Noname manuscript No.
(will be inserted by the editor)

Addressing Isolation Challenges of Non-blocking Caches for Multicore Real-Time Systems

Prathap Kumar Valsan · Heechul Yun · Farzad Farshchi

Received: date / Accepted: date

Abstract In multicore real-time systems, cache partitioning is commonly used to achieve isolation among different cores. We show, however, that space isolation achieved by cache partitioning does not necessarily guarantee predictable cache access timing in modern COTS multicore platforms, which use non-blocking caches. We find that special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs), which track the status of outstanding cache-misses, can be a significant source of contention that is not addressed by conventional cache partitioning.

We propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our approach includes a low-cost hardware extension that enables dynamic control of per-core memory-level parallelism (MLP) by the OS. Using the hardware extension, the OS scheduler then globally controls each core's MLP in such a way that eliminates MSHR contention and maximizes overall throughput of the system. We implement the hardware extension in a cycle-accurate full-system simulator and the scheduler modification in Linux 3.14 kernel. Extensive experimental results demonstrate the significance of the MSHR contention problem and the effectiveness of the proposed solution.

1 Introduction

As embedded real-time systems become more intelligent and complex, high-performance multicore processors are increasingly demanded to meet their performance and computing capacity demands while saving cost and reducing size, weight, and power (SWaP) requirements. However, consolidating multiple tasks, potentially with different criticality (a.k.a. mixed-criticality systems [41, 7]), on a commercial-off-the-shelf

Prathap Kumar Valsan, Heechul Yun, Farzad Farshchi
University of Kansas
E-mail: {prathap.kumarvalsan, heechul.yun, farshchi}@ku.edu

(COTS) multicore processor is challenging because interference in the shared hardware resources can significantly alter the tasks' timing characteristics.

Shared last-level cache (LLC) is one of the major sources of interference in multicore. Tasks sharing an LLC, if uncontrolled, can evict each other's valuable cache-lines, thereby affect each other's execution times. Such co-runner dependent execution time variations are highly undesirable for real-time systems. This is especially problematic for critical systems such as those in aviation that require certification [8].

To address the problem of unwanted cache-line evictions, cache partitioning is a well known solution in the real-time systems community [30,42,26,36,9]. The basic idea of cache partitioning is to assign dedicated space to each task (or core). It can be done in software, with a OS-level technique known as page coloring [23,43,28], or in hardware, with a technique known as way partitioning on some supported hardware platforms [33,35]. In either case, the effect of cache partitioning is the elimination of unwanted cache-line evictions (spatial isolation). Most literature assumes that cache partitioning guarantees that access timing to a dedicated cache partition would not be affected by concurrent accesses to different cache partitions (temporal isolation). Unfortunately, this is not necessarily the case in *non-blocking caches* [27], which are commonly used in modern COTS multicore processors to exploit memory-level parallelism (MLP).

In this article, we experimentally show that cache partitioning does not guarantee cache access timing isolation on COTS multicore platforms. We use a set of carefully designed synthetic benchmarks, which we call *IsolBench*, as well as macro benchmarks from EEMBC [1] and SD-VBS [40] benchmark suites; and evaluate the temporal isolation effect of cache partitioning on five COTS multicore platforms (six different architecture configurations). From the results, we observe significant Worst-Case Execution Time (WCET) increases—up to 21X—even though each evaluated task runs on a dedicated core, accessing a dedicated cache partition, and almost all of the memory accesses are cache hits. The WCET increases are observed in all but one COTS architecture we tested.

We attribute the source of the problem to contention in special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs) [27]. In a non-blocking cache, the cache can continue to serve memory requests even under multiple cache-misses (hit-under-multiple-misses) and MSHRs are used to track the outstanding cache-misses. The number of MSHRs effectively determines the cache's MLP. When the MSHRs are exhausted, however, the cache will block any further accesses until some of the outstanding misses are serviced—hence freeing the corresponding MSHR entries [2]. Because retrieving data from DRAM can take long time (i.e., 100 CPU cycles), the blocking caused by exhausting MSHRs of a non blocking cache, especially the shared LLC, can stall cores a long duration of time. We call this phenomenon as the *MSHR contention* problem.

To further validate the problem, we use a cycle accurate full system simulator and investigate isolation and throughput impacts of different MSHR configurations in private and shared non-blocking caches. We find that an insufficient number of MSHRs in the shared LLC can be highly detrimental to temporal isolation of the multicore system due to the MSHR contention problem. On the other hand, we also find that a large number of MSHRs in private L1 caches are often under-utilized.

Motivated by the findings, we propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our approach includes a low-cost hardware extension in each core's private L1 cache that enables the OS to dynamically control the number of valid MSHRs of the cache. Using the hardware extension, the OS scheduler then globally controls each core's MLP in a way that eliminates MSHR contention and maximizes overall throughput of the system.

We have implemented the hardware extension in a cycle-accurate full-system simulator, which models a quad-core ARM Cortex-A15 processor, and modified the scheduler of Linux 3.14 kernel, which runs on top of the simulator. We evaluate the effectiveness of our approach using a set of synthetic and macro benchmarks on a number of different configurations. Compared to baseline cache partitioning, our approach has shown to significantly improve the cache access timing isolation with minimal throughput impact.

This article is an **extended version** of [39]. The **additional contributions** compared with the original work are:

- We evaluate an additional COTS multicore platform, NVIDIA Tegra TK1. This allowed us to compare two ARM Cortex-A15 based platforms with different architectural features (en/disable L2 prefetcher). Also, we enhance our evaluation of the proposed MSHR partitioning technique with an additional experiment with eight EEMBC and SD-VBS benchmarks.
- In [39], we only used a synthetic memory intensive benchmark as co-runners to cause maximal MSHR contention. In this article, we also use real-world memory intensive benchmarks from SPEC2006 benchmark suite as co-runners, to investigate how that affects to the MSHR contention problem.
- We evaluate the effect of the various hardware architectural characteristics—i.e., core aggressiveness, clock-speed, and hardware prefetchers—to the MSHR contention problem.
- The MSHR partitioning algorithm in [39] could waste MSHRs in some corner cases. In this article, we improve the MSHR partitioning algorithm to remove MSHR waste in the corner cases.

The rest of the paper is organized as follows. Section 2 describes additional background on non-blocking caches and cache partitioning techniques. Section 3 demonstrates the problem of MSHR contention using real COTS multicore platforms. Section 4 further validates the MSHR contention problem and investigates isolation and throughput impacts of MSHRs in private and shared non-blocking caches. Section 5 presents our hardware and OS collaborative technique to eliminate MSHR contention. Section 6 presents evaluation results of the proposed technique. Section 7 investigates other factors that could influence the MSHR contention problem. We discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section, we provide necessary background on non-blocking caches and common cache partitioning techniques.

2.1 Non-blocking Caches and MSHRs

A non-blocking cache is a type of cache that can service multiple memory requests at the same time. In contrast, a blocking cache can only serve one request a time. Figure 1 shows the differences between the two cache types. Because a non-blocking cache can continue to service under multiple cache misses (miss-under-miss), it can hide cache-miss penalties and therefore improves performance.

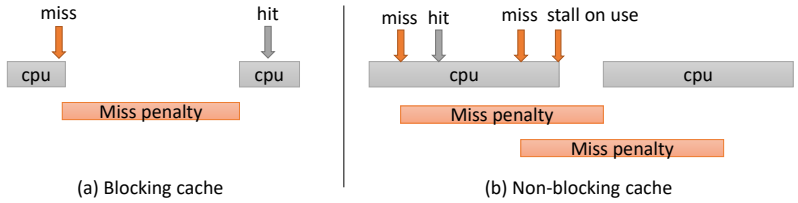


Fig. 1 Blocking vs. Non-blocking cache

Non-blocking caches are widely used in modern COTS multicore processors. In out-of-order processors, which are increasingly common in high-performance embedded processors [31,12], each core can generate multiple outstanding memory requests. Therefore, non-blocking caches are crucial in all layers of cache hierarchies (both private and shared caches) to feed data to the processors. Even in in-order processors where each core can only generate one outstanding memory request at a time, the cores collectively can generate multiple requests to the shared memory subsystems—shared cache and memory. Therefore, memory subsystems must be able to handle multiple parallel memory requests. The degree of parallelism supported by a memory subsystem is called *Memory-Level Parallelism (MLP)* [13]. Non-blocking caches are essential to provide high MLP in multicore processors.

When a cache-miss occurs on a non-blocking cache, the cache controller records the miss on a special register, called Miss Status Holding Register (MSHR) [27], which tracks the status of the ongoing request. The request is managed at a cache-line granularity. Multiple misses to the same cache-line are merged and notified together by a single MSHR entry. The MSHR entry is cleared when the corresponding memory request is serviced from the lower-level memory hierarchy. In the meantime, the cache can continue to serve cache (hit) access requests. Multiple MSHRs are used to support multiple outstanding cache-misses and the number of MSHRs determines the MLP of the cache. It is important to note that MSHRs in the shared LLC are also shared resources with respect to the CPU cores [17]. Moreover, if there are no remaining MSHRs, further accesses to the cache—*both* hits and misses—are blocked until free MSHRs become available [2], because whether a cache access is hit or miss is not immediately known at the time of the access [37]. In other words, cache hit requests can be delayed if all MSHRs are used up. This situation can happen even if the cache space is partitioned among cores, as we will show in Section 3.

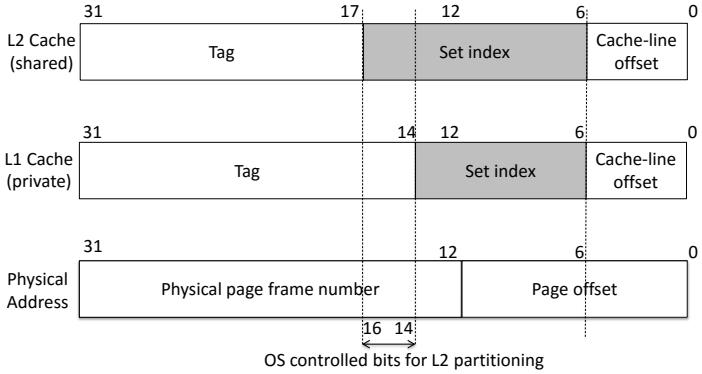


Fig. 2 Physical address and cache mapping of Cortex-A15.

