Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems

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

Abstract—**In this paper, we show that cache partitioning does not necessarily ensure predictable cache performance in modern COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism (MLP).**

Through carefully designed experiments using three real COTS multicore platforms (four distinct CPU architectures) and a cycle-accurate full system simulator, we show 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; we observe up to 21X WCET increase in a real COTS multicore platform due to MSHR contention.

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 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. We evaluate the effectiveness of our approach using a set of synthetic and macro benchmarks. In a case study, we achieve up to 19% WCET reduction (average: 13%) for a set of EEMBC benchmarks compared to a baseline cache partitioning setup.

I. Introduction

Multicore processors are increasingly used in intelligent embedded real-time systems—such as unmanned aerial vehicles (UAVs) and autonomous cars—that require high performance and efficiency to execute compute intensive tasks (e.g., vision based sense-and-avoid) in real-time.

Consolidating multiple tasks, potentially with different criticality (a.k.a. mixed-criticality systems [36], [7]), on a single multicore processor is, however, extremely challenging because interference in the shared hardware resources can significantly alter the tasks' timing characteristics. One of the major sources of interference is shared last-level cache (LLC). Tasks sharing a LLC, if uncontrolled, can evict each other's valuable cache-lines, thereby affect their execution times. Such co-runner dependent execution time variations are highly undesirable for real-time systems.

Cache-partitioning, which partitions the cache space among the cores and tasks, is a well-known solution which has been studied extensively in the real-time systems community [27], [37], [24], [32], [8]. Once a cache space is partitioned (spatial isolation), most literature assumes 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 [25], which are commonly used in modern multicore processors to exploit memory-level parallelism (MLP).

In this paper, we first experimentally show that cache partitioning does not guarantee cache access timing isolation on COTS multicore platforms. We use a set of carefully chosen synthetic and macro benchmarks (EEMBC [1], SD-VBS [35]) and evaluate their worst-case execution times (WCETs) on cache-partitioned COTS multicore systems (four CPU architectures). We observe significant WCET increases—up to 21X—even though the evaluated tasks run on a dedicated core, accessing a dedicated cache partition, and almost all of the memory accesses are cache hits. We attribute this to contention in special hardware registers in non-blocking caches, known as Miss Status Holding Registers (MSHRs), which support parallel outstanding cache-misses.

We validate the problem of MSHR contention using a cycle accurate full system simulator and investigate isolation and throughput impacts of different MSHR configurations in private and shared caches. We find that an insufficient number of MSHRs in the shared LLC can be detrimental to isolation 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.

Based on 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 that enables dynamic control of per-core MLP by the OS. 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. In a case study, we achieve up to 19% WCET reduction (average: 13%) for a set of EEMBC benchmarks compared to the baseline cache partitioning setup.

Contributions: Our contributions are as follows.

- We show that cache partitioning does not guarantee cache access timing isolation in non-blocking caches and identify MSHR contention as the root cause of the phenomenon.

- We provide extensive empirical evaluation results, collected on four COTS multicore architectures, showing the MSHR contention problem. We also provide the source code of the used synthetic benchmarks, necessary kernel patches, and testing scripts for replication study [1].
- We propose a hardware and system software (OS) collaborative approach that efficiently addresses the MSHR contention problem at a low hardware cost. To the best of our knowledge, this is the *first* paper that proposes a MSHR partitioning method to improve cache access timing isolation.
- We implement the proposed hardware and OS mechanisms in a cycle-accurate full system simulator and Linux kernel and present empirical evaluation results with a set of synthetic and macro benchmarks.

The rest of the paper is organized as follows. Section II describes necessary background. Section III demonstrates the problem of MSHR contention using real COTS multicore platforms. Section IV further validates the MSHR contention problem and investigates isolation and throughput impacts of MSHRs in private and shared non-blocking caches. Section V presents our hardware and OS collaborative technique to eliminate MSHR contention. Section VI presents evaluation results of the proposed technique. We discuss related work in Section VII and conclude in Section VIII.

