Work in Progress: Identifying Unexpected Inter-core Interference Induced by Shared Cache

Denis Hoornaert*, Shahin Roozkhosh†, Renato Mancuso† and Marco Caccamo*
*Technical University of Munich †Boston University
{denis.hoornaert, mcaccamo}@tum.de, {shahin, rmancuso}@bu.edu

***Abstract*—In modern real-time multicore systems, understanding and adequately managing shared caches is essential to ensure the temporal isolation of critical tasks. Recent research has identified and extensively studied the sources of unpredictability imputable to shared caches, heavily promoting techniques such as cache partitioning and internal resources management.**

In this article, we highlight the existence of an enigmatic source of inter-core interference: the CPU-brainfreeze. Experiments realized on a development board show that benchmarks (selected from the San-Diego Vision Benchmark Suite) can exhibit up to a 10-fold increase in their execution time. The same experiment shows that for extreme cases, the core cluster can be stalled indefinitely.

***Index Terms*—Multi-processors Systems, Real-Time Systems, Non-blocking Shared Caches**

I. Introduction

In modern high-performance mutliprocessor system-on-a-chips (MPSoCs), caches have become an essential piece of hardware bridging the gap between the speed of the processing elements and the main memory. The growing demand for high-performance system has engendered the emergence of *non-blocking caches*, a type of shared cache capable of accommodating several concurrent accesses to main memory and hiding the cache-miss penalty.

Unfortunately, while non-blocking shared caches offer high average bandwidth, their behavior is opaque and unpredictable. Understanding the cache behavior is of the utmost importance for safety-critical hard real-time systems where timing constraints must be respected and guaranteed. For instance, the *Federal Aviation Administration* (FAA) mandates the use of a single processor unless the impact of all the temporal interference channels existing in multi-core platforms can be appropriately identified, mitigated, or bounded.

A great deal of research has been conducted on cache management for real-time applications on MPSoCs. The two main sources of unpredictability imputed to the *last-level cache* (LLC) are (1) the inter-core cache line eviction and (2) the opaque management of shared internal resources.

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

The inter-core cache line eviction is a well-studied source of unpredictability that arises when the memory accesses of two independent cores lead to the eviction of each other cache lines in a destructive way. Such source of unpredictability can be prevented by enforcing the *spatial isolation* of the cores through *way-based* or *set-based* partitioning [3], [5], [8].

Inter-core interferences caused by shared internal resources such as the *Miss-Status-Holding-Registers* or the *write-back* unit have been recently studied in [2], [10]. If left unmanaged, the contention on these resources can create significant interferences even if the cores are spatially isolated.

In this article, we show the existence of a third source of inter-core interference linked to the speed at which the cacheable target memory reacts. In other words, if a target memory acknowledges a transaction coming from one core but waits to deliver the response, the execution time of tasks running on independent cores (co-runners) is impacted. Experiments performed on the ARM Cortex-A53 core cluster [1] show that a co-running task can see its execution time multiplied up to 10 times. Furthermore, we show that if the target memory acknowledges a single read transaction but never provides a response, the whole core cluster is frozen indefinitely. Considering the advertised Memory-Level-Parallelism of modern core clusters such as the ARM Cortex-A53, such a result is unexpected and counter-intuitive.

To the best of our knowledge, this article is the first to report inter-core interference caused by isolated and unserved read transactions.

II. Background

A. Non-blocking Caches

Caches in modern MPSoCs are crucial components that efficiently circumvent the performance gap between the speed of the processing elements and the main memory. However, as good as they are at providing high bandwidth, *blocking caches* are ineffective at hiding cache-miss penalty because they stall the processing elements until the data is received from the main memory. In order to hide this penalty and improve the cache performance, [4] proposed the first *Miss-Handling-Architecture* (MHA). This type of cache referred to as *Non-blocking* relies on the introduction of a set of new registers called *Miss-Status-Holding-Register* (MSHR), which are in charge of tracking the status of cache line misses.

