Boston University

OpenBU http://open.bu.edu

BU Open Access Articles BU Open Access Articles

2020-06

Reconciling predictability and coherent caching

Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, Marco Caccamo. 2020. "Reconciling Predictability and Coherent Caching." 2020 9th Mediterranean Conference on Embedded Computing (MECO). 2020 9th Mediterranean Conference on Embedded Computing (MECO). 2020-06-08 - 2020-06-11. https://doi.org/10.1109/meco49872.2020.9134262

https://hdl.handle.net/2144/43023

Downloaded from DSpace Repository, DSpace Institution's institutional repository

Reconciling Predictability and Coherent Caching

Ayoosh Bansal *, Jayati Singh *, Yifan Hao*, Jen-Yang Wen*, Renato Mancuso† and Marco Caccamo‡
* University of Illinois at Urbana-Champaign, {ayooshb2, jayati, yifanh5, jwen11}@illinois.edu
† Boston University, rmancuso@bu.edu
‡ Technical University of Munich, mcaccamo@tum.de

***Abstract*—Real-time systems are required to respond to their physical environment within predictable time. While multi-core platforms provide incredible computational power and throughput, they also introduce new sources of unpredictability. For parallel applications with data shared across multiple cores, overhead to maintain data coherence is a major cause of execution time variability. This source of variability can be eliminated by application level control for limiting data caching at different levels of the cache hierarchy. This removes the requirement of explicit coherence machinery for selected data. We show that such control can reduce the worst case write request latency on shared data by 52%. Benchmark evaluations show that proposed technique has a minimal impact on average performance.**

***Index Terms*—hardware/software co-design, worst-case execution time, cache coherence, memory contention**

I. Introduction

The last decade has witnessed a profound transformation in the way real-time systems are designed and integrated. At the root of this transformation are the ever growing data-heavy and time-sensitive real time applications. As scaling in processor speed has reached a limit, multi-core solutions have proliferated. Not only does this add a new dimension to scheduling, but the fundamental principle that worst-case execution time (WCET) of applications can be estimated in isolation has been shaken.

In multi-core systems, major sources of unpredictability arise from contention over shared memory resources. Resource partitioning techniques present a suitable approach to mitigate temporal interference between cores [1]. These solutions, however, are well suited only for systems where data is not exchanged between cores.

Modern platforms generally feature a multi-level cache hierarchy, with the first cache level (L1) comprised of private per-core caches. When multiple threads access the same memory locations, it is crucial to ensure the coherence of different copies of the same memory block in multiple L1 caches. Dedicated hardware circuitry, namely the *coherence controller*, exists to maintain this invariant. Because maintaining coherence requires coordination among distributed L1 caches, it introduces overheads.

Cache coherence introduces two main obstacles for real-time systems. First, hardware coherence protocols are a preciously guarded intellectual property of hardware manufacturers. As such, scarce details are available to study the worst-case behavior for coherent data exchange. Second, coherence controllers are designed to optimize throughput and not worst-case behavior.

This paper proposes a new approach to achieve predictable time access to coherent data. The key intuition is that if data accessed by multiple cores is cached only in cache levels visible to *all* cores that access this data, all accesses to the data are served by the same cache and data coherence is trivially satisfied. Section II describes this further. Control over caching is provided to the developer and/or compiler. The **contribution** of this work is a novel solution for predictable access time to shared data by elimination of variability induced by cache coherence mechanisms. A longer preprint of this work can be found here [2].

II. Solution Overview

The main idea is to allow application developers or compilers to use their knowledge of the application to choose between the trade-offs of worst-case vs. average use access time with fine granularity. The choice of cacheability determines which level of caches can the data be cached in and which levels of cache data cannot be cached. Data locations for which strong worst-case latency guarantees are required would be selectively cached in shared levels. Data coherence is achieved as all accesses go to the same cached copy of such data. Coherence overheads and variability are avoided.

