Vol.2019-ARC-240 No.38
Vol.2019-SLDM-191 No.38
Vol.2019-EMB-53 No.38
2019/2/28

Analysis of Performance Degradation on Shared Cache in Multicore Systems

YANG QIN[1,a)] SHINYA HONDA[1]
HIROAKI TAKADA[1] GANG ZENG[2]

Abstract: Modern microprocessors usually employ shard cache to keep high-speed communication with programs. Since many tasks in real-time embedded systems have implicit timing requirements, unpredictable execution time due to cache contention has become a major challenge. With carefully engineered concurrent applications, up to 168X execution time increase was observed on a real embedded multicore platform. Moreover, cache contention can cause different degrees of performance degradation concerning the characteristics of concurrent applications and processing processors. In this paper, we focus on analyzing the main factors contributing to varying performance degradations caused by the cache contention in multicore platforms. We evaluate eight SPEC 2017 benchmarks and two separate regulations doing read and write operations in memory. Evaluation results show that the application characteristics, CPU designs and implementation platforms cause significant effects on the variabilities in performance degradation.

Keywords: Shared cache, Performance Degradation, Application characteristics, CPU designs, Implementation platforms

1. Introduction

Multicore processors are widely used in real-time embedded systems to perform more tasks with better system performance. In modern multicore processors, shared cache architecture has become common since it brings more benefits than does dedicated cache, such as increasing cache utilization, offering faster inter-core communication, reducing cache coherency complexity and false sharing penalty. However, since many tasks in real-time embedded systems may have implicit performance requirements, it also becomes a major concern to achieve predictable performance in multicore systems because of the shared resources. For example, the shared L2 cache technology allows all the cores in the system to access the entire L2 freely, which in turn leads to resource contention and decreases the application performance.

Non-blocking caches are often used as shared caches in multicore processors to server concurrent memory requests. The problem is when any of the internal buffers become full, the cache becomes blocked and no further requests can be served any more until the buffers are available again. If a program induces shared cache blocking, it may cause significant performance impacts on the tasks executing on other cores, as none of the cores can access the cache when it is blocked. Therefore, it is to attribute the cache contention in special hardware buffers in non-blocking caches. In this work, we aim at exploring the performance impacts on non-blocking shared caches in multicore platforms.

Many research effort has been focused on improving memory performance in multicore platforms. For shared cache, cache partitioning technology which divides the cache space among applications based on their memory demands to provide capacity benefits with performance isolation, has been extensively studied [1]. However, Valsan et.al first experimentally showed that such partitioning solution does not necessarily guarantee performance isolation in modern processors that use non-blocking caches to exploit Memory-Level-Parallelism (MLP) [2]. In recent research, memory bandwidth as well as internal hardware structures of non-blocking cache, e.g., Miss-Status-Holding-Registers (MSHRs) and write buffers, have gained much attention to analyze and control the timing impacts on a shared cache. For example, Yun et.al suggested a memory bandwidth reservation system, which uses a hardware performance counter to regulate the maximum memory bandwidth, to support efficient memory performance isolation in multicore platforms [3]. Valsan et.al identified that MSHRs in non-blocking caches are a significant source of contention, and a hardware and OS collaborative approach was suggested to help eliminate the MSHR contention to improve the cache access [2]. Recently, Bechtel et.al identified cache MSHR and WriteBack (WB) buffer as two important attack vectors in denial-of-service (DoS), and an OS-level solution based on MemGuard was suggested to mitigate shared cache DoS attackers [4].

