© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, "PASoC: A Predictable Accelerator-Rich SoC," in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

# PASoC: A Predictable Accelerator-rich SoC

Susmita Tadepalli
University of Waterloo
Waterloo, Ontario, Canada
susmita.tadepalli@uwaterloo.ca

Zhuanhao Wu
University of Waterloo
Waterloo, Ontario, Canada
zhuanhao.wu@uwaterloo.ca

Hiren Patel
University of Waterloo
Waterloo, Ontario, Canada
hiren.patel@uwaterloo.ca

## ABSTRACT

We present a model of a predictable accelerator-rich system-on-chip (PASoC) for safety-critical systems. The PASoC allows the integration of multiple coherent agents to interact with each other over a shared memory bus. These agents can be a cluster of cache-coherent homogeneous cores, and fully or one-way coherent hardware accelerators. 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 MSI. We analyze the per-request worst-case latency, a memory request from any of the agents can experience in the PASoC. Finally, we present some observations based on our analysis that can help in future designs of PASoCs.

## CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; **Heterogeneous (hybrid) systems**; **System on a chip**.

## KEYWORDS

Predictability, SoCs, Hardware accelerators, Safety-critical systems

**ACM Reference Format:**
Susmita Tadepalli, Zhuanhao Wu, and Hiren Patel. 2023. PASoC: A Predictable Accelerator-rich SoC. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 9–12, 2023, San Antonio, TX, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587496

## 1 INTRODUCTION

Modern system-on-chips (SoCs) combine general-purpose multicores with one or more hardware accelerators. These hardware accelerators offer high performance for domain-specific algorithms. Common hardware accelerators include those that implement vision processing and deep learning [16]. 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 as well. 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 [3].

A key challenge in designing ASoCs involves deciding the manner in which the accelerators interact with the other agents within the ASoC. An agent can be a multicore cluster or a hardware accelerator. Such decisions involve determining the coupling of the accelerators via interconnect with other agents and their coherence modes in accessing the shared memory [16]. The coupling refers to how loosely or tightly interconnected the accelerators are with other agents in the ASoC. For example, CXL [1] connects accelerators with other agents in the system via the system bus. The coherence modes dictate the coherence activity the accelerators respond to in accessing the shared memory. Examples include one-way coherent and fully-coherent.

This particular challenge has resulted in industry-proposed solutions that focus on integrating accelerators on an SoC via cache-coherent interconnect standards such as the CXL [1] and ACE [17]. It has also resulted in exciting academic research discussing the modes of cache coherence best suited for the application [19].

We find that there has been limited exploration on the predictability of such ASoCs [15] [9]. Predictability is an important requirement for safety-critical systems as it enables analyses to compute worst-case execution time for applications that are deemed critical. In this paper, we attempt to take some initial steps towards exploring the design of a predictable ASoC (PASoC). Our presented PASoC consists of multiple agents interacting over the shared memory bus that share a last-level cache (LLC).

In this paper, we mainly concentrate our efforts on analyzing the worst-case access latency (WCL) of a memory request from any agent in the PASoC.

Our main contributions to this work are as follows.

- We model a predictable accelerator-rich SoC architecture (PASoC). This PASoC integrates multiple coherent agents together that share a LLC. These agents consist of hardware accelerators and clusters of predictable cores [8]. Each agent in the PASoC can support one of the two cache coherence modes: fully-coherent or one-way coherent.
- PASoC provides support to accelerators for coherence modes by employing state-of-the-art proposals on predictable cache coherence and combining them together. Specifically, PASoC uses linear coherence [13] within cluster and predictable modified-shared-invalid (PMSI) [10] between the agents.
- We present an analysis that computes the WCL of a memory access from any of the agents in the PASoC. Through this analysis, we identify opportunities to improve the design of future PASoCs for safety-critical systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CPS-IoT Week Workshops '23, May 9–12, 2023, San Antonio, TX, USA*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0049-1/23/05…$15.00
https://doi.org/10.1145/3576914.3587496

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, “PASoC: A Predictable Accelerator-Rich SoC,” in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

## 2 RELATED WORK

**One-way coherence.** One-way coherence, also referred to as I/O coherence, or coherent DMA [19], is an approach for integrating an accelerator to an SoC. In one-way coherence, the accelerator can snoop the private caches of other agents, but does not respond to any coherence activity by other agents in the system.