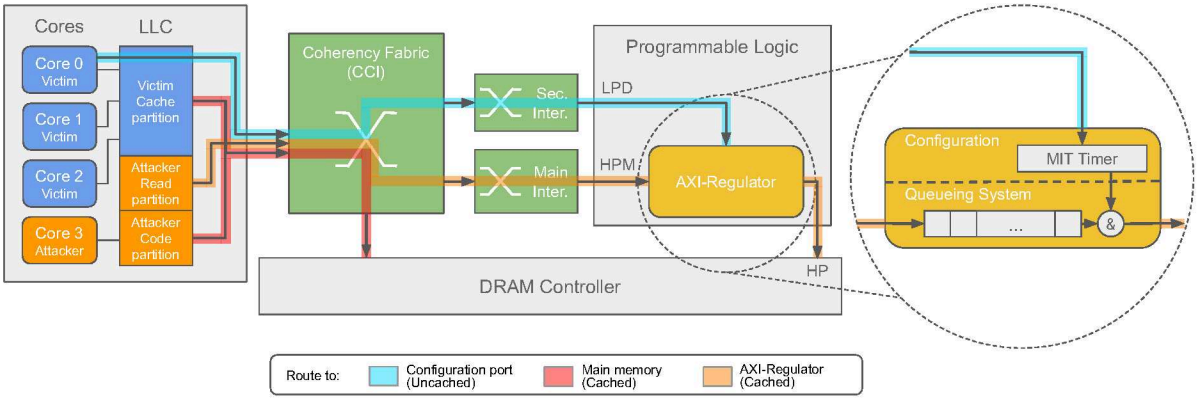

Fig. 1: Schematic view of the considered setup with the partitioning of the core cluster (both cores and LLC) on the left, the different path taken by the transactions highlighted in cyan, orange and red, and the AXI-Regulator on the right.

Each MSHR stores important information regarding the cache-misses such as the target address and the location of the cache line to refill. For each level of cache in a system, the amount of MSHRs denotes the number of outstanding (i.e., simultaneous) transactions it can handle. This amount is known as the *Memory-Level-Parallelism* (MLP).

At run time, a non-blocking cache behaves as follows. When a cache-miss occurs, the metadata of the cache-miss is stored in one of the available MSHRs. In case the same cache-miss already happened and one MSHR already holds the metadata, the two requests are merged. It is only once the cache line refill request has been served and placed in the proper cache line that the MSHR is made available to store new cache-misses requests. If none of the MSHRs are available, the system stops until one of them becomes available.

B. Programmable Logic In the Middle (PLIM)

The *Programmable Logic In the Middle* (PLIM) is a new paradigm introduced by [6] that takes advantage of the newly available platforms associating a traditional *Processing System* (PS side) with a tightly integrated Programmable Logic (PL side). In a PLIM system, the PL side is leveraged to create a secondary route between the core cluster and the main memory as displayed in Figure 1 with the orange route. This secondary route is populated with custom logic IPs, called PLIM modules, allowing the CPU-originated traffic to be manipulated before reaching the main memory. Despite the inherent cost of re-routing the traffic through the PL side, the use of PLIM modules broadens the control over memory operations considerably as it becomes possible to manipulate the memory traffic at the granularity of individual transactions. For instance, [6] tackles significant constraints imposed by the cache coloring technique via a PLIM module, called *bleacher*, which applies a configurable transformation on each incoming transaction address.

In this paper, the PLIM paradigm is used to have fine control over certain transactions and as a mean to showcase the CPU-brainfreeze.

III. Related work

A sizable amount of research has focused on addressing the challenges of isolating the cores sharing the same cache in order to prevent unpredictable temporal behavior. Most of this research [5], [8] has aimed at *spacially isolating* the cores (i.e., avoiding inter-core cache line eviction by constraining each core data and instructions in a specific region of the shared cache). Hardware-based solutions such as *lockdown per way* [3] are efficient, but not integrated in every platform. On the other hand, software-based solutions such as cache coloring [5], [9] can be deployed on most platforms but come at the cost of increased memory space requirements.