II. Background

In this section, we provide necessary background on non-blocking caches and the page-coloring technique.

A. Non-blocking caches and MSHRs

A typical modern COTS multicore architecture is composed of multiple independent processing cores, multiple layers of private and shared caches, and a shared memory controller(s) and DRAM memories. To support high performance, recent embedded processors are adopting out-of-order designs in which each core can generate multiple outstanding memory requests [28], [11]. 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 LLC and memory. Therefore, each shared-memory subsystem must be able to handle multiple parallel memory requests. The degree of parallelism supported by the shared memory subsystem is called *Memory-Level Parallelism (MLP)* [12].

At the cache-level, non-blocking caches are used to support MLP. 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) [25], which tracks the status of the ongoing request. The MSHR 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 [16]. 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 known at the time of the access [33]. 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 III.

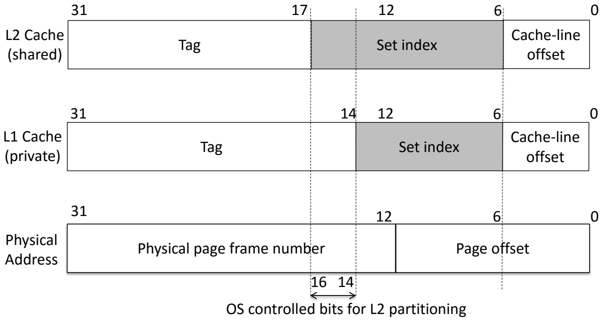


Fig. 1. Physical address and cache mapping of Cortex-A15.

B. Page Coloring

In this paper, we use a page-coloring based technique [39] to partition shared caches. 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 1 shows the address mapping of a Cortex-A15 platform, which we use in Section III. 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 1, only bit 14, 15, and 16 should be used to partition only the shared L2 cache.

III. 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. [2]

[1] https://github.com/CSL-KU/IsolBench

[2] Section III is based on our preliminary workshop paper [41] but extends it by using new hardware platforms (Exynos 5422 for Cortex-A7 and A15; Exynos 4412 for Cortex-A9) and by studying macro benchmarks from EEMBC [1] and SD-VBS [35] benchmark suites.

TABLE I
EVALUATED COTS MULTICORE PLATFORMS.

	Cortex-A7	Cortex-A9	Cortex-A15	Nehalem
Core	4 cores 1.4GHz in-order	4 cores 1.7GHz out-of-order	4 cores 2.0GHz out-of-order	4 cores 2.8GHz out-of-order
LLC	512KB,8way	1MB,8way	2MB,16way	8MB,16way
DRAM	2GB,16bank	2GB,16bank	2GB,16bank	4GB,16bank

TABLE II
LOCAL AND GLOBAL MLP

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

A. COTS Multicore Platforms

We use three COTS multicore platforms: Intel Xeon W3553 (Nehalem) based desktop machine and Odroid-XU4/U3 single-board computers (SBC). 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 [13] configuration. Thus, we use the Odroid-XU4 platform for both Cortex-A15 and Cortex-A7 experiments. The Odroid-U3 equips a Samsung Exynos 4412 processor which includes four Cortex-A9 cores. Table I shows the basic characteristics the four CPU architectures we used in our experiments. We run Linux 3.6.0 on the Intel Xeon platform, Linux 3.10.82 on the Odroid-XU4 platform, and Linux 3.8.13 on the Odroid-U3 platform; all kernels are patched with PALLOC [39] to partition the shared LLC at runtime.

B. Memory-level Parallelism

We first identify memory-level parallelism (MLP) of the four multicore architectures using an experimental method described in [10]. More detailed explanation of the methodology and the experimental results obtained in our tested platforms can be found in Appendix A.