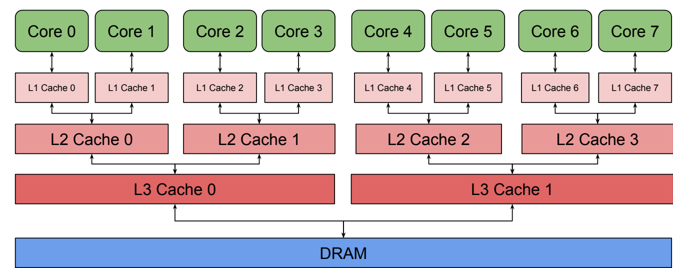

Fig. 1. Generalized multi-socketed multi-cluster processor.

Consider a hypothetical system as shown in Figure 1. For each core (C) consider a set M which consists of all

memory blocks that are in its data pipeline. A memory block can be a specific cache ($) or DRAM. For example:

$$M(C0) = \{L1\$0, L2\$0, L3\$0, DRAM\} \tag{1}$$
$$M(C1) = \{L1\$1, L2\$0, L3\$0, DRAM\} \tag{2}$$
$$M(C7) = \{L1\$7, L2\$3, L3\$1, DRAM\} \tag{3}$$

We propose that for any cache line which is assessed by only a subset of cores, the cache line be stored/cached on only the intersection of the sets M of these cores. Consider a cache line accessed by Core 0 and Core 1. This cache line can be stored in L2 Cache 0, L3 Cache 0 and DRAM but not in L1 Cache 0 or L1 Cache 1.

$$M(C0) \cap M(C1) = \{L2\$0, L3\$0, DRAM\} \tag{4}$$

Consider another cache line that is accessed from Core 0 and Core 7. Based on our proposal this data block should not be cached at all and be stored in DRAM only.

$$M(C0) \cap M(C7) = \{DRAM\} \tag{5}$$

Under these restrictions, explicit mechanisms to maintain data coherence are not required. The restrictions should be expressed at application level, possibly as granular as a cache line, so hard real time applications can leverage predictable access time to shared data, while non real time applications running on the same system can enjoy the higher throughput of hardware managed cache coherence as they are not affected by the occasional spikes.

In reality, the closest existing support is cacheability control for two cache levels expressed at memory page granularity in ARVv8-A ISA, as further described in Section IV-B. The implementation and evaluation are based on an ARVv8-A processor simulator, Figure 5, and limited to two cacheability levels.

III. Related Work

Multi-core systems have enabled multi-threaded real-time workloads. Cache contention among parallel tasks is a major cause of inter-core interference and execution time variability [3]. Mitigation approaches include selective caching [4] and cache partitioning [5]. But strict resource partitioning among cores is effective only for independent tasks with scarce data sharing. In MC2 [6], the authors acknowledge that data-sharing between tasks is inevitable in mixed criticality multi-core systems. Our work focuses on tightening and simplifying the analysis of the WCET bound by making shared data accesses immune to temporal effects of cache coherence controllers.

The problems introduced by data sharing in real-time signal processing applications were studied in [7], which demonstrate that the overhead from cache coherence protocols can severely diminish the potential gains from parallelism in multi-core systems. To address these problems the works [7] and [8] disallow concurrent access of shared data, but that only works at task level granularity and may force idle times on processor. Predictable MSI [9] uses a TDMA coherence bus and a modified coherence protocol to remove variability. The solution is invisible to software but imposes slowdown on all memory accesses and requires major changes to hardware coherence controllers. MC2 [6] avoided coherence effects by making the shared memory uncacheable or restricting tasks with shared memory accesses to run on the same core. The scheduling option is restrictive and like [7] may force processor idling. Extra accesses to uncached memory, i.e. main memory, are slow and can increase the WCET.

Our proposed solution does not impose any scheduling restrictions on the application and does not burden it to maintain coherence. It provides the developer freedom to choose which data to cache where and tradeoff average and worst-case performance. Our solution can be implemented without any changes to hardware coherence controllers and works with any coherence protocol. Minimal changes to the cache controller's logic are required.

IV. Background

This section provides background knowledge on cache coherence and memory types in existing processors.

