Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms

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

***Abstract*—Tasks running on a Commercial Off-The-Shelf (COTS) multicore processor can suffer significant execution time variations due to inter-core interference in accessing shared hardware resources such as shared last-level cache (LLC). Page-coloring is a well-known OS technique, which can partition the LLC space among the cores, to improve isolation.**

In this paper, we evaluate the effectiveness of page-coloring based cache partitioning on three COTS multicore platforms. On each platform, we use two carefully designed micro-benchmarks and perform a set of experiments, which generate very high interference at the shared LLC, with and without cache partitioning.

We made two interesting findings: (1) Without cache-partitioning, a task can suffer up to 103X slowdown due to interference at the shared LLC. (2) More surprisingly, we found that cache partitioning does not necessarily eliminate interference in accessing the LLC, even when the concerned task only accesses its dedicated cache partition (i.e., all memory accesses are cache hits); we observe up to 14X slowdown in such a configuration. We attribute this to contention in the Miss Status Holding Registers (MSHRs) of the LLC.

I. Introduction

Commercial Off-The-Shelf (COTS) multicore processors are increasingly being adopted in autonomous cars, unmanned aerial vehicles (UAV), and other critical cyber-physical systems (CPS). While these COTS multicore processors offer numerous benefits, they do not provide predictable timing—a highly desired property in many CPS applications.

In a COTS multicore system, the execution time of a task is determined not only by the task and the underlying hardware architecture, but also by co-runners on different cores due to interference in the shared hardware resources. One of the major source of interference is shared last-level cache (LLC). When more than two tasks execute in parallel on cores that share the LLC, tasks can evict each other's valuable cache-lines, which cause negative performance impacts. Cache-partitioning, which partitions the cache space among the cores, is a well-known solution to counter this problem [11], [15].

In this paper, we evaluate the effectiveness of cache partitioning in improving timing predictability on three modern COTS multicore platforms: one in-order (ARM Cortex-A7) and two out-of-order (ARM Cortex-A15 and Intel Nehalem) architecture based quad-core platforms. We use two carefully designed micro-benchmarks and perform a set of experiments to investigate the impacts of shared LLC to the application execution times—*with and without applying cache-partitioning*. In designing the experiments, we consider memory-level-parallelism (MLP) of modern COTS multicore architecture—non-blocking caches and DRAM bank parallelism—and intend to find worst-case scenarios where a task's execution time suffers the most slowdown due to cache interference.

From the experiments, we made several interesting findings. First, unlimited cache sharing can cause unacceptably high interference; we observe up to 103X slowdown (i.e., the task's execution time is increased by 103 times due to co-runners on different cores). Second, cache-partitioning is effective especially in the in-order architecture, as it almost completely eliminates cache-level interference. In out-of-order architectures, however, we observe significant interference even after cache partitioning is applied. Concretely, we observe up to 14X slowdown even when the task under consideration only accesses its dedicated cache partition (i.e., all memory accesses are cache hits). We attribute this to contention in the shared miss-status holding registers (MSHRs) [8] in the LLC (See Section V).

Our contributions are as follows: (1) experiment designs that help expose the degree of interference in the shared LLC; (2) detailed evaluation results on three COTS multicore platforms showing the performance impacts of the cache-level interference. To the best of our knowledge, this is the *first* paper that reports the worst-case performance impact of MSHR contention on COTS multicore platforms.

The rest of the paper is organized as follows. Section II describe necessary background on modern COTS multicore architecture. Section III describe the three COTS multicore platforms we used in this paper. Section IV experimentally analyze MLP of the hardware platforms. Section V investigate the impacts of cache (LLC) interference on the tested platforms. We conclude in Section VI.

II. Background

In this section, we provide necessary background on COTS multicore architecture and software based resource partitioning techniques.

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, processing cores in many embedded/mobile processors are adopting out-of-order designs in which each core can generate multiple outstanding memory requests [12], [4]. Even if the cores are based on in-order designs, in which one core can only generate one outstanding memory request at a time, they collectively can generate multiple requests to the shared memory subsystem. Therefore, the memory subsystem must be

TABLE I: Evaluated COTS multicore platforms.

	Cortex-A7	Cortex-A15	Nehalem
Core	4cores@0.6GHz in-order	4cores@1.6GHz out-of-order	4cores@2.8GHz out-of-order
LLC	512KB, 8way	2MB, 16way	8MB, 16way
DRAM	2GB, 16banks	2GB, 16banks	4GB, 16banks

able to handle multiple parallel memory requests. The degree of parallelism supported by the shared memory subsystem—the caches and main memory—is called *Memory-Level Parallelism (MLP)* [5].