In this work, we experimentally investigate the performance effects of non-blocking caches in real multicore platforms. According to the evaluation results of concurrent experiments, it is found that read and write operations in memory cause different performance degradation degrees as they may stress different internal structures of cache. It is also observed that cache blocking occurs not only in out-of-order processors but also in in-order processors, and the former architecture causes server execution time increases without a doubt. While it is surprisingly observed that up to 168X increase of execution time occurs on an in-order architecture based platform. Besides, we investigate the performance degradation that the processes suffer due to memory bandwidth constraints. Experiments show that main memory bandwidth contention negatively impacts the process performance; in two embedded platforms with the same core type, performance degradation difference can grow up to a hundred times for some of the applications. In summary, three main factors that cause different performance degradations are analyzed in real-world embedded platforms. The observations are expected to be applied to the improvement of performance issues

1 Graduate School of Informatics, Nagoya University, Chikusa-ku, Nagoya 464-8603, Japan

2 Graduate School of Engineering, Nagoya University, Chikusa-ku, Nagoya 464-8603, Japan

in both the industry and academia in the future.

The remainder of the paper is organized as follows. In section 2, the basic background of a non-blocking cache is introduced. In section 3, we present the relevant cache contention model, motivational examples, and memory access code. In section 4, evaluation experiments of different concurrent applications on different embedded platforms are conducted. Finally, section 5 concludes the paper and addresses some possible future work.

2. Non-Blocking Cache

Non-blocking cache is one of the most effective techniques used for tolerating cache-miss latency in modern processors. It was first proposed by Kroft in [5], which allows execution to proceed concurrently subsequent cache accesses after a cache miss occurs. In this section, we present the fundamental background of a non-blocking cache, together with the flowcharts of read and write operations correspondingly.

2.1 Internal organization

The internal organization of a non-blocking cache is shown in Fig.1. In order to allow non-blocking operations and multiple misses, Miss-Status-Holding-Registers (MSHRs) are used to record the miss related information, i.e., the address and cache line of the data block, the word that caused the miss, and the function unit or register to which the data is to be routed [6]. Usually, a non-blocking cache is capable of serving multiple outstanding misses, and the degree to which this can occur depends on the size of MSHR structure. As long as the corresponding cache line is fetched from next-level memory hierarchy, the MSHR entry will be cleared; otherwise, it can continue serving further cache misses, until the MSHR becomes full.

Another key hardware structure on a non-blocking cache is write buffer. If the cache is write-back, a WriteBack (WB) buffer is needed. In the write miss case, the request is sent to the main memory and space is only allocated when this request is answered. However, we do not want to send the data to the main memory only for it to be returned. Therefore, the WriteBack buffer can be temporarily used to hold data being written from the next memory hierarchy level. The detailed information will be introduced in section 2.2.

It is noteworthy to mention that when the MSHR or the write buffer becomes full, i.e., all buffer entries are occupied, the cache is blocked. In other words, the cache can no longer accept any other memory requests until both of MSHR and WriteBack buffer are available.

2.2 Memory operation

Generally speaking, memory operation can be simply classified as read and write operations. When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. In this section, we introduce basic steps taken by a cache memory to resolve to a memory access.

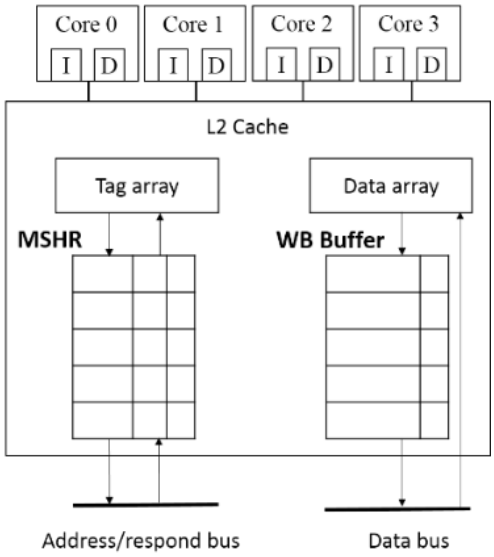


Figure 1: Structure of a non-blocking shared cache (adopted from Fig.1 in [4])

- **Read operation**

The flow chart of memory read operation is illustrated in Fig.2. When the CPU has a read request for data, the tag array of the cache will be accessed. If there is a cache read hit, the data will be returned from cache directly. Since all the read operations are finished in the cache, there is no need to access the main memory. On the contrary, if there is a cache miss, which implies that the data is not in the cache, the main memory has to be fetched to bring the data. In this case, the CPU will first try to locate a free cache block to use. Then the missed related information (introduced in section 2.1) will be recorded into the MSHR table. If there is no pending request for that cache block, a fill request is generated (read memory data to cache). When response of the fill request is received at the cache, the cache line is inserted into the cache and the corresponding MSHR entry is marked as filled. As presented above, as long as all the requests waiting at the filled MSHR entry have been responded to and serviced, the MSHR entry will be freed.