A. Cache Coherence

In traditional cache architectures, it is fundamental that the contents of private levels of caches are kept *coherent* across multiple cores. A hardware cache coherence controller is present for this purpose. It ensures that any valid copies of a cache line contain the same data. Cache coherence controllers work by assigning additional states to cache lines. ***MSI*** cache coherence protocol [10] with its states and transitions is shown in Figure 2.

S*hared* state cache lines contain valid data that can be read by the processor. Other caches may have the same cache line in ***S****hared* state. ***M****odified* state cache lines can be used to read or write data. Other caches cannot contain this line in any valid state. **I**nvalid state is the initial state and identifies an unused cache line. A cache line transitions between these state based on *Self* vs. *Other* load/store (LD/ST) events, as shown in Figure 2. Here, *Self* refers events generated by the the core under analysis. Evictions are cache line replacements. *Other* refers to messages to handle events generated by other cores.

B. Memory Types

A vast majority of modern multi-core embedded systems are implemented using ARM architectures. We focus on the latest major version ARMv8-A, extensively used in current platforms. This includes recent versions of Nvidia Tegra, Qualcomm Snapdragon and Samsung Exynos, among others. There are 100+ mobile and embedded SoC compliant with ARMv8-A Instruction Set Architecture (ISA). In this architecture, a uniform physical memory address space describes traditional memory resources (e.g. DRAM space), as well as configuration space for on-chip and external devices.

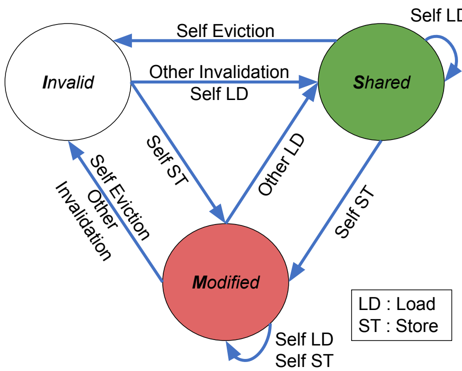

Fig. 2. MSI States and Transitions

In order to adopt the correct caching policy for any given memory region, the hardware allows specifying a set of meta-data, or *memory type*, for each memory page. The memory type specification informs the hardware of how load/store operations within a given memory range should be handled. Memory type attributes are encoded in each virtual memory page table descriptor. In setting up virtual memory, the OS is responsible for encoding the correct memory type in the page table entry (PTE) of any portion of memory being accessed.

ARMv8-A standard allows defining *cacheability* in the memory type, i.e., whether or not a memory location should be cached or not. There exist two cacheability attributes: *inner cacheability* and *outer cacheability*. If a memory region is marked as inner (resp., outer) cacheable, its content can be cached in the inner (resp., outer) cache levels. Our general solution would require an expansion of the notion of cacheability levels, one for each cache level. In this work, we focus on a memory types possible within the current ISA [11]. Specifically, Table I defines the memory attributes used throughout the paper. The default memory type is *Normal Cacheable*. This type of memory is cacheable at all levels of caches. The other frequently used memory type is *Uncacheable*. This memory type is typically used to describe I/O memory. We define a new memory type: *Inner Non Cacheable, Outer Cacheable **(INC-OC)*** and also address kernel support in Section VI-A2. This type of memory is accessed by the processor cores only. It is cached in all shared (outer) cache levels but not cached in any caches private (inner) to any cores. While ISA support exists, hardware implementations treat INC-OC as Uncacheable memory. X86 and MIPS ISA do not support granular cacheability control for different cache levels and are hence not considered in this work.

V. Motivation

Cache coherence mechanisms are troublesome for hard real time applications. The reasons are explored in this section.

A. Coherence Complexity

Hardware cache coherence simplifies the development of general purpose multi-threaded software. Many applications are served well by the transparent handling of data coherence by the hardware. But for real-time applications this creates another uncontrolled source of unpredictability in their worst-case execution time. Cache coherence protocols in SoCs are defined by vendors with only the main stable states [12]. There are a plethora of transient states in coherence state machines and many low level details that impact the overall coherence state machine operation [13]. This makes any analysis on existing cache coherence controllers difficult. Hard real-time systems cannot be certified if they suffer from this kind of unknowable behavior and execution variability. Our approach of caching data only in selected cache levels removes the cache coherence controller from the shared data access process.

