments with accurate timings as further work.

MHA Reconfiguration An MHA which includes the features needed to support AMHA is shown in Figure 6. This MHA is changed slightly compared to the generic MHA in Figure 2. The main difference is the addition of a *Usable (U)* bit to each MSHR. If this is set, the MSHR can be used to store miss data. By manipulating these bits, it is possible to change the number of available MSHRs at runtime. The maximum number of MSHRs is determined by the number of

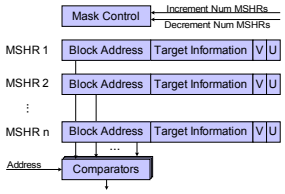


Fig. 6. The New MHA Implementation

physical registers and decided at implementation time. As in the conventional MSHR file, the *Valid (V)* bit is set if the MSHR contains valid miss data.

The other addition needed to support AMHA is *Mask Control*. This control unit manipulates the values of the U bits subject to the commands given by the *AMHA Engine*. For instance, if the *AMHA Engine* decides that the number of MSHRs in cache A should be reduced, cache A's *Mask Control* sets the U bits for some MSHRs to 0. In the current implementation, the number of available MSHRs is increased or decreased to the nearest power of two.

When the number of MSHRs is decreased, it is possible that some registers that contain valid miss data are taken out of use. Consequently, these registers must be searched when a response is received from the next memory hierarchy level. However, the cache should block immediately to reflect the decision of the *AMHA Engine*. This problem is solved by taking both the V and U bits into account on a cache miss and for the blocking decision. Furthermore, all registers that contain valid data (i.e. have their V bit set) are searched when a response is received.

We have chosen to restrict the adaptivity to the number of available MSHRs, but it is also possible to change the amount of target storage available. In other words, it is possible to manipulate the number of simultaneous misses to the same cache block that can be handled without blocking. This will increase the implementation complexity of AMHA considerably. Furthermore, it is only a different way to reduce the number of requests injected into the memory system. The reason is that the cache is blocked for a shorter amount of time with more targets which indirectly increases the bandwidth demand. For these reasons, AMHA keeps the amount of target storage per MSHR constant.

AMHA only requires slightly more area than a conventional MHA with the same maximum number of MSHRs as each MSHR only needs to be extended with one additional bit. Furthermore, the AMHA Engine needs a few registers and logic to compute and compare application speedups. In addition, the control functions in both the AMHA Engine and the reconfigurable MHAs require a small amount of logic.

5 Experimental Setup

We use the system call emulation mode of the cycle-accurate M5 simulator [25] to evaluate the conventional MHAs and AMHA. The processor architecture param-

Table 4. Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional Units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch Predictor	Hybrid, 2048 local history registers, 2-way 2048 entry BTB

Table 5. Memory System Parameters

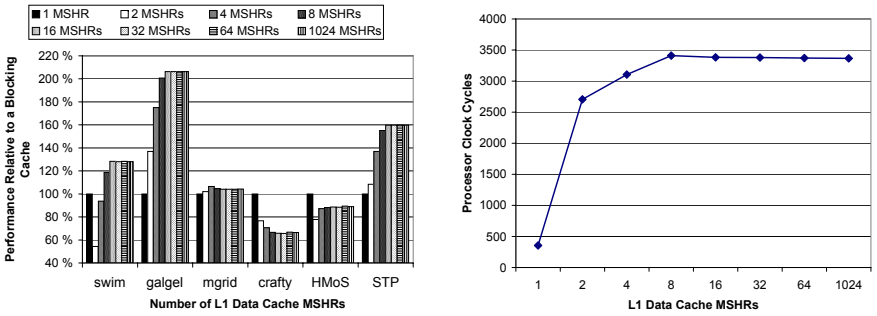
Parameter	Value
Level 1 Data Cache	64 KB 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 8 targets per MSHR, 1 cycle latency
Level 2 Unified Shared Cache	4 MB 8-way set associative, 64B blocks, 14 cycles latency, 16 MSHRs per bank, 8 targets per MSHR, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 8 cycles latency, 64B wide transmission channel
Memory Bus and DRAM	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [13], closed page policy

eters for the simulated 4-core CMP are shown in Table 4, and Table 5 contains the baseline memory system parameters. We have extended M5 with an AMHA implementation, a crossbar interconnect and a detailed DDR2-800 memory bus and SDRAM model [26]. The DDR2-800 memory bus is a split transaction bus which accurately models overlapping of requests to different banks, burst mode transfer as well as activation and precharging of memory pages. When a memory page has been activated, subsequent requests are serviced at a much lower latency (page hit). We refer the reader to Cuppu et al. [27] for more details on modern memory bus interfaces. The DDR2 memory controller uses Rixner et al.'s First Ready - First Come First Served (FR-FCFS) scheduling policy [13] and reorders memory requests to achieve higher page hit rates.