2.2 Cache Partitioning Techniques

Cache partitioning is a well-known solution to achieve cache space isolation—that is preventing unwanted cache-line evictions due to cache space sharing. A typical set-associative cache is organized in sets and and ways. Part of memory address bits are used as index bits to determine the set address of the cache. For each set, there are multiple ways and data can be located in any one of them. To partition a cache, two commonly used techniques are way-based partitioning and set-based partitioning.

In way-based cache partitioning [35], the cache space is partitioned at the granularity of cache ways, which requires hardware-level support. Some embedded processors and a few recent Intel Xeon processors support way-based cache partitioning [12, 19], but most processors do not support it. For example, none of the hardware platforms used in this study support hardware way partitioning.

On the other hand, in set-based partitioning, the cache space is partitioned with respect to cache-sets, which can be implemented in software (OS) with a technique known as page coloring [23, 28]. Page-coloring can be implemented in any hardware that equips memory-management unit (MMU). Therefore, in this paper, we use a page-coloring based technique [45] to partition the shared caches of all hardware platforms we tested. In page coloring, the OS controls the physical addresses of memory pages such that the pages are placed in specific cache locations (sets). By allocating memory pages over non-overlapping sets of the cache, the OS can effectively partition the cache. In order to apply page-coloring, the OS must understand how the cache sets are mapped onto the physical address space. Figure 2 shows the address mapping of a Cortex-A15 platform, which we use in Section 3. The address mapping of a cache is determined by the size of the cache, cache-line size, and the number of ways of the cache. Once the cache set-index bits are identified, the OS controls the subset of the index bits, called page colors, in allocating pages. When multiple layers of caches are used as in the case of Cortex-A15, care must be taken to partition only the shared LLC but not the private L1 caches. For example, in Figure 2, only bit 14, 15, and 16 should be used to partition only the shared L2 cache.

3 Evaluating Isolation Effect of Cache Partitioning on COTS Multicore Platforms

In this section, we present our experimental investigation on the effectiveness of cache partitioning in providing cache access performance isolation on COTS multicore platforms.

Table 1 Evaluated COTS multicore platforms.

	Cortex-A7	Cortex-A9	Cortex-A15^O	Cortex-A15^T	Nehalem
CPU	4 cores 1.4GHz in-order	4 cores 1.7GHz out-of-order	4 cores 2.0GHz out-of-order	4 cores 2.3Ghz out-of-order	4 cores 2.8GHz out-of-order
LLC	512KB	1MB	2MB	2MB	8MB
LLC prefetcher	No	No	Yes	No	No
DRAM	2GB	2GB	2GB	2GB	4GB
Platform	Odroid-XU4	Odroid-U3	Odroid-XU4	Tegra TK1	Dell T3500

3.1 COTS Multicore Platforms

We use four COTS multicore platforms (five architectures): Odroid-XU4, Odroid-U3, Tegra TK1 single-board computers (SBC) and a Dell T3500 desktop computer. The Odroid-XU4 board equips a Samsung Exynos 5422 processor which includes both four Cortex-A15 and four Cortex-A7 cores in a big-LITTLE [14] configuration. Thus, we use the Odroid-XU4 platform for both Cortex-A7 and Cortex-A15 experiments. The Odroid-U3 equips a Samsung Exynos 4412 processor which includes four Cortex-A9 cores. The Tegra TK1 platforms equips a NVIDIA's Tegra K1 processor which also has four Cortex-A15 cores. We disabled L2 cache prefetcher on the Tegra TK1, while it is enabled in the Odroid-XU4 platform. Lastly, the Dell T3500 platform equips an Intel Xeon W3553 (Nehalem) processor; we disabled hyperthreading and all hardware prefetchers. In all platforms, we disabled power saving features such as dynamic CPU and memory frequency scaling. Table 1 shows the basic characteristics of the five CPU architectures we used in our experiments. Note that Odroid-XU4's Cortex-A15 is denoted as Cortex-A15^O while TK1 is denoted as Cortex-A15^T Note that in each architecture, LLC and DRAM are shared by all four cores. For OS, we run Linux 3.10.82 on the Odroid-XU4 platform, Linux 3.8.13 on the Odroid-U3 platform, Linux 3.10.40 on the Tegra TK1 platform, and Linux 3.6.0 on the Intel Xeon platform, all kernels are patched with PALLOC [45] to partition the shared LLC at run-time.

3.2 Memory-Level Parallelism

We first identify memory-level parallelism (MLP) of the multicore architectures using an experimental method described in [11]. The method uses a pointer-chasing micro-benchmark shown in Figure 3 to identify memory-level parallelism. The benchmark

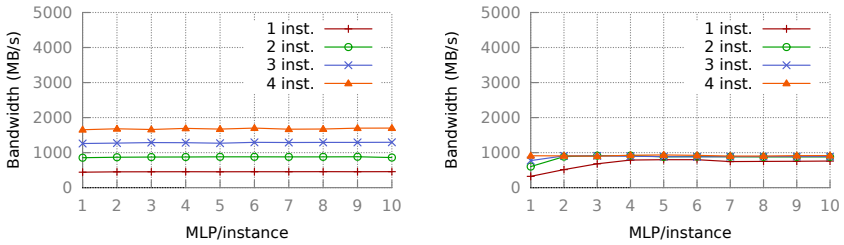
```
1  static int* list[MAX_MLP];
2  static int next[MAX_MLP];
3
4  long run(long iter, int mlp)
5  {
6      long cnt = 0;
7      for (long i = 0; i < iter; i++) {
8          switch (mlp) {
9          case MAX_MLP:
10             .
11             .
12         case 2:
13             next[1] = list[1][next[1]];
14             /* fall-through */
15         case 1:
16             next[0] = list[0][next[0]];
17         }
18         cnt += mlp;
19     }
20     return cnt;
21 }
```

Fig. 3 MLP micro-benchmark. Adopted from [11].

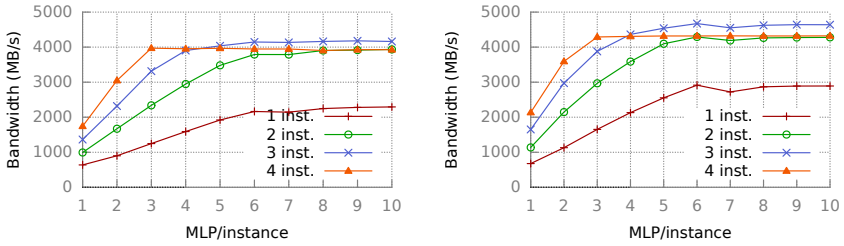
traverses a number of linked-lists. Each linked-list is randomly shuffled over a memory chunk of twice the size of the LLC. Hence, accessing each entry is likely to cause a cache-miss. Due to data-dependency, only one cache-miss can be generated for each linked list. In an out-of-order core, multiple lists can be accessed at a time, as it can tolerate up to a certain number of outstanding cache-misses. Therefore, by controlling the number of lists and measuring the performance of the benchmark, we can determine how many outstanding misses one core can generate at a time, which we call *local MLP*. We also vary the number of benchmark instances from one to four and measure the aggregate bandwidth to investigate the parallelism of the entire shared memory hierarchy, which we call *global MLP*.

Figure 4 shows the results. Let us first focus on a single instance results. For Cortex-A7, increasing the number of lists (X-axis) does not have any performance improvement. This is because Cortex-A7 is in-order architecture in which only one outstanding request can be made at a time. For Cortex-A9, Cortex-A15, and Nehalem—all out-of-order architecture based—performance improves as the number of lists increases until 4, 6, and 10 lists, respectively, suggesting their local MLPs. As we increase the number of benchmark instances, the point of saturation becomes shorter in the out-of-order cores. In case of Cortex-A15^O and Cortex-A15^T, the aggregate bandwidth saturates at three lists when four instances are used. This suggests that the global MLP of both Cortex-A15 is close to 12; according to [3], Cortex-A15's integrated LLC (L2) can support up to 11 outstanding cache-misses (global MLP of 11). In case of Nehalem, performance saturates when per-instance MLP is about four with four instances, suggesting the global MLP of 16. [1] Lastly, in case of Cortex-A9,

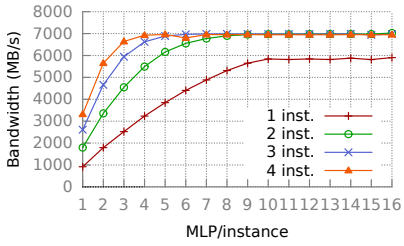
[1] According to [17], Nehalem architecture supports up to 32 outstanding LLC cache-misses. The observed global MLP in the platform is likely limited by MLP of the DRAM, not by MLP of the LLC.



(a) Cortex-A7 (b) Cortex-A9



(c) Cortex-A15^O (d) Cortex-A15^T



(e) Nehalem

Fig. 4 Aggregate memory bandwidth as a function of MLP (the number of lists) per benchmark.

Table 2 Local and global MLP

	Cortex-A7	Cortex-A9	Cortex-A15^O	Cortex-A15^T	Nehalem
local MLP	1	4	6	6	10
global MLP	4	4	11	11	16

both local and global MLP appear to be 4. [2] Table 2 shows the identified MLP of each platform.

Note first that all architectures, including in-order based Cortex-A7, support significant parallelism in the shared memory hierarchy (global MLP). This suggests that non-blocking caches are used in COTS multicore processors. In case of the Cortex-

[2] Cortex-A9 was released much earlier (2007) than Cortex-A7 (2011) and its cache-line size is smaller (32B/line) than the others (64B/line); we suspect these are the reasons of its relatively low memory performance.

A7, its local MLP is one because it is in-order architecture based and only one outstanding request can be made at a time. On the other hand, the other three architectures are out-of-order based and therefore can generate multiple outstanding requests. Moreover, note that the aggregated parallelism of the cores (the sum of local MLP) exceeds the parallelism supported by the shared memory hierarchy (global MLP) in the out-of-order architectures. As we will demonstrate in the next subsection, this can cause serious additional interference that is not handled by the existing cache partitioning techniques.