However, recent research [2], [10] has highlighted that, while cache partitioning is successful in most cases, in some situations, contention on shared internal units such as the MSHRs or the write-back unit can also introduce substantial inter-core interferences. In [10], the authors evaluate the impact of inter-core interference originated at the MSHRs on multiple platforms and propose a solution to eliminate this contention. The solution is based on a combination of a small hardware module and an OS-level controller. Their experiments show that, if left unmanaged, the execution time of independent cores is multiplied respectively by 10.6 and 21.3 under read and write workloads. Via simulation, they prove that their approach successfully provides the best overall throughput for each core while mitigating the inter-core interference caused by the MSHRs. Similarly, [2] investigates the contention in caches caused by shared internal units in the case of *Denial-of-service* (DOS) attacks and propose an OS-level solution enabling finer management of the system bandwidth. In contrast to [10], the internal unit studied and exploited is the write-back unit. They report that, by exploiting this unit efficiently, one can increase the execution time of a victim task by a factor of 346.

The CPU-brainfreeze studied in this paper differentiates itself from the one in [2], [10] as inter-core interference is not caused by the saturation of shared resources. Instead, the CPU-brainfreeze arises when a single outstanding transaction is left unserved by the target memory for an extended period.

IV. System Model

The system model assumed in this paper is composed of two isolated actors: a *victim* and an *attacker*. On one hand, the victim is defined as a set of trusted *hard real-time tasks* (HRTs). On the other hand, the attacker is a lightweight application in charge of disturbing the victim. The attack consists in a continuous flow of single sequential read transactions emitted towards a slow memory, the AXI-Regulator[1].

This section is divided into three parts, each giving further details on the system model components. First, in Section IV-A details regarding the isolation of the actors are given. Secondly, a complete description of the attacker's design is provided in Section IV-B. Finally, Section IV-C explains the AXI-Regulator architecture and mechanism.

A. Processing System Organization

As displayed in Figure 1, the actors are located on the same core cluster but are allocated non-overlapping sets of resources. In other words, they run on different cores and have private LLC partitions. This measure enforces the independence of the two actors and ensures that the observed interference cannot be imputed to either a common software stack or inter-core cache line evictions. Moreover, the private LLC partition of the attacker is subdivided into two. The first half allows the attacker to access the main memory, where its code is located (red route in Figure 1), whereas the second half is dedicated to the data read through the AXI-Regulator (orange route in Figure 1).

Assuming the attacker accesses to the main memory introduce little to no inter-core interference, the two actors can be deemed as properly isolated.

B. Attacker's Design

Even with the precautions mentioned in Section IV-A (i.e., actors isolation), the design of the attacker must be thought carefully to highlight the CPU-brainfreeze. Because, if not under control, a read memory bomb will steadily fetch data, creating many cache-misses. Following the non-blocking cache mechanism, these cache-misses will be inserted in one of the available MSHRs until all of them are used. In this situation, the non-blocking cache controller will stop the whole machinery, leading to the phenomenon reported by [10].

This effect can only be avoided by throttling down the attacker's core. We enforce this by following each read request by a *Data Synchronization Barrier* (`DSB`). This instruction ensures that at each instant, there will not be more than one transaction targeting the AXI-Regulator and, by extension, it guarantees at most one MSHR is occupied by the attacker.

C. AXI-Regulator IP

In our system model, the AXI-Regulator is a PLIM module [6] located on the secondary route to the main memory (orange route in Figure 1) and used to act as a slow cacheable target memory. Moreover, the AXI-Regulator prevents potential interference introduced by the DRAM controller as the former intercepts every transaction before they reach the latter.