Table II shows the identified MLP of each platform. In the table, how many outstanding misses one core can generate at a time is referred as *local MLP*, while the parallelism of the entire shared memory hierarchy (i.e., shared LLC and DRAM) is referred as *global MLP*. First, note that all architectures, including in-order based Cortex-A7, support significant parallelism in the shared memory hierarchy (global MLP)[3]. The results show that non-blocking caches are used in COTS multicore processors. In case of the Cortex-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. Note that the aggregated parallelism of the cores (the sum of local MLP) exceeds the parallelism supported by the shared LLC and DRAM (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.

TABLE III
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)

C. 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.

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 dependency, 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.

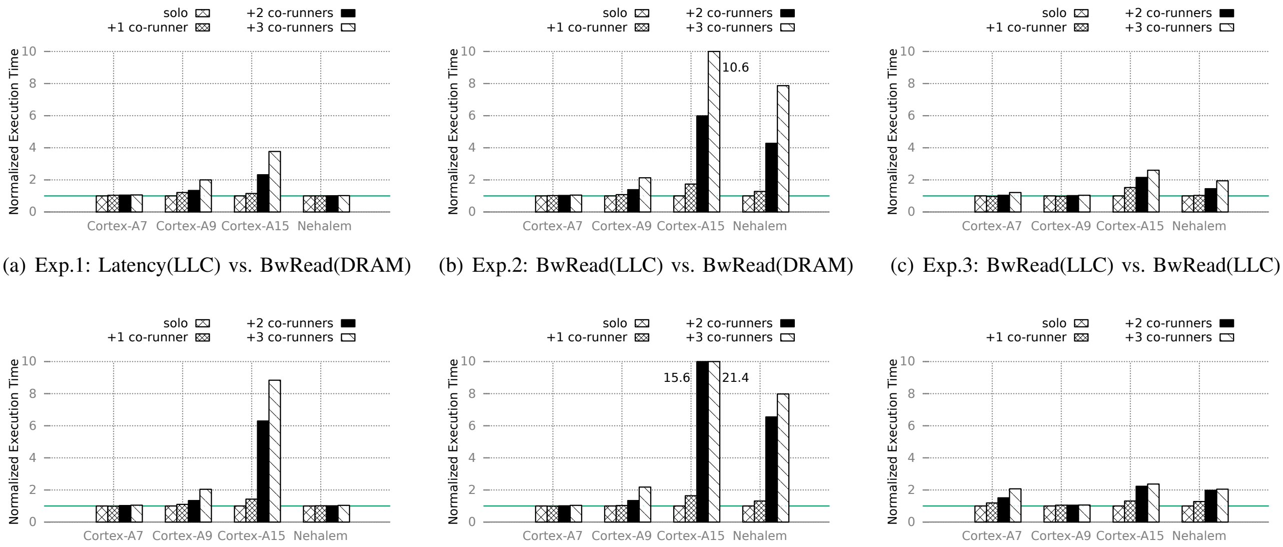
Table III shows the workload combinations we used. Note that the texts with parentheses—(LLC) and (DRAM)—indicate working-set sizes of the respective benchmark. In case of (LLC), the working size is configured to be smaller than 1/4 of the shared LLC size, but bigger than the size of the last core-private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

2) Exp. 1: Latency(LLC) vs. BwRead(DRAM): In the first experiment, we use the Latency benchmark as a subject and

[3] The global MLP of our Nehalem platform is determined by DRAM, while it is determined by LLC in the other platforms. See Appendix A for details.