- **Write operation**

Assume that the cache uses write-back policy, in which the data is written only to the block in the current cache, not to lower-level caches. Dirty bit is commonly used to indicate whether the contents of a particular cache block are different to what is stored in the main memory. Any dirty cache lines are written back to the system using writeback buffer.

If the CPU has a write request and there is a cache write hit, i.e., the address we want to write to is already loaded in cache, the data will be written into cache directly. Since there is no update on the main memory, the used cache block will be marked as dirty. On the other hand, in case of a cache write miss, the missed information will be recorded into MSHR. Meanwhile, it locates a cache block to use. If the block is dirty, it will be updated and recorded into writeback buffer. Note that the modified cache block is written to the main memory only when it is replaced; otherwise, if the block is clean, it updates the block in main memory and brings the block to the cache. The flowchart of write operation is presented in Fig.3.

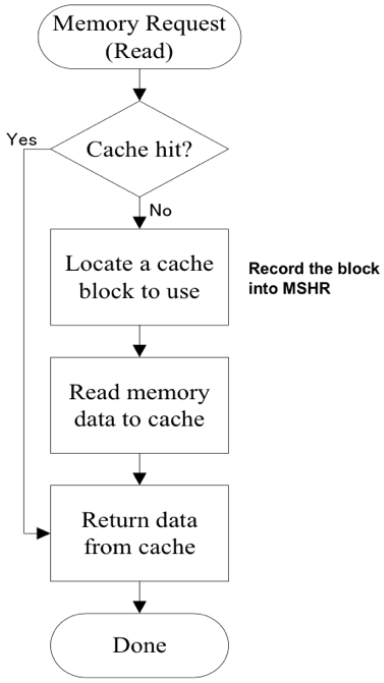


Figure 2: Flowchart of read operation

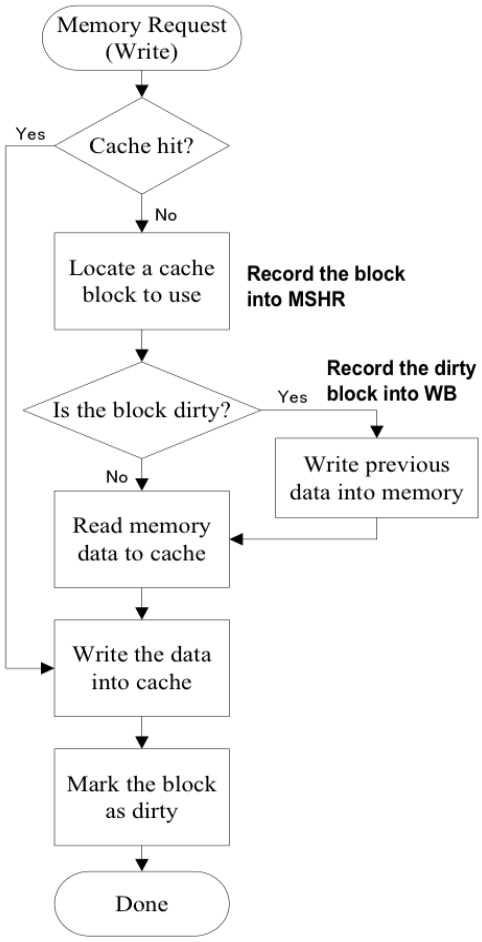


Figure 3: Flowchart of write operation

3. Cache Contention on Multicore Platforms

In this section, we first introduce a general cache contention model on a typical multicore platform. Then, a motivational example of real benchmarks on a real platform is given to explain cache contention. After that, the source code of memory-intensive programs aiming at causing as much as cache contention is introduced.