B. Coherence Cost

Cache coherence introduces a new dimension and source of latency to cache behavior. Consider a 2 core processor with 2 cache levels, each core has a private L1 cache and a L2 cache that is shared between both cores. In this example, the core attached to L1 Cache1 initiates a store operation. This cache does not have the data block for the Store, but L1 Cache2 has the data block in ***Modified*** state. Cache2 has to invalidate its cache line and write back the dirty data to the shared L2 cache. The L2 cache can then send the data to L1 Cache1 which can finally execute the Store. L1 Cache1 now contains the cache line in Modified state. These series of events can lead to long latency in executing a single memory access. We refer to this situation as a *Dirty Miss* in this paper. Figure 3 illustrates this. The latency to handle a *Dirty Miss* can far exceed the latency for a L2 cache hit.

C. Private vs Shared Access

In this section we discuss the difference between accessing Shared vs Private data on two real platforms. First, a 4 core ARMv8-A Cortex-A53 processor, and second a 14 core Intel Xeon E5-2658 processor.

TABLE I
Memory Types

Name	Cacheability	Description
Normal Cacheable	Inner Cacheable, Outer Cacheable	Data caching allowed in all caches
Uncacheable	Inner Non-Cacheable, Outer Non-Cacheable	Data caching not allowed
INC-OC	Inner Non-Cacheable, Outer Cacheable	Data caching allowed in Shared caches only

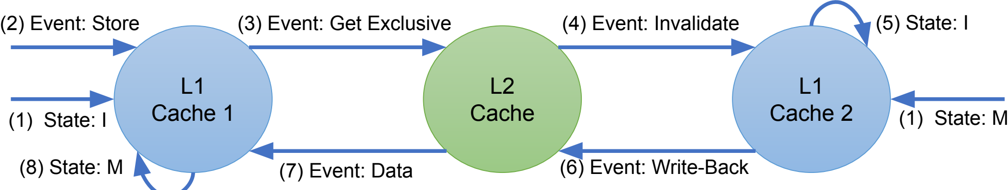

Fig. 3. Transitions of a Dirty Miss

We developed custom benchmarks to study the effect of cache coherence on real platforms. They measure the average latency to complete a Load or Store to data already present in L1 caches. All cores do the same operations simultaneously. The resulting average latency is a combined effect of single access latency, parallelization, bandwidth contention and opportunistic hardware optimization like prefetchers. Figure 4 shows the comparison for private where each core accesses dedicated memory ranges only vs shared data accesses. The accesses can be reads, writes followed by reads or locks. The Lock latency is the average latency to acquire and release a spinlock. Spinlock implementation is below and uses gcc built-in atomics. Once acquired, locks are immediately released. Private locks are accessed only by one core and is included only as a reference point. Shared locks are accessed by all cores but since the locks are immediately released minimal time is spent spinning on the lock itself.

```
lock: while(__sync_lock_test_and_set(&lck,1)){};
unlock: __sync_lock_release (&lck);
```

Listing 1. Spin Lock and Unlock code snippets

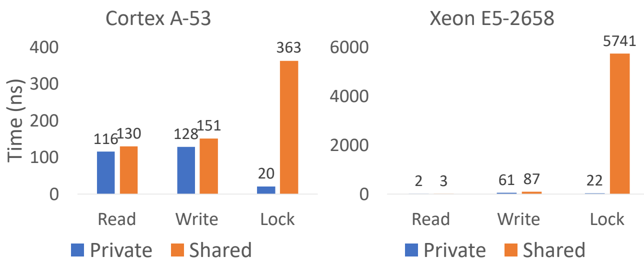

Fig. 4. Private vs Shared Access Latency

While exact overheads of cache coherence are hard to measure on real platforms, it is evident that the latency to access data is dependent on whether it is being shared across different cores. The Load/Store measurements represent latency differences near full memory bandwidth and hence the effect of coherence itself is diminished. Locks on the other hand require that the underlying micro-ops/instructions complete in order. Locks are hence affected more by the overheads of maintaining coherence.