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



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

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

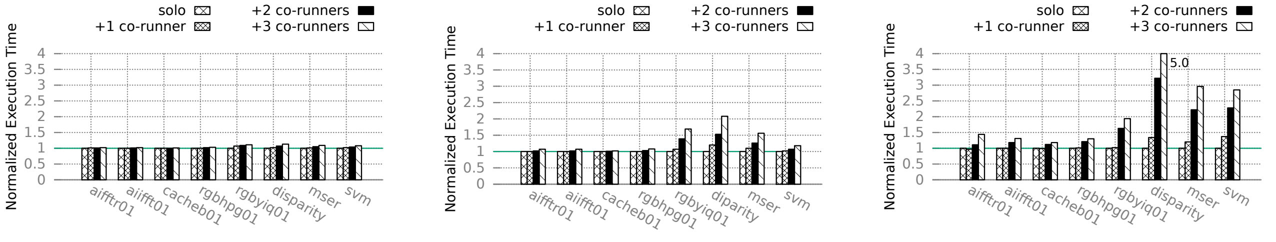
(c) Exp.3: BwRead(LLC) vs. BwRead(LLC)

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

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

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

Fig. 2. 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.



(a) Cortex-A7

(b) Cortex-A9

(c) Cortex-A15

Fig. 3. MSHR contention effects on WCETs of EEMBC and SD-VBS benchmarks.

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., ARM Cortex-A9, A15 and Intel Nehalem). Figure 2(a) shows the results. For Cortex-A7 and Intel Nehalem, Cache-partitioning is shown to be effective in providing timing isolation. For Cortex-A15 and A9, however, the response time is still increased by up to 3.7X and 2.0X, 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) 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 2(b) shows the results. While cache partitioning is still effective for Cortex-A7, the same is not true for the other platforms: Cortex-A9, A15, and Nehalem now suffer up to 2.1X, 10.6X, 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 [39]. The next experiment, however, shows it is not the case.

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 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 2(c), however, does not support this hypothesis. On the contrary, the observed slowdowns in all out-of-order cores are much less, compared to the previous experiment in which co-runners' memory accesses are cache misses and therefore use less cache bandwidth.

TABLE IV
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 [35]. (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

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 2(d), 2(e), and 2(f) show the results. Compared to BwRead, using BwWrite generally results in even worse interference to the subject tasks.

MSHR contention: To understand this phenomenon, we first need to understand how non-blocking caches processes cache accesses from the cores. As described in Section II, MSHRs are used to allow multiple outstanding cache-misses. If all MSHRs are in use, however, the cores can no longer access the cache until a free MSHR becomes available. 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 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 traffics delay the processing of line-fills, which in turn exacerbate the shortage of MSHRs.

D. 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 [35] 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 IV shows the Miss-Per-Kilo-Instructions (MPKI) characteristics of the benchmarks on a Cortex-A15 setting (32KB L1-I/D, 512KB L2 cache partition [5]).

TABLE V
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 [15], open-adaptive page policy
DRAM module	LPDDR2@533MHz, 1 rank, 8banks

We measured their execution times first alone in isolation and then with multiple instances of the BwWrite(DRAM), which has shown to cause 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 3 shows the results.[6] As expected, Cortex-A7 shows good isolation while Cortex-A9 and A15 show significant execution time increases in many of the benchmarks, even though they all access their own private cache partitions, due to MSHR contention. In Cortex-A9, we observe up to 2.08X (108%) WCET increase for the *disparity* benchmark; in Cortex-A15, we observe up to 5.0X WCET increase for the same benchmark. 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.

IV. UNDERSTANDING ISOLATION AND THROUGHPUT IMPACTS OF CACHE MSHRS

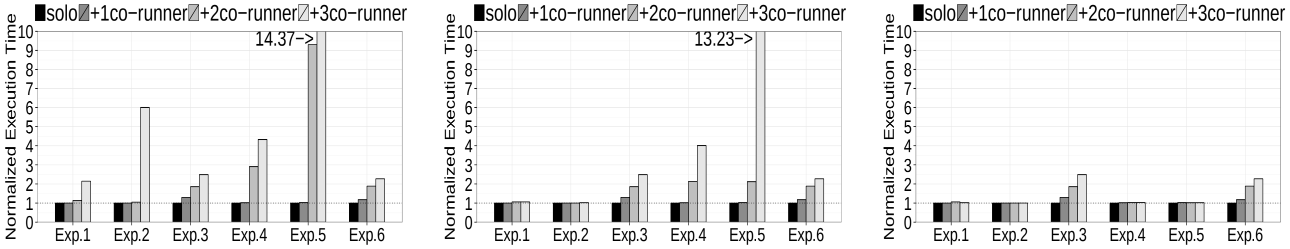
In this section, we study isolation and throughput impacts of MSHRs in non-blocking caches, by exploring different MSHR configurations using a cycle accurate full system simulator.