A. Non-blocking caches and MSHRs

At the cache-level, non-blocking caches are used to handle multiple simultaneous memory accesses. On a cache-miss, the cache controller allocates a MSHR (miss status holding register) to track the status of the ongoing request and the entry is cleared when the corresponding memory request is serviced from the lower-level memory hierarchy. For the last-level cache (LLC), each cache-miss request is sent to the main memory (DRAM). As such, the number of MSHRs in the LLC effectively determines the maximum number of outstanding memory requests directed to the DRAM controller. It is important to note that MSHRs are typically shared among the cores [7] and when there are no remaining MSHRs, further accesses to the cache—*both* hits and misses—are prevented until free MSHRs become available [1]. Because of this, even if the cache space is partitioned among cores using software cache partitioning mechanisms, in which each core is guaranteed to have its dedicated cache space, accessing the cache partition does not necessarily guarantee interference freedom as we will demonstrate in Section V.

B. DRAM and memory controllers

At the DRAM-level, a DRAM chip is divided into multiple *banks*, which can be accessed in parallel. As such, the number of banks determines the parallelism available on DRAM. To maximize the bank-level parallelism, DRAM controllers typically use an *interleaved mapping*, which maps consecutive physical addresses into different DRAM banks.

C. Cache and DRAM bank Partitioning

Cache partitioning has been studied extensively to provide better isolation and efficiency. Page coloring is a well-known software technique which partitions cache-sets among the cores [11], [15], [9], [16]. Also, there are a variety of hardware based partitioning mechanisms such as cache-way based partitioning [13], which is supported in some commercial processors [4]. More recently, several DRAM bank partitioning methods, mostly based on page-coloring, have been proposed to limit bank-level interference [17], [10], [14].

III. Evaluation Setup

In this paper, we use two COTS multicore platforms: an Intel Xeon W3553 (Nehalem) based desktop machine and an Odroid-XU+E single-board computer (SBC). The Odroid-XU+E board equips a Samsung Exynos 5410 processor which includes both four Cortex-A15 and four Cortex-A7 cores in a big-LITTLE [6] configuration. Thus, we use the Odroid-XU+E platform for both Cortex-A15 and Cortex-A7 experiments. Table I shows the basic characteristics the three platform configurations we used in our experiments. We run Linux 3.6.0 on the Intel Xeon platform and Linux 3.4.98 on the Odroid-XU+E platform; both kernels were patched with PALLOC [17] to be able to partition the shared LLC at runtime. When cache-partitioning is applied, the shared LLC is evenly partitioned among the four cores (i.e., each core gets 1/4 of the LLC space).

```
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 }
```

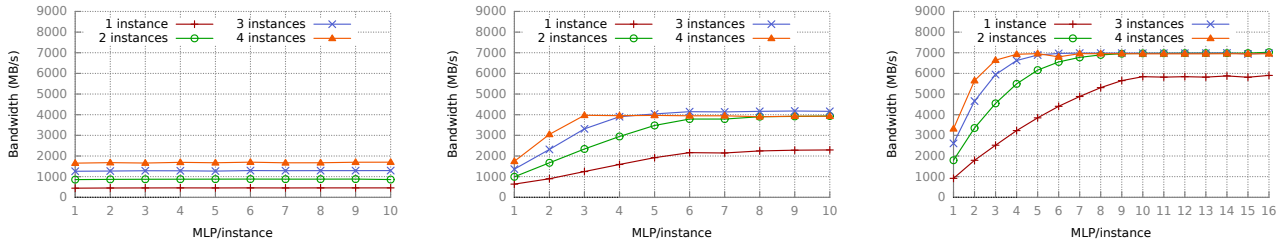
Fig. 1: MLP micro-benchmark. Adopted from [3].

IV. Understanding Memory-Level Parallelism

In this section, we identify memory-level parallelism (MLP) of the three multicore platforms using an experimental method described in [3].

In the following, we first briefly describe the method for better understanding. The method uses a pointer-chasing micro-benchmark shown in Figure 1. 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 (determined by *mlp* parameter in Figure 1) 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 varied the number of benchmark instances from one to four and measure the aggregate performance to investigate the parallelism of the entire shared memory hierarchy, which we call *global MLP*.

Figure 2 shows the results. Let us first focus on 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. On the other



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 2: Aggregate memory bandwidth as a function of MLP/benchmark.