3.1 Cache contention model

A typical multicore architecture is composed of multiple independent processing cores, multiple layers of caches, a shred memory controller, and DRAM memories. In this section, we introduce the contention model on a general multicore architecture, as illustrated in Fig.4. We call the affected program a subject program, and call the affecting program a concurrent program or a co-runner. Assume that the subject and concurrent programs are co-located on a multicore processor, in which the processing cores have a split L1 cache (i.e., instruction cache and data cache) while sharing an integrated L2 cache. In addition, we assume that there is a core and memory isolation between the subject and concurrent programs. To be more precise, the co-runners cannot be assigned on the same core as the subject program, moreover, they cannot access the memory of subject program directly. The main purpose of the concurrent programs is to analyze the delay of the subject program's execution time, also called performance degradation in this paper.

The performance degradation of the subject is defined as follows: firstly, we run the subject program on core 0, and measure its solo execution time, denoted as *T_solo*. Secondly, we locate the number of co-runners on Cores 1-3 executing concurrently with subject, and measure the response time of subject program again, denoted as *T_corun*. Then, the performance degradation can be computed as the ratio of co-run execution time to its solo execution time, which is *T_corun/T_solo*.

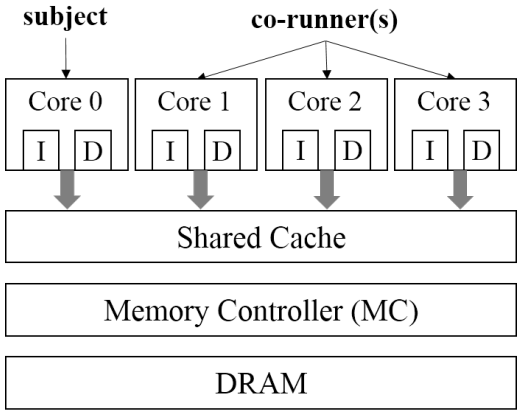
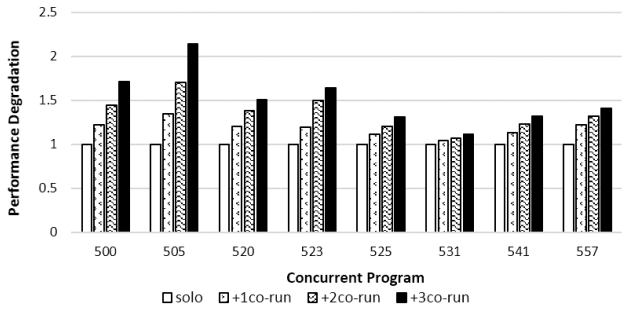


Figure 4: Cache contention model on a multicore architecture

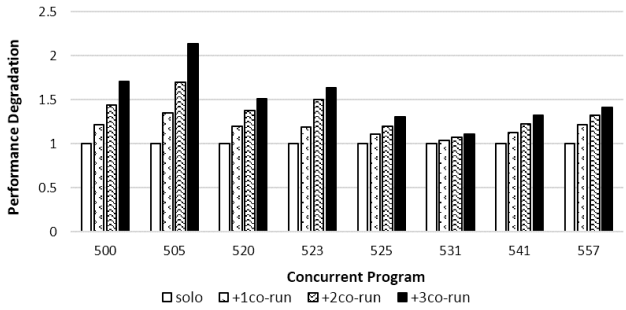
3.2 Motivational Example

In this section, we give a motivational example to explain the performance impacts of cache contention. We start the analysis from a set of real benchmarks SPEC 2017 [7] on a real board R-Car H3 [8]. The experiment setup is the same as described above,

where we first measure the solo execution time of a subject program on one CPU core, and co-execute it with three concurrent programs which are assigned on the other three CPU cores. In this example, we set eight SPEC 2017 Integer benchmarks as subject programs individually. The performance impact of each program on Cortex-A57 and Cortex-A53 is drawn in Fig.5, respectively. Evaluation results show that the performance degradation of each program is large than 1, which reflects that the execution time is indeed delayed because of cache contention. Also, it is noted that when the subject program is set as *505.mcf_r,* the performance degradation becomes worse than other programs. This is because, the application *mcf* in SPEC benchmark suits executes a significant number of memory related operations and has larger working sets [9]. Requiring more accesses to the memory hierarchy makes it more vulnerable to performance degradation due to memory contention.