The mechanism enabling the characteristics mentioned just above is the following. Upon the reception of a read transaction coming from the core cluster via the HPM port, the IP inserts this transaction within a queue where it waits to be relayed out of the AXI-Regulator. The decision to relay the transaction stored at the head of the queue to the DRAM controller is done according to a timer. The latter is located within the AXI-Regulator (*Configuration* in Figure 1) and, from the DRAM controller perspective, enforces a minimal inter-arrival time (MIT) between two consecutive transactions. The MIT is expressed in clock cycles (CC) and is dynamically reprogrammable thanks to a configuration port accessible by all cores with uncached transactions (cyan route in Figure 1).

V. Evaluation

For the experiments, we use the Jailhouse-RT project [7] to partition the four ARM Cortex-A53 cores [1], and the 1 MB of shared LLC offered by the Xilinx ZCU102 platform [12] according to the system model presented in Section IV. As illustrated in Figure 1, the victim is allocated three cores and half of the LLC, whereas the attacker is left with the remaining. Software-wise, the victim runs a selection of benchmarks issued from the San-Diego Vision Benchmark Suite [11] on top of Linux 4.14. On the other hand, the attacker runs a lightweight bare-metal version of the memory bomb described in Section IV-B. Finally, the PL side and the AXI-Regulator are clocked at 250 MHz.

To highlight the impact created by the attacker introducing the CPU-brainfreeze interference, we run the selected set of benchmarks[2] for all the available input sizes[3] (x axis in Figure 2) and for different configurations of the AXI-Regulator[4]. The result of each combination of a benchmark and input size has been normalized with respect to the same benchmark running alone. This baseline is referred to as *Solo* and represented by the leftmost bar of each bar cluster bars in Figure 2.

From the results displayed in Figure 2, two observations can be made. Firstly, the benchmarks have different sensitivity to the attacker. In fact, the `mser` benchmark is more affected by the attacker than `disparity`. The former particularly suffers for a small input size (i.e., `sim`), with its execution time increased by a factor of 10. On the other hand, disparity seems unaffected by the attacker, meaning that spatial isolation of the cores is enough. The increments of execution time observed in this experiment are in the same range as those reported by previous research. However, in our case, this interference is caused by only one outstanding memory transaction instead of a continuous flow of transactions generated by three cores. Secondly, big MITs tend to introduce more inter-core interfer-

[1] *Advanced eXtensible Interface (AXI)*, the bus communication protocol used for PS-PL communications