6 Results

6.1 Conventional MHA Performance in CMPs

In Section 4.1, we established that increasing the number of MSHRs improves throughput but reduces HMoS performance. However, the cause of this trend was not explained in detail. In this section, we shed some light on this issue by thoroughly analyzing the performance of the RW12 workload. This workload consists of the benchmarks *swim*, *mgrid*, *crafty* and *galgel* which are responsible for 53%, 39%, 5% and 3% of the memory bus requests with 16 MSHRs, respectively.



(a) Performance (b) Average Memory Bus Queue Latencies

Fig. 7. MHA Performance with RW12

Figure 7(a) shows the speedups relative to the equal allocation baseline plotted relative to the benchmark's speedup with a blocking cache configuration. In addition, the figure shows the performance trend for the system performance metrics HMoS and STP. The only benchmark that experiences a performance improvement with every increase in MHA size is *galgel*. For the other benchmarks, memory bus congestion causes more complex performance trends.

For *crafty*, performance is reduced substantially when the number of MSHRs is increased to 2. Performance is further reduced until the MHA contains 8 MSHRs before it stabilizes. Figure 7(b) shows the average memory bus queue latency as a function of the number of MSHRs. By comparing the performance trend of *crafty* with the average queue latency, we can see that for every increase in average queue latency there is a decrease in *crafty*'s performance. Since the HMoS metric is dominated by the program with the lowest performance, the HMoS metric has its highest value with the 1 MSHR MHA. However, the STP metric hides this effect and reports a throughput improvement with every increase in MHA size.

When *galgel* is provided with more MSHRs, its ability to hide the memory latencies improves enough to remove the effects of bus congestion which result in a net performance improvement. *Swim* needs a lager number of MSHRs to experience a performance improvement, but otherwise the performance trend is similar to that of *galgel*. The 2 and 4 MSHR MHAs both result in a performance reduction for *swim* because they provide to little miss parallelism to hide the long memory latencies. However, adding more MSHRs improve *swim*'s ability to hide the memory latency and result in a performance improvement. Changes in MHA size has a small performance impact on *mgrid*, and the performance difference between its best and worst MHA is only 6%.

Our study of workload RW12 has identified three properties that an adaptive MHA should be aware of. Firstly, programs with fewer memory requests are more sensitive to MHA size than memory intensive programs. Consequently, the MHA size of the memory intensive programs can be reduced to speed up the con-

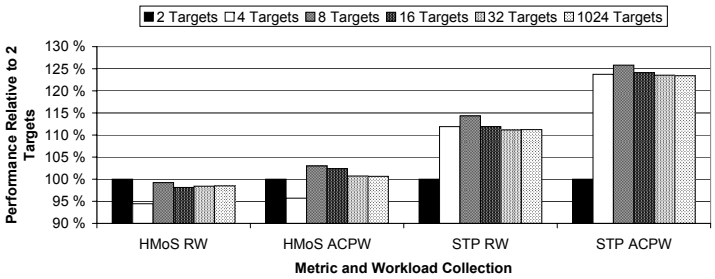


Fig. 8. Target Performance with 16 MSHRs

gestion sensitive programs without creating an unnecessarily large throughput degradation. Secondly, the impact of bus congestion on program performance is application dependent. Therefore, we can only rely on memory bus measurements to detect congestion while performance measurements are needed to determine the effects of an MHA change. Finally, the performance impact on an application from a change in MHA size depends on the relationship between the program's ability to hide memory latencies and the combined load the workload puts on the memory bus.

6.2 The Performance Impact of the Number of Targets per MSHR

Figure 8 shows the results from varying the number of outstanding misses to the same cache block address that can be handled without blocking (i.e. the number of targets). We investigated the performance impact of varying this parameter for L1 caches with 2, 4, 8 and 16 L1 data cache MSHRs, but only report the results for the 16 MSHR case because the performance trends are very similar. The main difference is that the performance impact of adding more targets is larger with more MSHRs. If there is one target per MSHR, the cache has to block on the first miss, and this is equivalent to a blocking cache.