3.3 Understanding Interference in Non-blocking Caches

While most previous research on shared cache has focused on unwanted cache-line evictions that can be solved by cache partitioning, little attention has been paid to the problem of shared MSHRs in non-blocking caches. As we will see later in this section, cache partitioning does not necessarily provide cache access timing isolation even when the application's working-set fits entirely in a dedicated cache partition, due to contention in the shared MSHRs.

3.3.1 Methodology and synthetic workloads

To find out worst-case interference, we use various combinations of two micro-benchmarks, *Latency* and *Bandwidth*, which we call the *IsolBench* suite. Latency is a pointer chasing synthetic benchmark, which accesses a randomly shuffled single linked list. Due to data dependencies, Latency can only generate one outstanding request at a time. Bandwidth is another synthetic benchmark, which sequentially reads or writes a big array; we henceforth refer *BwRead* as Bandwidth with read accesses and *BwWrite* as the one with write accesses. Unlike Latency, Bandwidth can generate multiple parallel memory requests on an out-of-order core as it has no data dependency.

Table 3 shows the workload combinations we used. Note that the texts with parentheses—(LLC) and (DRAM)—indicate working-set sizes of the respective benchmark. In case of (LLC), the working size is configured to be smaller than 1/4 of the shared LLC size, but bigger than the size of the last core-private cache.[3] As such, in case of (LLC), all memory accesses should be LLC hits. In case of (DRAM), the working-set size is the twice the size of the LLC so that all memory accesses result in LLC misses.

In all experiments, we first run the subject task on Core0 and collect its solo execution time. We then co-schedule an increasing number of co-runners on the other cores (Core1-3) and measure the response times of the subject task. Note that in all cases, we evenly partition the shared LLC among the four cores (i.e., each core gets 1/4 of the LLC space) and each task is assigned to a dedicated core and a dedicated cache partition. Note also that the working-set of each subject benchmark is accessed multiple times to warm-up the cache.

[3] The last core-private cache is L1 for ARM Cortex-A7, A9, and A15 while it is L2 for Intel Nehalem.

Table 3 Workloads for cache-interference experiments.

Experiment	Subject	Co-runner(s)
Exp. 1	Latency(LLC)	BwRead(DRAM)
Exp. 2	BwRead(LLC)	BwRead(DRAM)
Exp. 3	BwRead(LLC)	BwRead(LLC)
Exp. 4	Latency(LLC)	BwWrite(DRAM)
Exp. 5	BwRead(LLC)	BwWrite(DRAM)
Exp. 6	BwRead(LLC)	BwWrite(LLC)

3.3.2 Exp. 1: Latency(LLC) vs. BwRead(DRAM)

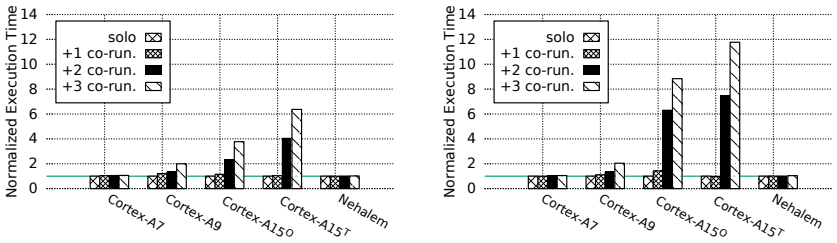
In the first experiment, we use the Latency benchmark as a subject and the BwRead benchmark as co-runners. Recall that BwRead has no data dependency and therefore can generate multiple outstanding memory requests on an out-of-order processing core (i.e., Cortex-A9, Cortex-A15^O, Cortex-A15^T and Intel Nehalem). Figure 5(a) shows the results. For Cortex-A7 and Intel Nehalem, Cache-partitioning is shown to be effective in providing timing isolation. For Cortex-A9, A15^O, and A15^T, however, the response times are still increased by up to 2.0X, 3.7X and 6.4X, respectively. This is an unexpectedly high degree of interference considering the fact that the cache-lines of the subject benchmark, Latency, are not evicted by the co-runners as a result of cache partitioning; in other words, the cache-hit accesses of the Latency benchmark are being delayed by co-runners.

3.3.3 Exp. 2: BwRead(LLC) vs. BwRead(DRAM)

To further investigate this phenomenon, the next experiment uses the BwRead benchmark for both the subject task and the co-runners. Therefore, both the subject and co-runners now generate multiple outstanding memory requests to the shared memory subsystem in out-of-order architectures. Figure 5(c) shows the results. While cache partitioning is still effective for Cortex-A7, the same is not true for the other platforms: Cortex-A9, A15^O, A15^T and Nehalem now suffer up to 2.1X, 10.6X, 10.4X and 7.9X slowdowns, respectively. The results suggest that *cache-partitioning does not necessarily provide expected performance isolation benefits in out-of-order architectures*. We initially suspected the cause of this phenomenon is likely the bandwidth contention at the shared cache, similar to the DRAM bandwidth contention [45]. The next experiment, however, shows it is not the case.

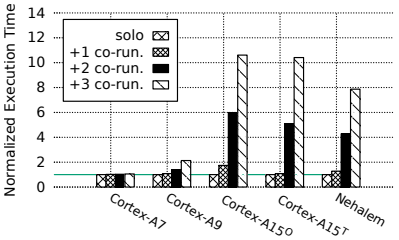
3.3.4 Exp. 3: BwRead(LLC) vs. BwRead(LLC)

In this experiment, we again use the BwRead benchmark for both the subject and the co-runners but we reduce the working-set size of the co-runners to (LLC) so that they all can fit in the LLC. If the LLC (bus) bandwidth contention is the problem, this experiment would cause even more slowdowns to the subject benchmark as the co-runners now need more LLC bandwidth. Figure 5(e), however, does not support this hypothesis. On the contrary, the observed slowdowns in all out-of-order cores

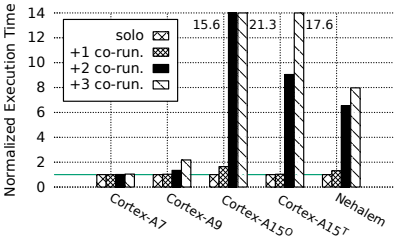


(a) Exp.1: Latency(LLC) vs. BwRead(DRAM)

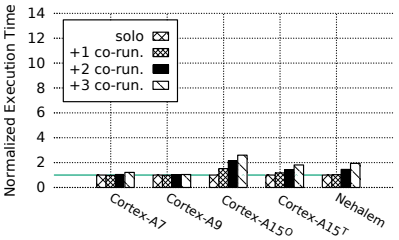
(b) Exp.4: Latency(LLC) vs. BwWrite(DRAM)



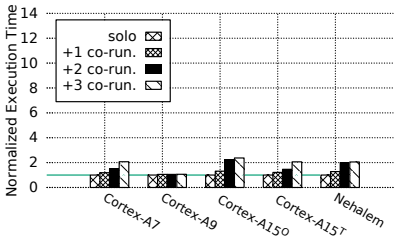
(c) Exp.2: BwRead(LLC) vs. BwRead(DRAM)



(d) Exp.5: BwRead(LLC) vs. BwWrite(DRAM)



(e) Exp.3: BwRead(LLC) vs. BwRead(LLC)



(f) Exp.6: BwRead(LLC) vs. BwWrite(LLC)

Fig. 5 Normalized execution times of the subject tasks, co-scheduled with co-runners on cache partitioned quad-core systems. Each task (both subject and co-runners) runs on a dedicated core and a dedicated cache partition. Also, each subject task's working-set size is smaller than its dedicated cache partition size. (i.e., its memory accesses are cache hits.)

are much less, compared to the previous experiment in which co-runners' memory accesses are cache misses and therefore use less cache bandwidth.

3.3.5 Exp. 4,5,6: Impact of write accesses

In the next three experiments, we repeat the previous three experiments except that now we use BwWrite benchmark as co-runners. Note that BwWrite updates a large array and therefore generates a line-fill (read) and a write-back (write) for each memory access. Figure 5(b), 5(d), and 5(f) show the results. Compared to BwRead, using BwWrite generally results in even worse interference to the subject tasks.

3.3.6 The sources of interference

MSHR contention: To understand this phenomenon, we first need to understand how non-blocking caches process cache accesses from the cores. As described in Section 2, MSHRs are used to allow multiple outstanding cache misses. If all MSHRs are in use, however, the cores can no longer access the cache—both hits and misses—until a free MSHR becomes available [2]. Because servicing memory requests from DRAM takes much longer than doing it from the LLC, cache-miss requests occupy MSHR entries longer. This causes a shortage of MSHRs, which will in turn block additional memory requests to the cache even when they are cache hits. The subject tasks generally suffer even more slowdowns when running write heavy co-runners (e.g., BwWrite) because the additional write-back traffic delays the processing of line-fills, which in turn exacerbate the shortage of MSHRs. The high-degree of interference observed in Exp. 1, 2, 4 and 5 on the out-of-order cores (all but the Cortex-A7, which is in-order) can be explained as the result of the MSHR contention because the co-runners stress the MSHRs, but not the cache bus (cache bus will be idle when the cache is blocked due to full MSHRs.) In Cortex-A7, the only in-order architecture, however, the observed interference is near zero in the four experiments because the MSHRs in its L2 cache never be stressed as each core only can generate one outstanding miss at a time.

Cache bus contention: On the other hand, the observed interference in Exp. 3 and 6 is not the result of MSHR contention because, in the two experiments, nearly all accesses to the LLC—from both the subject and co-runners—are cache-hits and therefore LLC's MSHRs are not stressed. Instead, the chief source of interference is likely the cache bus contention as all cores are trying to access the same shared bus to the LLC. Note, however, that the degree of interference in Exp. 3 and 6—the effect of cache bus contention (up to 2.6X slowdown on Cortex-A15^O)—is much *smaller* than those in Exp. 1, 2, 4 and 5—the effect of MSHR contention (up to 21.3X slowdown on Cortex-A15^O). This is because cache bus speed is much faster than DRAM speed.

In summary, from the experiments, we deduce that MSHR contention is a major interference source in out-of-order architectures. We further validate the finding with a cycle-accurate architecture simulator in Section 4.

3.4 Impact to Real-Time Applications

So far, we have shown the impact of MSHR contention using a set of synthetic benchmarks. The next question is how significant the MSHR contention problem is to worst-case execution times (WCETs) of real-world real-time applications.