TABLE II: Local and global MLP

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

hand, for Cortex-A15, the performance improves up to six lists and then saturates. This suggests that the Cortex-A15's local MLP is six. In case of Nehalem, performance improves up to ten concurrent lists, suggesting its local MLP is ten. As we increase the number of benchmark instances, the point of saturation become shorter in both Cortex-A15 and Nehalem. When four instances are used in Cortex-A15, the aggregate performance saturates at three. This suggests that the global MLP of Cortex-A15 is close to 12; according to [2], 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. 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). In the 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 MSHRs; according to [7], the Nehalem architecture supports up to 32 outstanding cache-misses. Table II shows the identified local and global MLP of the the three platforms we tested.

V. Understanding Cache Interference

In this section, we investigate performance impacts of cache-level interference on COTS multicore platforms.

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, which also can cause interference. As we will see later in this section, cache partitioning does not necessary provide isolation even when the application's working-set fits entirely in a dedicated cache partition, due to contention in the shared MSHRs.

To find out worst-case interference, we use various combinations of two micro-benchmarks: *Latency* and *Bandwidth* [18]. 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: 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)

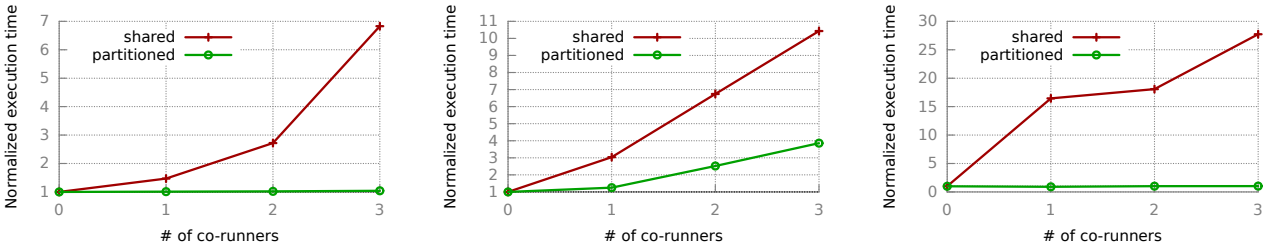
Table III shows the workload combinations we used. Note that the texts with parentheses—(LLC) and (DRAM)—indicate working-set sizes of the respective benchmark. In case of (LLC), the working size is configured to be smaller than 1/4 of the shared LLC size, but bigger than the size of the last core-private cache. [1] As such, in case of (LLC), all memory accesses are LLC hits in both cache partitioned and non-partitioned cases. 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. We repeat the experiment on the three test platforms with and without cache partitioning.

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

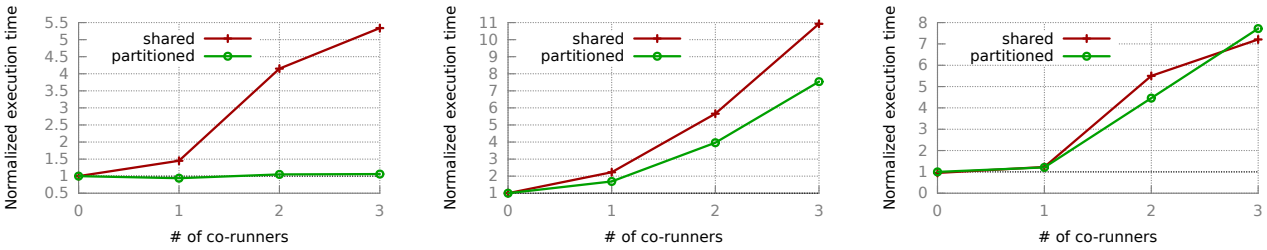
In the first experiment, we use the Latency benchmark as a subject and the BwRead benchmark as co-runners. Recall that BwRead has no data dependency and therefore can generate multiple outstanding memory requests on an out-of-order processing core (i.e., ARM Cortex-A15 and Intel Nehalem core). Figure 3 shows the results. When cache-partitioning is not applied, *shared*, the response times of the Latency benchmark are increased dramatically in all three platforms—up to 6.7X in Cortex-A7, 10.4X in Cortex-A15, and 27.7X in Nehalem. This is because cache-lines of the Latency benchmark are