(a) evaluation results of performance degradation on Cortex-A57 cores



(b) evaluation results of performance degradation on Cortex-A53 cores

Figure 5: Performance degradation of SPEC 2017 Integer benchmarks on R-Car H3 (+3 concurrent programs)

3.3 Subject and Concurrent Code

The motivational example described above not only proves that the execution time is extended due to memory contention, but also shows that the programs with more memory accesses are more likely to suffer memory contention. As we focus on exploring different levels of performance degradation in this work, a high amount of memory requests are required to generate to observe the severity of cache contention. Therefore, besides an evaluation of SPEC benchmarks, we specially design two separate regulations doing read and write operations in memory. Fig.6 shows the source code of originally generated programs.

Assume that there is a multicore platform with 2GB-sized shared cache. We intend to generate as many as cache accesses of the co-runners, thus they can cause more delays of the subject program's execution time. We define an array set to a set of a changing variable *N*. More precisely, when we prepare the *read.c/write.c* as the concurrent program, we define *N* in Fig.6 as 4, which implies that the array size is four times the cache size, and the cache miss is expected to be very high. While when we define the *read.c/write.c* as the subject program, we set the array size as 1/4 of the cache size (i.e., *N*=1/4), so that it is expected to always hit the shared L2 cache. In other words, we wish the cache contention is caused by the co-execution with other programs, but not caused by the subject itself. It is also noteworthy to mention that when the processor needs to transfer data between cache and memory, it always operates in a cache line unit. Since the cache line size in our evaluated platform is 64 Bytes, the stride of array operation is defined as 16 (=64 *Bytes/sizeof(int)*) to ensure more cache misses occur. Finally, to measure the execution time of each program, we simply set the iteration of array operation as 10000.

```
#define array_size 2*1024*1024/sizeof(int)*N
volatile int array[array_size]
int c;
int main()
{
   for (int i=0; i<array_size; i++)
   {
      array_size[i]=0;
   }

   for (int j=0; j<10000; j++)
   {
      for (int k=0; k<array_size; k=k+16)
      {
            c=array[k];
      }
   }
   return 0;
}
```

(a) *read.c* source code

```
#define array_size 2*1024*1024/sizeof(int)*N
volatile int array[array_size]
int c=0;
int main()
{
    for (int j=0; j<10000; j++)
   {
      for (int k=0; k<array_size; k=k+16)
      {
            array[k]=c;
      }
   }
   return 0;
}
```

(b) *write.c* source code

Figure 6: Read and write operations in memory

4. Experimental Evaluation

In this section, we present the embedded multicore platforms, evaluation procedures, and experimental results of the performance analysis. The purpose of this work is to explore the main factors that contribute to different levels of performance degradations caused by cache contention.

4.1 Embedded multicore platforms

We evaluate the performance impact on two embedded multicore platforms: R-Car H3 and Raspberry Pi 3 Model B+ [8, 10]. The former R-Car H3 is designed based on ARM big.LITTLE architecture [11]. It employs four Cortex-A57 cores as the high-performance core, which are performing out-of-order execution, and chooses four Cortex-A53 cores as the energy-efficient core, which are performing in-order execution. While the Raspberry Pi 3 Model B+ equips a quad-core processor, i.e., an ARM Cortex-A53 Quad Core Processor. Note that every core of the two platforms has its own independent L1 cache, while the identical cores are arranged into an integrated cluster and share a common L2 cache. Besides, R-Car H3 has a 4GB LPDDR4 SDRAM, with a 12.8GB/s data transfer, while Raspberry equips a 1GB LPDDR2, offering 8.5GB/s speed. The platform specifications can be seen in Table 1. Since the performance is highly dependent on the operational frequency, to make a fair comparison, in this work, we assume all the processing cores are operating at the maximum frequency.