To find out, we use a set of benchmarks from EEMBC [1] and SD-VBS [40] benchmark suites as real-time workloads. To focus on contention at the shared cache-level, we carefully chose the benchmarks with the following two characteristics: 1) high L1 miss rates and 2) low LLC miss rates. The first is to filter out those benchmarks which can fit entirely in private L1 cache and the second is to filter out those that heavily depend on DRAM performance. Table 4 shows the Miss-Per-

Table 4 Benchmark characteristics

Benchmark	L1-MPKI	L2-MPKI	Description
EEMBC Automotive, Consumer [1]			
aifftr01	3.64	0.00	FFT (automotive)
aiifft01	3.99	0.00	Inverse FFT (automotive)
cacheb01	2.14	0.00	Cache buster (automotive)
rgbhpg01	1.59	0.00	Image filter (consumer)
rgbyiq01	3.81	0.01	Image filter (consumer)
SD-VBS: San Diego Vision Benchmark Suite [40]. (input: sqcif)			
disparity	56.92	0.13	Disparity map
mser	16.12	0.57	Maximally stable regions
svm	7.81	0.01	Support vector machines

Kilo-Instructions (MPKI) characteristics of the benchmarks on a Cortex-A15 setting (32KB L1-I/D, 512KB L2 cache partition [4]).

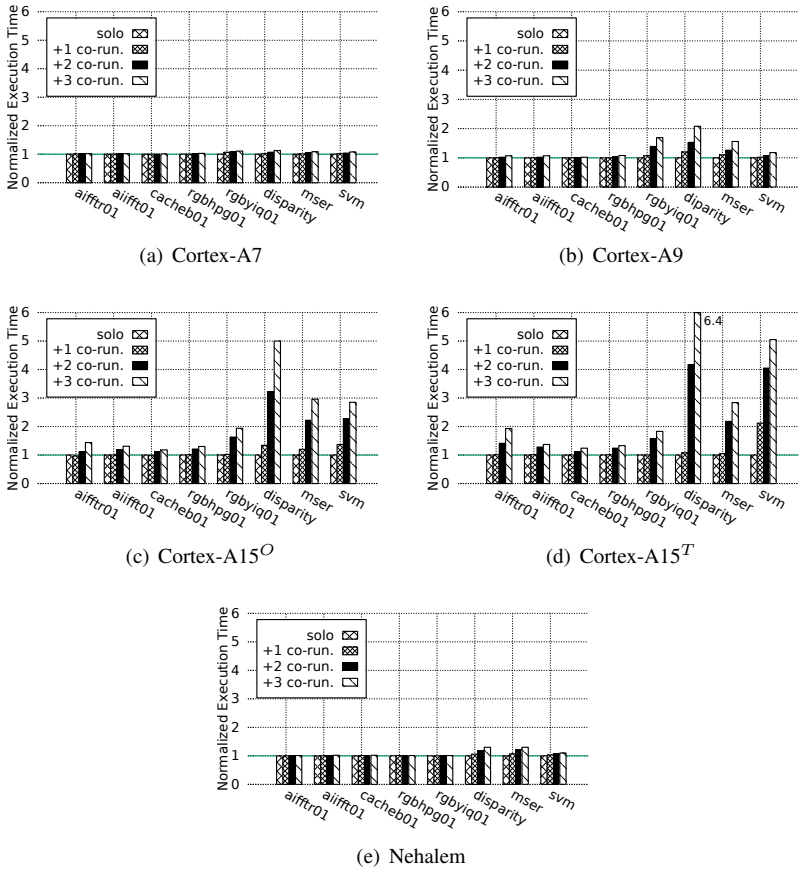
We measured their execution times first alone in isolation and then with multiple instances of the BwWrite(DRAM), which has shown to cause the highest delays in the previous synthetic experiments. In all experiments, the LLC is evenly partitioned on a per-core basis and the benchmarks are scheduled using the `SCHED_FIFO` real-time scheduler in Linux to minimize OS interference.

Figure 6 shows the results. As expected, Cortex-A7 shows good isolation as we observe no significant WCET increases across all benchmarks. On the other hand, Cortex-A9, A15^O, A15^T show significant execution time increases in many of the benchmarks, even though they all access their own private cache partitions. The cause of this is again likely due to MSHR contention. In Cortex-A9, we observe up to 2.08X (108%) WCET increase for the *disparity* benchmark; in Cortex-A15^O and Cortex-A15^T, we observe up to 5.0X and 6.4X WCET increase, respectively, for the same benchmark. Lastly, unlike the IsolBench results in the previous subsection (Section 3.3.1), Nehalem shows good isolation performance for the EEMBC and SD-VBS benchmarks. This is because it has additional private L2 cache (256KB) that absorbs most of L1 cache misses of these benchmarks; as a result, its shared LLC (L3) is rarely accessed when running the benchmarks and the observed WCET increases are much smaller (up to 30% WCET increase) than the other out-of-order cores.

While the overall trend is similar for both EEMBC and SD-VBS benchmarks, the latter tend to suffer substantially higher delays than the former benchmarks. This is because the SD-VBS benchmarks access the shared LLC much more frequently (i.e., higher L1 MPKI rates) than the EEMBC benchmarks and, therefore, suffer more from LLC lock-ups due to MSHR contention.

In summary, while cache space competition is certainly an important source of interference, eliminating it, via cache-partitioning, does not necessarily provide expected isolation in modern COTS multicore platforms due to MSHR contention.

[4] We used the gem5 cycle-accurate simulator, described in Section 4, to analyze the MPKI characteristics of the benchmarks



(a) Cortex-A7 (b) Cortex-A9

(c) Cortex-A15^O (d) Cortex-A15^T

(e) Nehalem

Fig. 6 MSHR contention effects on WCETs of EEMBC and SD-VBS benchmarks.

4 Understanding Isolation and Throughput Impacts of Cache MSHRs

In this section, we study isolation and throughput impacts of MSHRs in non-blocking caches, by exploring different MSHR configurations using a cycle accurate full system simulator.

4.1 Isolation Impact of MSHRs in Shared LLC

In this experiment, we study how the number of MSHRs at the shared LLC affects the MSHR contention problem of a multicore system. For the study, we use the Gem5 simulator [5] and configure the simulator to approximately model a Cortex-A15 quad-core system, which has been shown to suffer the highest degree of MSHR contention in our real platform experiments. The baseline simulation parameters are shown in Ta-

Table 5 Baseline simulator configuration

Core	Quad-core, out-of-order, 1.6GHz ROB: 40, IQ: 32, LSQ: 16/16 entries
L1-I/D caches	private 32/32 KiB (2-way)
L2 cache	shared 2 MiB (16-way), no h/w prefetcher
DRAM controller	64/64 read/write buffers, FR-FCFS [16], open-adaptive page policy
DRAM module	LPDDR2@533MHz, 1 rank, 8banks

ble 5 [5]. On the simulator, we run a full Linux 3.14 kernel, patched with PALLOC [45] to partition the LLC, as we have done in the real platform experiments.

Using the simulator, we evaluate three different MSHR configurations: *MSHR(6/8), MSHR(6/12), and MSHR(6/24)*. The numbers in a parenthesis represents L1 (data) and L2 MSHRs, respectively. At MSHR(6/8), for example, each core's private L1 cache has 6 MSHRs (i.e., up to 6 outstanding misses per core) and the shared L2 cache has 8 MSHRs (up to 8 outstanding misses of all cores). For each MSHR configuration, we repeat the cache interference experiments described in Section 3. Again, as in the previous real platform experiments, the LLC is evenly partitioned among the four cores and all tasks (both the subject and co-runners) are given their own private cache partitions. In other words, observed delays, if any, are not caused by cache space evictions.

Figure 7 shows the results of the six IsolBench workloads (Table 3). As expected, when the number of L2 MSHRs is not big enough to support parallelism of the cores, the subject tasks suffer significant delays due to cache (shared L2) lock-ups caused by MSHR contention. At MSHR(6/8), we observe up to 14.4X slowdown, which is driven by a sharp increase in the number of blocked cycles of the L2 cache. As we increase the L2 MSHRs, however, the delays decrease. At MSHR(6/24), in all but Exp. 3 and Exp. 6, the subject tasks achieve near perfect isolation as increased L2 MSHRs eliminates MSHR contention. In cases of the Exp. 3 and Exp. 6, eliminating MSHR contention does not result in ideal isolation because the main source of the delays is limited cache bandwidth, not MSHR contention. Note that in the two experiments, almost all memory accesses of both subject and co-runners are L2 cache hits, which do not allocate MSHRs.

Figure 8 shows the results of EEMBC and SD-VBS benchmarks, which are measured with three instances of BwWrite(DRAM) co-runners. The results are in tandem with the IsolBench results. At MSHR(6/8), the subject task suffers contention—up to 1.43X slowdown for EEMBC *cacheb01* and 4.3X slowdown for SD-VBS *disparity*. At MSHR(6/24), interference is almost completely eliminated for most benchmarks. Notable exceptions are *disparity* and *mser* from the SD-VBS benchmark suite. For the two benchmarks, while isolation performance is significantly improved, they still

[5] We use gem5's detailed out-of-order CPU model, O3CPU. The CPU parameters are largely based on gem5's default ARM configuration, which is, according to [15], similar to Cortex-A15. However, because not all details of Cortex-A15 are publicly available by ARM, some of the parameters could be different from a real one. For example, the reorder buffer (ROB) size of Cortex-A15 is referred as 128 in [34], 60 in [6], and 40 in the default arm configuration of gem5. We do not know which is the correct ROB value. However, we would like to stress that our main focus is not in accurate modeling of a Cortex-A15 platform but in understanding relative impacts of MSHRs in out-of-order cores.