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



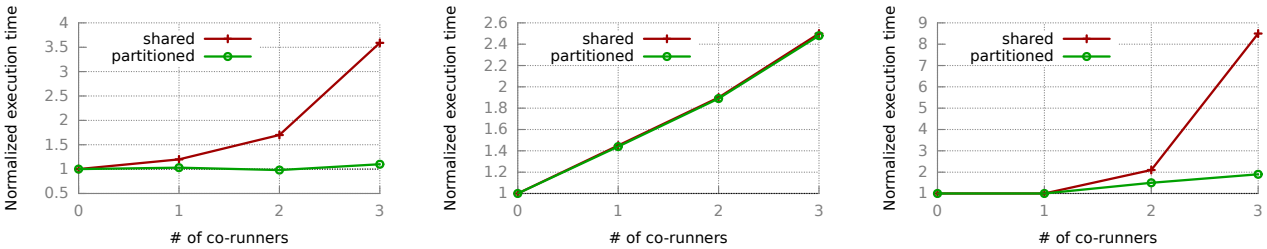
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 3: [Exp.1] Slowdown of Latency(LLC) with BwRead(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 4: [Exp.2] Slowdown of BwRead(LLC) with BwRead(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 5: [Exp.3] Slowdown of BwRead(LLC) with BwRead(LLC) co-runners.

evicted by the co-running BwRead benchmark instances. If not the co-runners, those cache-lines would never have been evicted. On the other hand, applying cache-partitioning is shown to be effective in preventing such cache-line evictions hence providing performance isolation, especially in Cortex-A7 and Intel Nehalem platforms. In the Cortex-A15 platform, however, the response time is still increased by up to 3.9X even after partitioning the cache. 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.

B. 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 4 shows the results. Note that while the behavior of Cortex-A7 is similar to the previous experiment, the behaviors of Cortex-A15 and Nehalem are considerably different. In the Nehalem platform, in particular, *the performance isolation benefit of cache partitioning is completely eliminated* as the subject benchmark suffers from the similar degree of slowdowns regardless of cache-partitioning. In other words, the results suggest that cache-partitioning does not necessary provide expected performance isolation benefits in out-of-order architectures. We initially suspected the cause of this phenomenon is likely the bandwidth competition at the shared cache, similar to the DRAM bandwidth contention [17]. The following experiment, however, shows it is not the case.

C. 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 reduced 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 5, however, does not support this hypothesis. On the contrary, the observed slowdowns in both Cortex-A15 and Nehalem are much less, compared to the previous experiment in which co-runners' memory accesses are cache misses and therefore use less cache bandwidth.

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 stall additional memory requests even when they are cache hits.

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

In the next experiments, we further validate the problem of MSHR contention by using the BwWrite benchmark as co-runners. BwWrite updates a large array and therefore generates a line-fill (read) and a write-back (write) for each memory access. The additional write-back requests add more pressure in DRAM and therefore delay the processing of line-fill requests, which in turn further exacerbate the shortage of MSHRs. Figure 6, Figure 7, and Figure 8 show results. As expected, the subject tasks generally suffer even more slowdowns due to the additional write-back memory traffic.

E. Summary

Figure 9 show the maximum observed slowdowns in all experiments. When the LLC is partitioned, we observed up to 14.2X slowdown on Cortex-A15, 7.9X slowdown on Nehalem, and 2.1X slowdown on Cortex-A7. When the LLC is not partitioned, we observed up to 26.3X slowdown on Cortex-A15, 103.7X slowdown on Nehalem, and 6.8X slowdown on Cortex-A7.

In summary, while cache space competition (i.e., cache-line evictions) is certainly an important source of interference, eliminating the space competition through cache-partitioning does not necessary provide ideal isolation in COTS multicore platforms due to the characteristics of non-blocking caches. Through a series of experiments, we demonstrated that the MSHR competition can also cause significant interference, especially in out-of-order cores.

VI. Conclusion

Many prior works focus on cache partitioning to ensure predictable cache performance. In this paper, we showed that cache partitioning does not necessarily provide predictable cache performance in modern COTS multicore platforms that use non-blocking caches to exploit memory-level-parallelism (MLP). We quantified the degree of MLP on three COTS multicore platforms and performed a set of experiments that are specially designed to expose worst-case interference in accessing the shared LLC among the cores.

The results showed that while cache-partitioning help reduce interference, it can still suffer significant interference—up to an order of magnitude slowdown—even when the task under consideration accesses its own dedicated cache partition (i.e., all cache-hits). This is because there are other important shared resources, particularly MSHRs, which need to be managed in order to provide better isolation on COTS multicore platforms. We plan to address the issue as our future work.

References

[1] Memory system in gem5. http://www.gem5.org/docs/html/gem5MemorySystem.html.

[2] ARM. *Cortex-A15 Technical Reference Manual, Rev: r2p0*, 2011.

[3] 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.

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

[5] A. Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea Session98*, 1998.

[6] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

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

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

[9] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[10] 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.

[11] 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.

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

[13] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 117–128. IEEE, 2002.