A. Isolation Impact of MSHRs in Shared LLC

In this experiment, we study how the number of MSHRs at the shared LLC affects to 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

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

[6] We exclude Nehalem because it has additional private L2 cache (256KB) that absorbs most of L1 cache misses; as a result, its shared LLC (L3) is rarely accessed when running the benchmarks and therefore we observe no significant WCET increases in Nehalem.

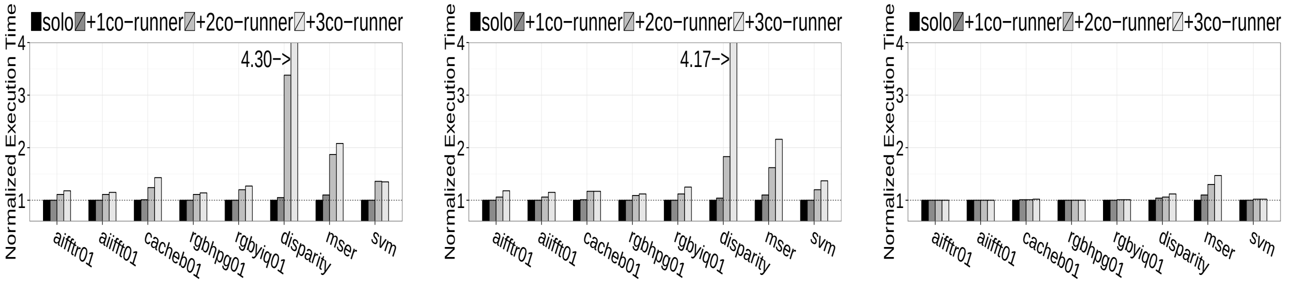


(a) MSHR(6/8)

(b) MSHR(6/12)

(c) MSHR(6/24)

Fig. 4. Effects of MSHR configurations on WCETs of IsolBench

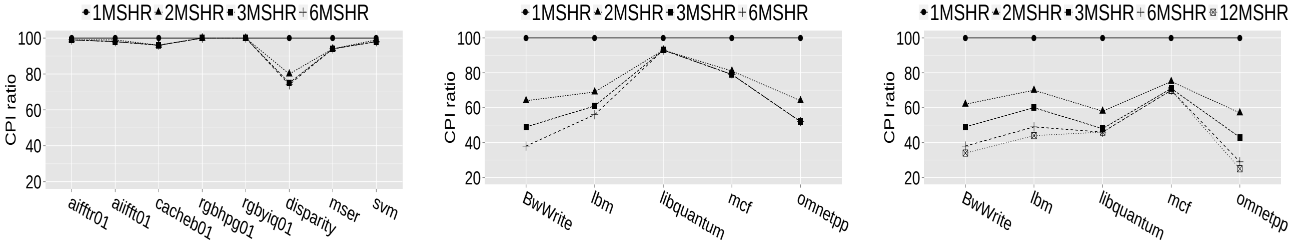


(a) MSHR(6/8)

(b) MSHR(6/12)

(c) MSHR(6/24)

Fig. 5. Effects of MSHR configurations on WCETs of EEMBC and SD-VBS benchmarks



(a) EEMBC, SD-VBS

(b) SPEC2006 & BwWrite

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

Fig. 6. Performance impact of MSHRs in private L1 cache.

shown in Table V [7]. On the simulator, we run a full Linux 3.14 kernel, patched with PALLOC [39] 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 III. 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 4 shows the results of the six IsolBench workloads (Table III). 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

[7] The CPU parameters are largely based on gem5's default ARM configuration, which is, according to [14], 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 [30], 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.