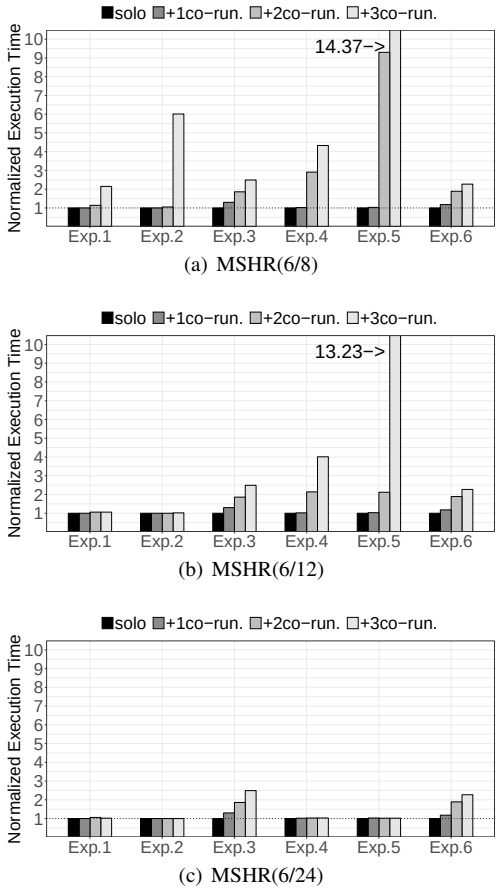
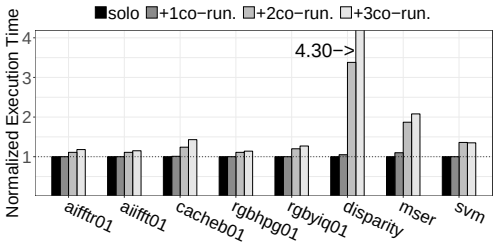


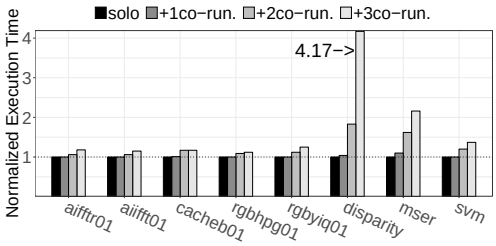
Fig. 7 Effects of MSHR configurations on WCETs of IsolBench

suffer considerable delays. This can be explained as a result of their relatively high DRAM access rates (see L2 MPKI values at Table 4). Because the co-runners—BwWrite(DRAM) instances—are highly memory (DRAM) intensive, they cause severe contention at the DRAM controller queues, which in turn delays memory requests from the subject benchmarks; we observe a large increase in the average queue length and the average memory access latency in the memory controller statistics of the simulator. (COTS DRAM controller-level contention is an important orthogonal problem, which has been actively studied in recent years [25,46,22,24].)

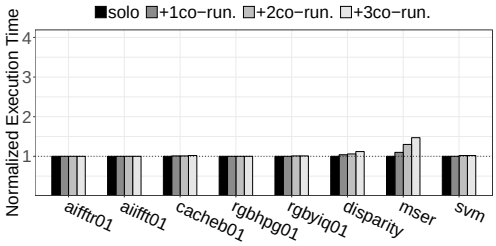
The results validate that MSHRs in a shared LLC can be a significant source of contention, which causes frequent cache lockups even when the cache is spatially partitioned. The results also show that eliminating MSHR contention, by increasing the number of MSHRs in the shared LLC, significantly improves isolation performance.



(a) MSHR(6/8)



(b) MSHR(6/12)



(c) MSHR(6/24)

Fig. 8 Effects of MSHR configurations on WCETs of EEMBC and SD-VBS benchmarks

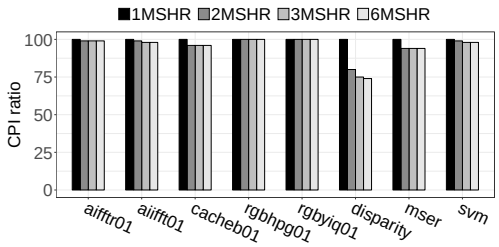
4.2 Throughput Impact of MSHRs in Private L1 Cache

Increasing the number of MSHRs in the shared LLC is, however, not always desirable because supporting many highly associative MSHRs can be challenging due to increased area and logic complexity [37]. Furthermore, it becomes even more difficult as the number of cores increases and each core supports more memory-level parallelism (higher local MLP).

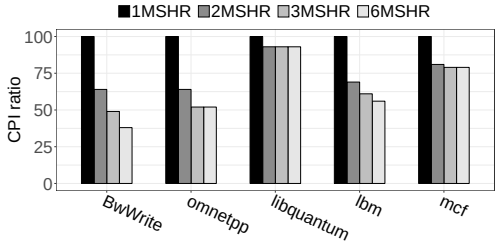
Another simple solution to eliminate MSHR contention is reducing the number of MSHRs in the private L1 caches (reduction of local MLP), instead of increasing the number of LLC MSHRs. However, an obvious downside of this approach is that it could affect the core's single-thread performance. The question is, then, how important is the core-level memory-level parallelism (local MLP) to application performance?

In the following experiments, we evaluate the single-thread performance impact of the number of L1 MSHRs using a set of benchmarks from EEMBC, SD-VBS,

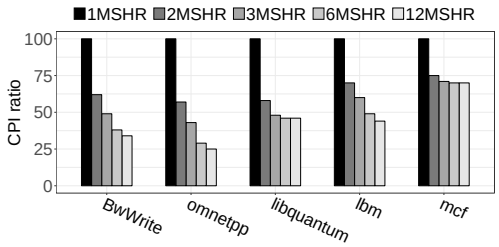
and SPEC2006 benchmark suites. The benchmarks from EEMBC and SD-VBS are the same as the ones used in previous experiments: cache intensive (high L1 MPKI) but not DRAM intensive (low L2 MPKI). On the other hand, we also choose highly memory (DRAM) intensive SPEC2006 benchmarks for better comparison. On the simulator, we vary the number of L1 MSHRs from 1 to 6, while fixing the number of L2 MSHRs at 12. Note that one L1 MSHR means that the cache will block on each miss and therefore is equivalent to a blocking cache. For each L1 MSHR configuration, we measure each benchmark's Cycles-Per-Instructions (CPI).



(a) EEMBC, SD-VBS



(b) SPEC2006 & BwWrite



(c) SPEC2006 & BwWrite (inf. core resources)

Fig. 9 Performance impact of MSHRs in private L1 cache. (Lower is Better)

Figure 9(a) shows the results of EEMBC and SD-VBS benchmarks, normalized to the one L1 MSHR configuration. For EEMBC benchmarks, performance does not improve much as the number of L1 MSHRs increases. For example, we observe only 4% improvement for *cacheb01* with 2 MSHRs and additional MSHRs do not make any difference in performance. For SD-VBS vision benchmarks, performance im-

provement is more significant. In particular, *disparity* shows up to 26% improvement with 6 MSHRs, although the difference between 6MSHRs and 2MSHRs is relatively small. These results can be explained as follows: The working sets of the EEMBC and SD-VBS benchmarks fit in the L2 cache and therefore most L1 misses result in L2 cache hits. Because L2 cache is relatively fast, compared to DRAM, the L1 MSHRs quickly become available as soon as the L2 cache returns the data. As a result, only a small number of MSHRs can deliver most of the performance benefits of out-of-order cores.

On the other hand, Figure 9(b) and 9(c) show the results of SPEC2006 and BwWrite benchmarks. The two figures differ in that in Figure 9(c), we significantly increased the sizes of Instruction Queue (IQ), Reorder buffer (ROB), and Load/Store Queue (LSQ) to simulate more aggressive out-of-order cores. In general, memory-intensive benchmarks greatly benefit from the increase of L1 MSHRs as it reduces memory related stalls. And the performance improvements are even greater on more aggressive out-of-order cores. For example, with 6 MSHRs, *BwWrite*, *lbm*, *libquantum*, and *omnetpp*, achieve more than 50% performance improvements on the aggressive out-of-order core setting.

These results show that throughput impact of the number of MSHRs at core-private L1 caches is highly *application dependent*. This observation motivates us to propose a solution to eliminate MSHR contention problem without increasing MSHRs as we will describe in Section 5.

5 OS Controlled MSHR Partitioning

In this section, we propose a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for real-time systems.

5.1 Assumptions

We consider a multicore system with m identical cores. The cores are out-of-order architecture-based and each core equips a non-blocking private L1 data cache with N_{mshr}^{L1} MSHRs (i.e., local MLP of N_{mshr}^{L1}). Also, there is a non-blocking shared LLC (L2) with N_{mshr}^{LLC} MSHRs (i.e., global MLP of N_{mshr}^{LLC}). We assume the sum of the local MLP is bigger than the MLP of the shared cache—$m \times N_{mshr}^{L1} > N_{mshr}^{LLC}$—as we experimentally observed in the real COTS multicore platforms shown in Section 3.2. This means that the shared LLC can suffer from MSHR contention when its MSHRs are exhausted. We assume the task system is composed of a mix of critical real-time tasks and best-effort tasks. We assume that the tasks are partitioned on a per-core basis and each core uses a two-level hierarchical scheduling framework that first schedules the real-time tasks with a fixed priority scheduler and then schedules the best-effort tasks with a fairness focused general purpose scheduler (e.g., CFS in Linux). Note that any core may execute both real-time tasks and best-effort tasks. In other words, there are no designated "real-time cores."

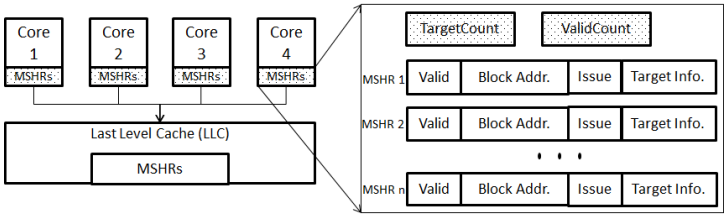


Fig. 10 Proposed MSHR Architecture. Grey areas indicate our proposed addition.

5.2 MSHR Partitioning Hardware Mechanism

In order to eliminate MSHR contention, we propose to dynamically control the number of usable MSHRs in the private L1 caches. We achieve this via a low cost extension to the L1 caches. Figure 10 shows the proposed extension. We add two hardware counters $TargetCount$ and $ValidCount$ for each L1 cache controller. The $ValidCount$ tracks the number of total valid MSHR entries (i.e., entries with outstanding memory requests) of the cache and is updated by the hardware. The $TargetCount$ defines the maximum number of MSHRs that can be used by the core and is set by the system software (OS). If $ValidCount_i >= TargetCount_i$, the cache immediately locks up. System software can update $TargetCount$ registers by executing privileged instructions (e.g., `wrmsr` instructions in Intel [18]). By controlling the value of $TargetCount$, the OS can effectively control the core's local MLP. The added area and logic complexity is minimal as we only need two additional counter registers and one comparator logic.