**Predictable hardware 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 predictability of a memory request with defined WCL bound. Prior works on designing predictable HCC have certain requirements. First, the implementations should honour certain invariants. Second, several architectural and coherence protocol changes must be made in the system to ensure predictability of a memory request, and also to improve its WCL bound [10] [18] [13].

*PMSI.* For example, *PMSI* [10] 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 identified sources of unpredictability with a conventional MSI coherence protocol and proposed minimal architectural changes like adding *first-in-first-out* (FIFO) arbitration between requests to the same cache line in the shared memory and between write-back responses at every core. PMSI's 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.

PMSI's WCL bound is further improved in *linear coherence* [13] via coherence protocol changes and the use of a *cache-to-cache* (C2C) data bus to transfer modified data directly between the caches. Neither of these proposals supports a LLC in the system. Another approach, *PISCOT* [12], 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. A recent work, *ROC* [18] highlighted the unpredictable behavior due to back-invalidations in multicore systems with inclusive shared LLC. ROC deployed a *set sequencer* hardware architecture to prevent the unbounded WCL. To our knowledge, all these predictable HCC mechanisms are designed for CPU multicores with fully-coherent data sharing. Authors in CoMPSoC [9] proposed a predictable SoC platform template that removed all interference between applications through resource reservations. However, their SoC does not include caches, sharing of caches and coherence. Further, none of these works studies the predictability of HCC mechanisms when integrating hardware accelerators with different coherence modes with a predictable multicore system.

**Coherence for heterogeneous platforms.** There are various standards and commercial solutions that support coherent data sharing among CPU clusters and accelerators. For example, ARM's AMBA standard [17] has the AXI Coherence Extension (ACE) for full coherence in accelerators and accelerator coherency port (ACP) for I/O coherency in accelerators. The CXL standard [1] utilizes PCIe physical layer to provide coherence across the host processor and the accelerators, supporting both fully-coherent and one-way coherent accelerators. The industrial adoption of accelerator coherence has resulted in concrete platforms such as NVIDIA's Tegra [2] that features I/O coherence between the processor and the GPU, Xilinx's Zynq Ultrascale+ [5] that adopts the AMBA standards providing coherence between ARM cores and the FPGA, and Intel's Agilex [7] that supports coherence between the x86 cores and the FPGA with CXL. Research projects such as embedded scaleable platform (ESP) [14] integrates accelerators with fully-coherent or non-coherent modes, where Cohmeleon [19] further enables dynamic coherence mode changes. Other works [4, 6] provide ways to integrate accelerators, such as GPUs, into a fully-coherent multicore systems, and enable fully-coherent data sharing among cores and the accelerators. However, these approaches do not consider predictable HCC 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. We deploy some minimal architectural and cache coherence protocol changes from conventional ASoCs to prevent unbounded WCLs.

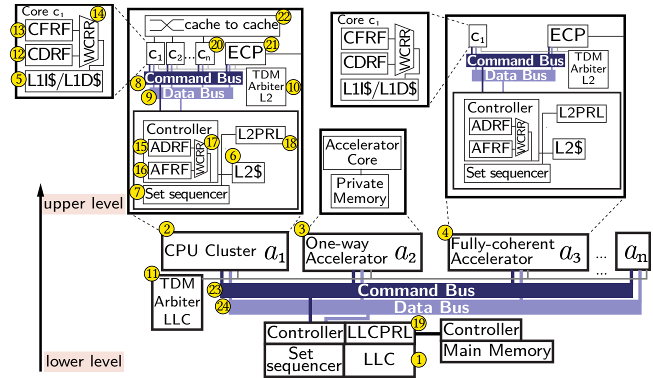


**Figure 1: Diagram of system model**

## 3 SYSTEM MODEL

**PASoC setup.** We employ Figure 1 to guide our system model. We assume our predictable accelerator-rich SoC to have $N_A$ agents that share a LLC (1). An agent can be one of the following: (1) a cluster of homogeneous predictable cores [8] (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 one-way coherent agent (OCA). Accelerators have only one processing element in it whereas a CPU cluster have multiple predictable cores. For brevity, we assume PASoC only supports one CPU cluster and $N_A - 1$ accelerators, and every agent can make only one outstanding memory request to the LLC.

**Coherent accelerator configurations.** We support fully-coherent and one-way coherent agents as described in this section.

**Coherence protocols.** We use state-of-the-art predictable coherence protocols to construct PASoC. Within the CPU cluster, we use

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, “PASoC: A Predictable Accelerator-Rich SoC,” in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

a predictable cache coherence protocol called linear coherence [13]. Between the agents, we use PMSI [10]. The main modifications in PMSI require the OCA to force write-through stores and ignore any coherence activity on the shared command bus.

**Cache hierarchy, inclusion policy and set sequencer.**

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

*Cache Inclusion.* LLC is inclusive of all L2 caches and L2 is inclusive of 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 eviction of cache lines for the same address in upper-level caches. This is called *back-invalidation* (BI). For our 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.

*Set Sequencer.* We employ the approach in [18] 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. 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 was released for an older request. This eliminates the unbounded WCL scenario.

**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 co-ordinates 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. We use work conserving TDM arbitration [11] $TDM_{LLC}$ (10) and $TDM_{L2}$ (11) to arbitrate accesses on the shared command bus to LLC and L2, respectively. We allocate one slot for each requester 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.

**FIFOs and PRLUTs.** We use two FIFOs to buffer incoming messages at each L1 and L2 cache controllers: a Demand request FIFO (DRF) and a Forward Response FIFO (FRF). A DRF buffers incoming read, write, and write-back requests originating from a core or agent to L2 or LLC respectively. A FRF buffers the write-back responses that the core or agent sends to L2 or LLC, respectively. These write-back responses from L1 cache controller are to the requests forwarded from other cores within the cluster, or from other external agents in the PASoC. The write-back responses from 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 [10]. Note that, an OCA does not have a FRF because it does not respond to any coherence activity in the system.

*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, which is buffered in *ADRF*. *AFRF* is of size $N_A$.

*PRLUTs.* At the L2 within the FCA, and the LLC, we use a FIFO arbitration between pending requests to the same cache line. We use a look-up table to queue pending requests *(PRLUT)* [10]. The PRLUT queues requests in the order of their arrival at the corresponding cache. We call the PRLUT at the L2, *L2PRL* (18), and at LLC, *LLCPRL* (19), respectively.

**Fully-coherent agent (FCA).** A FCA consists of $n_c$ homogeneous cache-coherent predictable cores [8, 13] (20) and an *external coherency port (ECP)* (21) that allows it to receive commands from agents external to the FCA. 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. The direct C2C data bus only connects the L1s [13, 17]. We allocate $n_c + 1$ 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. We further assume that there can only be one pending memory request from a core or external agent to L2.

**One-way coherent agent (OCA).** Similar to [19], we support a one-way coherent accelerator that does not have a private cache and is coherent with other cache-coherent agents in the system. Thus, a read from an OCA retrieves the most up-to-date data. On a write, the OCA updates the contents in the LLC. Note that a request from an OCA can cause coherence activity for other agents; however, the OCA does not respond to any coherence activity itself.

## 4 WORST-CASE LATENCY ANALYSIS

We 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 agent with only one core in it, and a one-way coherent agent.

### 4.1 WCL components

We begin by defining the WCL of components that we use to determine the WCL analysis of a memory request. These latency components represent the WCL for accessing shared caches such

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, “PASoC: A Predictable Accelerator-Rich SoC,” in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

as the L2 and LLC or for granting access to the shared bus by the arbitration. Due to space constraints, we only describe the intuition behind the proofs for the worst-case formulations.

**Worst-case TDM arbitration latency.** We use TDM arbitration 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 [10].

Lemma 4.1. *The WCL of a requester accessing a shared cache using TDM arbitration is given by*

$$WCL_{TDMArb}(u, s) = u \times s, \quad (1)$$

*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. Given that we allocate one slot per requester, the WCL is simply the product of the number of requesters and the slot width.

**Worst-case intra-core arbitration latency.** As described in Section 3, when a demand request and a forward response are both ready in their corresponding FIFOs, the WCRR arbitration 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 is termed intra-core arbitration latency.

Lemma 4.2. *The WCL of intra-core arbitration of a request from a core in a FCA is given by $WCL_{intraCoreArb}(y) = y$ where $y$ is the number of cycles for a WCRR round.*

We assume that when the message is buffered in a FIFO (DRF or FRF), in the worst case, the WCRR selects a message from the other FIFO. Hence, it has to wait for its next WCRR round to place the message on the shared bus. 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 4.1.

**Worst-case back invalidation latency.** As described in Section 3, 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. 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 4.3. *The WCL of BI at a shared cache is given by*

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

*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 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 4.1 and 4.2, 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.

**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 our system model that we deploy a set sequencer [18] (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 [18]:

Lemma 4.4. *The WCL of a replacement at a shared cache is*

$$WCL_{repl}(u, v, x, m, w) = u \times (WCL_{BI}(v, x) + w + m), \quad (3)$$

*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.*

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 [18]. 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. We refer readers to [18] for a complete proof.

## 4.2 WCL of request from a core in a CPU cluster

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 4.4) 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 and LLC, in Lemma 4.5. We explain the Lemma with the help of a timing diagram in Figure 2.

Lemma 4.5. *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 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 4.5 to both the L2 and LLC caches.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, "PASoC: A Predictable Accelerator-Rich SoC," in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

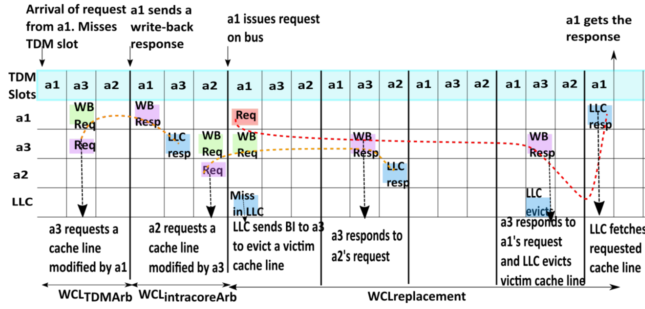


**Figure 2: Timing diagram explaining the WCL of a demand request from a1, in a system with three cores accessing the shared cache.**

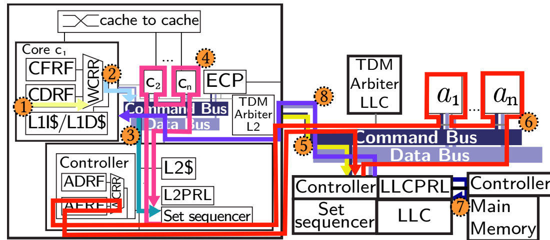


**Figure 3: Critical instance scenario.**

Corollary 4.6. *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 4.6 is that a demand request that hits in 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 4.5 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 example in Figure 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 (1) causing the demand request to experience the WCL of intra-core arbitration. 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 $TDM_{L2}$ grants the request access to the L2, the request is enqueued in *ADRF* at L2 (3). 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. 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.

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 4.5 (5-8).

We present the WCL a demand request experiences at the LLC in Lemma 4.7. $sw_x$ is the slot width of $TDM_x$. $L_{mem}$ is the WCL to access main memory.

Lemma 4.7. *The WCL of a demand request processed at the LLC is*

$$WCL_{DreqLLC} = WCL_{TDMArb}(N_A, SW_{LLC}) + WCL_{intraCoreArb}(N_A \times SW_{LLC}) + WCL_{repl}(N_A, N_A, N_A \times SW_{LLC}, L_{mem}, L_{mem}), \quad (4)$$

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

We also derive the WCL for performing a write-back from the L2 to the LLC, as required in Lemma 4.4, 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 4.8. *The WCL of a write-back request from the L2 to the LLC is given by*

$$WCL_{wbLLC} = WCL_{intraCoreArb}(N_A \times SW_{LLC}) + WCL_{TDMArb}(N_A, SW_{LLC}), \quad (5)$$

We present Theorem 4.9 that gives the WCL of a demand request that misses in the L1 of a core.

Theorem 4.9. *The WCL of a demand request sent from L1 is*

$$WCL_{totalFCCReq} = WCL_{TDMArb}(n_c + 1, SW_{L2}) + WCL_{intraCoreArb}((n_c + 1)SW_{L2}) + WCL_{repl}(n_c{+}N_A, n_c{+}N_A, (n_c{+}1)SW_{L2}, WCL_{DreqLLC}, WCL_{wbLLC}), \quad (6)$$

*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 replacement latency at L2, there are $n_c + N_A$ cores requesting a cache line in L2, each back-invalidation may be stalled by $n_c + N_A$ write-back responses 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 ($n_c = 1$).

## 4.3 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 4.10.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1145/3576914.3587496

S. Tadepalli, Z. Wu, and H. Patel, "PASoC: A Predictable Accelerator-Rich SoC," in Proceedings of Cyber-Physical Systems and Internet of Things Week 2023, New York, NY, USA, 2023, pp. 325–330. doi: 10.1145/3576914.3587496.

THEOREM 4.10. *The WCL of a demand request from an OCA is given by*

$$WCL_{totalOCAReq} = WCL_{TDMArb}(N_A, SW_{LLC}) + WCL_{repl}(N_A, N_A, N_A \times SW_{LLC}, L_{mem}, L_{mem}). \quad (7)$$

This follows directly from Lemma 4.5. 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 my cause coherence activity in other coherent agents as captured by the WCL of a replacement, which includes the WCL of back-invalidations.

## 4.4 Observations

PASoC is our attempt at employing state-of-the-art research in modelling a predictable accelerator-rich SoC with coherent accelerators. We find that this effort allows us to make observations that may be beneficial for a future version of PASoC. Specifically, we make three observations that we hope to further explore.

**Back-invalidations.** Our WCL analysis revealed that a large contributor to the WCL was back-invalidations. This is evident when we expand equations $WCL_{totalFCCReq}$ and $WCL_{totalOCAReq}$, 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.

**OCA vs. FCA.** Our WCL analysis further revealed that the WCL of a demand request originating from an OCA ($WCL_{totalOCAReq}$) is lower compared to one from an FCA ($WCL_{DreqLLC}$). This is because an OCA 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 $WCL_{intraCoreArb}$ latency that would appear in the WCL for FCA. While this indicates the benefit of having an OCA with respect to the WCL, it is a small reduction in the WCL.

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 agents' activities must also be removed from the transitions for the OCA. Further, the protocol must be extended to support coherence activities as a result from OCA's write-through to the LLC.

**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 units (GPUs). 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.

# 5 CONCLUSION

We present 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. We also provide 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] 2023. *Compute Express Link*. Retrieved February 3, 2023 from https://www.computeexpresslink.org/
[2] 2023. Cuda for Tegra. Retrieved Feb 9, 2023 from https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html
[3] 2023. *NVIDIA AGX Orin SoC*. Retrieved February 3, 2023 from https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/
[4] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 261–274.
[5] Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. 2015. UltraScale+ MPSoC and FPGA families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–37.
[6] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 155–166.
[7] Ilya K Ganusov, Mahesh A Iyer, Ning Cheng, and Alon Meisler. 2020. Agilex™ generation of intel® fpgas. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–26.
[8] Sebastian Hahn and Jan Reineke. 2018. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *RTSS*.
[9] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. 2009. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14, 1 (2009), 1–24.
[10] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. 2017. Predictable Cache Coherence for Multi-core Real-Time Systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 235–246.
[11] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet. 2020. Work-conserving dynamic time-division multiplexing for multi-criticality systems. *Real-Time Systems* 56, 2 (2020), 124–170.
[12] Salah Hessien and Mohamed Hassan. 2022. PISCOT: A Pipelined Split-Transaction COTS-Coherent Bus for Multi-Core Real-Time Systems. *ACM Trans. Embed. Comput. Syst.* (jul 2022).
[13] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. 2021. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *IEEE Trans. Comput.* 70, 12 (2021), 2098–2111.
[14] Paolo Mantovani et al. 2020. Agile SoC Development with Open ESP *(ICCAD '20)*. ACM, New York, NY, USA, Article 96, 9 pages.
[15] Francesco Restuccia and Alessandro Biondi. 2021. Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. 441–454.
[16] Yakun Sophia Shao and David Brooks. 2015. *Research infrastructures for hardware accelerators*. Morgan & Claypool Publishers.
[17] Ashley Stevens. 2011. *Introduction to AMBA® 4 ACE™ and big. LITTLE™ Processing Technology.*
[18] Zhuanhao Wu and Hiren Patel. 2022. Predictable Sharing of Last-Level Cache Partitions for Multi-Core Safety-Critical Systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. ACM, New York, NY, USA, 1273–1278.
[19] Joseph Zuckerman et al. 2021. Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. ACM, New York, NY, USA, 350–365.