[2] `disparity`, `mser`, `sift` and `tracking`

[3] with the exception of `sim_fast` and `full_hd`

[4] I.e., MITs of 2^{10}, 2^{15}, 2^{17} and 2^{19} clock cycles

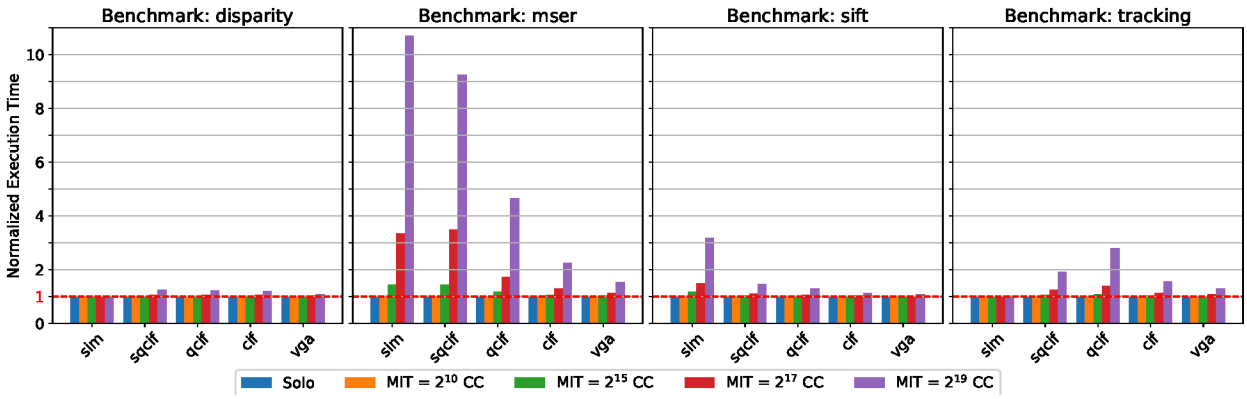

Fig. 2: Normailized execution time for different combinations of benchmark (inset), input sizes (x axis) and MITs (bar).

ence than their counterpart. For instance, a MIT of 2^{19} always introduces some visible interference, with varying magnitude.

Emulating an infinite MIT by configuring the AXI-Regulator to accept the transaction but never answer systematically leads the whole system to be suspended indefinitely. In other words, if a task tries to fetch data from a non-responding memory, not only its core stalls, but the whole core cluster is suspended. This result suggests that for extremely big MITs, significant increases in execution could be observed and that tasks deployed on all the cores might be affected by the inter-core interference.

VI. Discussion

Non-blocking caches are advertised as capable of seamlessly hiding the cache-miss penalty and managing multiple simultaneous memory accesses unless either all the MSHRs are occupied or if the write-back unit buffer is full. A priori, nothing in the non-blocking cache architecture suggests that a single outstanding read transaction could introduce inter-core interference. However, our experiments suggest otherwise.

While the exact source of the observed inter-core interference is unclear to the authors, all the precautions taken during the experiment (i.e., isolation of the inmates and partition of the cache) and the result suggest that the source originates from the LLC controller itself.

The authors acknowledge that the described phenomenon is unlikely to occur in a normal situation (i.e., all the inmates target the main memory), and if it does, the consequences should be negligible. Nonetheless, this experiment has the merit of pinpointing a clear issue in the LLC controller design in ARM Cortex-A53 clusters [1]. It is a reminder of the gap between the theoretical models, the expectations on the hardware, and real-world behavior.

VII. Conclusion

In this article, we have highlighted the existence of an enigmatic source of inter-core interference in the ARM Cortex-A53 clusters caused by isolated and delayed read transactions. The interference increases the execution time of co-runners by a factor of 10 in the worst case despite all the precautions taken to eliminate known sources of interferences. Even worst, the experiment shows that in the most extreme case, the whole core cluster can be stalled, jeopardizing any isolation.

Extensions of this work will focus on strengthening the experimental setup, narrowing down the scope of potential sources of the observed interference, and propose a solution to prevent it. In addition, the authors plan to experiment with other wide-spread ARM Cortex CPUs to see whether the issue is specific to the A53 model.

References

[1] ARM, "ARM Cortex-A53 MPCore Processor Technical Reference Manual," Tech. Rep., 2014. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf

[2] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *RTAS 2019*, 2019, pp. 357–367.

[3] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2830555

[4] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 81–87.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," *RTAS 2013*, pp. 45–54, 2013.

[6] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *RTAS 2020*, Sydney, Australia, 2020.

[7] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-WarP: a system-wide framework for memory bandwidth profiling and management," in *41st IEEE RTSS 2020*, Houston, TX, USA, Dec. 2020.

[8] 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 *2013 IEEE 16th International Conference on Computational Science and Engineering*, 2013.

[9] M. S. T. Kloda, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems," in *25th IEEE RTAS 2019*, Montreal, Canada, April 2019, conference, pp. 1–14.

[10] P. K. Valsan, H. Yun, and F. Farshchi, "Addressing isolation challenges of non-blocking caches for multicore real-time systems," *Real-Time Systems*, vol. 53, pp. 673–708, 2017.

[11] 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 *IISWC 2009*, 2009, pp. 55–64.

[12] Xilinx, "Zynq UltraScale+ Device Technical Reference Manual," Tech. Rep., 2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf