PASoC: A Predictable Accelerator Rich SoC for Safety-Critical Systems

by

Susmita Tadepalli

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Abstract

This thesis presents a model of a Predictable Accelerator-rich System-on-Chip (PASoC) for safety-critical systems, which guarantees timing predictability of a memory access in the system. Earlier adoption of accelerator-rich SoCs was for general-purpose computing and thus timing predictability of such systems was not well explored, despite being used in safety-critical systems. This thesis takes initial steps in exploring the predictability of ASoCs by combining CPU clusters with one or more hardware accelerators. The PASoC allows the integration of multiple coherent agents to interact with each other over a shared memory bus and a shared LLC. These agents can be a cluster of cache-coherent homogeneous cores, and fully or one-way coherent hardware accelerators. PASoC ensures the predictability of a memory request through some modifications in hardware architecture and cache coherence protocols. PASoC supports predictable cache coherence within the cluster of cores and across agents. The former uses linear cache coherence, and the latter uses a modified version of predictable Modified Shared Invalid (MSI) cache coherence protocol. PASoC analyzes the per-request worst-case latency of a memory request from any of the agents and evaluates the design on the gem5 simulator. Finally, this work presents some observations based on the analysis that can help in future designs of PASoCs.

Acknowledgements

I want to thank my supervisor, Professor Hiren Patel, for his guidance throughout my graduate studies. I also acknowledge and thank the rest of the Computer Architecture and Embedded Systems Research (CAESR) team at the University of Waterloo. Thanks to Zhuanhao Wu for his continued support over the past two years. I also want to thank my family for their ongoing support of my endeavors and for helping me get motivated at the most challenging times.

Table of Contents

List of Figures

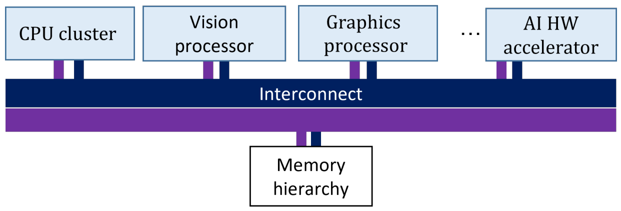
List of Tables

Chapter 1

Introduction

Modern System-On-Chips (SoCs) combine general-purpose multicores with one or more hardware accelerators. These hardware accelerators offer performance, power, and energy improvements for several domain-specific algorithms. Common hardware accelerators include those that implement vision processing and deep learning as shown in Figure 1.1a. While much of the early adoption of such Accelerator-Rich SoCs (ASoCs), was in the domain of general-purpose embedded computing, ASoCs are steadily being adopted in safety-critical systems such as transportation (e.g., automotive, railway, avionics, space) and industrial machinery (e.g., wind turbines, industrial control robots) [23] [30]. One popular use case is in autonomous driving, where a combination of clusters of cores are interconnected with hardware accelerators for vision processing and deep learning. Naturally, industries are also designing specific SoCs for autonomous driving that are accelerator-rich. For example, NVIDIA's DRIVE AGX Orin SoC promotes itself to be specifically designed for next-generation autonomous driving [10].

The Orin SoC has ARM-based multicores, a Graphics Processing Unit (GPU), and



(a) Block diagram of ASoC

(b) Applications of Safety-critical systems

Figure 1.1: ASoCs for safety-critical systems

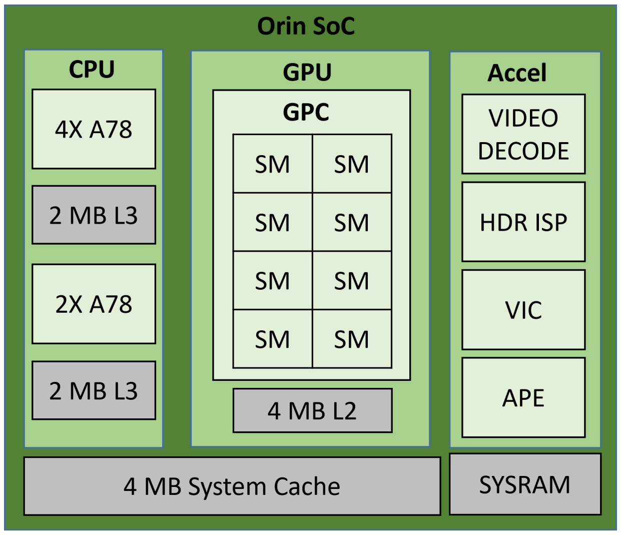


Figure 1.2: NVIDIA Orin SoC [10]

hardware accelerators for deep learning and vision processing, as shown in Figure 1.2.

1.1 Decisions in Designing ASoCs

A key design decision in designing ASoCs involves deciding the manner in which the accelerators interact with other agents within the ASoC. An agent can be a multicore cluster or a hardware accelerator. It is widely known that caching of data and their coherent communication is one way to allow multiple cores or agents to interact with each other while delivering high performance. As specialized accelerators are integrated with CPU clusters, many heterogeneous architectures are moving towards unified, coherent address space for ease of programmability and inter-device communication [35] [31] [34]. Designing such platforms involves determining the accelerators' coupling via interconnect with other agents and their coherence modes in accessing the shared memory [30].

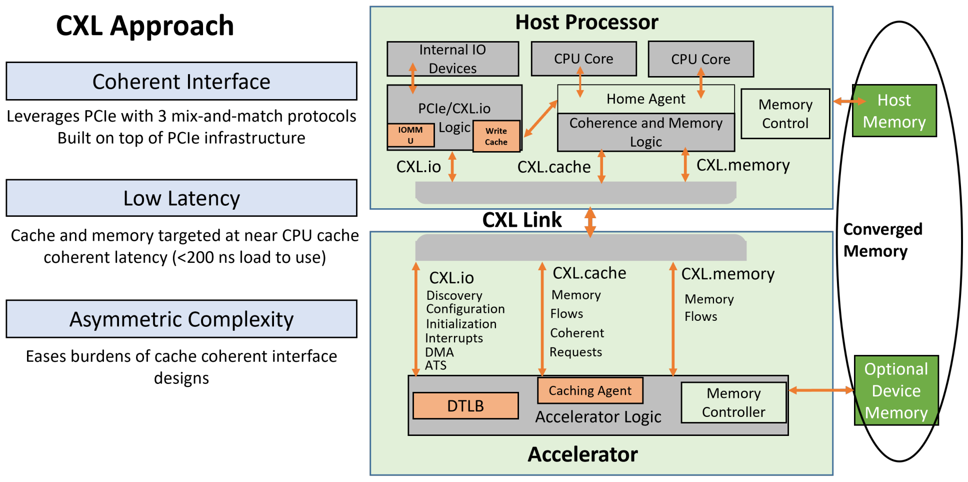
Coupling defines where accelerators are deployed in the system. Today's system hierarchy typically includes a pipelined processor core with multiple levels of caches attached to the memory interconnect and then connected to I/O devices through the I/O interconnect. Conceptually, accelerators can be attached to all levels of this hierarchy, and hence the degree of coupling can be discretized into the following categories; accelerators that are part of the pipeline, accelerators that are attached to cache or memory via on-chip memory interconnect, and accelerators that are attached off-chip via the I/O interconnect. As accelerators are more tightly coupled, such as those that are part of the processor's pipeline, they require more modifications of the host processor designs but promise lower invocation latency. And, as accelerators are more loosely coupled, such as those attached to SoC via a memory interconnect, they often incur high invocation costs. But are free from the design constraints of the host core and are more flexible.

This particular decision has resulted in industry-proposed solutions that focus on integrating accelerators to an SoC via cache-coherent interconnect standards such as the Compute Express Link (CXL [31]) and AXI Coherence Extension (ACE [34]). As shown in Figures 1.4a and 1.4b, CXL connects accelerators with other agents in the system via a cache-coherent IO interconnect, and ACE attaches the accelerators on SoC via an on-chip cache coherent interconnect.

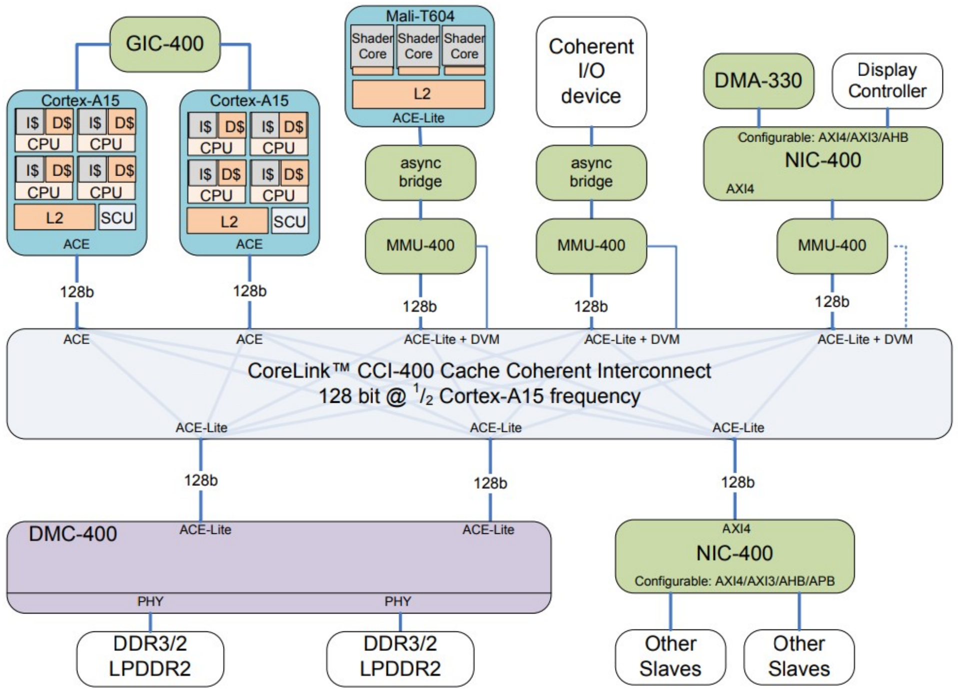
Another key decision involves the coherence modes, which dictate the coherence activity the accelerators respond to in accessing the shared memory. Accelerators in ASoCs can have a wide range of memory demands, motivating different cache coherence strategies. For example, CPU applications tend to prefer low latency memory accesses. As a result, they use cache coherence protocols like MSI, which obtain persistent read and write permissions for data at line granularity. However, such protocols are complex and can incur high coherence overheads for throughput-oriented applications. This leads to different cache coherence modes. Examples include one-way coherent and fully coherent. One-way coherence, also called I/O coherence or coherent DMA, is an approach for integrating an accelerator into an SoC [39]. In one-way coherence, the accelerator can snoop on the private caches of other agents but does not respond to any coherence activity by other agents in the system. In fully coherence, the accelerator can snoop on other agents private caches and respond to any coherent activity by other agents in the system.

This has resulted in exciting academic research discussing the modes of cache coherence best suited for the application [39] [1]. For example, Cohmeleon [39], as shown in Figure 1.4a, integrates accelerators with different coherence modes to the SoC and applies reinforcement learning to select the best coherence mode at runtime. Spandex [1], as shown in Figure 1.4b, designs a flexible LLC that can efficiently integrate devices with different coherent strategies.

Despite being used in safety-critical systems, the impact of these design decisions on the predictability of ASoCs is not well explored.

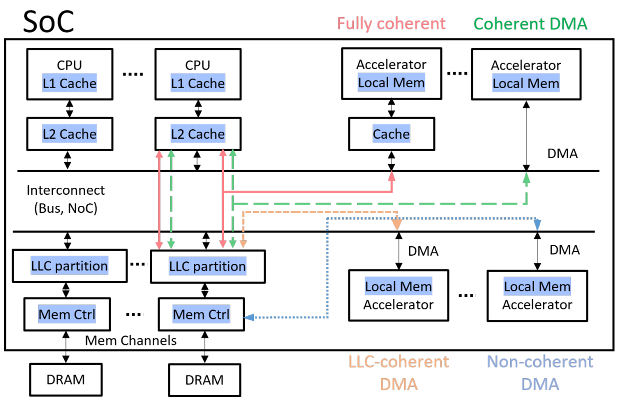


(a) CXL: Accelerator coupling via IO bus [31]

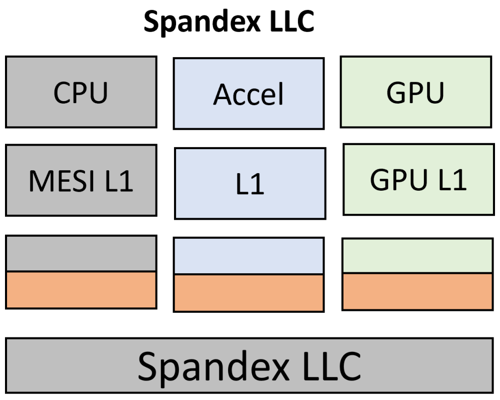


(b) ACE: Accelerator coupling via on-chip cache coherent interconnect [34]

Figure 1.3: Accelerator coupling



(a) Cohmeleon: Accelerators interaction with different coherence modes [39]



(b) Spandex: Flexible LLC integrating devices with different coherence strategies [1]

Figure 1.4: Accelerator coherence modes

1.2 Predictability of ASoCs for Safety-Critical Systems

Safety is defined as the "absence of catastrophic consequences on the user(s) and the environment." Safety-critical systems are programmable electronics systems whose failure can lead to catastrophic consequences, like environmental damage or human casualties (e.g., autonomous driving system accident) [23]. Thus, predictability is an essential requirement for safety-critical systems as it enables analyses to compute worst-case execution times for deemed critical applications. In the context of homogeneous multicores for safety-critical systems, researchers have developed predictable cache coherence that exploits cache coherence for performance while providing low Worst Case Latency (WCL) bounds [15] [37] [20]. However, there has been limited exploration on the predictability of heterogeneous systems [27] [14]. Safety-critical applications deployed on ASoCs must often deliver high performance and predictability. The former allows the application to deliver its quality of service, and predictability enables certification to ensure that worst-case execution times of safety-critical tasks are always honored. Balancing the need for high performance and predictability is a recurring challenge in designing predictable systems, which is further worsened with heterogeneous computing elements.

This work attempts to take some initial steps towards exploring the design of a Predictable ASoC (PASoC). The presented PASoC consists of multiple agents interacting over the shared memory bus that share a Last-Level Cache (LLC). An agent can be a cluster of predictable cores [13] or an accelerator. PASoC supports cache coherence within the cluster of predictable cores using a predictable cache coherence protocol called linear coherence [20]. Between the agents, PASoC employs a modified predictable MSI protocol [15]. The main focus of this thesis is to analyze the WCL of a memory request from any agent in the PASoC and identify opportunities to improve the design of future PASoCs. Prior research in developing predictable cache coherence protocols for safety-critical systems assumed that all cores were fully coherent, and they did not consider the integration of accelerators.

1.3 Thesis Objectives

The main objectives of this thesis are as follows.

- Model a PASoC and integrate multiple coherent agents that share an LLC. These agents consist of hardware accelerators and clusters of predictable cores [13]. Each agent in the PASoC can support one of the two cache coherence modes: fully coherent or one-way coherent.
- Implement the PASoC model by employing state-of-the-art proposals on predictable cache coherence and combining them. Specifically, PASoC uses linear coherence [20] within a cluster and Predictable Modified-Shared-Invalid (PMSI) [15] between the agents.
- Analyze the WCL of a memory access from any of the agents in the PASoC.
- Through this analysis, identify opportunities to improve the design of future PASoCs for safety-critical systems.

1.4 Thesis Outline

The thesis is organized into chapters as follows.

- Chapter 1 discusses the position and objectives of the thesis in the broader context of ASoCs for safety-critical systems. It outlines the decisions in designing ASoCs and explores the predictability of such ASoCs. It outlines the design and implementation of PASoC using state-of-the-art hardware cache coherence proposals for predictable systems.

- Chapter 2 presents background on predictable cache coherence models for multi-core systems and coherence for heterogeneous systems. The discussion of predictable cache coherence models covers the hardware architecture and cache coherence protocol changes to ensure predictability. Coherence for heterogeneous systems discusses different cache coherence modes and interconnects for integrating accelerators into an SoC.

- Chapter 3 reviews prior works in designing predictable SoCs for safety-critical systems. It also reviews works that examine the cache coherence models for accelerator-rich SoCs.

- Chapter 4 describes the system model of PASoC. It discusses the hardware architecture, memory hierarchy and coherence protocols deployed in designing PASoC.

- Chapter 5 presents the WCL analysis for a memory access in PASoC. It describes about different latency terms used in the analysis and explains the critical instance scenario in detail.

- Chapter 7 presents the evaluation of the PASoC system model on the gem5 simulator and discusses the observations obtained from evaluation results.

- Chapter 8 reiterates the contributions and findings of the thesis. It outlines the limitations of the work and subsequent opportunities for future work.

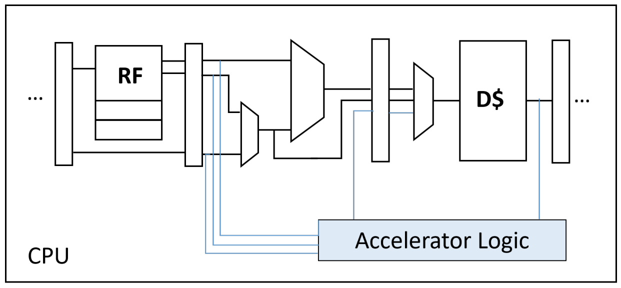
Chapter 2

Background

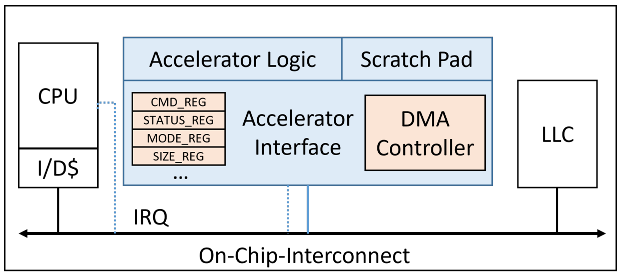
The integration of accelerators into an SoC has been a trend in a variety of computing systems, ranging from embedded devices to server-class computers, to open up the possibilities to computationally demanding applications such as automated drive, robotics, uncrewed aerial vehicles, and in health care [23]. The CPU and accelerator interaction is crucial in leveraging the performance benefits of these ASoCs. To that end, this chapter presents a background on accelerator's coherence and accelerators coupling to the SoC. Additionally, safety-critical systems also require that the integration of accelerators into SoC does not affect the predictability of the system, ensuring Worst Case Execution Time (WCET) bounds for a memory access. Interference from cache coherence can negate the benefits of parallelism, and it is essential to study cache-coherence effects when deriving WCET bounds [15]. Hence, this chapter also discusses predictable cache coherence models, which are incorporated for designing PASoC.

2.1 CPU Accelerator Coupling

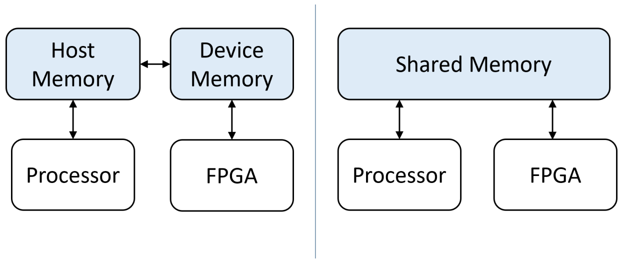
One of the key decisions while integrating accelerators to SoC is the CPU-accelerator coupling in the system. Conceptually, accelerators can be attached to any hierarchy level in the system, ranging from CPU's pipeline to off-chip. This work studies the implementation of accelerators following three models of coupling as presented below.



(a) Tightly-coupled accelerators [9]



(b) Loosely-coupled accelerators [9]



(c) Off-chip accelerators [8]

Figure 2.1: CPU-accelerator coupling

2.1.1 Tightly Coupled Accelerators (TCAs)

TCAs are located very close to the host CPU and share key resources with CPU like register file, L1 data cache, and Memory Management Unit (MMU) as shown in Figure 2.1a [9]. Hence, the CPU stalls until TCA finishes its execution. TCAs do not require internal storage since they share the CPUs' L1 data cache. They provide nil runtime overhead of invocation because of being located close to the CPU. They are used to accelerate critical portions of an application kernel, for example, the body of an inner loop of an algorithm. The tight coupling of accelerators with CPUs poses integration challenges and further complicates the CPU design because of hardware dependencies.

2.1.2 Loosely Coupled Accelerators (LCAs)

LCAs are located outside CPU cores and interact with them through an on-chip interconnect like NoC and AXI, as shown in Figure 2.1b [9]. This allows LCA to implement private local memories or caches, which helps to accelerate coarse-grained accelerator logic blocks with complex data paths, for example, a complete application kernel for image encoding.

LCAs share the same system memory with the host and are connected to either the LLC or DMA of the system. This also allows LCAs to support different cache coherence protocols based on their memory demands, which we discuss in the next section. Decoupling LCAs from the CPUs provides greater flexibility in design compared to the TCA model. Hence, they offer better integration because their design is independent of CPU cores. ACE [34] architecture is an example of this category.

2.1.3 Off-Chip Accelerators

Off-chip accelerators are located off-chip and are attached to the CPU using IO interconnects like CXL, CAPI, and PCIe, as shown in Figure 2.1c [8]. This approach is used for workloads that must be accelerated off-chip due to prohibitive area requirements or need reconfigurability, like FPGAs. This type of accelerator can have its memory subsystem with independent address space or have a unified shared address space between CPUs and accelerators. In the case of separate memory, the data shared between the host and accelerators must be allocated in both the memories and explicitly copied between them, resulting in extra latency and overhead. Microsoft Catapult [25] is an example of this category. The ideal case would be to have a unified shared address space where only a single allocation is necessary instead of allocating two copies in both host and device memory. This eliminates explicit data copies and increases the performance of fine-grained memory access. CXL and IBM CAPI platforms are examples of off-chip accelerators with unified shared coherent address space with the system [31] [35].

As accelerators become more tightly coupled and heterogeneous applications become more capable and diverse, the LCA approach has a higher scope to support a broad range of access patterns between CPUs and accelerators. Hence, this work focuses on the LCA model where CPU clusters and accelerators are integrated over SoC by an on-chip interconnect. The accelerators and CPUs share LLC and the main memory of the system.

2.2 Accelerator Coherence

Accelerators are heterogeneous in nature, and this diversity is reflected in data access patterns and communication properties in the system. The literature proposes many modes for interaction between accelerators and memory hierarchy, ranging from accelerators accessing off-chip memory directly bypassing the cache hierarchy to accelerators having their private caches. Similar to CPUs, accelerators can also benefit from having private caches and hardware cache coherence. To that end, this work takes initial steps in integrating accelerators with different cache coherence modes to the SoC, as discussed below.

2.2.1 Non-Coherent Accelerators

Non-coherent accelerators do not have a private cache, and their memory requests bypass the cache hierarchy of the SoC and access main memory directly [39]. In this approach, coherence is implicitly managed by software. If the accelerator data is allocated in a cacheable memory region, a flush of the caches is required before invoking the accelerator to ensure the main memory contains the updated data version. The non-coherent accelerators are best suited for executing large streaming workloads that do not fit on system caches [39].

2.2.2 Fully Coherent Accelerators

A fully coherent accelerator is equipped with a private cache, to which it sends memory requests instead of sending them directly on the system interconnect. Hardware cache coherence protocols used for general-purpose processors are required to keep the data coherent in the system. This work uses standard protocols for accelerators like MSI, MESI, or MOESI that are used for multicore processors. Besides, there are other options like Fusion [21], DeNoVo [7], and GPU-like [1] coherence protocols for accelerators. Designing PASoC with these coherence protocols is left for the future scope of the work.

2.2.3 One-way Coherent Accelerators

One-way coherence, also referred to as I/O coherence or coherent DMA, is an approach for integrating an accelerator to an SoC where the accelerators do not have private caches, and its memory requests are sent directly to the LLC interconnect [39]. One-way coherence sits between non-coherent accelerators and fully coherent accelerators in terms of coherence activity. In one-way coherence, the accelerator can snoop on the private caches of other agents but does not respond to any coherence activity by other agents in the system. The LLC, in turn, fetches or invalidates the requested data and responds to the accelerator. In this case, the CPUs and accelerators follow producer and consumer relationship [34]. For example, the ARM Mali-T604 GPU supports one-way coherency between the processors and GPU. The processor writes GPU commands and data to memory. Without one-way coherency, these commands must be flushed from the processor's cache so that the GPU can consume them. Cache flushing could take longer for larger LLC, consuming the precious CPU's bandwidth. With one-way coherence, the cache flushing is eliminated, and external memory accesses are reduced. Since memory bandwidth is increasingly the main system performance bottleneck and external memory accesses use far more energy than internal accesses, overall system performance and energy efficiency are enhanced with one-way coherent accelerators.

2.3 Hardware Cache Coherence

A Hardware Cache Coherence (HCC) mechanism avoids data incoherence by deploying a set of rules to ensure that cores access and cache the correct version of data at all times. The main component in a hardware cache coherence mechanism is a cache *coherence protocol*, a state machine that deploys the set of rules. The protocol state machine implements these rules with a set of coherence states that convey access permissions (read, write), and other information about the cache line data, and coherence state transitions between states triggered based on the memory activity of cores on the shared data. Cache coherence protocols deployed in current multi-core platforms consist of three fundamental stable states, which establish the Modified-Shared-Invalid (MSI) protocol: Modified (M), Shared (S), and Invalid (I) [33]. A cache line in M means that the current core has written to it and has not propagated the updated data to the shared memory yet. Only one core can have a specific cache line in a modified state and is called the owner. A core that has a cache line in M and observes remote memory activity on the cache line must update the cache line copy in the shared memory. This operation is called write-back to shared memory. A cache line in S means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in a shared state to allow for fast read accesses. Cores that have the same cache line in the shared state are referred to as sharers. This constraint of one owner for a cache line or multiple cores sharing a cache line is called the Single-Writer Multiple-Reader (SWMR) invariant [33]. A cache line in I denotes the unavailability of that line in the cache or that its data is outdated and no longer valid.

Figure 2.2 shows the implementation of an MSI coherence protocol for fully coherent and one-way coherent cores. Cores c_1 and c_2 are fully coherent cores, and ACC is a one-way coherent core. All the cores access shared memory via a shared interconnect. Initially, all cores have cache line A in I state (①). Core c_1 issues a store request to cache line A, which is a miss. Hence, it retrieves the data from shared memory. Shared memory remembers that A is modified in core c_1 (②). Next, core c_2 performs a load on A, which is a miss in its private cache (③). c_2 retrieves the most up-to-date value from c_1, maintaining a coherent

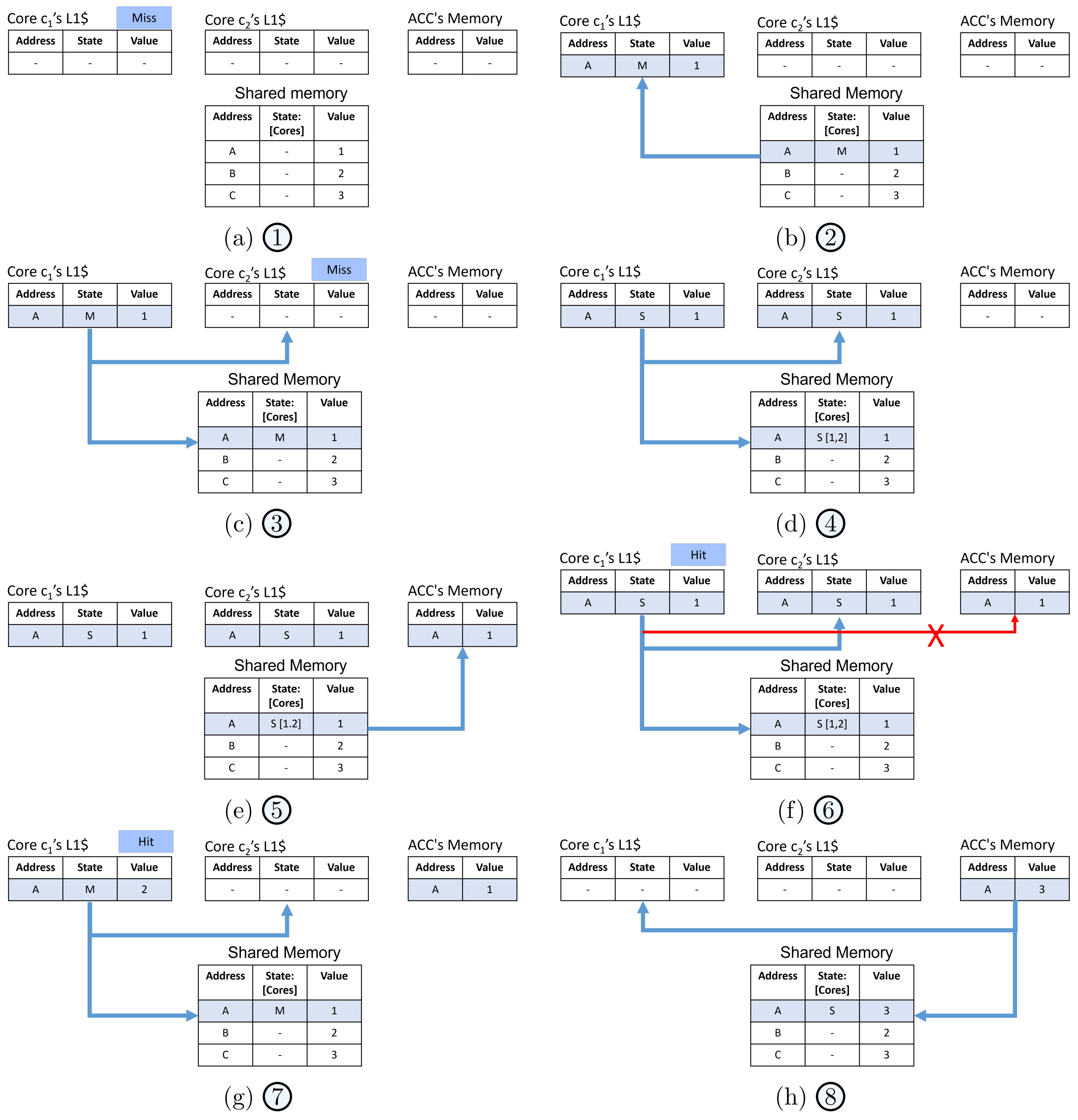


Figure 2.2: Cache coherence example

view of the data across all cores. Shared memory remembers that cores c_1 and c_2 share A in S state (④). Next, *ACC* performs a load on A and retrieves the most up-to-date data from shared memory (⑤). Next, c_1 performs a store on shared line A. This triggers invalidation in core c_2 so that its next load gets the most up-to-date value. However, *ACC* may not respond to c_1*'s* request when there is already a copy, as it does not need the most up-to-date data (⑥). This is true for producer consumer relationship between the CPU core and the accelerator. Shared memory remembers that c_1 modified A (⑦). Next, when *ACC* core issues a store to cache line A, it triggers invalidations in c_1 and c_2 so that the next load gets the most up-to-date value written by *ACC* (⑧).

2.4 Predictable Cache Coherence Models

Conventional cache coherence protocols can lead to scenarios where a memory request has unbounded memory latency. A predictable HCC mechanism ensures the predictability of a memory request with a defined WCL bound. Prior works on designing predictable HCC have certain requirements. First, the implementations should honor certain design invariants to address the unbounded memory latency. Second, several architectural and coherence protocol changes must be made in the system to ensure the predictability of a memory request and also to improve its WCL bound [15] [37] [20]. This section discusses some of the previous works in predictable cache coherence models that are used in designing PASoC.

2.4.1 Design Invariants for Predictable Cache Coherence

Prior work PMSI [15] proposed certain design invariants to address the unpredictable scenarios for a fully coherent multi-core processor that accesses a shared memory and implements a conventional MSI protocol. The proposed invariants are general design guidelines, which are independent of the adopted cache coherence protocol implementation and the underlying platform architecture. Hence, some of these design invariants are also applicable to PASoC system model, which are discussed below.

Invariant 1. A predictable bus arbiter must manage coherence messages and data on the bus such that each core broadcasts a coherence request or communicates data on the bus if and only if it is granted an access slot to the bus

To implement a coherence protocol, a core initiates memory requests by communicating coherence messages with other cores in the system and shared memory. This communication is done over a shared memory bus using an arbiter that manages accesses to the shared bus such that at any time instance it grants access to a single core. A predictable arbiter guarantees that each requesting core is granted the bus eventually in a defined upper-bound amount of time.

Invariant 2. The shared memory services requests to the same line in the order of their arrival to the shared memory

An unpredictable scenario in a cache coherence protocol arises when multiple cores read the same modified cache line, say A. If a core requests to modify A, it has to wait for the owner to write back A to the shared memory. This can lead to repeated broadcasts and unpredictable scenarios for the requester if the updated cache line in shared memory is again modified by another core before the requesting could access it. Invariant 2 requires memory to service requests to the same cache line in their arrival order; thus, it guarantees that a line being requested by a core will not be invalidated before the core accesses it.

Invariant 3. A core responds to coherence requests in the order of their arrival to that core

A second unpredictable scenario arises when multiple cores request different cache lines that are modified by the same core (owner). As a result, the owner has to write back the modified lines requested by the other cores to the shared memory. This can lead to unpredictable scenarios for the requester if the owner core picks to respond to another core's request, stalling the response for the requester core indefinitely. Invariant 3 imposes an order in servicing coherence messages from other cores (write-backs, for example). Note that this design invariant would not be needed for a one-way coherent core as it doesn't respond to coherence messages from other cores.

Invariant 4. Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores

An unpredictable scenario within a core arises due to multiple memory activities from the same core, such as a core's pending request and its response to a request from another core. This response is, for example, a write-back to a line that this core has in a modified state. This leads to indefinite stalling of a core request if it always picks to respond to other core requests. Invariant four states that any predictable arbitration mechanism between coherence requests of a core and responses from the same core is sufficient to address this scenario. For instance, Work-Conserving Round Robin (WCRR) arbitration between a core's request and response can prevent this unpredictable scenario.

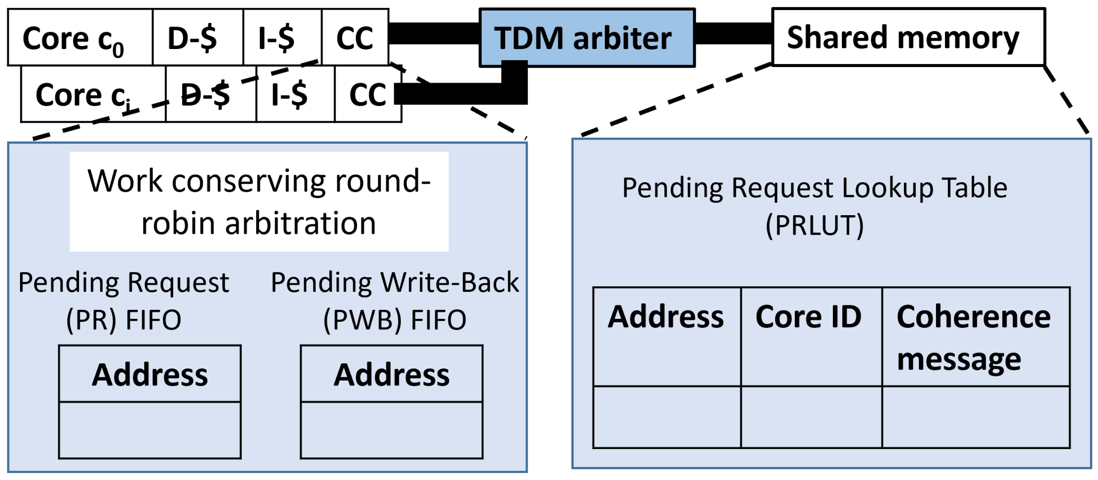

Core c_0
D-\$
I-\$
CC
Core c_i
D-\$
I-\$
CC
TDM arbiter
Shared memory
Work conserving round-robin arbitration
Pending Request (PR) FIFO
Address
Pending Write-Back (PWB) FIFO
Address
Pending Request Lookup Table (PRLUT)
Address
Core ID
Coherence message

Figure 2.3: PMSI: Architectural changes [15]

2.4.2 Predictable MSI

Hassan et al. [15] applied the proposed design invariants to a conventional MSI protocol, resulting in Predictable MSI (PMSI) coherence protocol. To ensure that the invariants discussed in the above section hold, PMSI protocol proposed architectural modifications and additional coherence states. For example, PMSI protocol utilizes a shared command and data bus with Time-Division Multiplexing (TDM) arbitration for interconnecting the private caches of cores and the shared memory. The TDM arbitration grants one slot for each core to access the shared memory. The TDM slot width allows for one data transfer between shared memory and private cache. This satisfies the design Invariant 2.4.1. The shared memory uses a First-In-First-Out (FIFO) arbitration between requests to the same cache line. This arbitration is implemented using a Look-Up Table (LUT) to queue Pending Requests (PR), denoted as PRLUT, as shown in Figure 2.3. Each entry consists of the address of the requested line, the identification of the requesting core, and the coherence message. The PRLUT queues requests by the order of their arrival. When the memory has the updated data of a cache line, it services the oldest pending request for that line. This satisfies Invariant 2.4.1. Each core buffers the pending write back responses in a FIFO queue, which Figure 2.3 denotes as the Pending Write-Back (PWB) FIFO. This modification cooperates with the proposed transient states to satisfy Invariant 2.4.1. Each core deploys a work-conserving TDM arbitration between the PR and PWB FIFOs. This arbitration complies with Invariant 2.4.1.

PMSI's analysis showed that per-request WCL bound scales *quadratically* with the

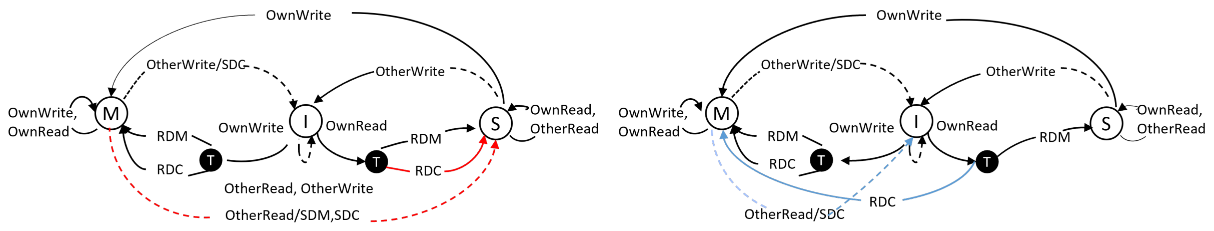


Figure 2.4: Linear cache coherence [19]

number of cores in the system because of coherence activity between all fully coherent cores. PASoC implements a modified PMSI coherence protocol between its agents accessing the shared LLC.

2.4.3 Linear Cache Coherence

PMSI's WCL bound is further improved in *linear coherence* via coherence protocol changes and the use of a point-to-point *Cache-to-Cache* (C2C) data bus to transfer modified data directly between the caches [20]. In the linear coherence protocol described in Figure 2.4, a core receives the requested cache line either from the shared memory (RDM) or from another core (RDC). A core receives the requested cache line from the shared memory only when there does not exist another core that has the same cache line in the M state. Hence, there do not exist any cores that will respond with shared bus accesses in response to the requesting core's memory request. In the worst case, the requesting core waits for its next allocated slot to broadcast its request, and will receive the requested cache line in the same slot. On the other hand, a core receives the requested cache line from another core that has the cache line in M state. However, the transitions in linear cache coherence cause the core that has the cache line in the M state to transition to the I state after sending the cache line data (SDC). Furthermore, a core that receives the requested cache line from another core moves to the M state. Worst-case latency analysis for Linear coherence showed that per-request WCL bound scales *linearly* with the number of cores in the system. PASoC implements a modified linear cache coherence protocol between cores of a fully coherent agent that accesses a shared cache within the agent.

2.4.4 Predictable Sharing of LLC for Multicore System

As the number of cores in an SoC increases, shared LLCs are used in multicore systems to deliver high performances. Cores access shared LLC when they experience misses in their private caches. Modern SoCs group cores into clusters, and all the clusters access the shared LLC [34]. This approach in the system model provides performance and energy benefits. However, multiple cores accessing a shared, inclusive LLC introduces inter-core temporal coherence interference due to back-invalidations. Suppose a core's request to the cache line is a miss in all its private caches and LLC. Then, for the LLC to respond with the provided data, the LLC must ensure: (1) that there is a vacant entry in the set that the cache line maps to, (2) the cache line from main memory is fetched, and (3) the response to the requesting core is sent in its slot. To create a vacant entry in the set that is full, the LLC has to evict a victim cache line. An important property of inclusive caches is that an eviction of a cache line in the lower-level cache requires the eviction of cache lines for the same address in upper-level caches. This is called *Back-Invalidation* (BI). BI's can lead to unpredictable scenarios in a multicore system where one core occupies the vacant cache entry created for another core. *ROC* highlighted this unpredictable behavior due to back-invalidations in multicore systems with inclusive shared LLC and proposed *set sequencer* hardware architecture to enforce ordering constraint on which requests occupy vacant cache line entries [37]. The set sequencer orders requests that are mapped to the same cache set, which prevents a younger request from occupying a vacant entry in the respective caches that were released for an older request, preventing the unbounded WCL.

Chapter 3

Related Work

There exists a range of prior works that integrate accelerators to SoC and discuss different coherence modes while coupling accelerators to CPU cores. There has been extensive research on designing predictable hardware cache coherence for safety-critical systems. PASoC work takes inspiration from these prior works on accelerator coupling and predictable hardware cache coherence in designing a predictable accelerator-rich SoC. This chapter introduces these related works. Section 3.1 starts by introducing the related work done in coherence for heterogeneous platforms. Section 3.2 goes into more detail with past works proposing one-way coherence for accelerators. Finally, Section 3.3 discusses the related work done in designing predictable hardware cache coherence models. Findings from these works inspire and direct the system model as will be discussed in Chapter 4.

3.1 Coherence for Heterogeneous Platforms

There are various standards and commercial solutions that support coherent data sharing among CPU clusters and accelerators. The industries propose various standards and implementations that guide the integration of accelerators.

Industrial Adoption of Heterogeneous Coherence. For example, to make a complete SoC that combines heterogeneous processing elements, ARM's AMBA standard [34] has proposed a flexible cache coherency protocol, AXI Coherence Extension (ACE). ACE works not just for coherence amongst the CPUs but also between GPUs and hardware accelerators. ARM has developed the CCI-400 Cache Coherent Interconnect to support up to two clusters of CPUs and three ACE-Lite I/O coherent masters. The ACE-Lite port is dedicated to hardware accelerators or GPUs to snoop the CPU caches, which reduces the external memory accesses and improves performance.

The CXL standard [31] utilizes the PCIe physical layer to provide coherence across the host processor and the accelerators, supporting both fully coherent and one-way coherent accelerators. CXL uses three protocols allowing for access to shared unified, coherent address space between CPU host and accelerators. CXL also supports accelerators with own memory, which can be coherently shared between the host CPU and other CXL devices. CXL proposed host-bias and device-bias modes for accelerators with memory. An accelerator in device-bias mode can access its own memory directly without host intervention, whereas a request from an accelerator in host-bias mode has to go through the host because the host may have cached copies. This facilitates easier, fine-grained sharing between accelerators without host intervention. Cabrera et al. [6] showed that using CXL caching instead of host memory in a heterogeneous system with GPUs and FPGA results in a 1.31× speedup, while a more tightly-coupled pipelined implementation using CXL-enabled hardware would result in a speedup of 1.45x.

The industrial adoption of accelerator coherence has resulted in concrete platforms such as NVIDIA's Tegra [36] that features I/O coherence between the processor and the GPU, Xilinx's Zynq Ultrascale+ [3] that adopts the AMBA standards providing coherence between ARM cores and the FPGA, and Intel's Agilex [12] that supports coherence between the x86 cores and the FPGA with CXL.

Academic Research on Heterogeneous Coherence. Research projects such as Embedded Scaleable Platform (ESP) [22] integrate accelerators with fully coherent, non-coherent, or one-way coherent modes. Zuckerman et al. [39] proposed Cohmeleon architecture, enabling dynamic coherence mode changes for accelerators in heterogeneous SoCs. Cohmeleon architecture applies reinforcement learning to automatically and adaptively select the optimal cache coherence mode at the time of each accelerator's invocation, which shows speedup by 38% over state-of-the-art design time approaches.

Other works provide a flexible LLC to integrate accelerators, such as GPUs, into fully coherent multicore systems and enable fully coherent data sharing among cores and the accelerators [1] [7]. Alsop et al. [1] directly interfaced devices with diverse coherence properties and memory demands, such as self-invalidation-based reads and write-through stores, enabling each device to communicate in a manner appropriate for its specific access properties and showing performance improvements by 16% over the MESI coherence interface.

Kumar et al. [21] developed a lightweight coherent cache hierarchy for accelerators called Fusion by leveraging temporal coherence to optimize data movement between accelerators and minimize data exchanges with the host LLC. Fusion architecture improves performance by 4.3x compared to DMA, which pushes data into the scratchpad.

Near Data Accelerators. In the context of near-data computing, there has been potential research in integrating Near Data Accelerators (NDAs), which reside off-chip close to main memory and can yield further benefits than on-chip accelerators. Enforcing coherence with the rest of the system becomes more difficult for NDAs. This is because (1) the cost of communication between NDAs and CPUs is high, and (2) NDA applications generate a lot of off-chip data movement. Boroumand et al. [4] proposed a specialized optimistic coherence mechanism called CoNDA for integrating NDAs. This mechanism gathers information on memory accesses and uses this information to avoid performing unnecessary coherence requests. This improves performance by 19.6% over prior coherence mechanisms. Schwedock et al. [29] argue that the hardware-software interface is the problem in the current systems and proposed a polymorphic cache hierarchy architecture called Tako by allowing the software to observe data movement. Tako adds one programmable engine to each tile of the Chip-Level Multiprocessor (CMP) near the L2 cache to execute software callbacks on misses, evictions, and write-backs. These engines have their own L1 coherent data caches. Brana et al. [5] further simplified the integration of cache-attached accelerators by introducing a new directory structure called Miss-Direction Filter (MDF) within the L2 cache of each tile of CMP to track the accelerator's private cache and maintain local coherence between the core and accelerator, leaving the LLC protocol unchanged.

CPU-GPU Coherence. Many of the current heterogeneous platforms integrate GPUs and CPUs on a single chip with unified shared memory. However, throughput-oriented GPUs can overwhelm CPUs with coherence requests not well-filtered by caches. Hence, supporting coherence between CPU clusters and GPUs is a known challenge, and there has been continuous research development to find new ways to integrate CPUs and GPUs in a coherent manner. Power et al. [24] developed a heterogeneous system coherence for CPU-GPU systems by replacing a standard block-level directory with a region directory and adding region buffers to both CPU and GPU L2 caches to track regions over which the CPU or GPU currently holds permission. This approach improved performance by more than 2x over a conventional directory protocol and showed a significant reduction in bandwidth to the directory. GPUs typically employ self-invalidation protocols and write-through caches for their high-throughput applications [33]. Singh et al. [32] proposed a time-based coherence framework for GPUs, called Temporal Coherence, that exploits globally synchronized counters in single-chip systems to develop a streamlined GPU coherence protocol. Synchronized counters enable all coherence transitions, such as invalidation of cache blocks, to happen synchronously, eliminating all coherence traffic and protocol races.

However, maintaining globally synchronized counters and timestamps could be hard. An alternate approach is to use relaxed consistency models such as release consistency or Sequential Consistency for Data-Race-Free (SC for DRF). Hower et al. [18] introduced Sequential Consistency for Heterogeneous-Race-Free (SC for HRF), a class of memory models to intuitively and robustly define the behavior of scoped synchronization operations in heterogeneous systems. However, weaker consistency models require inserting fences or lock acquire/release correctly and can lead to bugs that are very difficult to debug. Hence, Ren et al. [26] proposed a SC-based GPU coherence protocol called Relativistic Cache Coherence (RCC) that can perform on par with non-SC designs. RCC uses logical timestamps to determine a global memory order and maintain SC ordering.

Overall, these approaches show that having a unified shared coherent memory between accelerators and host improves performance. However, they do not consider predictable hardware cache coherence mechanisms, which are essential for safety-critical systems. Consequently, these platforms do not strive to guarantee that the per-request WCL of a memory request is bounded. Hence, we present a model of a PASoC, which guarantees per-request WCL bound for any shared cache-coherent access in the system by integrating fully coherent and one-way coherent accelerators to CPU cluster. Literature proposed other interactions between host and accelerators, such as NDAs, self-invalidation-based protocols, and logical timestamp-based coherence. To our knowledge, there is no work that explores the predictability of such interactions with host. Analyzing the worst-case execution time for such interactions between accelerators and CPUs remains a future work for PASoC.

3.2 One-way Coherence

One-way coherence, also referred to as I/O coherence or coherent DMA, is an approach for integrating an accelerator to a SoC [39]. In one-way coherence, the accelerator can snoop on the private caches of other agents but does not respond to any coherence activity by other agents in the system. Zuckerman et al. [39] showed that one-way coherent mode achieves better performance execution time with fewer off-chip memory accesses for irregular access patterns. AMBA standard defines ACE-Lite I/O coherency in support for one-way coherent accelerators [34]. ACE-Lite enables uncached masters to snoop ACE coherent masters. This can enable interfaces such as Gigabit Ethernet to directly read and write cached data shared within the CPU. The ARM Mali-T604 GPU supports ACE-Lite I/O coherency, which provides system performance benefits by eliminating cache-flushing and off-chip memory accesses when integrating accelerators to SoCs. Industry standard for heterogeneous coherence such as CXL [31] also supports one-way coherence modes for accelerators in the device-bias mode where accelerators can snoop the host CPU caches but not allow the host to access the accelerator's memory. Hence, PASoC takes inspiration from these prior works, which showed the benefits of integrating one-way coherent accelerators into the SoC and guarantees that the per-request worst-case latency of a memory access is bounded.

3.3 Predictable Cache Coherence

Hardware Cache Coherence (HCC) mechanisms deploy a set of rules to ensure that cores access the correct version of data at all times. A predictable HCC mechanism ensures the predictability of a memory request with a defined WCL bound. Prior works on designing predictable HCC have certain requirements. First, the implementations should honor certain invariants. Second, several architectural and coherence protocol changes must be made in the system to ensure the predictability of a memory request and also to improve its WCL bound [15] [37] [20].

PMSI. Hassan et al. [15] proposed a predictable cache coherence protocol called *PMSI* for multicore real-time systems. *PMSI* coherence protocol utilizes a shared command and data bus with Time-Division Multiplexing (TDM) arbitration for interconnecting the private caches of cores and the shared memory. The TDM slot width allows for one data transfer between shared memory and private cache. With this system model, PMSI protocol identified sources of unpredictability with a conventional MSI coherence protocol and proposed minimal architectural changes like adding FIFO arbitration between requests to the same cache line in the shared memory and between write-back responses at every core. The analysis showed that per-request WCL bound scales *quadratically* with the number of cores in the system because of coherence activity between all fully coherent cores.

Linear Coherence. PMSI protocol's WCL bound is further tightened in *linear coherence* [20] via coherence protocol changes and the use of a *Cache-to-Cache* (C2C) data bus to transfer modified data directly between the caches. The WCL bound is linear with respect to number of cores. Neither of these proposals supports an LLC in the system. Another approach, *PISCOT* [17], separated the request bus and the response bus with different arbitration schemes to achieve improvement of performance and achieving a WCL that is linear with respect to the number of cores.

ROC. A recent work, *ROC* [37] highlighted the unpredictable behavior due to back-invalidations in multicore systems with inclusive shared LLC. Similar to prior works, every core is allocated one TDM slot to access the shared LLC. An unpredictable scenario occurs when a victim cache entry in a cache set that is full is occupied by a request from another core before the data from the actual request is filled. ROC deployed a *set sequencer* hardware architecture to prevent this unbounded WCL by enforcing ordering constraints on which requests occupy vacant cache line entries in a cache set. The WCL bound scales *cubically* with the number of cores in the system because of back-invalidations.

Resource or Task Scheduling. Hansson et al. [14] proposed a predictable SoC platform called CoMPSoC that removed all interference between applications through resource reservations. However, CoMPSoC does not include caches, shared caches, and coherence. Forsberg et al. [11] proposed a solution that separates memory and computation phases in real-time codes, then arbitrates memory phases from different tasks such that only one core at a time can access the DRAM. Such solutions allow only one core to access a cache line of shared data at a time. Other cores requesting this data have to stall. In the worst case, this is equivalent to sequential execution. On the other hand, PASoC allows tasks to simultaneously access shared data, which considerably improves performance. In addition, it does not pose any requirements on task scheduling techniques, and it does not require software modifications.

Overall, these predictable HCC mechanisms are designed for CPU multicores with fully coherent data sharing. Further, none of these works studies the predictability of HCC mechanisms when integrating hardware accelerators with different coherence modes with a predictable multicore system, which is what PASoC aims to do in this work.

Chapter 4

System Model

This chapter presents the system model for PASoC. It outlines the architectural modifications required to design a predictable system inspired by prior works described in Chapter 2 and applies them to the PASoC system model.

4.1 PASoC Setup

This section employs Figure 4.2 through 4.5b to guide the system model. This work assumes PASoC to have N_A agents that share an LLC (①), which is connected to the main memory. An agent can be one of the following: (1) a cluster of homogeneous predictable cores [13] (CPU cluster a_1 ②); (2) a fixed-function one-way coherent accelerator (a_2 ③) ; or, (3) a fixed-function fully coherent accelerator (a_3 ④). A CPU cluster and a fully coherent accelerator are both Fully Coherent Agents (FCAs) and a one-way coherent accelerator is a One-way Coherent Agent (OCA). Accelerators have only one processing element in it, whereas a CPU cluster has multiple predictable cores. For brevity, this work assumes that PASoC only supports one CPU cluster and $N_A - 1$ accelerators, and every agent can make only one outstanding memory request to the LLC. Every agent accesses the shared LLC using two buses: the shared command bus and the shared data bus. The access via buses to shared LLC is arbitrated using Time Division Multiplexing (TDM).

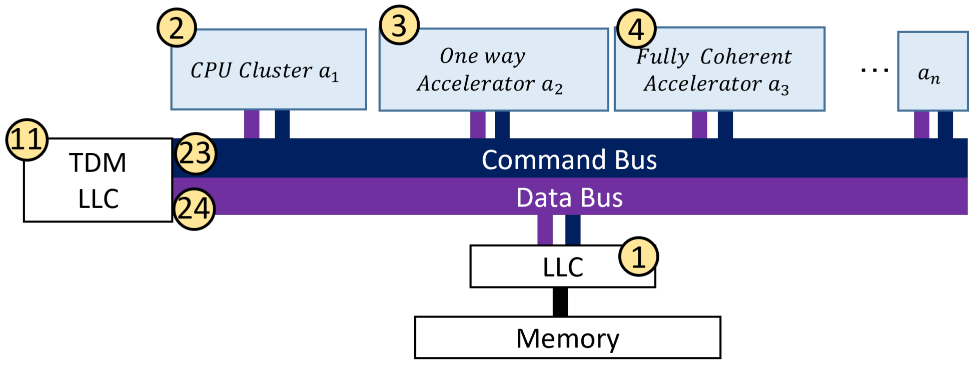


Figure 4.1: Block diagram of PASoC

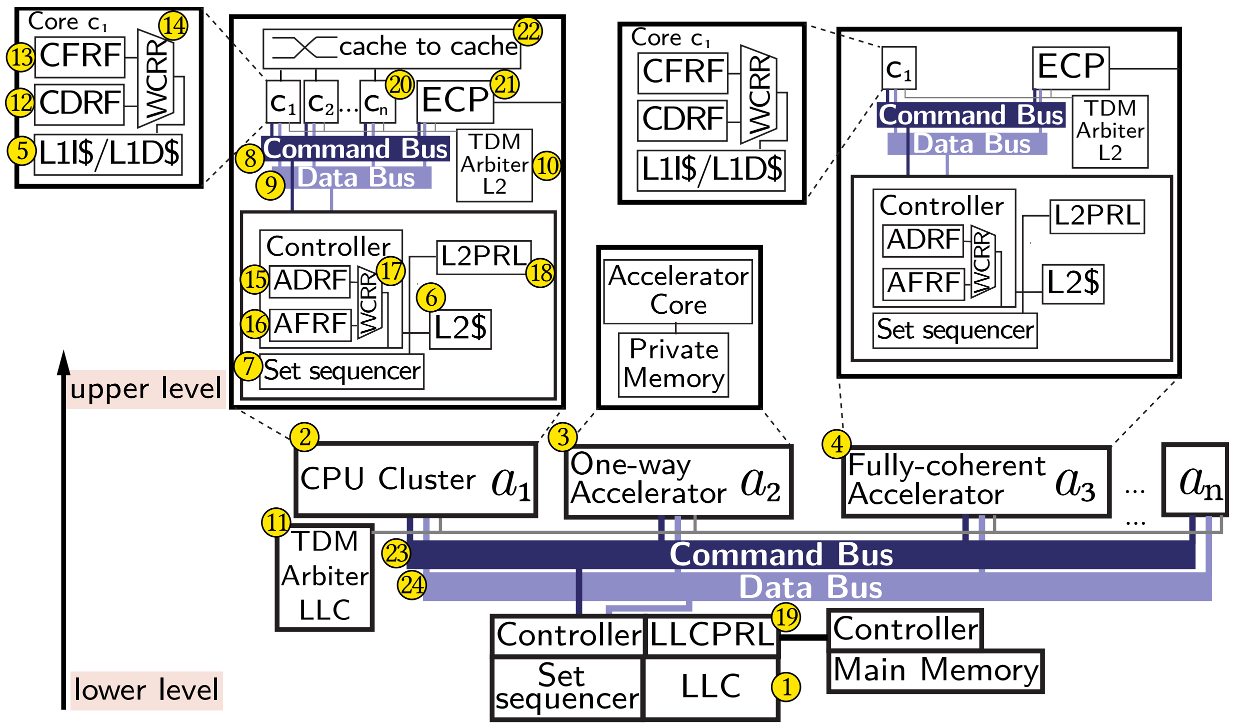


Figure 4.2: Overall view of PASoC system model

4.2 Coherent Accelerator Configurations

PASoC supports *fully coherent* and *one-way coherent* agents, as will be described later in this section. PASoC does not include non-coherent accelerators, where the interaction between such accelerators and other coherent agents requires cache flushes for data sharing. This approach is good for streaming large accelerator workloads that trigger more memory accesses. However, as accelerators become more tightly coupled and heterogeneous applications become more capable and diverse, the non-coherent mode does not generalize efficiently to a broader range of access patterns. Hence, PASoC emphasizes studying and analyzing only the interactions between coherent accelerators and CPU clusters on an SoC.

4.3 Cache Hierarchy

All agents in the PASoC share a *LLC*. A FCA consists of a two-level inclusive cache hierarchy. Each core in FCA has a private *L1* (5) instruction and data cache that is connected to a shared *L2* (6) cache. An OCA does not have a private cache. PASoC assumes all caches in the system are set-associative write-back caches with a write-allocate policy and a Least-Recently-Used (LRU) replacement policy.

4.4 Cache Inclusion

PASoC assumes that LLC is inclusive of all L2 caches and L2 is inclusive of all L1 caches. Suppose that a core's request to the cache line is a miss in all its private caches, L2, and the LLC. Then, for the LLC to respond with the provided data, the LLC must ensure (1) that there is a vacant entry in the set that the cache line maps to, (2) the cache line from main memory is fetched, and (3) the response to the requesting core is sent in its slot. To create a vacant entry in the set that is full, the LLC has to evict a victim cache line. An important property of inclusive caches is that an eviction of a cache line in the lower-level cache requires the eviction of cache lines for the same address in upper-level caches. This is called *Back-Invalidation* (BI). For this setup, an eviction in LLC forces BIs in both the L1 and L2 private caches of the agents that have the data, whereas an eviction in L2 forces BIs in L1 private cache that have the data.

4.5 Shared-Memory Bus Interconnect

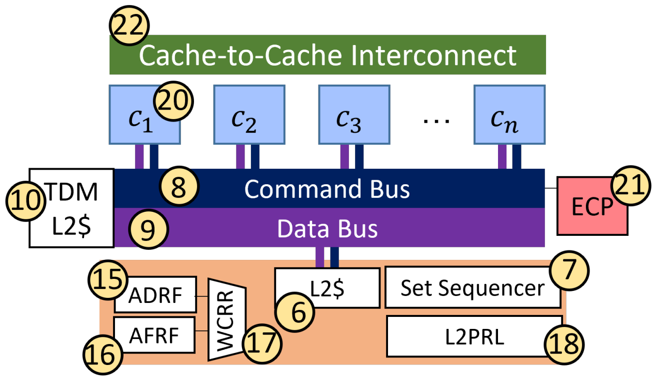
Each requester in PASoC communicates with shared memory (LLC or L2) via two shared buses: one for broadcasting commands *(command bus)* (⑧ and ㉓) and one for transferring data *(data bus)* (⑨ and ㉔). The shared cache controller accepts requests placed on the command bus and coordinates the necessary actions to complete the request in their broadcast order (the order received by the cache controller). The shared cache responds with the data by placing the data on the data bus. PASoC uses work conserving TDM arbitration [16] TDM_{LLC} (⑩) and TDM_{L2} (⑪) to arbitrate accesses on the shared command bus to LLC and L2, respectively. Each requester is allocated one slot to access the shared cache. A TDM slot is long enough to complete one data transfer between the requester and the shared memory and to snoop other coherent caches.

4.6 Coherence Protocols

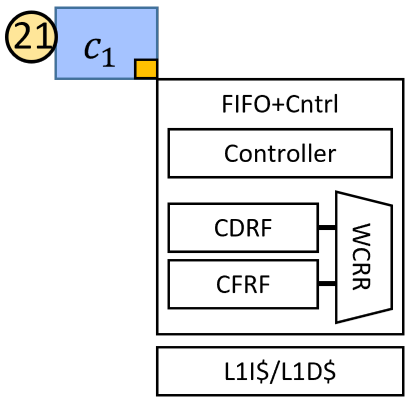
This work uses state-of-the-art predictable coherence protocols with modifications to construct PASoC. Within the CPU cluster, PASoC uses a predictable cache coherence protocol called linear coherence [20], and between the agents, PASoC uses PMSI [15]. The main modifications in PMSI require the OCA to force write-through stores and ignore any coherence activity on the shared command bus. The main modifications in Linear coherence require supporting requests from an External Coherency Port (ECP) that broadcasts commands from other agents to the FCA. Linear coherence allows the transfer of modified data between cores within an agent. Unlike the requests from L1 cores, the requests from an ECP allow for data transfer from cores only via the L2 cache using the shared data bus between cores and the L2 cache. The L2 cache serves as an intermediate cache between the core's private L1 cache and shared LLC, honoring requests from internal to agent as well as external to agents. Hence, the L2 cache supports a mix of both linear coherence and PMSI.

4.7 Fully Coherent Agent (FCA)

An FCA consists of n_c homogeneous cache-coherent predictable cores [20, 13] (⑳) and a two-level inclusive cache hierarchy as shown in Figure 4.3a. Each core has a private L1 instruction and data cache. These private caches are connected to an L2 cache. An *External Coherency Port (ECP)* (㉑) allows it to receive commands from agents external to



(a) Block diagram of FCA



(b) Block diagram of a core in FCA

Figure 4.3: Components of fully coherent agent

the FCA. This allows other agents to access data from within the FCA, allowing its private caches to be snooped by other agents. In the FCA, the communication among cores' caches, ECP, and L2 happens using two shared busses and a direct *C2C* (22) data bus. The shared command and data bus connects the L1s and ECP to L2. The ECP is only used to relay broadcast commands from other agents into the FCA. There is a separate point-to-point interconnect to allow for direct cache-to-cache data transfer between every core in the FCA [38]. The direct C2C data bus only connects the L1s [34] [20]. This cache-to-cache data bus allows the transfer of modified data between the L1s. PASoC employs a work-conserving Time-Division Multiplexing (TDM) bus arbitration TDM_{L2} [16] to arbitrate accesses to the L2. PASoC allocates $n_c + 1$ slots to $TDM_{L2}:$ one for each core and one for the ECP.

A special case of an FCA is when $n_c = 1$ where there is only one processing element, yet it is fully coherent with other agents. Note that there is no C2C data bus in this case. We assume a fully coherent accelerator is an FCA with one coherent core $n_c = 1$ and a CPU cluster is an FCA with $n_c \geq 1$ We further assume that there can only be one pending memory request from a core or external agent to L2.

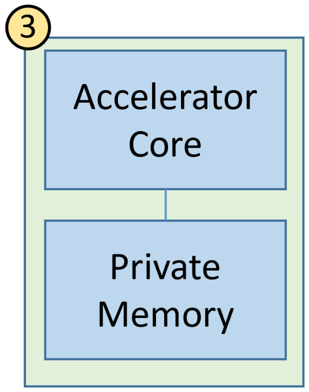


Figure 4.4: One-way coherent agent

4.8 One-way Coherent Agent (OCA)

Similar to [39], PASoC supports a fixed-function one-way coherent accelerator that does not have a private cache but can have a private memory, as shown in Figure 4.4. OCA enables producer-consumer interaction between a CPU cluster and accelerator, where the CPU cluster is the producer and the accelerator is the consumer. One-way coherence allows a simplified cache memory hierarchy for handling coherence traffic for accelerators, giving performance and energy benefits [34]. A request from an OCA can cause coherence activity for other agents; however, the OCA does not respond to any coherence activity itself. This means that the OCA can retrieve up-to-date data from the LLC for reads and by updating the LLC for writes. However, other agents cannot obtain data within the OCA. A request issued by the OCA is placed on the command bus to the LLC's controller. The LLC controller coordinates the necessary actions to complete the request. When the accelerator issues a write request to a cache line also cached by other FCAs, the LLC controller orchestrates the invalidation of all other private copies and updates its local copy before responding to the one-way coherent accelerator [34].

4.9 Ordering Hardware Components

In order to ensure the predictability of the system, PASoC employs certain hardware components for ordering the requests or responses at every controller, as discussed below in this section.

4.9.1 FIFOs

PASoC uses two FIFOs to buffer incoming messages at each L1 and L2 cache controller: a Demand Request FIFO (DRF) and a Forward Response FIFO (FRF), as shown in Figure 4.5a. A DRF buffer's incoming read, write, and write-back requests originating from a core or agent to L2 or LLC, respectively. An FRF buffer's the write-back responses that the core or agent sends to L2 or LLC, respectively. These write-back responses from the L1 cache controller are to the requests forwarded from other cores within the cluster, from the L2 cache during back-invalidation, or from other external agents in the PASoC. The write-back responses from the L2 cache controller are only to the requests that are forwarded from external agents. A predictable arbitration such as Work-Conserving Round-Robin (WCRR) (⑭) between DRF and FRF chooses from a request or a write-back response to send on the bus at the beginning of the TDM slot [15]. Note that an OCA does not have a FRF because it does not respond to any coherence activity in the system. The DRFs, FRFs, and WCRR arbitration between them complies with Design Invariant 2.4.1 and 2.4.1.

Input of L1 Cache Controller

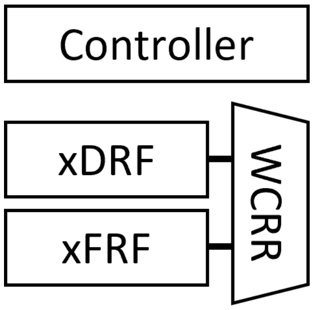
The DRF and FRF are named as *($CDRF$)* (⑫) and *($CFRF$)* (⑬), respectively. The maximum size of $CDRF$ is 1, and $CFRF$ is $n_c + N_A$.

Input of L2 Cache Controller

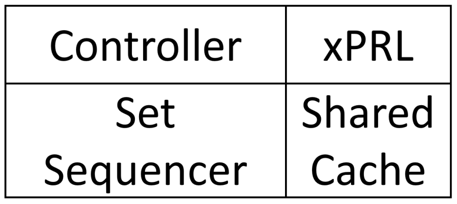
The DRF and FRF are termed *($ADRF$)* (⑮) and *($AFRF$)* (⑯), respectively. The $ADRF$ is of size $2n_c$, because, at worst, every demand request requires a write-back of a victim cache line to L2, which is also buffered in $ADRF$. $AFRF$ is of size N_A.

4.9.2 PRLUTs

At the L2 within the FCA and the LLC, we use a FIFO arbitration between pending requests to the same cache line, as shown in Figure 4.5b. PASoC uses a look-up table to queue pending requests *(PRLUT)* [15]. The PRLUT queues requests in the order of their arrival at the corresponding cache. The PRLUT at the L2 is called $L2PRL$ (⑱), and at LLC, $LLCPRL$ (⑲), respectively. The PRLUT complies with Design Invariant2.4.1.



(a) Block diagram of FIFOs at cache controllers L1 and L2



(b) Block diagram of PRL and Set Sequencer at shared cache L2 and LLC

Figure 4.5: Ordering components

4.9.3 Set Sequencers

PASoC employs the approach in [37] to enforce the Request Ordering Constraint (ROC) by deploying a hardware architecture named *set sequencer* (⑦) in each of the L2 caches and the LLC as shown in Figure 4.5b. The set sequencer orders requests that are mapped to the same cache set, which prevents a younger request from occupying a vacant entry in the respective caches that were released for an older request. This eliminates the unbounded WCL scenario triggered due to back invalidations.

4.10 Cache Operations in PASoC

This section explains the possible cache operations between L1, L2, and LLC in PASoC with an example, as shown in Figure 4.6.

4.10.1 Coherence Example

- A core requests data from memory using a *Load*, writes data to memory using a *Store* and writes back using *PutS* (clean copy of data) or *PutM* (dirty copy of data) to relinquish its private copy of victim cache line. These incoming demand requests access the L1 cache first (①).
- If the request is a hit in the L1 cache, then the data is returned to the core for a Load and written to the L1 cache for a Store (②).

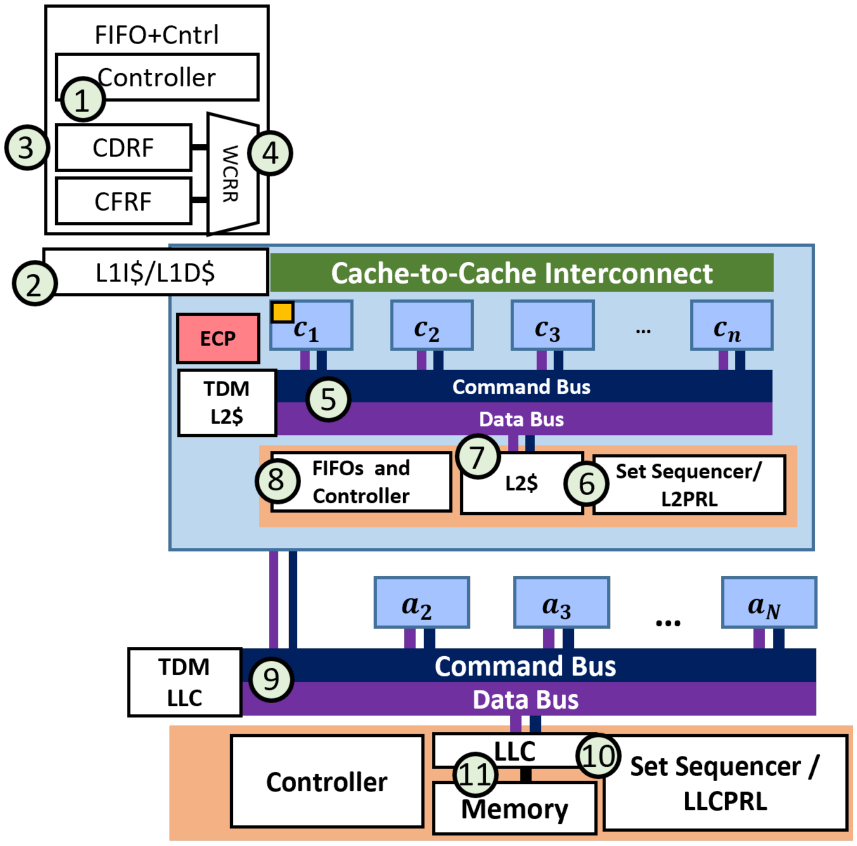


Figure 4.6: PASoC coherence operations example

- If these requests are a miss in L1 cache then they are buffered in $CDRF$ and forwarded to L2 cache (③).
- The core experiences WCRR arbitration between its own demand requests in $CFRF$ and pending write-back responses in $CFRF$ to be picked to submit to L2 cache (④).
- The core's request is placed on a shared command bus to submit to the L2 cache in its TDM slot (⑤).
- At the L2 cache, the set sequencer or $L2PRL$ orders requests if necessary to ensure bounded WCL. When the L2 cache is ready to process the request, it orchestrates the necessary communication between cores and sends the requested cache line to the core while maintaining the coherence of data between all cores (⑥).
- If the request is a miss in the L2 cache, then L2 triggers back invalidation to replace a victim cache line with the requested cache line fetched from LLC (⑦).
- The request is buffered in $ADRF$ at the L2 cache controller to forward to LLC (⑧).
- Once again, the request undergoes WCRR arbitration between $ADRF$ and $AFRF$ to be picked to submit to LLC and issues the request to LLC in its TDM slot on the shared command bus to LLC (⑨).
- Upon receiving a demand request from the agent, similar to L2 cache, $LLCPRL$ and set sequencer at LLC order requests to avoid any unbounded WCL. If the request is a hit in LLC, then LLC performs the necessary coherence activity and sends the requested cache line to the agent (⑩).
- If it is a miss in LLC, then LLC triggers back invalidation to replace a victim cache line with the requested cache line fetched from main memory and sends the data to the agent (⑪).

Chapter 5

Implementation

Chapter 4 detailed the system model of PASoC, explaining different hardware architectural components and their usage in designing a predictable system. In order to support different coherence modes and ensure predictability from agents, PASoC proposed some of the architectural changes in Chapter 4. It also gave a brief description of the coherence protocol that PASoC implements in order to maintain a coherent view of the system memory in a predictable manner. The purpose of this chapter is to discuss these implementation details when integrating hardware accelerators with different cache coherence modes with a predictable multicore system.

5.1 Limitations of Prior Work

The prior works on heterogeneous system coherence provide a solid foundation for integrating accelerators to multicore SoC. However, these approaches do not consider predictable hardware cache coherence mechanisms, which are essential for safety-critical systems. Consequently, these platforms do not strive to guarantee that the per-request WCL of a memory request is bounded. Some of the prior works that studied the predictable hardware cache coherence mechanisms did not consider integrating accelerators into multicore SoCs. Hence, there are two key limitations of prior works that PASoC addresses in this work.

1. **Integrating accelerators with different coherence modes to predictable multicore platforms**. Accelerators are heterogeneous in nature, and this diversity is reflected in their communication properties like memory access patterns and

interaction with system memory hierarchy. Integrating these accelerators to predictable multicore CPUs can affect the WCL or overall predictability of the system. As a result, it is necessary to study these interactions between CPUs and accelerators to design a predictable ASoC for safety-critical systems.

2. **There is no defined WCL analysis for a heterogeneous system**. WCL analysis is required to ensure the predictability of a system and guarantees that the latency of all possible memory requests in the system is bounded. Simultaneous access to shared hardware resources such as shared memory buses shared caches, and shared main memory in a system can impact the timing behavior and WCL bound. As a result, computing the WCL of a memory request is required for heterogeneous systems to be used safely in safety-critical systems.

   This chapter addresses the first limitation by proposing architectural and coherence protocol modifications for integrating accelerators to predictable multicore CPU platforms.

5.2 Architectural Modifications

This section introduces all the required architectural modifications for PASoC, specifically to support integrating CPU cluster and accelerators with different coherence modes over an SoC.

5.2.1 External Coherency port

The first architectural modification is introducing an External Coherency Port (ECP). As described in Section 4.7, ECP is used in a fully coherent agent to broadcast messages from other agents to this FCA. Effectively, in an FCA, the ECP acts like an additional core without any cache hierarchy that joins the FCA with other agents in the system. Hence, similar to other cores in the FCA, an additional TDM_{L2} slot is granted for ECP to issue its requests to the shared command bus in FCA. For example, if the number of cores in FCA is n, the TDM_{L2} arbiter allocates $n+1$ slots for each of the core and ECP to access the shared command bus.

ECP acts as a translator of coherence messages from external agents to commands that FCA understands. This helps in integrating cores with different coherence modes in a heterogeneous system. For example, OCA stores are write-through, unlike the ownership-based write-back stores of a CPU core. When an OCA broadcasts a store request to a cache line, all other cached copies of that cache line have to be invalidated in all other cores without any write-backs of modified data. To support this behavior, ECP translates the write-through request from OCA to an invalidation request, which invalidates the cache line without any write-back to the lower level cache.

Recall from Section 4.7 that there exists a C2C data bus between all L1 caches in FCA. This internal cache-to-cache data transfer improves performance. However, the data transfer between L1 and ECP is not direct and only happens through the L2 cache. That is, if a request from ECP requires the transfer of modified data from the L1 cache, then that data has to be first transferred to the L2 cache via a shared data bus that connects the L1 and L2 caches. L2, in-turn, transfers the data to the shared LLC, which forwards it to the requesting agent.

5.2.2 Support for One-way Coherence

As described in Section 4.8, PASoC supports accelerator with one-way coherence mode. A one-way coherent accelerator does not have a private cache and issues its request to get the most up-to-date from LLC directly. This triggers coherence activity in other cores, but the OCA does not respond to any coherence activity that happens in the system. The stores from OCA are write-through stores, which update the LLC directly, invalidating all other cached copies. PASoC introduces architectural changes to support this coherence mode.

In order to not respond to any coherence activity in the system, the shared command bus blocks from sending any coherence requests to the OCA. However, the OCA itself can issue commands over the shared command bus in its allocated TDM_{LLC} arbitration slot.

OCA read requests are like any other read requests from an FCA core. It gets the most up-to-date data from either the LLC or other agents or from main memory. However, the read data is not privately cached in OCA, and has to issue the request to LLC to read the data again. An OCA read will not impact the ownership of a modified cache line privately cached in an FCA core. That is, unlike an FCA read request to a modified cache line, which changes the access permission of the cache line to *read-only* or *Invalid*, the OCA read only gets the updated data from owner core and does not take away ownership permissions from the owner.

OCA stores are write-through stores, which means all other cached copies have to be invalidated, and the dirty data need not be written back. OCA updates the data in LLC directly. Hence, a write-through request from an OCA translates to an Invalidation request to all other cores. Similar to an OCA read, the OCA store is also not privately cached and has to issue the store request to LLC to update the data again. An OCA store will set the access permission of the cache line to *Valid* in LLC, meaning the subsequent requests to this cache line are a hit in LLC and do not generate any coherence activity in the system.

Effectively, OCA can be considered as a core with a private cache where the cache lines are always in *Invalid* state, requiring the core to always issue commands to LLC to get the updated data. Since the cache lines are always in an Invalid state, this triggers no replacement or back-invalidations of cache lines.

5.2.3 Intermediate L2 Cache

As described in Section 4.7, an FCA agent has an intermediate L2 cache, which is shared by all cores within FCA. The L2 cache controller acts as a local, low-latency shared cache inclusive of all L1 caches in the FCA. Similar to prior works [15] [37], the L2 cache uses PRLUT to order requests to the same cache line and set sequencer to order requests targeting the same cache sets to ensure bounded WCL.

The L2 cache controller filters out the unnecessary memory requests internal to FCA from being issued to external agents and vice-versa. That is, if a memory request from an FCA core is a hit in the cluster, then the L2 cache will not issue the request to LLC. In the same way, any request from external agents via ECP that does not require broadcast to FCA cores in the agent is processed by the L2 cache itself. On the other hand, if any requests to the L2 cache from an FCA core are a miss, then L2 forwards that request to LLC. In this case, L2 acts as a requester that submits a request to a shared LLC.

As described in Section 4.9.1, L2 buffers the demand requests to LLC in its ADRF, and there can only be one pending request to LLC from an agent. L2 cache also merges multiple requests to the same cache line and enqueues only one single request for that cache line in ADRF to be submitted to LLC. When L2 receives the response from LLC, it sends the response to all cores that requested the cache line within the agent.

5.3 Predictable Cache Coherence Protocol for PASoC: PMSI-Mix

Having described the architectural modifications for PASoC, this section describes the cache coherence protocol employed in the system, *PMSI-Mix*, which supports full coherence and one-way coherence modes.

5.3.1 Memory Operations

A core in FCA can issue *read*, *write* or *replacement* requests to memory. Since OCA does not have a private cache, it issues read and write requests only. If a core issues a read request, its cache controller generates $GetS()$ message. If a core issues a write request, its cache controller generates $GetM()$ message. If a core wants to write to a shared cache line in its private cache, its cache controller generates $Upg()$ message. If a core wants to evict a cache line that has been modified, the cache controller generates a $PutM()$ message and then writes back the updated data contents to shared memory. Consequently, an OCA cache controller generates $GetS()$ for read and $ReqWT()$ for writes.

5.3.2 Cache Coherence Protocol State Machine Modifications

This section discusses the protocol modifications to the PMSI [15] and Linear Coherence protocols [20] that work in tandem with the architectural modifications described earlier. The protocol modifications result in a new coherence protocol: PMSI-Mix. Table 5.2 shows the private caches' coherence states for a cache line and the transitions between these states for the PMSI-Mix protocol. Tables 5.3 and 5.4 show the shared memory coherence states and transitions for L2 cache and LLC, respectively. This section also introduces new transient coherence states, stable coherence states, and bus events for the PMSI-Mix protocol. Table 5.1 describes the new states added to the PMSI-Mix protocol.

New Stable Coherence States

PMSI-Mix introduces new stable coherence states (s-states) P and V for L2 cache and LLC, respectively. The s-state P in L2 cache coneys that a cache line is privately shared within an agent, and no other agent has this cache line in its private or shared caches. Hence, an upgrade to this cache line by a core in an agent need not broadcast the request externally to an agent and can be handled privately by the agent. The s-state V in LLC conveys that a cache line is valid in LLC and is not shared by any agents in the system. For example, the cache line in LLC after a write-though request from an OCA change to V state as shown in Table 5.4. This is because the cache line is invalidated in all other agents, and OCA updates the data in LLC directly.

New Transient States

PMSI-Mix introduces new transient states (t-states) as shown in Table 5.1. Transient states like MI^{dwb} and MS^{dwb} convey the mixed behavior of intermediate L2 cache as a shared memory within an agent and a private cache with respect to LLC. These states indicate that L2 is waiting for modified data from the L1 cache. Once the data is available, the L2 cache writes back the data to LLC in its next available slot. PMSI-Mix also introduces new t-states for the proposed s-states P and V, such as MP^{d}, P^{wb}, V^{D}. Note that previous works such as PMSI [15] or linear coherence [20] had no LLC. Hence, there were no states indicating the transitions between LLC and main memory. Hence, PMSI-Mix proposed new t-states such as V^{m}, S^{m}, M^{m}, and I^{m} as shown in Tables 5.4 and 5.1.

New Bus Events

PMSI-Mix introduces new bus events for L1, L2, and LLC, highlighted in red in Tables 5.2 through 5.4. The new bus events for the L1 cache, such as *ExtGetS*, *ExtGetM* are introduced to support requests from external coherency port, for which the data transfer happens via L2 cache instead of direct cache-to-cache. To support *ReqWT* message from OCA, new bus events such as *Inv*, *OtherReqWT*, and *ReqWT* are introduced for L1, L2, and LLC, respectively.

For Tables 5.1 through 5.4, changes to conventional PMSI [15] protocol are highlighted in red. Shaded cells represent transitions that are not possible under correct operation. Cells marked with "-" represent situations where no transition occurs, and the coherence state remains unchanged.

Initial state	**Transient state**	**Final state**	**Description**
M	MI^{dwb}	MI^{wb}	L2 cache has cache line A in M state. Another agent has requested A to modify. MI^{dwb} is necessary to reflect that L2 is waiting for modified data A from L1 cache to write-back to LLC in its next slot.
M	MS^{dwb}	MS^{wb}	L2 cache has cache line A in M state. Another agent has requested A to read. MI^{dwb} is necessary to reflect that L2 is waiting for modified data A from L1 cache to write-back to LLC in its next slot.
M	II^{dwb}	II^{wb}	L2 cache has cache line A in M state. The cache line is selected for replacement. II^{dwb} is necessary to reflect that L2 cache is waiting for modified data from owner core. Once the cache line is available L2 writes-back the cache line A to LLC.
M	MP^{d}	P	L2 cache has cache line A in M state. Owner L1 cache issued PutM to write-back A. MP^{d} is necessary to reflect that L2 cache is waiting for modified data to be written-back by L1 cache to L2 cache.
S/M/V	V^{D}	V	LLC has cache line A in S/M/V state. OCA agent has requested A to write-through. V^{D} is necessary to reflect that LLC is waiting for data to be written by OCA.
I	$S^{m}/M^{m}/V^{m}$	S/M/V	LLC has cache line A in I state. An agent issued GetS/GetM/ReqWT to LLC. $S^{m}/M^{m}/V^{m}$ is necessary to reflect that LLC is waiting for data from memory.
M	II^{Dm}	I^{m}	LLC has cache line A in M state. The cache line is selected for replacement. II^{Dm} is necessary to reflect that LLC is waiting for modified data from owner core. Once the cache line is available LLC writes-back the cache line A to memory.
V	I^{m}	I	LLC has cache line A which is dirty in V state. The cache line is selected for replacement. I^{m} is necessary to reflect that LLC has written-back the cache line to memory and is waiting for acknowledgement from memory to transition to I.

Table 5.1: Semantics of proposed t-states and s-states for PMSI-Mix protocol for L2 cache and LLC

State	Core Events			Bus Events							
	Read	Write	Replac-ement	Upg Ack	Own Data	Own PutM	Other GetS	Ext GetS	Other GetM	Ext GetM	Inv
I	Issue GetS/ IS^D	Issue GetM/ IM^D	▨	▨	▨	▨	-	-	-	-	-
S	Hit	Issue Upg/ SM^A	I	▨	▨	▨	-	-	I	I	I
M	Hit	Hit	Issue PutM/ MI^A	▨	▨	▨	Send data to requester/I	Send data to L2/S	Send data to requester/I	Send data to L2/I	I
IS^D	▨	▨	▨	▨	If L2 cache is data source, then complete read/S, else complete read/M	▨	-	-	IS^DI	IS^DI	IS^DI
IM^D	▨	▨	▨	▨	Complete write/M	▨	IM^DI	IM^DS	IM^DI	IM^DI	IM^DI
SM^A	Hit	▨	Stall	Complete write/M	▨	▨	▨	▨	Re-issue write/I	Re-issue write/I	Re-issue write/I
MI^A	Hit	Hit	▨	▨	▨	Write back data/I	-	-	-	-	-
IM^DI	▨	▨	▨	▨	Complete write, Send data/I	▨	-	-	-	-	-
IM^DS	▨	▨	▨	▨	Complete write, Send data to L2/S	▨	-	-	IM^DI	IM^DI	-
IS^DI	▨	▨	▨	▨	Complete read /I	▨	-	-	-	-	-

Table 5.2: L1 private cache states for PMSI-Mix. Issue msg/state means the core issues the message 'msg' and moves to state 'state'. A core issues a read/write request. Once the cache line is available, the core reads/writes it. A core needs to issue a replacement to write back a dirty block before eviction. Changes to conventional PMSI are highlighted in red. Shaded cells represent transitions that are not possible under correct operation. Cells marked with "-" represent situations where no transition occurs, and the coherence state remains unchanged.

State	L1 Core Events					LLC Bus Events					
	GetS	**GetM**	**PutM**	**Data from L1**	**Replacement L2**	**Own Data**	**Own Upg**	**Own PutM**	**Other GetS**	**Other GetM/ Upg**	**Other Req WT**
I	Issue GetS/ IS^{D}	Issue GetM/ IM^{D}							-	-	-
S	Hit, Send data	Hit, Issue Upg/ SM^{A}			Broadcast GetM/I				-	Broadcast GetM/I	Brodcast Inv/I
P	Hit Send data	Hit, Up-date Owner/ M			Broadcast GetM/PI^{wb}				PS^{wb}	Broadcast GetM/ PI^{wb}	Broadcast Inv/ I
M	Hit, Clear owner/ S	Hit, Up-date owner/ M	Clear owner/ MP^{d}		FwdGetM/ II^{dwb}				Fwd GetS/ MS^{dwb}	Fwd GetM/ MI^{dwb}	FwdInv to Owner/ I
IS^{D}	Stall	Stall	Stall			Fwd data to L1/S			Stall	Stall	Stall
IM^{D}	Stall	Stall	Stall			Fwd data to L1/M			Stall	Stall	Stall
SM^{A}					Stall		Fwd Upg Ack/M		Broadcast Inv/I	Broadcast Inv/I	Broadcast Inv/I
PI^{wb}/ MI^{wb}	Hit	Hit			-			Write back data/ I	Stall	Stall	Stall
PS^{wb}/ MS^{wb}	Hit	Hit			Stall			Write back data/ S	Stall	Stall	Stall
MP^{d}	Stall	Stall	Stall	Copy data/P	MI^{dwb}				MS^{dwb}	MI^{dwb}	II^{d}
MI^{dwb}				Copy data,Issue PutM/MI^{wb}	-				Stall	Stall	Stall
MS^{dwb}				Copy data,Issue PutM/MS^{wb}	MI^{dwb}				Stall	Stall	Stall
II^{d}				Broadcast Inv/I	MI^{dwb}				-	-	-
II^{dwb}				Issue PutM/II^{wb}				-	-	-	
II^{wb}							Write back data/I	-	-	-	

Table 5.3: L2 cache states for PMSI-Mix. Issue msg/state means the L2 issues the message 'msg' to LLC and move to state 'state'. L1 issues GetS/GetM/PutM messages to L2. Once the cache line is available, the L2 forwards (Fwd) or Broadcasts messages/data to L1. L2 issues replacement to write back a dirty block before eviction to LLC.

State	**Agent Events**						**Mem Events**	
	GetS	**GetM**	**PutM**	**ReqWT**	**Data from agent**	**Replacement**	**Mem Data**	**Mem Ack**
I	Issue MemRd /S^m	Issue MemRd /M^m	▨	Issue MemRd /V^m	▨	▨	▨	▨
S	Send data/S	Send data/M	▨	Update cacheline /V^D	▨	Broadcast Inv/I	▨	▨
M	Clear Owner /S^D	Update owner /M^D	Clear Owner /V^D	Clear Owner, Update cacheline/V^D	▨	FwdGetM to owner/II^{Dm}	▨	▨
V	Send data/S	Update Owner, Send data/M	▨	Write data to LLC/ V^D	▨	Issue MemWr/I^m	▨	▨
S^D	Stall	Stall	▨	Stall	Write data to LLC/S	I^D	▨	▨
V^D	Stall	Stall	▨	Stall	Write data to LLC/V	I^D	▨	▨
S^m	Stall	Stall	▨	Stall	▨	I^m	Write data to LLC/S	▨
M^m	Stall	Stall	▨	Stall	▨	I^m	Write data to LLC/M	▨
II^{Dm}	Stall	Stall	▨	Stall	Write data to LLC/I^m	▨	▨	▨
I^m	Stall	Stall	▨	Stall	▨	▨	Write data to LLC/I	I

Table 5.4: LLC cache states for PMSI-Mix. Issue msg/state means the LLC issues the message 'msg' to main memory and moves to state 'state'. An agent issues GetS/GetM/PutM/ReqWT messages to LLC. Once the cache line is available, the LLC sends data to an agent. LLC issues replacement to write back a dirty block before eviction to memory. Changes to conventional PMSI are highlighted in red. Shaded cells represent transitions that are not possible under correct operation.

Chapter 6

Worst Case Latency Analysis

Chapter 5 outlined two key limitations that this work addresses. Chapter 5 addresses the first limitation by describing the implementation details of PASoC. The purpose of this chapter is to address the second limitation, that is to derive the per-request WCL of a memory request issued by each of the following: a core within a CPU cluster with all cores being cache-coherent, a fully-coherent accelerator with only one core in it, and a one-way coherent accelerator. This chapter begins in Section 6.1 by defining the WCL components that are used in deriving the WCL of PASoC. Next, Section 6.2 outlines the critical instance scenario that elicits the worst-case behavior and derives the WCL for different agents.

6.1 WCL Components

We begin by defining the WCL components that we use to determine the WCL analysis of a memory request. These latency components represent the WCL for accessing shared caches such as the L2 and LLC or for granting access to the shared bus by the arbitration.

6.1.1 Worst-Case TDM Arbitration Latency

TDM arbitration is used to grant access to the shared caches (L2 and LLC have their own TDM arbiter). Like several prior works, we assume a TDM schedule where each requester is granted a slot in a period [15].

Lemma 1 *The WCL of a requester accessing a shared cache using TDM arbitration is given by*

$$WCL_{TDMArb}(u, s) = u \times s, \tag{6.1}$$

where u is the number of requesters, and s is the number of clock cycles of a slot width.

For any requester to access the shared cache via TDM arbitration, the WCL occurs when the requester misses the start of the TDM slot allocated to it, and it has to wait for the TDM arbiter to grant the requester its next slot, as shown in Figure 6.1. The schedule of TDM arbitration throughout the system grants one slot to each sharer per period. Hence, the request that misses the start of the slot will suffer the WCL. Otherwise, the request will wait for a lesser amount of time before reaching its next slot. Given that TDM allocates one slot per requester, the WCL is simply the product of the number of requesters and the slot width.

6.1.2 Worst-Case Intra-core Arbitration Latency

As described in Section 4.9.1, each core buffers incoming messages in two FIFOs: Demand Request FIFO (DRF) and Forward Response FIFO (FRF). When a demand request and a write-back response for forwarded requests are both ready in their corresponding FIFOs, the WCRR arbitration in PASoC picks one. The other message must wait until the one selected by WCRR completes before it can be placed on the shared bus to access the cache. This latency to arbitrate such request and response messages within a core is termed intra-core arbitration latency.

Lemma 2 *The WCL of intra-core arbitration of a request from a core in an FCA is given by $WCL_{intraCoreArb}(y) = y$ where y is the number of cycles for a WCRR round.*

When the message is buffered in a FIFO (DRF or FRF), in the worst case, the WCRR picks a message from the other FIFO. Hence, it has to wait for its next WCRR round to place the message on the shared bus, as shown in Figure 6.1. Note that in our system model, the number of cycles for a WCRR round is the same as the TDM period to allow access to a lower-level cache, as stated in Lemma 1.

6.1.3 Worst-Case Back-Invalidation Latency

As described in Section 4.4, eviction of a victim cache line from an inclusive shared cache requires back-invalidation of the victim cache line privately cached in all the upper-level caches. A BI occurs when access to the shared, inclusive cache targets a set that has no vacant entries. Thus, a victim cache line in the set is chosen to be evicted from the shared cache. This cache line is called the victim cache line. However, that victim cache line may be privately cached in upper-level caches. Since the cache hierarchy is inclusive, all copies of the victim cache line in the upper-level caches must also be invalidated. The worst-case back invalidation latency accounts for having to perform this BI by a shared cache in its upper-level caches so that the victim cache line is ready to be evicted.

Lemma 3 *The WCL of BI at a shared cache is given by*

$$WCL_{BI}(u, x) = u \times (2x), \tag{6.2}$$

where x is the TDM arbitration period for accessing the shared cache, and u is the number of pending write-back responses in upper-level caches before the write-back of the victim cache line.

In the worst case, the core that has the victim cache line in its private cache can have pending write-back responses queued in its FRF, and the victim cache line write-back is queued last. Thus, all u write-back responses are serviced before the write-back of the victim cache line. According to Lemmas 1 and 2, each of the u write-back responses takes two TDM arbitration periods to complete when they are ready to be serviced at the front of the FRF.

6.1.4 Worst-Case Replacement Latency

The worst-case replacement latency of a memory request is the latency that the demand request experiences when the cache line is a miss in the shared cache and requires a victim cache line eviction. Recall from the system model that PASoC deploys a set sequencer [37] (⑦) to order the memory requests to the same cache set and prevent a younger request from occupying a vacant entry in the shared caches. The worst-case replacement latency follows directly from [37]:

Lemma 4 *The WCL of a replacement at a shared cache is*

$$WCL_{repl}(u, v, x, m, w) = u \times (WCL_{BI}(v, x) + w + m), \tag{6.3}$$

where u is the requesters allowed to access the shared cache, v is the number of write-back responses from upper-level caches, x is the TDM arbitration period, m is the WCL to fill the vacant entry with the requested cache line from the lower-level memory, and w is the WCL to write-back the victim cache line entry to the lower-level memory.

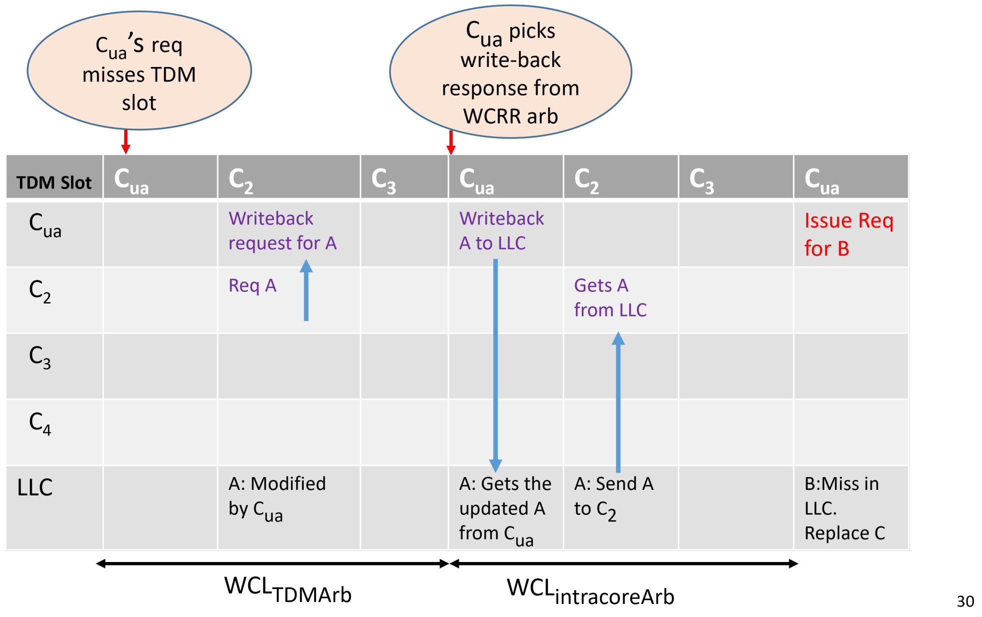


Figure 6.1: Timing diagram explaining the WCL_{TDMArb} and $WCL_{intraCoreArb}$ for core C_{ua}

A memory request r experiences the WCL for replacement in the following scenario. All u requesters, which are connected to the shared cache through the command bus and are capable of sending a demand request, issue a demand request to the shared cache to the same cache set that is full (no vacant entries). Since these requests are to the same cache set, they are enqueued in the order of arrival into the set sequencer [37]. Each request in the set sequencer experiences $WCL_{BI}(v,x)$ to get the write-back response from upper-level caches in the worst case. This write-back response from the upper-level caches is then written back to the lower-level memory, which vacates an entry in the targeted cache set. This allows the shared cache's controller to then retrieve the data being requested by r and fill in the vacant entry in the cache set.

Figure 6.2 explains the WCL_{repl}scenario for a demand request for cache line B from core C_{ua}. The request is a miss in LLC and triggers BI of cache line C modified by core C_3. Core C_3 has pending demand requests and write-back responses. Hence, the pending response for cache line C is queued last. Once the core C_3 writes back the modified cache line C to LLC, LLC replaces cache line C with B and sends the response to core C_{ua}.

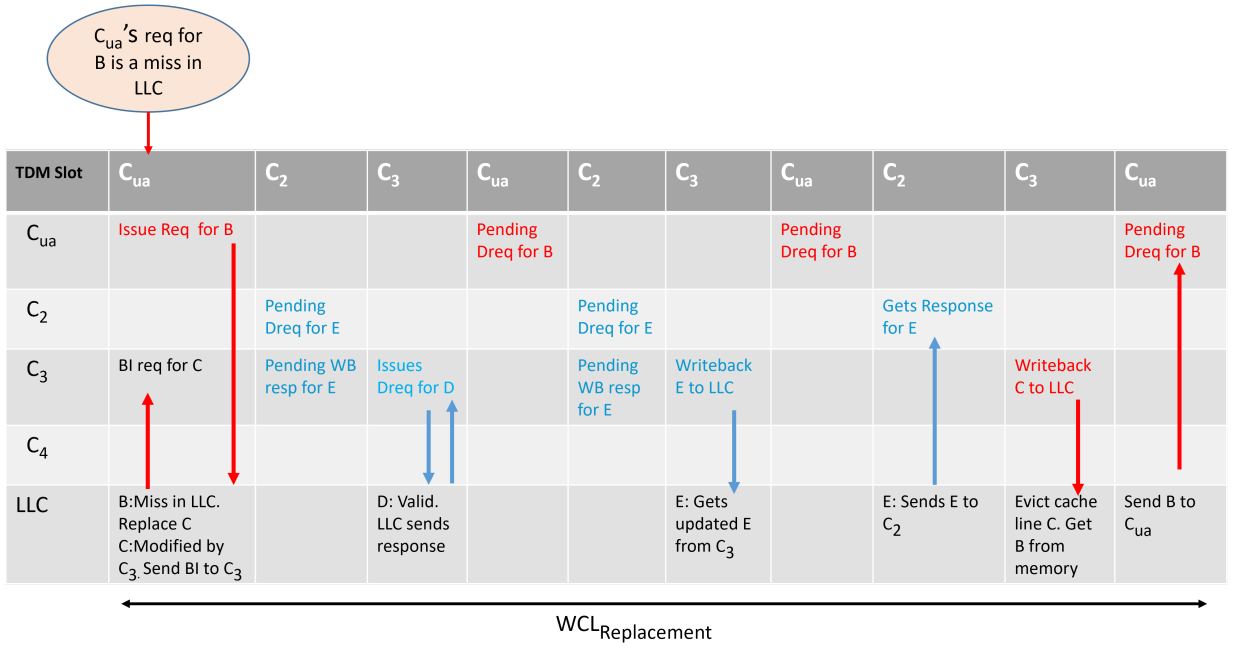


Figure 6.2: Timing diagram explaining the WCL_{repl} for core C_{ua}

6.2 WCL Analysis of Demand Request From a Core in a CPU Cluster

This section constructs the critical instance scenario for deriving the WCL of a demand request from the CPU core and derives the WCL.

6.2.1 Critical Instance

The critical instance of a demand request from a core in a CPU cluster happens when the request misses in all caches (L1, L2, and LLC) and experiences the WCL of replacement (Lemma 4) in both the L2 and the LLC. To present the critical instance, we first concentrate on the critical instance for fetching data from any shared cache, i.e., L2 or LLC, in Lemma 5.

Lemma 5 *The WCL of a demand request fetching data from a shared cache occurs under the following scenario: (1) The request experiences the WCL of TDM arbitration to access the shared cache; (2) the request is processed after a write-back request if the request*

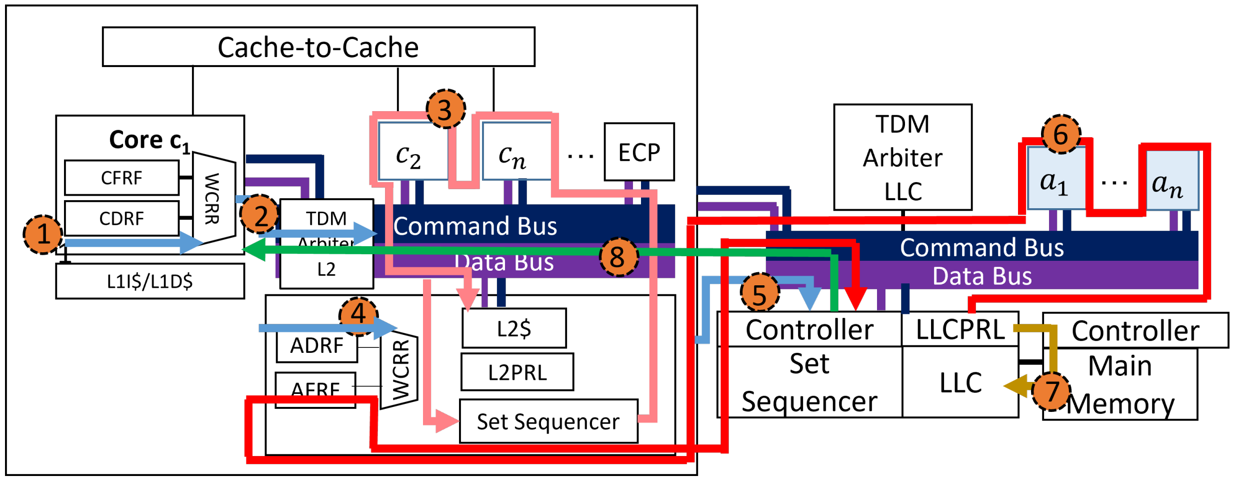


Figure 6.3: Critical instance scenario.

originates from a core and experiences the WCL of intra-core arbitration; (3) the request misses in the shared cache and requires a replacement to vacate an entry in the cache set resulting in back-invalidations and incurs the WCL of replacement.

For a demand request from a core in a CPU cluster to experience the WCL, it experiences the WCL at all cache levels, where we apply Lemma 5 to both the L2 and LLC caches.

Corollary 6 *The critical instance of a demand request from a core in a CPU cluster occurs when the request misses in L1 and L2 and LLC, and experiences the WCL in both L2 and LLC.*

The intuition behind Corollary 6 is that a demand request that hits any cache in the cache hierarchy can have its requested data returned by the cache sooner than if the request is a miss. Note that a miss must traverse further to the lower-level caches or memory. Hence, we can use Lemma 5 once for the shared L2 cache and again for the LLC to construct the critical instance such that the demand request experiences the WCL.

We illustrate the worst-case scenario with the critical instance in Figure 6.3. A request from a core gets enqueued in L1's $CDRF$. This request misses in L1, resulting in a demand request for the L2. However, before accessing the L2, a write-back response in $CFRF$ may interfere with this demand request in $CDRF$, causing the demand request to experience the WCL of intra-core arbitration (1). Next, the demand request gets placed on the command bus with TDM_{L2} arbitration. To get its slot, the request experiences the WCL

TDM arbitration latency (②). Once the command is issued to L2, the L2 processes this demand request by checking whether the targeted cache set is the same as prior requests enqueued in the set sequencer. In the worst case, the request targets a cache set that is also being accessed by all prior requests in the set sequencer initiated by all other L1s. The set sequencer orders all prior accesses to the same set first, and then, the request can access the L2. This can result in a replacement of a cache line to vacate an entry for the data to be brought in from the lower-level cache or memory (③). The request experiences the WCL of fetching data from the LLC, which constitutes the WCL of a demand request from the L2, as stated in Lemma 4 (④-⑧).

After that, this demand request is enqueued at the $ADRF$ of the L2 to be issued to LLC and experiences WCL of intra-core arbitration between $ADRF$ and $AFRF$ (④).

Note that the request experiences the same scenario when suffering a miss in the LLC and accessing the main memory; hence, we can re-apply Lemma 5 (⑤-⑧).

6.2.2 WCL for FCA

Since the WCL of a demand request in L2 happens when a request experiences the WCL in the LLC, we start the derivation of the WCL of a demand request by presenting the WCL request experiences at the LLC as explained in Lemma 7. The proof for all the below-derived equations follows directly from Lemma 5. Table 6.1 summarizes the symbols used for lateny derivations.

Lemma 7 *The WCL of a demand request processed at the LLC is*

$$\begin{aligned} WCL_{DreqLLC} = WCL_{TDMArb}(N_A, SW_{LLC}) + \\ WCL_{intraCoreArb}(N_A \times SW_{LLC}) + \\ WCL_{repl}(N_A, N_A, N_A \times SW_{LLC}, L_{mem}, L_{mem}), \end{aligned} \quad (6.4)$$

where, SW_{LLC} is the slotwidth of TDM_{LLC} arbiter, N_A is the number of agents in PASoC and L_{mem} is the WCL to access main memory. The TDM arbitration latency and intra-core arbitration arbitration latency is one TDM_{LLC} period, which is $N_A \times SW_{LLC}$. For the worst-case replacement latency at LLC, there can be N_A agents requesting a cache line in LLC. Each back-invalidation may be stalled by N_A write-back responses at an agent in the worst case. L_{mem} is the WCL to fill in the vacant entry with the requested cache line from the main memory or the WCL to write back the victim cache line entry to the main memory.

Symbol	Description
N_A	Number of agents in the system
n_c	Number of cores in FCA
SW_{L2}, SW_{LLC}	Slot width at TDM_{L2} and TDM_{LLC} respectively
L_{mem}	Latency to access main memory
$WCL_{DreqLLC}$	WCL of a demand request processed at LLC
WCL_{wbLLC}	WCL of a write-back request from L2 to LLC
$WCL_{totalFCCReq}$	WCL of a demand request sent from L1
$WCL_{totalOCAReq}$	WCL of a demand request from OCA

Table 6.1: Symbols used for WCL derivation

We also derive the WCL for performing a write-back from the L2 to the LLC, as required in Lemma 4, which follows from the fact that a write-back request completes in one slot because it does not require interaction with other agents; hence, it only experiences arbitration latencies.

Lemma 8 *The WCL of a write-back request from the L2 to the LLC is given by*

$$WCL_{wbLLC} = WCL_{intraCoreArb}(N_A \times SW_{LLC}) + WCL_{TDMArb}(N_A, SW_{LLC}), \quad (6.5)$$

We present Theorem 9 that gives the WCL of a demand request that misses in the L1 of a core.

Theorem 9 *The WCL of a demand request sent from L1 is*

$$\begin{aligned} WCL_{totalFCCReq} = {} & WCL_{TDMArb}(n_c + 1, SW_{L2}) + \\ & WCL_{intraCoreArb}((n_c + 1)SW_{L2}) + \\ & WCL_{repl}(n_c + N_A, n_c + N_A, (n_c + 1)SW_{L2}, WCL_{DreqLLC}, WCL_{wbLLC}), \end{aligned} \quad (6.6)$$

where, SW_{L2} is the slotwidth of TDM_{L2} arbiter, n_c is the number of cores in the CPU cluster. The TDM arbitration latency and intra-core arbitration arbitration latency is one

TDM_{L2} period, which is $(n_c+1)\times SW_{L2}$. For the worst-case replacement latency at L2, there can be n_c+N_A cores requesting a cache line in L2. Each back-invalidation may be stalled by n_c+N_A write-back responses at a core in the worst case. $WCL_{DreqLLC}$ is the WCL to fill the vacant entry with the requested cache line from LLC, and WCL_{wbLLC} is the WCL to write back the victim cache line entry to LLC.

Note that the WCL analysis discussed in this section applies to fully coherent agents with a single processing element ($n_c = 1$).

6.3 WCL for OCA

Recall that an OCA does not have private caches that are coherent with other agents in the PASoC. A request from an OCA directly accesses the LLC. Thus, the WCL of a request for an OCA is simply the WCL to access the LLC as shown in Theorem 10.

Theorem 10 *The WCL of a demand request from an OCA is given by*

$$\begin{aligned} WCL_{totalOCAReq} = {} & WCL_{TDMArb}(N_A, SW_{LLC}) + \\ & WCL_{repl}(N_A, N_A, N_A \times SW_{LLC}, L_{mem}, L_{mem}). \end{aligned} \tag{6.7}$$

where, SW_{LLC} is the slotwidth of TDM_{LLC} arbiter, N_A is the number of agents in the PASoC. Note that an OCA does not have a FRF; hence, there is no intra-core arbitration latency involved. However, a demand request from an OCA may cause coherence activity in other coherent agents as captured by the WCL of a replacement, which includes the WCL of back-invalidations.

6.4 Observations

PASoC is an attempt at employing state-of-the-art research in modeling a predictable accelerator-rich SoC with coherent accelerators. This effort allows us to make observations that may be beneficial for a future version of PASoC. Specifically, there are four observations that we hope to further explore.

6.4.1 Back-Invalidations

WCL analysis revealed that a large contributor to the WCL is back-invalidations. This is evident when we expand Equations 9 ($WCL_{totalFCCReq}$) and 10 ($WCL_{totalOCAReq}$), which show that the WCL of a demand request had n_c^3 and $n_c N_A^3$ components. Therefore, to further reduce the WCL in a PASoC, we need to explore solutions that either reduce or eliminate the contributions that back-invalidations introduce to the WCL.

6.4.2 OCA vs. FCA

WCL analysis further revealed that the WCL of a demand request originating from an OCA ($WCL_{totalOCAReq}$ (10)) is lower compared to one from an FCA ($WCL_{DreqLLC}$ (7)). This is because an OCA does not have private caches and employs one-way coherence, which means it does not respond to coherence activities from other cores. Thus, there is no need for a forward response FIFO, which eliminates the ($WCL_{intraCoreArb}$ (2)) latency that would appear in the WCL for FCA. This reduction in WCL, which has the N_A component, is marginal because the OCA can still cause back-invalidation in the LLC and causes the N_A^3 latency component in the WCL.

However, there is a large difference in the WCL of a demand request originating from a core in the CPU cluster ($WCL_{totalFCCReq}$ (9)) and from the core in OCA ($WCL_{totalOCAReq}$ (10)). The $WCL_{totalFCCReq}$is of the order of (n_c+ N_A) times of $WCL_{totalOCAReq}$. This is because of the intra-agent coherence that exists between the cores of the CPU cluster.

We also notice that predictable cache coherence protocols used for multicores, such as PMSI, are not readily applicable to an OCA. Specifically, to enable one-way coherence for PMSI, coherence activities by agents other than the OCA must be blocked for the OCA, and the coherence protocol transitions in response to other agent's activities must also be removed from the transitions for the OCA. Further, the protocol must be extended to support coherence activities as a result of OCA's write-through to the LLC.

6.4.3 WCL for Hierarchical Systems

PASoC follows a hierarchical cache structure for FCA agents, where each core in the cluster has a private L1 cache, all cores in the cluster share an L2 cache, and all agents in the system share an LLC. The critical instance scenario from Section 6.2.1 showed that the WCL analysis for a core in the cluster ($WCL_{totalFCCReq}$ (9)) is similar to the WCL analysis for a demand request from L2 to LLC ($WCL_{DreqLLC}$ (7)). This is evident when we expand the Equation 9, where the $WCL_{totalFCCReq}$ has WCL_{replL2}, which includes $WCL_{DreqLLC}$ term to fetch data from LLC. That is,

$$WCL_{totalFCCReq} = (n_c + N_A)WCL_{DreqLLC} \tag{6.8}$$

where n_c is the number of cores in cluster and N_A is number of agents in the system. Hence, the WCL analysis for hierarchical systems follows a recursive bottom-up approach, starting from the last-level cache in determining the WCL bound of a demand request from a core. This can be applied to any level of hierarchical cache coherence. This shows that latency components for hierarchical cache systems get multiplied for every level of cache hierarchy, and this is worsened for WCL_{repl}latency component.

6.4.4 Self-Invalidation-Based Coherence Mode

Although PASoC supports fully coherent and one-way coherent agents, a common accelerator that we do not discuss is Graphics-Processing Unit (GPU). GPUs are widely used in autonomous driving, and a careful study of their predictability with other agents must be done. GPUs are particularly unique because they typically use write-through and self-invalidation-based coherence approaches best suited for throughput-based applications. However, integrating such coherence approaches with existing predictable coherence approaches remains unexplored and is necessary if one of the accelerators is to be a GPU. There are some good examples, such as Spandex [1], that we can use for inspiration. However, there are specific challenges while integrating GPUs with the system because GPUs typically rely on a Data Race-Free (DRF) consistency model and synchronization accesses. Hence, it is important to carefully study the WCL analysis while integrating these accelerators.

Chapter 7

Evaluation

Chapters 5 and 6 presented the implementation and worst-case latency analysis of PASoC. The purpose of this chapter is to evaluate the implementation and worst-case latency analysis with benchmarks and synthetic workloads. All the implementation was completed in gem5 simulator [2] and evaluated using Splash 3 benchmark suite [28].

7.1 Methodology

PASoC is integrated into the gem5 simulator using the Ruby memory model, and Garnet interconnect, which is a cycle-accurate model with a detailed implementation of cache coherence events and the network interconnect. All the cores in the system are x86 cores running at 1GHz. The cores implement in-order pipelines, which are representative of cores used in real-time domains. The CPU cluster is a multi-core architecture, where each core has a private 1KB direct-mapped instruction and data cache. All cores in the CPU cluster share an L2 cache, which is a 4-way set associative cache of size 16KB. The fully coherent accelerator has a private 4KB 2-way set associative L1 instruction and data cache and a 32KB 4-way set associative L2 cache. One-way coherent accelerator does not have a private cache and always fetches the up-to-date data from LLC and issues write-through stores to LLC. All the agents share an LLC, which is a 16-way set associative cache of size 1MB. All the caches have a cache line size of 64 bytes. The CPU cluster and fully coherent accelerator consist of an interconnect bus that uses TDM arbitration amongst all cores and ECP, granting one slot per core. The interconnect bus consists of shared data and message bus between all cores, ECP and L2 cache, and point-to-point data interconnect between all cores. The L1 cache controller uses work-conserving TDM arbitration between a core's own requests and its responses to other core requests. The interconnect bus that connects all the agents in the system also uses TDM arbitration amongst all agents. The interconnect bus consists of shared data and message buses between all agents, and every agent uses work-conserving TDM arbitration between its own requests and responses to other agents. The shared L2 and LLC cache has a set sequencer to order requests to the same cache set. We do not run an operating system in the simulator, and hence, all memory addresses generated by the cores are physical memory addresses. For all the workloads, the number of CPU cores in the CPU cluster is $n_c = 4$, $SW_{L2} = 16$ cycles, $SW_{LLC} = 50$ cycles, and $L_{mem} = 200$ cycles. PASoC is evaluated using the SPLASH-3 benchmark suite developed by [28]. In addition, PASoC uses synthetic workloads to stress the WC behavior. Table 7.1 summarizes the simulation parameters as described in this section.

7.2 Verification

The correctness of PASoC is verified using Ruby Random Tester with gem5 [2] specifically to verify coherence protocols. PASoC is stressed with 10 million random requests that cover all possible transitions and states in PASoC. PASoC is executed with applications in the SPLASH-3 benchmark suite on gem5, and the data correctness is checked by checking the output of the application. All observed latencies conform to the bounds, which ensures the predictability of the system.

7.3 Experiments and Observations

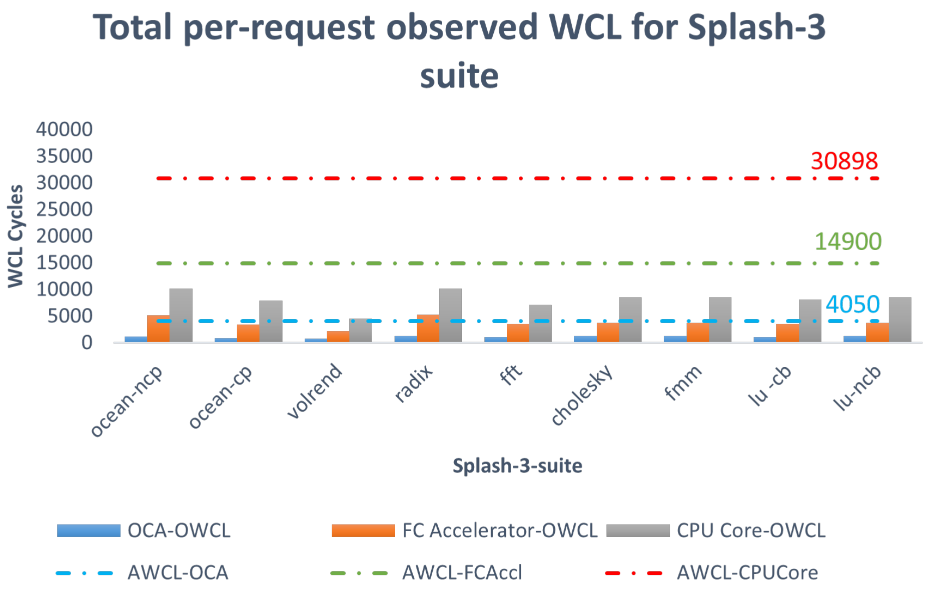
This section discusses different experiments conducted to obtain the WCL of all agents in PASoC for different workloads. For each experiment, we compare the results with analytical WCL and state the observations.

7.3.1 Total Per-request Observed Worst-Case Latencies

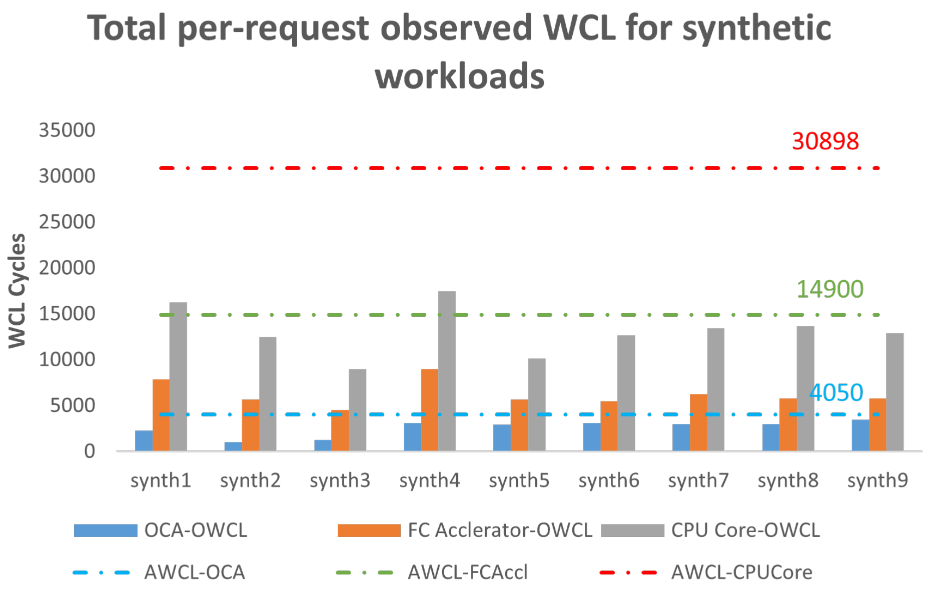
In this experiment, we study the effectiveness of PASoC to bound the total per-request worst-case memory latency. For SPLASH-3, the SPLASH-3 application was launched as multiple threads using single-threaded cores in CPU cluster, fully coherent, and one-way coherent accelerators, where only one application is used per experiment. Figure 7.1a depicts the findings. Since SPLASH-3 applications are optimized to minimize data sharing,

Parameter	**Configuration**
PASoC platform	CPU cluster, Fully coherent agent, One-way Coherent agent, one outstanding memory request per agent, 1GHz operating frequency
PASoC memory hierarchy	All agents share an LLC, 1MB 16-way set associative cache, cacheline size 64 bytes
CPU cluster	4 in-order cores, one outstanding memory request per core, private 1KB direct-mapped instruction and data cache, 16KB 4-way set associative shared L2 cache, cache line size 64 bytes
Fully coherent agent	Single in-order core, private 4KB 2-way set associative L1 instruction and data cache, 32KB 4-way set associative L2 cache, cache line size 64 bytes
One-way coherent agent	Single in-order core, no private cache
Inter-agent bus interconnect	Shared data and message bus interconnects between agents and LLC, TDM arbitration policy, one TDM slot per agent, slot width = 50 cycles
Intra-agent bus interconnect	Shared data and message bus interconnects between cores,ECP and L2 cache, TDM arbitration policy, one TDM slot per core, point-to-point data interconnects between cores, slot width = 16 cycles

Table 7.1: Simulation parameters



(a) Total per-request observed WC latencies for SPLASH-3 suite



(b) Total per-request observed WC latencies for synthetic workloads

Figure 7.1: Total per-request observed WC latencies of different agents in PASoC

they do not stress the coherence protocol. Therefore, to further stress the coherence protocol, PASoC was executed with synthetic experiments using nine synthetically generated workloads: Synth1 to Synth9 as shown in Figure 7.1b. In each synthetic experiment, identical instances of one workload are run simultaneously by assigning one instance on each core. These experiments represent the maximum possible sharing of data since each core generates the same sequence of memory requests. Figures 7.1a and 7.1b illustrate the total per-request observed WC latencies for a) a CPU core in a CPU cluster, b) a fully coherent accelerator, and c) a one-way coherent accelerator.

Observations.

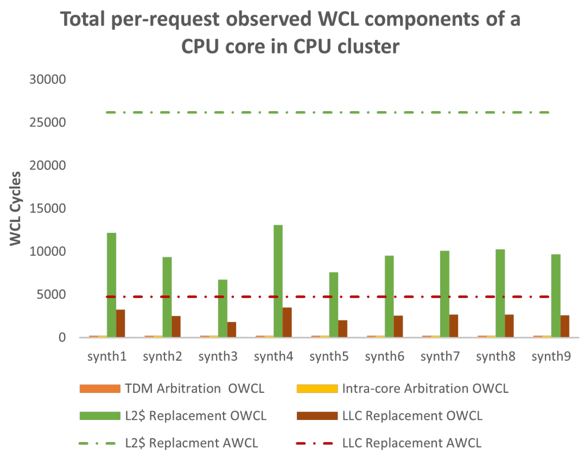
- The Observed WCL (OWCL) bound for each core in PASoC is always within its respective Analytical WCL (AWCL) bound.
- The OWCL bound for a CPU core in a CPU cluster (CPU Core-OWCL) is significantly higher than a fully coherent accelerator (FC Acclerator-OWCL) or one-way coherent accelerator (OCA-OWCL). This is because of intra-cluster coherence present between cores in CPU clusters, which causes additional coherence activity when compared to single-core agents like accelerators. Hence, for workloads that require frequent inter-agent communication, allowing an agent to have a clustered hierarchy with shared intermediate caches is not very efficient in terms of latency.
- The OWCL of an OCA core is comparatively lower compared to FCAs. This is because OCA does not respond to any coherence activity in the system and does not have any private caches. The worst-case latency is the latency to obtain the most up-to-date data from LLC. Hence, this shows the latency benefits of having simpler coherence protocols like one-way coherence instead of complex MSI-based fully coherence modes for communication between different agents.
- The OWCL for fully coherent agents, such as CPU clusters or fully coherent accelerators, is significantly higher than a one-way coherent agent. This is because of back-invalidations in the L2 cache. As mentioned in Chapter 6, back-invalidations are the major contributors to the worst-case latency, and its effect can get worsened or multiplied because of multiple inclusive cache hierarchy levels.

7.3.2 Profiling the Total Worst-Case Latencies

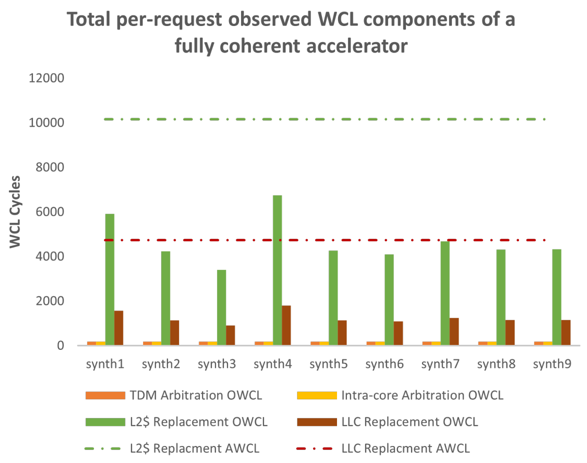
In this experiment, we study the contributions of each latency component to the total per-request worst-case memory latency for a) a CPU core in a CPU cluster, b) a fully coherent accelerator, and c) a one-way coherent accelerator. This experiment executes nine synthetically-generated workloads: Synth1 to Synth9 and uses profiling to collect the latency components as shown in Figures 7.2a, 7.2b, and 7.2c.

Observations.

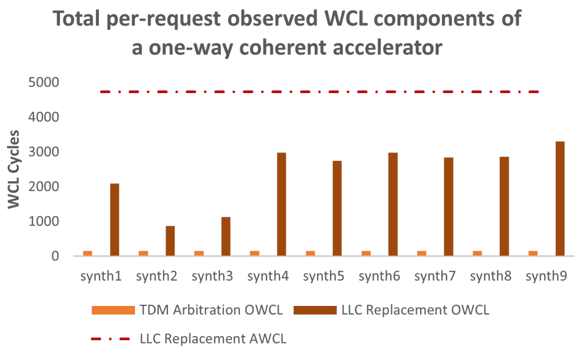
- The OWCL components for each core in PASoC are always within its AWCL bounds.
- The major contributor to the total per-request OWCL component for all cores is L2 cache's and LLC's Replacement latency. Hence, as mentioned in Chapter 6, we need solutions to eliminate back-invalidations in the system.
- The observed worst-case replacement latency in the L2 cache (L2 Replacement OWCL) is significantly higher than the observed worst-case replacement latency in the LLC (LLC Replacement OWCL) cache because replacement in L2 can trigger replacement in LLC to fetch the new cache line from the main memory. Hence, this shows that back-invalidations have a recursive effect on total worst-case latency in the hierarchical cache system model.



(a) Total per-request observed WCL components of a CPU core in CPU cluster



(b) Total per-request observed WCL components of a fully coherent accelerator



(c) Total per-request observed WCL components of a one-way coherent accelerator

Figure 7.2: Total per-request observed WCL components of different agents in PASoC

Chapter 8

Conclusions

This chapter summarizes the discussion and findings in Chapters 4 through 7, address the limitations of the implementation and analysis, and present opportunities for future work.

8.1 Discussion

The objective of this work is to present a model of a predictable accelerator-rich SoC, PASoC, that allows the integration of multiple agents with different coherence modes.

Chapter 4 discussed the system model of PASoC. Fundamentally, PASoC integrates a CPU cluster, fully coherent accelerator, and one-way coherent accelerator in an SoC that shares LLC via shared command and data bus. In order to ensure predictability for the system, architectural modifications are proposed, inspired by prior works [15] [20] [37]. These architectural modifications include TDM bus arbitration between cores to access shared memory, *FIFOs* to order pending requests and write-back requests from a cache controller, *PRLUTs* to order requests to the same cache line in a shared memory, and *set sequencers* to order requests to same cache set in a shared memory. This system model allows for designing a predictable accelerator-rich SoC and forms the base implementation for analyzing the critical instance scenario of memory access in the system.

Chapter 5 described the implementation details of the system model proposed for PASoC in Chapter 4. Architectural modifications include designing the ECP port to allow coherent interaction between agents and designing an intermediate L2 cache that filters out unnecessary interactions from within an agent to external agents and vice-versa. In order to support one-way coherence, architectural modifications required changes in the shared bus to block any coherence messages sent to OCA and also to allow for write-through stores from OCA. Protocol changes to PMSI [15] and linear coherence [20] that work in tandem with the proposed architectural modifications resulted in a new coherence protocol PMSI-Mix for PASoC. PMSI-Mix includes new coherent stable states, transient states, and bus events for the PASoC system model. Together, these architectural modifications and protocol changes form the base for performing worst-case latency analysis and defining the predictable WCL bounds for PASoC.

Chapter 6 presented the worst-case latency analysis for the proposed PASoC architecture, applying PMSI-Mix coherence protocol to the design. The latency analysis formulates different worst-case latency terms while accessing a shared memory and constructs a critical instance scenario of a memory request in PASoC. These latency terms, when applied to the critical instance scenario, derive the analytical WCL bounds of memory access from different agents in PASoC. This latency analysis led to some of the critical observations that can be applied to design better PASoCs in the future. The key observations include that back-invalidations are the main contributors to the WCL bound of a memory access and significant WCL differences between a request originating from an FCA core vs. an OCA core. The lower WCL of a demand request originating from the OCA core is because OCA does not respond to any coherence activity in the system and the absence of private caches in OCA. These observations form a base for designing better PASoCs for the future by finding solutions to eliminate back-invalidations in the system and allowing accelerators to integrate with simple coherence modes like one-way coherence instead of complex full-way coherence protocols like MSI.

Chapter 7 evaluated the PASoC implementation on the gem5 simulator with various benchmark applications from SPLASH-3 as well as using synthetic workloads. The worst-case latency was collected after running every benchmark suite. The empirical evaluation matches with the analytical worst-case latency analysis, where back-invalidations are the major contributors to the latency. The latency for different agents was evaluated and compared, where a memory request from a CPU core in a CPU cluster experiences higher latency when compared to other agents. All observed WCL bounds are within their analytical WCL bounds.

8.2 Limitations and Future Work

This thesis presents a predictable accelerator-rich SoC suitable for designing safety-critical systems by ensuring a defined WCL bound of memory access in the system. While PASoC employs novel architectural and coherence protocol modifications in modeling an SoC with coherent accelerators, there are some limitations and opportunities for enhancing this work for a future version of PASoC.

From the WCL analysis, it is evident that back-invalidations are the major contributors to WCL. The WCL analysis showed that back-invalidations contribute recursively to the total WCL bound. That is, in the worst-case a replacement in the L2 cache triggers back-invalidations in all L1 caches, and a replacement in LLC triggers back-invalidations in all L2 and L1 caches. Hence, future PASoCs should either eliminate back-invalidations in inclusive caches or explore other different cache inclusion techniques in the design. Hence, future work should aim to either reduce or eliminate the contributions of back-invalidations to WCL.

PASoC does not integrate GPUs, which are widely used in autonomous driving. GPUs have a completely different hardware architecture and programming model suitable for parallelism in high-throughput applications. GPUs have a cluster of processing units called Streaming Multiprocessors (SMs). Each SM is a highly multi-threaded scalar core capable of running on the order of a thousand threads. All threads mapped to an SM share the L1 cache, and all SMs share an L2 cache. This cache hierarchy is similar to a CPU cluster; however, GPU workloads do not share data or synchronize frequently. Hence, GPUs do not enforce SWMR invariant and implement relaxed consistency models with scoped synchronization such as SC for Data-Race-Free (DRF) consistency model [33] [32]. GPUs typically use write-through and self-invalidation-based coherence approaches. Hence, integrating GPUs with existing predictable coherence approaches that enforce SWMR invariant and sequential consistency models remains unexplored. There are some good examples, such as Spandex [1], that we can use for inspiration. However, it is important to carefully study the WCL analysis while integrating these accelerators with different consistency models. This opens up an exciting research opportunity in designing PASoC with GPUs and is left for future work.

Devices in heterogeneous systems are specialized for specific data access patterns and can have a wide range of memory demands in terms of spatial locality, temporal locality, fine-grain synchronization, latency sensitivity, throughput sensitivity, and cache inclusiveness [1]. Hence, there can be different coherent strategies for integrating accelerators and CPU cores over an SoC, which can also pose new challenges to the predictability of the system. Hence, there is a scope for future work to explore all such interactions in the SoC.

Finally, PASoC is an initial step in exploring predictability for ASoCs. As previously mentioned and explored in prior work, modern ASoCs are not designed with predictability in mind. Therefore, there is an opportunity for research work to redesign ASoCs to better serve real-time and safety-critical systems.

8.3 Summary

This work presented a model of a predictable accelerator-rich SoC, PASoC that allows the integration of multiple agents with different coherence modes. At the moment, PASoC enables integrating one CPU cluster of cache-coherent multicores with multiple fully coherent and one-way coherent agents. The CPU cluster supports a predictable cache coherence protocol called linear coherence, and the coherence protocol between the agents is a modified version of predictable MSI, resulting in a new coherence protocol, PMSI-Mix. PASoC also provided a WCL analysis that shows the latency a memory request would experience in the worst case when accessing data from any of the agents. Our future work plans to extend PASoC to support other forms of accelerators, such as GPUs, and to address the WCL contributions from back-invalidations.

References

[1] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 261–274. IEEE Press, 2018.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[3] Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. Ultrascale + MPSoC and FPGA families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–37. IEEE, 2015.

[4] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T Malladi, and Hongzhong Zheng. CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 629–642, 2019.

[5] Jennifer Brana, Brian C Schwedock, Yatin A Manerkar, and Nathan Beckmann. Kobold: Simplified Cache Coherence for Cache-Attached Accelerators. *IEEE Computer Architecture Letters*, 2023.

[6] Anthony M Cabrera, Aaron R Young, and Jeffrey S Vetter. Design and analysis of CXL performance models for tightly-coupled heterogeneous computing. In *Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions*, pages 1–6, 2022.

[7] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. De-

novo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, 2011.

[8] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[9] Emilio G Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[10] Michael Ditty. Nvidia Orin System-On-Chip. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–17. IEEE Computer Society, 2022.

[11] Björn Forsberg, Luca Benini, and Andrea Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544. IEEE, 2018.

[12] Ilya K Ganusov, Mahesh A Iyer, Ning Cheng, and Alon Meisler. Agilex™ Generation of Intel FPGAs. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–26. IEEE Computer Society, 2020.

[13] Sebastian Hahn and Jan Reineke. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *RTSS*, Dec 2018.

[14] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1–24, 2009.

[15] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. Predictable Cache Coherence for Multi-Core Real-Time Systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, 2017.

[16] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet. Work-Conserving Dynamic Time-Division Multiplexing for Multi-Criticality Systems. *Real-Time Systems*, 56(2):124–170, 2020.

[17] Salah Hessien and Mohamed Hassan. PISCOT: A Pipelined Split-Transaction COTS-Coherent Bus for Multi-Core Real-Time Systems. *ACM Trans. Embed. Comput. Syst.*, jul 2022.

[18] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 427–440, 2014.

[19] Anirudh Mohan Kaushik. Timing Predictable and High-Performance Hardware Cache Coherence Mechanisms for Real-Time Multi-Core Platforms. *University of Waterloo*, pages 27–88, 2021.

[20] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *IEEE Transactions on Computers*, 70(12):2098–2111, 2021.

[21] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design tradeoffs in coherent cache hierarchies for accelerators. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 733–745, 2015.

[22] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G Cota, Michele Petracca, Christian Pilato, and Luca P Carloni. Agile SoC development with open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[23] Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, Alejandro J Calderon, Francisco Cazorla, and Jose Luis Flores. GPU Devices for Safety-Critical Systems: A Survey. *ACM Computing Surveys*, 55(7):1–37, 2022.

[24] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467, 2013.

[25] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, and Jan Gray. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[26] Xiaowei Ren and Mieszko Lis. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636. IEEE, 2017.

[27] Francesco Restuccia and Alessandro Biondi. Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 441–454, 2021.

[28] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[29] Brian C Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. Täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 42–58, 2022.

[30] Yakun Sophia Shao and David Brooks. *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool Publishers, 2015.

[31] Debendra Das Sharma. Compute Express Link: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12. IEEE, 2022.

[32] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache Coherence for GPU Architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590. IEEE, 2013.

[33] Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[34] Ashley Stevens. Introduction to AMBA 4 ACE™ and big. LITTLE™ Processing Technology. *ARM White Paper, CoreLink Intelligent System IP by ARM*, 2011.

[35] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[36] Michael Toksvig, Parthasarathy Sriram, John Matheson, Brian Cabral, and Brian Smith. NVIDIA Tegra. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–28. IEEE, 2008.

[37] Zhuanhao Wu and Hiren Patel. Predictable Sharing of Last-level Cache Partitions for Multi-Core Safety-Critical Systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1273–1278, New York, NY, USA, 2022. ACM.

[38] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel Quickpath Interconnect Architectural Features Supporting Scalable System Architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

[39] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P Carloni. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 350–365, 2021.