For both workload collections, throughput is maximized with 8 targets per MSHR. The reason is that this creates a good compromise between latency tolerance and memory bus congestion. Unfortunately, the area cost of adding 8 targets is high. Consequently, the MHA with 4 targets is probably a better choice given the small performance benefit of increasing the number of targets beyond 4. The performance impact from adding more targets is larger for the ACPW collection because its workloads contain a larger number of memory intensive benchmarks by design. In other words, a greater number of benchmarks are memory intensive enough to benefit from increased miss parallelism. On the HMoS metric, adding more targets only slightly affects performance. Although the performance with 4 targets is the worst out of the examined target counts, the large increase in throughput and reasonable hardware cost makes a compelling argument for choosing this number of targets.

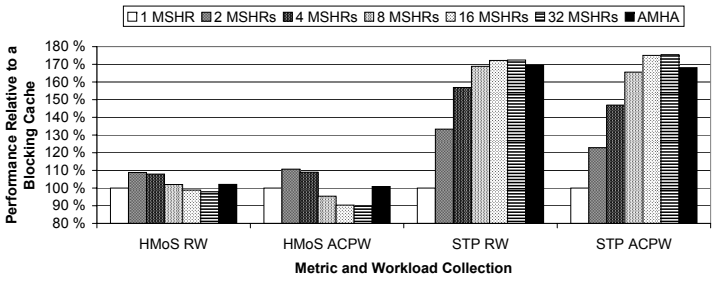


Fig. 9. AMHA Average Performance

6.3 Adaptive MHA Performance

In this section, we report the results from our evaluation of the Adaptive MHA. For the experiments in this section, AMHA has a maximum of 16 MSHRs available in the L1 data cache. Therefore, the area overhead of this configuration is comparable to the conventional MHA with 16 MSHRs. AMHA only changes the number of available MSHRs in the L1 data cache for each core, and we keep the number of MSHRs in the L1 instruction caches constant at 16 for all conventional and adaptive configurations. The number of targets is 4 in all MSHRs.

AMHA aims at improving the performance of the applications that are victims of memory bus bandwidth overuse by other programs. Consequently, we expect an improvement on the HMoS metric with a reasonable reduction in system throughput. Figure 9 shows AMHA's average performance compared to various conventional MHAs. For the RW collection, the performance impact by running AMHA is small on average, since AMHA only has a significant performance impact on 4 workloads. This is necessary for AMHA to give stable performance because reducing the number of available MSHRs can drastically reduce performance if the memory bus is not sufficiently congested. RW35 is the workload where AMHA has the largest impact with an HMoS improvement of 193% compared to a 16 MSHR MHA. If we only consider the 4 workloads where AMHA has a HMoS impact of more than 5% (both improvement and degradation), the result is an average HMoS improvement by 72% and a 3% average improvement in throughput. Consequently, we can conclude that with randomly generated workloads, AMHA has a large performance impact when it is needed and effectively turns itself off when it is not.

In the ACPW collection, the impact of AMHA is much larger since memory bus congestion is more likely for these workloads. Figure 10 shows the performance of AMHA relative to that of a conventional 16 MSHR MHA for the workloads where AMHA has a larger HMoS impact (both improvement and degradation) of more than 10%. Again, AMHA has a large HMoS impact when it is needed and improves HMoS by 52% on average and as much as 324%. In some cases AMHA also improve STP, but the common case is a small STP degradation. Since AMHA reduces the miss bandwidth of the memory bus intensive

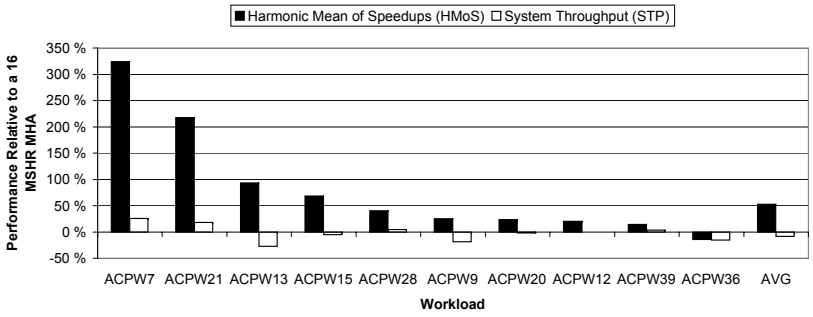
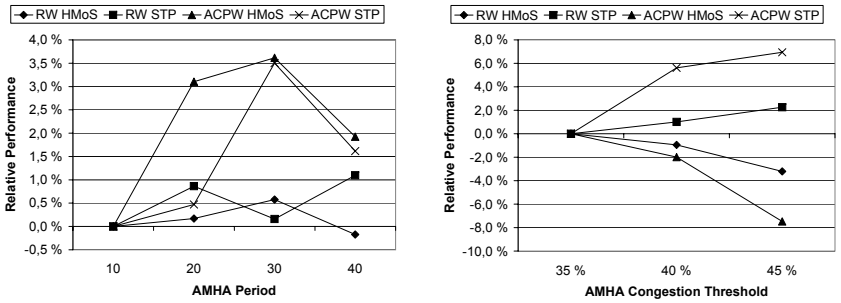


Fig. 10. AMHA Performance with High-Impact Workloads from ACPW



(a) AMHA Period (b) AMHA Congestion Threshold

Fig. 11. AMHA Settings

programs, it is likely that their performance is reduced which is shown in our measurements as a throughput reduction.