Table 1: Compared embedded platforms and core types

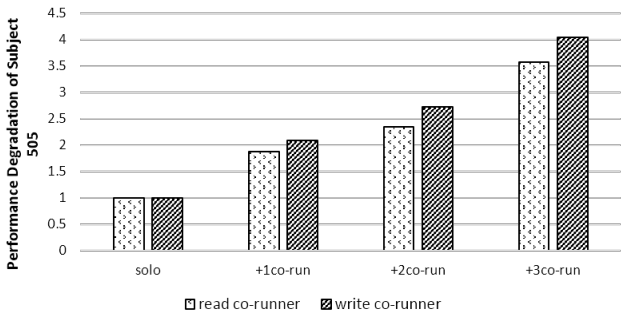
Platform	R-Car H3		Raspberry Pi 3 Model B+
CPU	4 Cortex-A57 (out-of-order)	4 Cortex-A53 (in-order)	4 Cortex-A53 (in-order)
L1 Cache	I: 48KB D: 32KB	I: 32KB D: 32KB	I: 32KB D: 32KB
L2 Cache	2MB	512KB	512KB
Memory	4GB LPDDR4		1GB LPDDR2
Bandwidth	12.8 GB/s		8.5 GB/s

4.2 Evaluation results

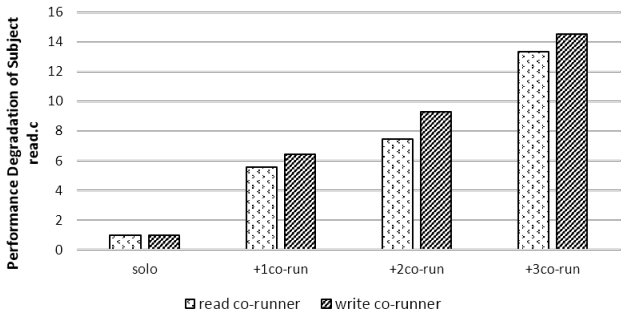
To demonstrate the affecting factors on performance degradation, we generate two separate programs doing read and write operations in memory as described in section 3.3. In this section, we conduct evaluation experiments of two assessment programs on two types of cores: Cortex-A57 and Cortex-A53. Below, the details of the procedures and results are introduced.

- **Impact of program characteristics**

In the first experiment, we investigate the impact of program characteristics on performance degradation. As illustrated in Fig.7, both *505.mcf* and *read.c* (N=1/4) are defined as subject programs executing on Core 0. We co-execute the subject program with read co-runners and write co-runners (N=4, executing on Cores 1-3), respectively. Results show that both the read and write co-runners cause execution time delay of subject programs, more than 14X delay as can be seen in Fig.7. Besides, it is also found that the degree of execution time delay has a dramatic effect on the memory operations of concurrent applications. Concretely, compared with read co-runners, write co-runners always cause server performance degradations. This is because, the generated read co-runners keep generating read transactions in memory, and they may always miss the cache since the array size is 4 times as the cache size. The missed loads then stress the MSHRs in the non-blocking cache and cause performance degradation. On the other hand, the write co-runners perform write operations in memory. Although they cause cache misses as the same as read co-runners, the write miss may trigger dirty conditions of cache lines (inconsistent cache and memory data). The dirty cache lines are required to be saved in WriteBack buffer. Summarily, in addition to stressing MSHRs as read co-runners do, the write co-runners also stress WriteBack buffer due to the existence of dirty cache blocks. Thus more timing impacts occur in the case of write co-execution.