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 5 shows the results of EEMBC and SD-VBS benchmarks. 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 suffer considerable delays. This can be explained as a result of their relatively high DRAM access rates (see L2 MPKI values at Table IV). 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 [23], [40], [21], [22].)

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.

B. 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 [33]. 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 of a blocking cache. For each L1 MSHR configuration, we measure each benchmark's Cycles-Per-Instructions (CPI).

Figure 6(a) shows the results of EEMBC and SD-VBS benchmarks, normalized to 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 improvement 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 6(b) and 6(c) show the results of SPEC2006 and BwWrite benchmarks. The two figures differ in that in Figure 6(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 the next section.

V. 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.

A. 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 III-B. This means that the shared LLC can

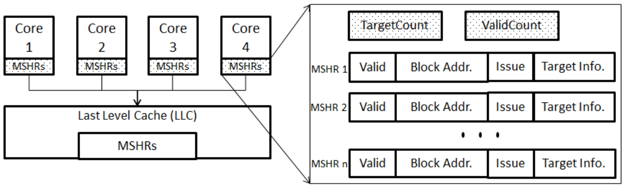


Fig. 7. Proposed MSHR Architecture

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 schedule 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."

B. 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 7 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 [17]). 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.

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

$$1 \leq TargetCount_i \leq N_{mshr}^{L1} \quad (2)$$

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 as 3 (half of the physically allowed number 6) to eliminate MSHR contention.

```
void prepare_task_switch(prev, next)
{
    // myid = local cpu index
    myid = smp_processor_id();
    if (next->mshr_reserve > 0) {
        // enable/update MSHR partitioning
        $R$ = next->mshr_reserve;
        mshr_part[myid] = $R$;
        $TargetCount_{myid} = R$;

        m_rt = 0;
        mshr_remain = $N_{mshr}^{LLC}$;
        for (i = 0...$m-1$) {
            if (mshr_part[i] > 0) {
                m_rt ++;
                mshr_remain -= mshr_part[i];
            }
        }
        $R_{nrt}$ = mshr_remain / ($m$ - m_rt);
        for (i = 0...$m-1$) {
            if (mshr_part[i] == 0) {
                $TargetCount_i = R_{nrt}$;
            }
        }
    } else if (prev->mshr_reserve > 0) {
        mshr_part[myid] = 0;
        for (i = 0...$m-1$) {
            if (mshr_part[i] > 0)
                return;
        }
        // disable MSHR partitioning
        for (i = 0...$m-1$) {
            $TargetCount_i = N_{mshr}^{L1}$;
        }
    }
}
```

Fig. 8. MSHR reservation algorithm in the CPU scheduler.

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 IV-B, assigning $TargetCount = 1$ to a core that executes the *lbm* SPEC2006 benchmark would cause more than 40% performance reduction.

C. 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 of the shared LLC on a per-task basis. 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 to N_{mshr}^{LLC}/m to guarantee reservation. 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. MSHR reservation of each real-time task is enforced globally by the OS scheduler by updating the $TargetCount$ registers of all cores to satisfy the Eqs. 1 and 2, which effectively partition LLC MSHRs among the cores.

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. We instrument the OS scheduler to start and stop MSHR reservation, if needed, at the time of a task switching.

Figure 8 shows the algorithm. The algorithm works on each context switching—from $prev$ task to $next$ task—on any core in the system. On a context switch, if the next scheduled task requires MSHR reservation of (Line 5-25), it configures the $TargetCount$ register of the corresponding core (Line 9). Note that R denotes the number of reserved MSHRs. It then determines the number of available MSHRs (excluding reserved MSHRs), which is then evenly distributed to the cores that execute best-effort tasks (Line 20-25). On the other hand, if no currently running tasks wish to reserve MSHRs, the scheduler resets the $TargetCount$ registers of all cores to the maximum (Line 33-35).