VI. Evaluation

The implementation and evaluation are based on gem5 [14] hardware simulator. A system as shown in Figure 5 is realized in the simulator. The implementation and raw results are here[1].

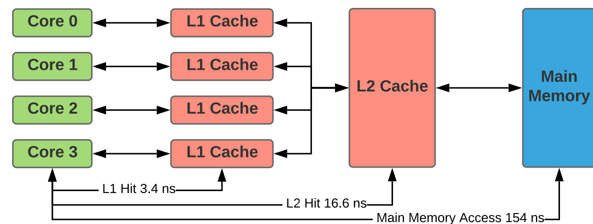

Fig. 5. ARM Cortex A-53 equivalent used for evaluation.

A. Implementation

1) Application: *mmap* is modified to accept additional flags that are used by the kernel to determine the cacheability of allocated pages. For example:

```
buf = mmap(0, size, PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_INCOC, fd, offset);
```

Listing 2. Sample memory allocation for INC_OC memory type

2) Kernel: For an OS to provide the cacheability control to the userspace application, two components are required: first, APIs to allow applications to choose the memory type, as shown above. Second, page table entries need to be set up with the right value as defined by the ISA to use the INC-OC memory type. We implemented both these components for Linux ARM64. Linux kernel defines 6 memory types in arch/arm64/include/asm/memory.h. The memory type field is 3 bit wide and hence two more memory types can be defined without significant changes. We defined one of the unused values as a new memory type INC-OC. The kernel changes can run on any ARMv8-A compliant platform and set the memory type bits for INC-OC as defined in the ARMv8-A ISA, but the eventual handling depends on the underlying hardware.

3) Caches: The cache controllers behave differently for Normal and INC-OC memory type.

Normal Memory: The coherence for Normal Memory type is maintained with the MSI protocol. Other protocols

[1] https://gitlab.engr.illinois.edu/rtesl/inc-oc

can also be used. This cache coherence mechanism exists in modern multicore systems and need not be modified.

INC-OC Memory: If a memory request is marked as Inner Non-Cacheable the request is directly forwarded to the L2 cache, bypassing the L1 cache hierarchy completely. Since INC-OC data blocks are never cached in the private L1 caches, multiple copies of the same data can not exist in caches, therefore, no extra logic is required to maintain coherence. Another change is to correctly handle Load/Store exclusive (LDXR/STXR) instruction pair in L2 Cache.

B. Worst Case Analysis

Evaluation I: Using a controlled execution mode of the gem5 simulator [9], the worst case scenario for a memory request is evaluated. A write request to the same address is generated by all 4 cores, simultaneously. The address was written to by one of the cores previously so the corresponding cache line is in the dirty state in one of the private caches. Figure 6 shows the timeline of the flow of first write request and additionally the time to completion of all write requests.

Observation: The total time to process the *dirty miss* is **52%** shorter for INC-OC memory. For all 4 write requests the total time was reduced by **74%**. At per request level INC-OC memory type has a large reduction in worst case access time.

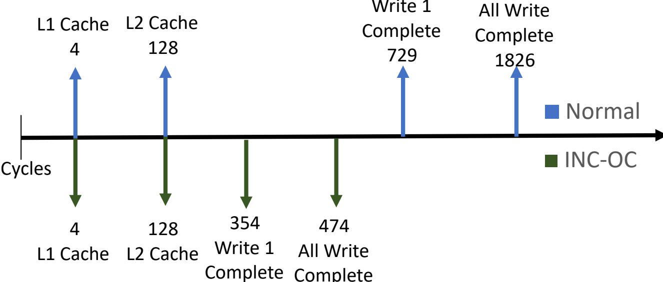

Fig. 6. Worst case write request contention

Evaluation II: Custom benchmarks are run within Linux running over gem5 hardware simulator. Figure 7 shows the results of running the custom benchmarks discussed in Section V on the simulation setup.

Observation: Latency to acquire locks reduces by **85%** by the use of INC-OC memory type. Load and Store time for INC-OC memory types increases. The forced ordering of locking primitives makes the Lock benchmark sensitive to latency only and hence coherence effects are observable. In case of loads and stores the combined effect of bandwidth and parallel handling of coherence of individual lines makes INC-OC average access latency higher.