To eliminate MSHR contention, the OS employs a partitioning scheme that limits the sum of $TargetCount$ values of all L1 caches be equal or less than the number of MSHRs of the (shared) LLC, while also respecting the maximum number of MSHRs of each private L1 cache. In other words, the OS would satisfy the following inequalities.

$$\sum_{i=1}^{m} TargetCount_i \leq N_{mshr}^{LLC}, \tag{1}$$

$$1 \leq TargetCount_i \leq N_{mshr}^{L1} \tag{2}$$

For example, in a quad-core system in which the LLC has 12 MSHRs and each core's L1 cache has 6 MSHRs, the OS may set $TargetCount$ value of all L1 caches to 3 (half of the physically allowed number 6) to eliminate MSHR contention.

However, care must be taken to minimize potential throughput reduction because some workloads may be greatly affected by the reduction of parallelism offered by the L1 cache. For example, according to our experiments in Section 4.2, assigning $TargetCount = 1$ to a core that executes the *lbm* SPEC2006 benchmark would cause more than 40% performance reduction.

```
1  int redist_be_mshr()
2  {
3      // fairly distribute non-reserved MSHRs
4      m_rt = 0;
5      mshr_remain = N_mshr^LLC;
6      for (i = 0...m-1) {
7          if (mshr_part[i] > 0) {
8              m_rt ++;
9              mshr_remain -= mshr_part[i];
10         }
11     }
12     R_nrt = ceiling(mshr_remain / (m - m_rt));
13     for (i = 0...m-1) {
14         if (mshr_part[i] == 0) {
15             TargetCount_i = min(R_nrt, mshr_remain);
16             mshr_remain -= R_nrt
17         }
18     }
19     return m_rt;
20 }
21
22 void prepare_task_switch(prev, next)
23 {
24     // myid = local cpu index
25     myid = smp_processor_id();
26     if (next->mshr_reserve > 0) {
27         // enable/update MSHR partitioning
28         R = next->mshr_reserve;
29         mshr_part[myid] = R;
30         TargetCount_myid = R;
31         redist_be_mshr();
32     } else if (prev->mshr_reserve > 0) {
33         mshr_part[myid] = 0;
34         m_rt = redist_be_mshr();
35         if (m_rt > 0)
36            return;
37         // disable MSHR partitioning
38         for (i = 0...m-1) {
39             TargetCount_i = N_mshr^L1;
40         }
41     }
42 }
```

Fig. 11 MSHR partitioning algorithm in the CPU scheduler.

5.3 OS Scheduler Design

We enhance the OS scheduler to efficiently utilize MSHRs while eliminating the MSHR contention. First, the OS provides a system call that allows users to reserve a certain number of MSHRs needed by the chosen task. We assume that all critical real-time tasks reserve MSHRs while best-effort tasks do not. The MSHR reservation information of each (real-time) task is kept in the OS (e.g., `task_struct` in Linux) and used by the scheduler when the task is being scheduled. We limit the maximum

number of reservable MSHRs of a task to N_{mshr}^{LLC}/m to guarantee reservation when the task is scheduled. This is needed because, in our model, all m cores may execute m real-time tasks, all of which request MSHR reservation, at the same time. The best-effort tasks equally share remaining MSHRs—i.e., those that are not used by the currently scheduled real-time tasks.

To minimize unnecessary throughput impact to best-effort tasks, we apply MSHR partitioning only when at least one core is executing a real-time task with MSHR reservation. When enabled, reserved MSHRs of each task is enforced globally by the OS scheduler by updating the $TargetCount$ registers of all cores to satisfy the Eqs. 1 and 2. Note that updating a core's $TargetCount$ register controls the core's maximum L1 MSHRs, which in turn control the maximum necessary LLC MSHRs for the core. Thus, by controlling each core's L1 MSHRs, the scheduler effectively partitions LLC MSHRs among the cores.

Figure 11 shows the algorithm. `prepare_task_switch()` is called by the CPU scheduler to prepare a context switch from $prev$ task to $next$ task. Note that in Linux, each core independently schedules the tasks in the core's run queue. On a context switch, if the next scheduled task is a real-time task that requires MSHR reservation (Line 27-32), we configures the $TargetCount$ register of the corresponding core (Line 31). Note that R denotes the number of reserved MSHRs of the *next* task. It then determines the number of available MSHRs (excluding reserved MSHRs), which is then fairly distributed to the cores that execute best-effort tasks by calling `redist_be_mshr()` (Line 32). On the other hand, if the $prev$ task reserves MSHRs and the $next$ task does not, then the previously reserved MSHRs of the $prev$ task will be fairly re-distributed to all best-effort tasks (Line 34-35). if no currently running tasks wish to reserve MSHRs, the scheduler resets the $TargetCount$ registers of all cores to the maximum (Line 38-41), which effectively disables MSHR reservation.

6 Evaluation

In this section, we evaluate isolation and throughput impacts of the proposed approach through a set of experiments.

6.1 Setup

We use the same experimental setup as explained in Section 4—a Quad-core Cortex-A15 platform model on the Gem5 simulator having 6 per-core L1 MSHRs and 12 L2 MSHRs—as the baseline hardware platform. On the simulator, we implement the proposed hardware extension by modifying its cache subsystem. We modify the Linux kernel's scheduler to communicate with the simulator to adjust the number of MSHRs (`prepare_task_switch()` in `kernel/sched/core.c`)

In the following, we compare two system configurations: (1) *'cache part'* and (2) *'cache+mshr part'*. In *cache part*, we apply only cache partitioning. In *cache+mshr part*, on the other hand, we use the proposed OS-controlled MSHR partitioning approach in addition to the cache partitioning. In this configuration, when a real-time

task is released, the OS reserves 2 MSHRs for the task and the rest of the non-reserved MSHRs are equally shared by the best-effort tasks. While the "right" number of reserved MSHRs for a task may vary depending on the task's characteristics, we choose to reserve 2 MSHRs because the real-time tasks used in our experiments do not benefit from more than 2 MSHRs (see Section 4.2.)

6.2 Isolation Effect of MSHR Partitioning

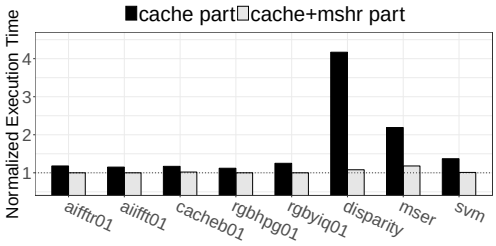


Fig. 12 Normalized exec. times of EEMBC and SD-VBS

First, we investigate the isolation impact of the proposed MSHR partitioning technique by repeating the experiment described in Section 4.1. Briefly, the experiment is as follows: we run each of the EEMBC and SD-VBS benchmarks as a real-time task, with three instances of the BwWrite benchmark as (non-real-time) co-runners; we then measure the response time of the real-time tasks under two system configurations: *'cache part'* and *'cache+mshr part'*. Again, in this setup, each core executes only one task each (either the real-time task or a co-runner).

Figure 12 shows the normalized execution time of the real-time tasks. Note that we use the performance measured in isolation under the equal cache partitioning setup—without applying MSHR partitioning—as baseline. As can be seen clearly in the figure, under *cache+mshr part*, the interference is almost completely eliminated because the MSHR contention problem has been eliminated by ensuring dedicated MSHRs in the shared LLC to the real-time tasks. In other words, we could eliminate MSHR contention without increasing the number of LLC MSHRs.

6.3 Case Study: A Mixed Criticality Task System

In this experiment, we model a mixed-criticality task system using four instances of EEMBC benchmarks—*aifftr01, aiifft01, cacheb01 and, rgbhpg01* [6]—as real-time tasks and four instances BwWrite(DRAM) as best-effort tasks, such that both real-time and best-effort tasks are co-scheduled on a single multicore system. We modified the EEMBC benchmarks to run periodically.

[6] We choose the benchmarks with (near) zero L2-MPKI values to avoid DRAM controller level contention.

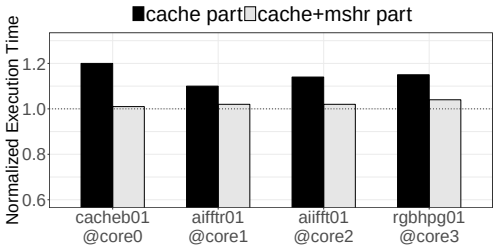


Fig. 13 WCETs of real-time tasks (EEMBC), co-scheduled with best-effort tasks.

The experiment procedure is as follows. We start four BwWrite benchmark instances on Core0, Core1, Core2 and Core3, respectively. While these Bandwidth instances are running in the background, we start the four EEMBC benchmarks, one per core, so that each core runs one real-time task and one best-effort task. As the LLC cache is partitioned on a per-core basis, the two tasks (one real-time and one best-effort) on each core use the same cache partition in this experiment. Our focus in this experiment is inter-core interference, not intra-core interference. Note that the EEMBC benchmarks are scheduled using the `SCHED_FIFO` real-time scheduler in Linux, and therefore they are always prioritized over the BwWrite instances. The EEMBC benchmarks have different periods—20ms, 30ms, 40ms, and 60ms for Core0, 1, 2, and 3 respectively—but their computation times are configured to be approximately 8 milliseconds. Each EEMBC benchmark runs to completion and then sleeps until the next period starts. During this time the core is yielded to the best-effort task (i.e., BwWrite). The experiment is performed for the duration of 120ms (two hyper-periods of the real-time tasks).

Figure 13 shows observed WCETs of the real-time tasks, normalized to their run-alone execution times on the baseline system configuration. In *cache part.*, the real-time tasks suffer significant WCET increases—up to 20% for *cacheb01*—even though they always execute on their own dedicated cores, accessing dedicated cache partitions, due to MSHR contention. In *cache+mshr part.*, on the other hand, the real-time tasks suffer almost no WCET increases because *MSHR contention is eliminated* by the proposed MSHR partitioning scheme. In terms of throughput of the best-effort tasks (BwWrite), we observe 3% throughput reduction in *cache+mshr part* as they are given fewer MSHRs. We believe it is an acceptable trade-off for real-time systems.