For ACPW36 (*facerec*, *mcf*, *art* and *sixtrack*), AMHA reduces both HMoS and STP. Here, bus utilization is low enough for AMHA to be turned off in most periods. However, there are two periods of bus congestion where AMHA's performance measurements indicate a large speed-up by significantly reducing *sixtrack*'s number of MSHRs. Although this is correct when the measurements are taken, AMHA keeps this configuration also after the brief period of congestion has passed. Consequently, the available miss parallelism is reduced more than necessary which results in a performance degradation on both metrics.

6.4 Choosing AMHA Implementation Constants

Up to now, we have used an AMHA implementation with a period of 30, a congestion threshold of 40% and an acceptance threshold of 10%. These values have been determined through extensive simulation of possible AMHA implementations. Figure 11(a) shows the performance impact of varying the AMHA period setting. Here, the value must be chosen such that the cost of searching for a good

MHA is amortized over a sufficiently long period as well as that a new search is carried our before the findings from the last search becomes obsolete. Figure 11(b) shows the parameter space for the congestion threshold setting which adjusts the bus utilization necessary to conduct an MHA search. Here, STP is maximized with a high threshold value and HMoS is maximized with a low threshold value. Since we in this work aim at increasing HMoS while tolerating a small STP reduction, the middle value of 40% is a good choice. However, choosing 45% as the threshold is appropriate if a more throughput friendly AMHA is desired.

Finally, AMHA also needs an acceptance threshold which determines how large the difference between the performance benefit and performance cost of a sample MHA must be for the sample MHA to be used for the remainder of the AMHA period. Here, we investigated values in the 2% to 10% range and found that 10% gave the best results. For the RW collection this parameter had nearly no impact while for the ACPW collection both HMoS and STP was maximized by choosing 10%. In general, the acceptance threshold must be large enough to filter out reduction operations that are not justified and small enough to carry out the MHA reductions when they are needed.

7 Discussion

AMHA works well for the CMP architecture used in this paper. However, it is important that it will also work well in future CMP architectures. Since AMHA improves performance when there is congestion in the memory bus, the performance gains are closely tied to the amount of congestion. The width of the memory bus and the clock frequency are both subject to technological constraints [3]. Consequently, it is unlikely that bus bandwidth can be improved sufficiently to match the expected increase in the number of processing cores [28]. Unless a revolutionary new memory interface solution is discovered, off-chip bandwidth is likely to become an even more constrained resource in the future [29]. Consequently, techniques like AMHA will become more important.

Currently, AMHA does not support multithreaded applications or processor cores with SMT. To support multithreaded applications, we need to treat multiple processor cores as a single entity when allocating miss bandwidth. This can be accomplished by letting the operating system provide some simplified process IDs as discussed by Zaho et al. [30] and communicate this to the Adaptive MHA Engine. Furthermore, some logic must be added to keep instructions committed in busy wait loops out of AMHA's performance measurements. Introducing SMT further complicates matters as each core now supports more than one hardware thread. Here, we need to further extend the MHA to allocate a different number of L1 MSHRs to each hardware thread. We leave the exact implementation and evaluation of such extensions as further work.

By targeting the victims of memory bus congestion and improving their performance, one might argue that AMHA is a fairness technique. However, AMHA only target unfairness in one situation, namely when the memory bus is severely

congested. Furthermore, AMHA makes no guarantees of how much miss bandwidth each processor is given. Therefore, it is better to view AMHA as a simple performance optimization that can be applied when certain conditions are met.

8 Conclusion

When designing Miss Handling Architectures (MHAs), the aim has been to support as many outstanding misses as possible in an area efficient manner. Unfortunately, applying this strategy to a CMP will not realize its performance potential. The reason is that allowing too much miss parallelism creates congestion in the off-chip memory bus.