VI. Evaluation

In this section, we evaluate isolation and throughput impacts of the proposed approach though a case study.

A. Setup

We use the same experiment setup as explained in section IV—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 (`prepare_task_switch()` at `kernel/sched/core.c`) to communicate with the simulator to adjust the number of MSHRs.

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.

B. Case Study: A Mixed Criticality System

In this experiment, we model a mixed-criticality task system using four instances of EEMBC benchmarks—*aifftr01, aiifft01, cacheb01 and, rgbhpg01* [8]—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.

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 a 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).

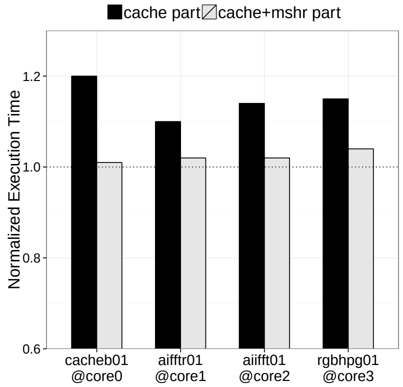


Fig. 9. WCETs of real-time tasks (EEMBC), co-scheduled with best-effort tasks.

Figure 9 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.

VII. 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 [31] 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 [11],

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

[18]. 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 [24], [27], [37], [38]. As discussed in II-B, 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 [26], [32], [39] and TLB [29]. 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 [27] in combination with page coloring. In MC^2 project [8], 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 [9], [19], [20] 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 they are similar to our work in the sense they also control the number of MSHRs, they do so dynamically via complex hardware implementations (no OS involement) and do not guarantee the absence of MSHR contention. In contrast, we provide a simple hardware mechanism that enables software (OS) based control of MSHRs to guarantee the absence of MSHR contention.

VIII. 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 [34] 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)*. IEEE, 2013.

[7] A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[8] 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.

[9] 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.

[10] 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.

[11] Freescale. *e500mc Core Reference Manual*, 2012.

[12] A. Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea*, 1998.

[13] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[14] 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.

[15] 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.

[16] Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[17] Intel. *Intel®64 and IA-32 Architectures Software Developer Manuals*, 2012.

[18] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, April 2015.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA)*, pages 81–87. IEEE Computer Society Press, 1981.

[26] 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.

[27] 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.

[28] NVIDIA. *NVIDIA Tegra K1 Mobile Processor, Technical Reference Manual Rev-01p*, 2014.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] H. Yun and P. Valsan. Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2015.

Appendix

A. Memory-level Parallelism (MLP) Identification

We use a pointer-chasing micro-benchmark shown in Figure 10 to identify memory-level parallelism. The benchmark 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*.