7 Other Factors Influencing MSHR Contention

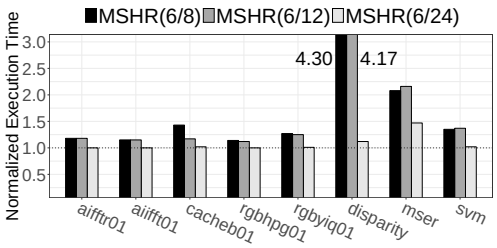
In this section, we study other factors that could influence the MSHR contention problem. Specifically, we would like to answer the following three questions:

1. Is the MSHR contention problem still significant when more realistic (less aggressive) co-runners are used?
2. Do more aggressive and faster out-of-order cores make MSHR contention worse?

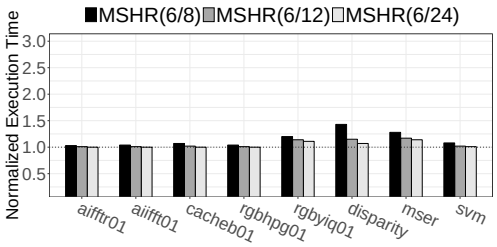
3. What are the impacts of simultaneous multi-threading (SMT) and hardware prefetchers in private and shared caches?

7.1 Aggressiveness of Co-runners

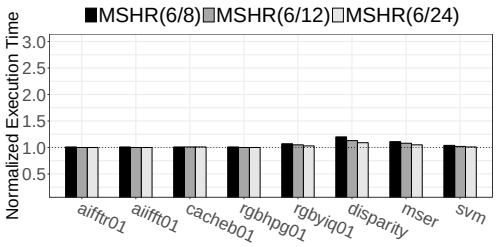
So far, we have used synthetic memory intensive tasks with a high degree of memory-level parallelism (MLP) as co-runners. This is to generate the maximum degree of MSHR contention. An interesting question is whether we can observe significant MSHR contention when more realistic workloads are used as co-runners. To find out, we use two SPEC2006 benchmarks, *lbm* and *omnetpp*, as co-runners and repeat the subset of experiments in Section 4.1 that use EEMBC and SD-VBS benchmarks as the subject tasks. The two SPEC2006 benchmark are chosen as they are relatively memory intensive real-world applications. For this experiment, we use the gem5 simulator as in Section 4.



(a) vs. 3 x BwWrite



(b) vs. 3 x lbm



(c) vs. 3 x omnetpp

Fig. 14 Effects of co-runners on WCETs of EEMBC and SD-VBS benchmarks

Figure 14 shows the results. Note that we observe less severe MSHR contention when we use *lbm* or *omnetpp* as co-runners instead of the synthetic *BwWrite*. In inset (a), in which three instances of synthetic *BwWrite* are used as co-runners [7], the subject tasks (EEMBC and SD-VBS benchmarks) suffer up to 4.3X slowdown (The *disparity* benchmark). In inset (b), in which *lbm* is used as co-runners instead, we observe only up to 1.43X slowdown. In inset (c), in which *omnetpp* is used as co-runners, the impact of MSHR contention is further reduced as we observe only up to 1.20X slowdown. Also, notice that the effect of increasing MSHRs in the shared L2 cache is less dramatic as we use *lbm* and *omnetpp* as co-runners.

In short, the degree of MSHR contention problem is less severe when realistic applications—less aggressive in terms of memory intensity and parallelism than the synthetic ones—are co-scheduled. We would argue, however, that the fact that carefully designed synthetic applications could cause severe slowdowns—by several factors—is an important security and safety issue. For example, a malicious program can potentially mount timing attacks to critical real-time tasks by stressing shared cache MSHRs.

7.2 Aggressiveness of Cores

In an out-of-order core, a stalled instruction, for example, due to a cache-miss or data dependency, does not necessarily block the entire processor pipeline. Instead, a subsequent instruction can be executed ahead of the stalled instruction as long as it is ready—that is, all operands are available and there is an available execution unit.

To enable the out-of-order execution, a typical out-of-order core design employs a number of hardware resources such as Reorder Buffer (ROB), Instruction Queue (IQ) and Load/Store Queue (LSQ). The ROB provides temporary storage for execution results and ensure that the instructions are still appeared to be executed in the instruction order (in-order completion). The LSQ and IQ provide buffer spaces for memory (load and store) and non-memory instructions, respectively, until they become ready to be executed regardless of the instruction order (out-of-order execution). The more these resources are available, the more aggressive (potentially faster) out-of-order execution is possible in general. However, it would also increase the possibility of more severe MSHR contention as more memory instructions can be executed in parallel at a time. As such, we expect that the MSHR contention problem would become *worse* on more advanced (aggressive) processors.

We evaluate this hypothesis with the gem5 simulator. We compare three core configurations: *Baseline@1.6GHz*, *Aggressive@1.6GHz*, and *Aggressive@2.0GHz*. Baseline@1.6GHz is the same setting as in Table 5. In Aggressive@1.6GHz, the sizes of ROB, IQ, and LSQ are increased by a factor of three, to simulate more aggressive cores. Lastly, in Aggressive@2.0GHz, we increase each core's clock speed from 1.6GHz to 2.0Hz to further stress the shared memory hierarchy. Note that the DRAM memory speed is unchanged (533MHz LPDDR2). Note that in all configurations, the MSHR setting is fixed at MSHR(6/12)—i.e., 6 MSHRs in L1 (private)

[7] Note that this is a redrawing of Figure 8, focusing only on the case of three co-runners.

and 12 MSHRs in L2 (shared)—and the L2 cache space is equally partitioned on a per-core basis. For each simulator configuration, we repeat the experiment described in Section 4.1 using the IsolBench, EEMBC, and SD-VBS benchmarks.

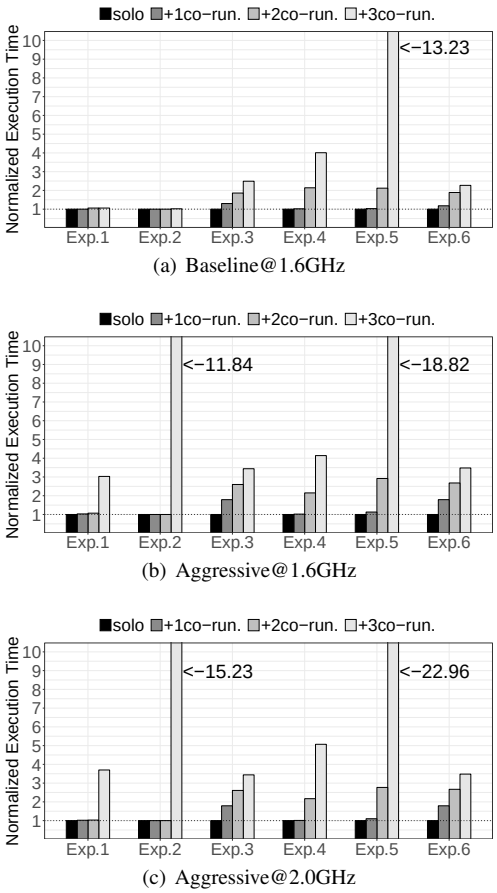
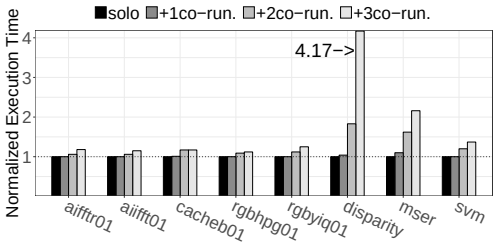
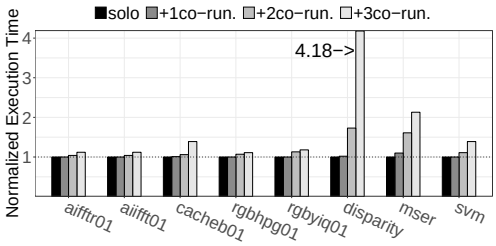


Fig. 15 Effects of core aggressiveness on MSHR contention of IsolBench.

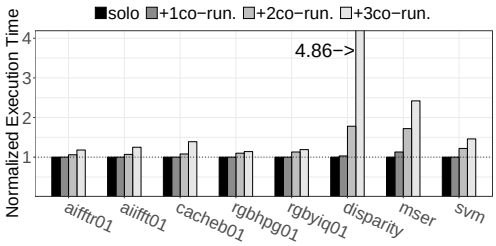
Figure 15 shows the results for the IsolBench suite (see Table 3 for the workload details). Comparing insets (a) and (b), which only differ in their core resources, we can observe significant WCET increases in the latter. In Exp.2, for example, the subject task BwRead(LLC) suffers a dramatic WCET increase when three instances of BwRead(DRAM) are co-scheduled as co-runners. This is because, as we expected, the increased resources result in more slowdowns due to increased MSHR contention—validating our hypothesis. Also as expected, the degree of contention further increases as we increase the speed of the cores in inset(c), because the memory subsystem is further stressed.



(a) Baseline@1.6GHz



(b) Aggressive@1.6GHz



(c) Aggressive@2.0GHz

Fig. 16 Effects of core aggressiveness on MSHR contention of EEMBC and SD-VBS benchmarks.

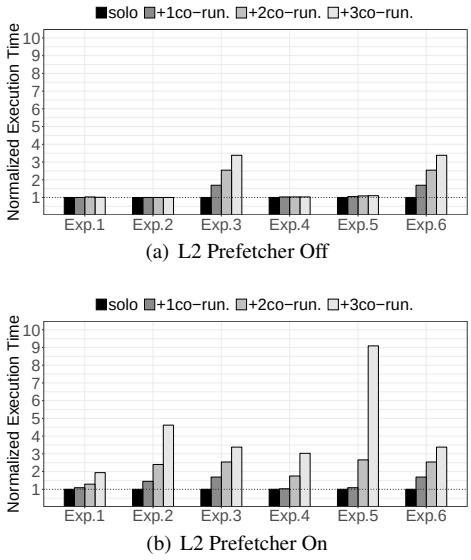
Figure 16 shows the results for the EEMBC and SD-VBS benchmarks. Unlike the IsolBench results (Figure 15), the observed WCET increases are much smaller when EEMBC and SD-VBS benchmarks are the subject benchmarks (The co-runners are still the same BwWrite(DRAM)). This is mainly because they are not memory intensive and, for them, increased core resources do not make much of a difference in their baseline performance.