C. Average Performance Analysis

Evaluation: Run time for randomly selected SPLASH2 [15] benchmarks. For *Normal* all memory used is normal cacheable. For *INC-OC* all shared memory locations that can be written to by different cores are allocated as INC-OC memory type. Figure 8 shows normalized results.

Observation: *INC-OC* memory type support does not cause a general slowdown of the application. Note that gem5 simulator aims to have deterministically repeatable execution. Consequently the observed execution time variability is less than 1%.

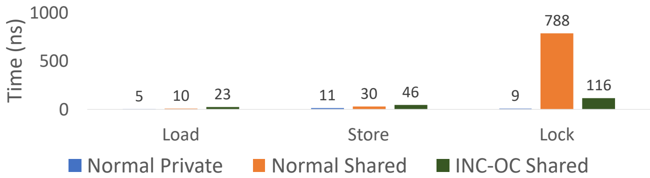

Fig. 7. Custom benchmarks that compare normal private and shared accesses to INC-OC type.

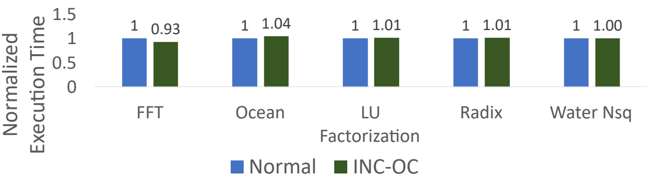

Fig. 8. SPLASH2 Benchmark evaluation.

VII. Future Work

In this work we assume that an application programmer manually selects cacheability for each memory allocation that requires limited cacheability. This creates manual overheads and a higher cost of adoption of this process into application development. So in future work we will present a compiler extension for INC-OC. A compiler extension can statically analyze the application, identifying memory locations that are accessible by different cores in parallel. The cacheability of these memory locations can then be directly modified by compiler. Use of INC-OC would become a compiler option while continuing to support manual annotations by application developers.

VIII. Conclusion

This paper presents a solution for memory access time variability caused by cache coherence mechanisms. INC-OC memory type bypasses private caches, hence avoiding coherence overheads and reducing memory request worst-case access latency.

Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1646383. M. Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. 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.

References

[1] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale *et al.*, “Single core equivalent virtual machines for hard real—time computing on multicore processors,” Tech. Rep., 2014.

[2] A. Bansal, J. Singh, Y. Hao, J.-Y. Wen, R. Mancuso, and M. Caccamo, “Cache where you want! reconciling predictability and coherent caching,” *arXiv preprint arXiv:1909.05349*, 2019.

[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, pp. 32:1–32:36, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2830555

[4] B. Lesage, D. Hardy, and I. Puaut, “Shared data caches conflicts reduction for wcet computation in multi-core architectures.” in *18th International Conference on Real-Time and Network Systems*, 2010, p. 2283.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS 2013*, 2013, pp. 45–54.

[6] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, “Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 57–68.

[7] G. Gracioli and A. A. Fröhlich, “On the design and evaluation of a real-time operating system for cache-coherent multicore architectures,” *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 2, pp. 2–16, Jan. 2016.

[8] A. Pyka, M. Rohde, and S. Uhrig, “Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems,” in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, July 2014, pp. 107–114.

[9] M. Hassan, A. M. Kaushik, and H. Patel, “Predictable cache coherence for multi-core real-time systems,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017, pp. 235–246.

[10] T. Suh, D. M. Blough, and H. . S. Lee, “Supporting cache coherence in heterogeneous multiprocessor systems,” in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, Feb 2004, pp. 1150–1155 Vol.2.

[11] Arm Holdings, “Arm cortex-a series programmer’s guide for armv8-a,” 2018. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CEGDBEJE.html

[12] Arm Holding, “6.2.5. Data cache coherency,” 2019. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500j/ch06s02s05.html

[13] D. Abts, S. Scott, and D. J. Lilja, “So many states, so little time: Verifying memory coherence in the cray x1,” in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 10–pp.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” in *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.