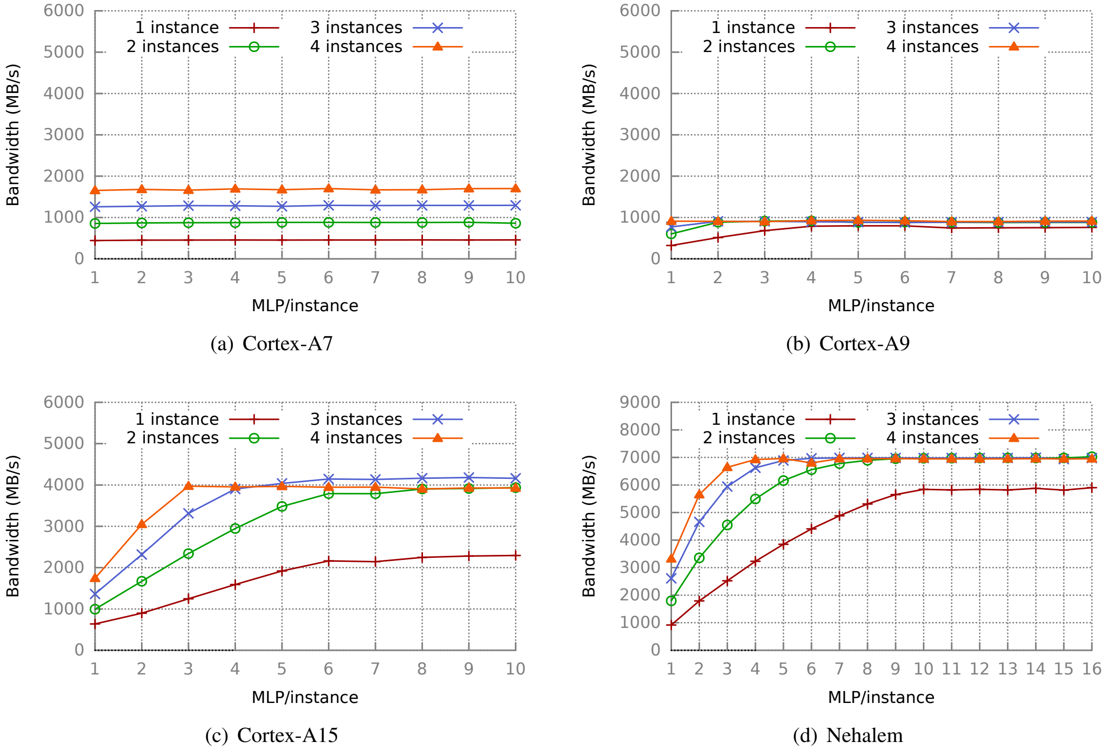
```
static int* list[MAX_MLP];
static int next[MAX_MLP];

long run(long iter, int mlp)
{
    long cnt = 0;
    for (long i = 0; i < iter; i++) {
        switch (mlp) {
        case MAX_MLP:
            .
            .
        case 2:
            next[1] = list[1][next[1]];
            /* fall-through */
        case 1:
            next[0] = list[0][next[0]];
        }
        cnt += mlp;
    }
    return cnt;
}
```

Fig. 10. MLP micro-benchmark. Adopted from [10].

Figure 11 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 MLP. As we increase the number of benchmark instances, the point of saturation becomes shorter in the out-of-order cores. When four instances are used in Cortex-A15, the aggregate bandwidth saturates at three lists. This suggests that the global MLP of Cortex-A15 is close to 12; according to [3], the LLC can support up to 11 outstanding cache-misses (global MLP of 11). Note that the global MLP can be limited by either of the two factors: the size of MSHRs in the shared LLC or the number of DRAM banks [9]. In the case of Cortex-A15, the limit is likely determined by the number of MSHRs of the LLC (11), because the number of banks is bigger than that (16 banks). In case of Nehalem, on the other hand, the performance saturates when the global MLP is about 16, which is likely determined by the number of banks, rather than the number of LLC MSHRs; according to [16], the Nehalem architecture supports up to 32 outstanding cache-misses. In other words, the MLP of its shared LLC is 32, while the MLP of the DRAM is 16. Lastly, in case of Cortex-A9, both local and global MLP appear to be 4. Cortex-A9 was released much earlier (2007)

[9] The number of DRAM banks determines DRAM-level parallelism, as banks can be accessed in parallel.



(a) Cortex-A7

(b) Cortex-A9

(c) Cortex-A15

(d) Nehalem

Fig. 11. Aggregate memory bandwidth as a function of MLP (the number of lists) per benchmark.

than Cortex-A7 (2011) and its cache-line size is also smaller (32B/line) than the others (64B/line). We suspect these are the reasons of its relatively low memory performance.

In summary, caches are non-blocking in modern multicore processors. In in-order processors, while each individual core may block at each cache-miss at its private L1 cache, the shared LLC still allows non-blocking accesses to improve performance. In out-of-order processors, both private and shared caches support significant amount of parallelism to minimize blocking of the cores.