Consistency-aware and Predictable Memory Processing for Safety-critical Out-of-order Multicores

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

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

***Abstract*—We introduce an approach that facilitates predictable processing of multiple outstanding memory requests in safety-critical out-of-order multicores. A primary challenge addressed by this work is ensuring that multiple outstanding memory requests maintain a memory consistent model while minimizing a low worst-case latency. Adhering to a memory consistency model is crucial for ensuring the correctness of programs executed on such multicores. Our approach, termed predictable processing of multiple outstanding requests (PPP), leverages micro-architectural enhancements to tackle this challenge. Experimental results show that PPP delivers speedups of 2.07×, 2.79×, and 3.38× over serialization for 2, 4, and 8 cores, while maintaining the worst-case latency.**

I. Introduction

Multicore systems provide high performance while minimizing size, weight, and power in today's safety-critical embedded systems [1]–[3]. The implementation of multicore systems for safety-critical embedded systems must strike a balance between ensuring timing predictability—specifically, the ability to establish worst-case execution time (WCET) bounds for critical tasks—and delivering high performance to satisfy the application's quality-of-service requirements. This is typically accomplished through meticulous design of the multicore microarchitecture to accommodate a range of performance-enhancing techniques while guaranteeing predictability. These include sophisticated in-order pipelines, cache hierarchies, hardware-managed cache coherence [4]–[7], main memory controllers [8], and interconnects [9], [10]. However, we observe that many of these multicores allow each core to issue only one outstanding memory request to the memory hierarchy [4]–[7]. We suspect it is to maintain predictability and deliver worst-case latency (WCL) bounds on memory requests. Unfortunately, this limitation presents three significant drawbacks. (1) Non-blocking caches cannot be used because multiple requests from a core cannot be issued. (2) Out-of-order cores cannot exploit memory operations being executed out-of-order because they are eventually completed in-order. (3) The overall memory-level parallelism within the memory hierarchy remains unexploited. These drawbacks limit the implementation of several crucial techniques designed to enhance the performance of multicore systems for safety-critical embedded systems.

In this work, we overcome these drawbacks by building multicores that adopt out-of-order pipelines for cores, a predictable cache coherence mechanism between cores' caches, non-blocking caches for higher performance, and timing predictability with a low WCL on memory accesses. By timing predictability, we mean **delivering a guaranteed upper bound on WCL of memory accesses**. To achieve this, we enable cores to issue multiple outstanding memory requests, allowing them to take full advantage of various micro-architectural performance techniques. However, two key challenges must be addressed. (1) We need to ensure that multithreaded programs honour a memory consistency model, which was not necessary when restricting each core to only one outstanding request per core as done in state-of-the-art related works. (2) We must design a technique that guarantees a WCL, and offers a lower WCL when compared to employing existing techniques to ensure memory consistency. We address these challenges in **p**redictable **p**rocessing of multi**p**le outstanding requests (PPP), which enables out-of-order cores, non-blocking caches, and a hardware cache coherence mechanism while preserving the total store order (TSO) memory consistency model and timing predictability. Our main contributions in this work are as follows.

1) We identify that introducing memory-level parallelism naively leads to memory consistency model violations.
2) We examine approaches used in general-purpose systems to ensure the memory consistency model and show that those approaches result in a large WCL empirically.
3) We propose micro-architecture extensions, collectively called PPP, that ensure both TSO and *timing predictability*. The resulting multicores have out-of-order pipelines, non-blocking caches, and predictable hardware cache coherence. PPP integrates hardware that monitors for potential memory consistency model violations. On detecting a violation, PPP delays the request until the request does not causes memory consistency model violations.
4) We provide a WCL analysis of memory accesses for PPP.
5) We implement PPP in the gem5 [11] simulator, and compare PPP against two mechanisms that enforce TSO: serialization of memory requests, and retry memory requests.

Benefits of PPP. PPP offers three benefits over state-of-the-art safety-critical multicores.

1) **Performance.** PPP directly addresses the performance limitations in state-of-the-art safety-critical multicores caused by only allowing a single outstanding memory request per core in the memory hierarchy. PPP enables multiple outstanding memory requests per core for safety-critical multicores. Our results show that PPP achieves a speedup of $2.07\times$, $2.79\times$, and $3.38\times$ compared to serialization for 2, 4, and 8 cores configurations, while maintaining the analytical WCL bound. Moreover, PPP offers comparable performance to general-purpose implementations with only a 5% performance loss.
2) **Functional Correctness.** PPP enforces sequential consistency and total store order memory models. This functional correctness requirement is non-negotiable even in general-purpose multicores; failure in doing so, such as integrating prior works naively [12], is a hardware bug. This is because failing to honour a consistency model results in unexpected outputs for multithreaded programs.
3) **Low Worst-Case Latency.** PPP delivers a lower worst-case latency than its counterparts while offering performance comparable to general-purpose approaches. We observed the worst-case latency of retry and general-purpose approaches are up to 6.0× and 5.4× of the observed worst-case latency of PPP.