The first contribution of this paper is a thorough investigation of conventional MHA performance in a CMP. The main result of this investigation was that a majority of applications need large miss parallelism. However, this must be provided in a way that avoids memory bus congestion. Our Adaptive MHA (AMHA) scheme serves this purpose and is the second contribution in this paper. AMHA increases CMP performance by dynamically adapting the allowed number of outstanding misses in the private L1 data caches to the current memory bus utilization.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach, Fourth Edition. Morgan Kaufmann Publishers (2007)
2. Burger, D., Goodman, J.R., Kägi, A.: Memory Bandwidth Limitations of Future Microprocessors. In: ISCA '96: Proc. of the 23rd An. Int. Symp. on Comp. Arch. (1996)
3. ITRS: Int. Tech. Roadmap for Semiconductors. `http://www.itrs.net/` (2006)
4. Tuck, J., Ceze, L., Torrellas, J.: Scalable Cache Miss Handling for High Memory-Level Parallelism. In: MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarchitecture. (2006) 409–422
5. Kroft, D.: Lockup-free Instruction Fetch/Prefetch Cache Organization. In: ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch. (1981) 81–87
6. Farkas, K.I., Jouppi, N.P.: Complexity/Performance Tradeoffs with Non-Blocking Loads. In: ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch. (1994) 211–222
7. Sohi, G.S., Franklin, M.: High-bandwidth Data Memory Systems for Superscalar Processors. In: ASPLOS-IV: Proc. of the fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (1991) 53–62
8. Belayneh, S., Kaeli, D.R.: A Discussion on Non-Blocking/Lockup-Free Caches. SIGARCH Comp. Arch. News **24**(3) (1996) 18–25
9. Mutlu, O., Moscibroda, T.: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture. (2007)
10. Nesbit, K.J., Aggarwal, N., L., J., Smith, J.E.: Fair Queuing Memory Systems. In: MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarchitecture. (2006) 208–222

11. Rafique, N., Lim, W.T., Thottethodi, M.: Effective Management of DRAM Bandwidth in Multicore Processors. In: PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques. (2007) 245–258
12. Shao, J., Davis, B.: A Burst Scheduling Access Reordering Mechanism. In: HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch. (2007)
13. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory Access Scheduling. In: ISCA '00: Proc. of the 27th An. Int. Symp. on Comp. Arch. (2000) 128–138
14. Qureshi, M.K., Patt, Y.N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch. (2006) 423–432
15. Dybdahl, H., Stenström, P.: An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In: HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch. (2007)
16. Nesbit, K.J., Laudon, J., Smith, J.E.: Virtual Private Caches. In: ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch. (2007) 57–68
17. Chang, J., Sohi, G.S.: Cooperative Cache Partitioning for Chip Multiprocessors. In: ICS '07: Proc. of the 21st An. Int. Conf. on Supercomputing. (2007) 242–252
18. Scott, S.L., Sohi, G.S.: The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control. IEEE Trans. Parallel Distrib. Syst. (1990) 385–398
19. Thottethodi, M., Lebeck, A., Mukherjee, S.: Exploiting Global Knowledge to achieve Self-tuned Congestion Control for k-ary n-cube Networks. IEEE Trans. on Parallel and Distributed Systems (2004) 257–272
20. Martin, M., Sorin, D., Hill, M., Wood, D.: Bandwidth Adaptive Snooping. In: HPCA '02: Proc. of the 8th Int. Symp. on High-Performance Comp. Arch. (2002) 251
21. SPEC: SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`
22. Luo, K., Gummaraju, J., Franklin, M.: Balancing Throughput and Fairness in SMT Processors. In: ISPASS. (2001)
23. Snavely, A., Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In: Arch. Support for Programming Languages and Operating Systems. (2000) 234–244
24. Eyerman, S., Eeckhout, L.: System-Level Performance Metrics for Multiprogram Workloads. IEEE Micro **28**(3) (2008) 42–53
25. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The M5 Simulator: Modeling Networked Systems. IEEE Micro **26**(4) (2006) 52–60
26. JEDEC Solid State Technology Association: DDR2 SDRAM Specification. (May 2006)
27. Cuppu, V., Jacob, B., Davis, B., Mudge, T.: A Performance Comparison of Contemporary DRAM Architectures. In: Proc. of the 26th Inter. Symp. on Comp. Arch. (1999) 222–233
28. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley (December 2006)
29. Huh, J., Burger, D., Keckler, S.W.: Exploring the Design Space of Future CMPs. In: PACT '01: Proc. of the 2001 Int. Conf. on Parallel Architectures and Compilation Techniques. (2001) 199–210
30. Zhao, L., Iyer, R., Illikkal, R., Moses, J., Makineni, S., Newell, D.: CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In: PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques. (2007) 339–352