(a) performance impact on *505.mcf*



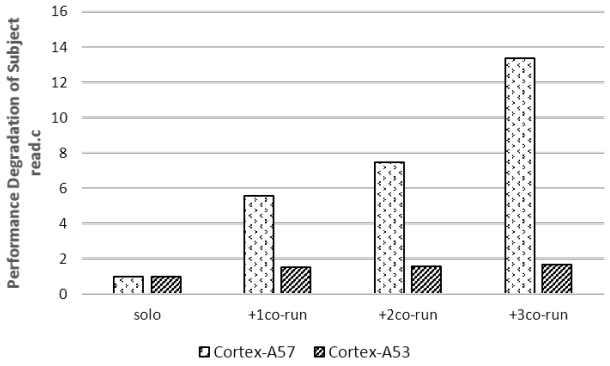
(b) performance impact on *read.c*

Figure 7: Comparison of read/write co-runners on R-Car H3 (e.g., on Cortex-A57)

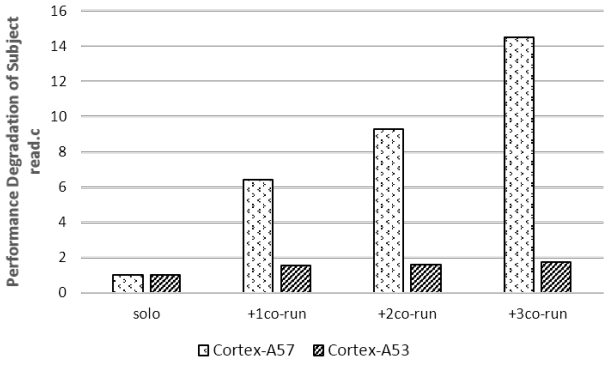
- **Impact of CPU designs**

In this experiment, we evaluate the performance impacts of non-blocking caches on CPU designs. Without loss of generality, the processor can be classified as in-order and out-of-order. Therefore, we conduct the experiments on two CPU-designed ARM processors: out-of-order Cortex-A57 processors and in-order Cortex-A53 processors. We take read program (N=1/4) as the subject, and co-execute it with three read co-runners and write co-runners separately. The results in Fig.8 show that the out-of-order designed CPUs suffer server performance impacts than in-order designed CPUs. We consider the reasons as follows: each core in out-of-order processors can generate multiple outstanding memory

requests at a time, while each core in in-order processors can only generate one. In other words, out-of-order cores can generate more concurrent cache access than in-order processors. The higher degree of parallelism supported by out-of-order execution is more likely to cause MSHR contention and thus degrade the performance. It is noteworthy that the same results can be observed by evaluating SPEC programs. Considering the limited space of paper, the experiment results are not showed here.



(a) co-execute with read co-runners



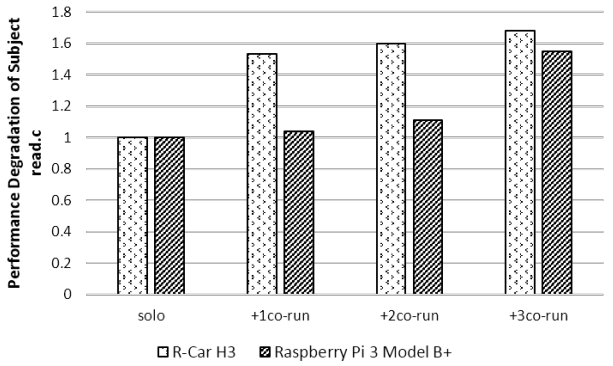
(b) co-execute with write co-runners

Figure 8: Comparison of CPU designs on R-Car H3

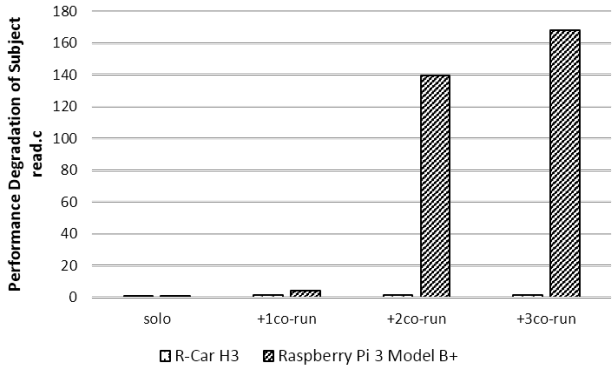
- **Impact of implementation platforms**

Memory bandwidth is also an important aspect for performance analysis, as it affects how quickly the OS can get data into and out of memory for processing. In this experiment, we evaluate the effect of memory bandwidth on performance degradation. Instead of comparing two types of CPUs in the last experiment, we repeat the experiments in the type of core, but different implementation platforms.