7.3 SMT Threads and Hardware Prefetchers

In Simultaneous Multi-threading (SMT) processors, multiple hardware threads share not only the memory hierarchy but also much of important core resources such as arithmetic logic units (ALUs). Although such design increases hardware resource utilization (thus improves throughput), it also results in highly unpredictable timing behavior. As such, in real-time systems, it is common to disable SMT threads if it is

possible in the considered processor. For example, most Intel processors allow disabling SMT in BIOS. Similar effect can be achieved at the OS-level by not scheduling tasks on certain logical processors (SMT). Therefore, in this paper, we do not consider SMT-level sharing.

On the other hand, hardware prefetchers in both private and shared caches require more careful consideration. A hardware prefetcher is tightly coupled with a cache and monitors the memory access patterns to the cache and prefetch predicted cach-lines into the cache to minimize latency. A hardware prefetcher in a private L1 cache can increase the likelihood of MSHR contention by generating more memory requests to the shared LLC, starving the LLC's MSHRs. Similarly, a hardware prefetcher in the shared LLC could result in more severe MSHR contention because additional prefetched memory requests to DRAM can quickly exhaust the MSHRs in the shared LLC. Note that our proposed MSHR partitioning scheme in Section 5, which controls the number of valid L1 MSHRs, would still eliminate MSHR contention even if hardware prefetchers are used at the L1 caches because the L1 prefetchers, just like the cores, would use the same L1 MSHRs (which is the case in the gem5 model.) However, if a hardware prefetcher is used at the shared LLC, it may fail to eliminate MSHR contention because the prefetcher's use of LLC MSHRs is not controlled by the OS. Recall that in our proposed scheme, the OS ensures that the sum of valid L1 MSHRs does not exceed the number of LLC MSHRs. However, if the hardware prefetcher in the LLC allocates additional MSHRs to serve prefetch requests, the cores' memory requests can still suffer MSHR contention.

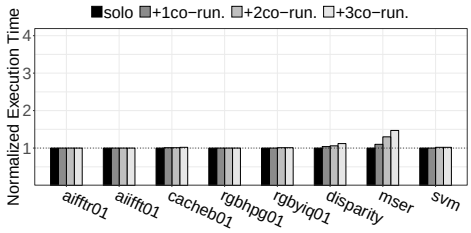


(a) L2 Prefetcher Off

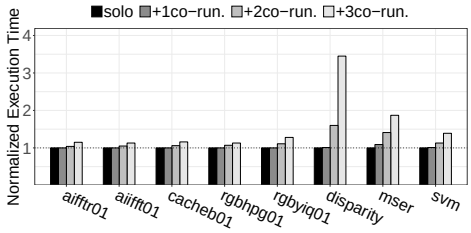
(b) L2 Prefetcher On

Fig. 17 Effects of L2 Prefetcher on Gem5 at MSHR(6/24) for IsolBench.

We investigate the potential impact of the hardware prefetcher of the shared LLC using the gem5 simulator. On the simulator, we repeat the IsolBench, EEMBC, and



(a) L2 Prefetcher Off



(b) L2 Prefetcher On

Fig. 18 Effects of L2 Prefetcher on Gem5 at MSHR(6/24) for EEMBC and SDVBS.

SD-VBS experiments in previous sections after enabling the LLC prefetcher; and compared the results with the ones without the prefetcher. Note that in both configurations, each of the private L1 caches has 6 MSHRs while the shared L2 cache has 24 MSHRs. In other words, the cores alone cannot generate more than 24 concurrent memory requests at a time. Therefore, if MSHR contention is observed, it would have been caused by the LLC prefetcher, which generates additional requests to the LLC.

Figure 17 shows the results for IsolBench. As expected, when the L2 prefetcher is enabled, we observe significantly higher interference—up to 9.2X—than the case of without the prefetcher. This is because the additional prefetch memory requests starve the MSHRs in the shared L2 cache, causing MSHR contention.

Figure 18 shows the results for EEMBC and SD-VBS benchmarks. The overall trend is similar in the sense that with the L2 prefetcher we observe significant MSHR contention even with the increased MSHRs at the shared L2 cache.

A solution to this problem would require close coordination between the cores and the LLC prefetcher. We leave this problem for future work.

8 Related Work

Cache space sharing is a well-known source of timing unpredictability in multicore platforms [4]. Various hardware and software cache partitioning methods have been studied to improve cache access timing predictability. Way-based cache partitioning [35] is the most well-known hardware based approach, which partitions the cache space at the granularity of cache ways. Some embedded processors and a few recent

Intel Xeon processors support way-based cache partitioning [12,19]. However, not all COTS multicore processors support such hardware mechanisms.

Page-coloring is a software-based cache partitioning technique that does not require any special hardware support other than the standard memory management unit (MMU). Therefore, it is more readily applicable to most COTS multicore platforms and has been studied extensively in the real-time systems community [26,30,42,44]. As discussed in 2.2, in page coloring, the OS carefully controls the physical addresses of memory pages so that they can be allocated in specific sets of the cache. By allocating memory pages over non-overlapping sets of the cache, the OS can effectively partition the cache. In recent years, page-coloring has also been applied to partition DRAM banks [29,36,45] and TLB [32]. In this paper, we also use a page-coloring based technique to partition the shared cache.

Cache locking is another technique to improve cache access timing predictability, which has been explored in [30] in combination with page coloring. In the MC^2 project [9], both hardware-based way-partitioning and page-coloring are used to gain more flexibility in partitioning the cache.

While all the aforementioned techniques are effective in eliminating cache space contention problem, they however do not address the problem of MSHR contention.

In the context of general purpose computing systems, hardware based adaptive management of MSHRs has been studied in [10,20,21] to improve throughput and fairness. They use sophisticated hardware mechanisms to periodically estimate the slowdown ratios of the cores and adaptively control the number of MSHRs to reduce memory pressure of the cores that cause high interference. While our work is similar to these works in the sense we also control the number of MSHRs, the key difference is that in our approach it is controlled by the OS to guarantee the absence of MSHR contention in real-time tasks, while in their works it is controlled by complex hardware implementations (no OS involvement), which do not guarantee the absence of MSHR contention.

9 Conclusion

We have shown that cache partitioning does not guarantee predictable cache access timing in COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism (MLP). Through extensive experimentation on real and simulated multicore platforms, we have identified that special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs), can be a significant source of contention. We have proposed a hardware and system software (OS) collaborative approach to efficiently eliminate MSHR contention for multicore real-time systems. Our evaluation results show that the proposed approach significantly improves the cache access timing isolation without noticeable throughput impact.

As future work, we plan to integrate the proposed OS-controlled MSHR management technique with a DRAM management technique [38] to further improve isolation of high-performance multicore real-time systems.

Acknowledgements

This research is supported in part by NSF CNS 1302563.

References

1. EEMBC benchmark suite. `www.eembc.org`.
2. Memory system in gem5. `http://www.gem5.org/docs/html/gem5MemorySystem.html`.
3. ARM. *Cortex-A15 Technical Reference Manual, Rev: r2p0*, 2011.
4. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.
5. N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.
6. E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2013.
7. A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.
8. Certification Authorities Software Team. CAST-32: Multi-core Processors (Rev 0). Technical report, Federal Aviation Administration (FAA), May 2014.
9. M. Chisholm, B. Ward, N. Kim, , and J. Anderson. Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems. In *Real-Time Systems Symposium (RTSS)*, 2015.
10. E. Ebrahimi, C. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335, 2010.
11. D. Eklov, N. Nikolakis, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: quantitative characterization of memory contention. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.
12. Freescale. *e500mc Core Reference Manual*, 2012.
13. A. Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea*, 1998.
14. P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.
15. A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *Performance Analysis of Systems and Software (ISPASS)*, pages 13–22. IEEE, 2014.
16. A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM controllers for future system architecture exploration. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.
17. Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.
18. Intel. *Intel®64 and IA-32 Architectures Software Developer Manuals*, 2012.
19. Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, April 2015.
20. M. Jahre and L. Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 1–10. ACM, 2009.
21. M. Jahre and L. Natvig. A high performance adaptive miss handling architecture for chip multiprocessors. In *Transactions on High-Performance Embedded Architectures and Compilers IV*, pages 1–20. Springer, 2011.
22. J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *Real-Time Systems Symposium (RTSS)*, pages 207–217. IEEE, 2014.
23. R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.
24. H. Kim, D. Bromany, E. Lee, M. Zimmer, A. Shrivastava, J. Oh, et al. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–326. IEEE, 2015.

25. H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
26. H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Real-Time Systems (ECRTS)*, pages 80–89. IEEE, 2013.
27. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA)*, pages 81–87. IEEE Computer Society Press, 1981.
28. J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 213–224. IEEE, 1997.
29. L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 367–376. ACM, 2012.
30. R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.
31. NVIDIA. *NVIDIA Tegra K1 Mobile Processor, Technical Reference Manual Rev-01p*, 2014.
32. S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–13. IEEE, 2015.
33. P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, volume 28. ACM, 2000.
34. A. L. Shimpi and B. Klug. NVIDIA Tegra 4 Architecture Deep Dive, Plus Tegra 4i, Icera i500 & Phoenix Hands On. `http://www.anandtech.com/show/6787/nvidia-tegra-4-architecture-deep-dive-plus-tegra-4i-phoenix-hands-on`.
35. G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture (HPCA)*. IEEE, 2002.
36. N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.
37. J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *International Symposium on Microarchitecture (MICRO)*, pages 409–422. IEEE, 2006.
38. P. Valsan and H. Yun. MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore based Embedded Systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2015.
39. P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.
40. S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *International Symposium on Workload Characterization (ISWC)*, pages 55–64. IEEE, 2009.
41. S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium (RTSS)*, pages 239–243. IEEE, 2007.
42. B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.
43. A. Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, 2(3):315–327, 1994.
44. Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 381–392. ACM, 2014.
45. H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
46. H. Yun, R. Pellizzoni, and P. Valsan. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2015.