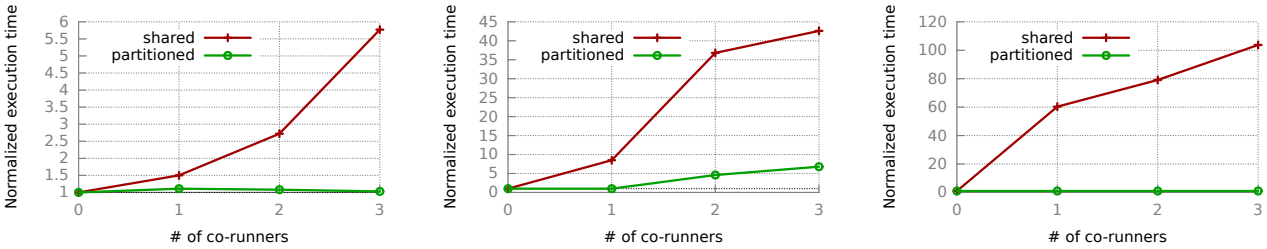
[14] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.

[15] 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.

[16] 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.

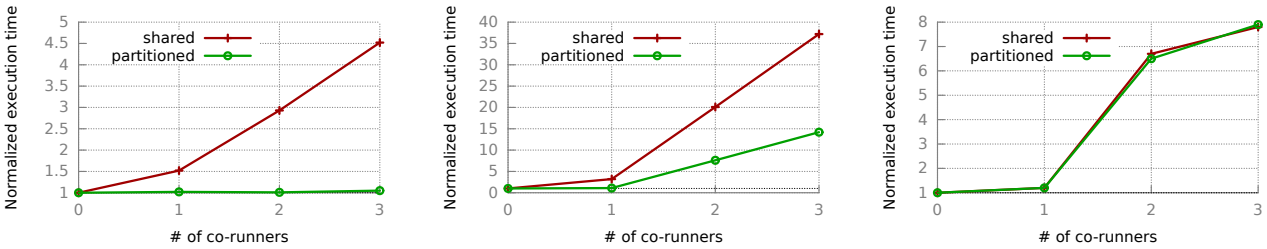
[17] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.



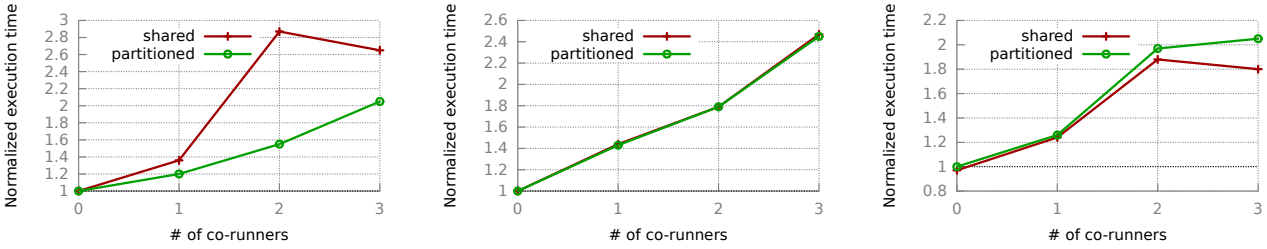
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 6: [Exp.4] Slowdown of Latency(LLC) with BwWrite(DRAM) co-runners.



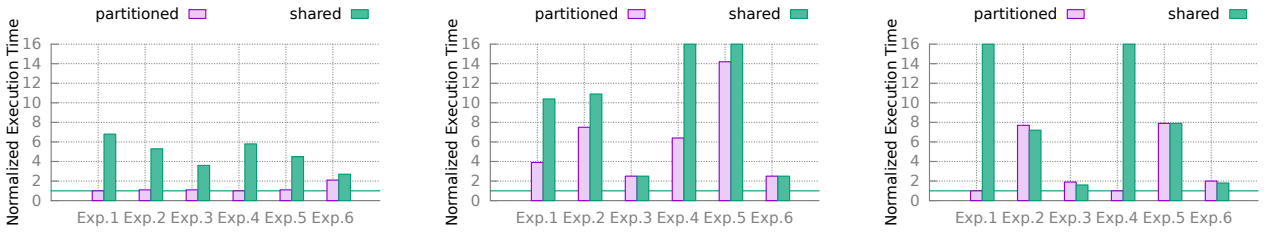
(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 7: [Exp.5] Slowdown of BwRead(LLC) with BwWrite(DRAM) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 8: [Exp.6] Slowdown of BwRead(LLC) with BwWrite(LLC) co-runners.



(a) Cortex-A7 (b) Cortex-A15 (c) Nehalem

Fig. 9: Maximum observed slowdowns in all experiments.