We first co-execute the subject program (*read.c*, N=1/4) with read and write co-runners (N=4) on two recent embedded platforms, R-Car H3 and Raspberry Pi 3 Model B+, both of which are equipped with four in-order Cortex-A53 cores. Experiment results show that when the subject program is co-executed with read co-runners, the execution time increase induced by cache contention is close, ranging from 1.04X to 1.68X, as shown in Fig.9(a). However, when it is co-executed with write co-runners, execution time increase occurred on Raspberry is much worse than that of R-Car H3. A gap of more than one hundred times was observed to be more precise. As illustrated in Table 1, both R-Car H3 and Raspberry Pi 3 Model B+ has four Cortex-A53 cores with the same sized cache, while R-Car H3 has a larger memory size and bandwidth. Thus, the performance degradation degree in R-Car H3 is expected to be smaller than that in Raspberry. The interesting observation is that the extreme difference is only observed on write co-execution experiments, as can be seen in Fig.9(b), while the results in Fig.9(a) were not the case. Therefore, we suspect that because write operation causes more cache misses than read (illustrated in previous experiments), they are more susceptible to memory bandwidth.



(a) co-execute with read co-runners



(b) co-execute with write co-runners

Figure 9: Comparison of implementation platforms (e.g., on Cortex-A53)

5. Conclusion

In this paper, we made an analysis of performance degradation on a shared cache in multiprocessor systems. We evaluated SPEC 2017 benchmarks and two separate read and write memory-intensive programs, and found that the degrees to performance degradation are closely related to program characteristics. In particular, write concurrent applications cause higher memory contention and thus contributes to server performance degradation than read concurrent applications. Besides, we performed experiments on two recent ARM processors: in-order Cortex-A53 and out-of-order Cortex-A57. Evaluation results showed that

compared with in-order processors, out-of-order processors are more likely to suffer MSHR related cache blocking thus cause more serious performance degradations. Finally, we evaluated the experiments on the same processing CPU, but different implementation platforms. It was observed that even for the same processors, memory bandwidth leads to significant effects on the variabilities in performance degradation. Besides the observations presented in this paper, we also found that the cache prefetching technique affects the program performance and cache miss significantly. For future work, we plan to investigate the configuration and mechanism of hardware prefetchers in the performance issue.

Reference

[1] Mittal, S. (2017). A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)*, *50*(2), 1-39.

[2] Valsan, P. K., Yun, H., & Farshchi, F. (2016, April). Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (pp. 1-12). IEEE.

[3] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., & Sha, L. (2013, April). Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (pp. 55-64). IEEE.

[4] Bechtel, M., & Yun, H. (2019, April). Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (pp. 357-367). IEEE.

[5] Kroft, D. (1998, August). Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)* (pp. 195-201).

[6] Chen, T. F., & Baer, J. L. (1992). Reducing memory latency via non-blocking and prefetching caches. *ACM SIGPLAN Notices*, *27*(9), 51-61.

[7] "SPEC CPU® 2017." https://www.spec.org/cpu2017.

[8] "R-Car H3." https://www.renesas.com/kr/en/solutions/automotive/soc/r-car-h3.html.

[9] Singh, S., & Awasthi, M. (2019, April). Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (pp. 285-292). ACM.

[10] "Raspberry Pi 3 Model B+." https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[11] Jeff, B. (2012). Advances in big. little technology for power and energy savings. *ARM White paper*, 33.

Acknowledgments The author Yang Qin thanks for the financial support from China Scholarship Council (CSC, 201606090182).