II. Background

A. Predictable hardware cache coherence

Predictable hardware cache coherence is a state-of-the-art data-sharing mechanism for safety-critical embedded system multicores, allowing cores to privately cache data with a consistent view of the data across all cores and offering high performance [4], [7], [12], [13]. Predictable hardware cache coherence ensures each memory request has a bounded worst-case latency [4], [7], [12]. Predictable hardware cache coherence operates at the granularity of cache lines, typically 32-byte or 64-byte. We refer to the cache line whose first byte is at address A as: *cache line A* or simply *A*. An example of predictable hardware cache coherence is the MSI-based protocol, PMSI* [14]. PMSI* assigns each cache line a state: **M**odified, **S**hared, or **I**nvalid. A cache line in **M** state is privately cached by only one core and this core can read from or write to the cache line. A cache line in **S** state can be privately cached by one or more cores. Finally, a cache line is in **I** state if the core does not privately cache the cache line.

B. Memory consistency model

A *memory consistency model* describes the correctness of shared memory accesses for multithreaded programs. Specifically, the memory consistency model identifies the values loads return when executing multithreaded programs, and the eventual state of the memory. Consequently, a memory consistency model defines the expected order of memory accesses seen by all cores. To help our exposition, we introduce several definitions that formalize sequential consistency (SC) and TSO memory consistency models. This formalization will help readers understand the challenge in enforcing memory consistency models in multicore systems, and allows us to provide an understanding of correctness behind PPP. We will use example programs in Figure 1 to aid the illustration.

c_1	c_2
(i1) $[C] = 1$	(i4) $[B] = 1$
(i2) $r_1 = [A]$	(i5) $[A] = 1$
(i3) $r_2 = [B]$	(i6) $r_3 = [C]$
Forbidden in SC and TSO:	
(a) $r_1 = 1, r_2 = 0, r_3 = 0$	

c_1	c_2
(i1) $[C] = 1$	(i4) $[B] = 1$
(i2) $r_1 = [C]$	(i5) $[A] = 1$
(i3) $r_2 = [B]$	(i6) $r_3 = [C]$
Forbidden in SC, allowed in TSO:	
(b) $r_1 = 1, r_2 = 0, r_3 = 0$	

Fig. 1: (a) The example shows results disallowed in SC and TSO. (b) Forwarding value from (i1) to (i2) causes execution result disallowed in SC.

Definitions and Terminology. We focus on memory operations of a multithreaded program when running on a set of N cores $\mathbb{C} = \{c_1, c_2, \cdots, c_N\}$. Hence, we denote the set of load and store operations of a multithreaded program as $\mathbb{L}$ and $\mathbb{S}$, respectively. To represent initial values in memory, we augment $\mathbb{S}$ with initial stores s_0^a that populate the initial state of the memory at the address a. Throughout this paper, we assume the initial state in memory to be 0 for all addresses, unless specified otherwise. For a memory operation $o \in \mathbb{O}$, $\mathbb{S}|_o$ identifies the set of stores to the address targeted by o. We start by describing the program order of a program executed by a specific core c in Definition 1. The main intuition behind this definition is that $\xrightarrow{p(c)}$ is a total order on the operations issued by core c, allowing us to determine which operation comes before another in the program.

Definition 1 (Program order per core)**.** *The program order of a program to be executed on core c, denoted as* $\xrightarrow{p(c)}$*, is defined as:*

1) $\xrightarrow{p(c)}$ *is a total order, and*
2) $\forall o_1, o_2 \in \mathbb{O}, (o_1.core = o_2.core = c) \wedge (o_1 \neq o_2) \implies o_1 \xrightarrow{p(c)} o_2 \vee o_2 \xrightarrow{p(c)} o_1$. $o_1.core$ *is the core issuing* o_1.

We use $\xrightarrow{p(o)}$ for a memory operation $o \in \mathbb{O}$ to mean the program order of the core that issued the memory operation. We construct the *program order* by combining $\xrightarrow{p(c)}$ from individual cores, as defined in Definition 2. The program order allows us to discuss memory operations from any cores.

Definition 2 (Program order)**.** *Two operations* $o_1, o_2 \in \mathbb{O}$ *are in program order* $o_1 \xrightarrow{p} o_2$ *iff* $\exists c \in \mathbb{C}, o_1 \xrightarrow{p(c)} o_2$.

Next, we introduce the notion of a memory order in Definition 3 and the closest store in Definition 4. The memory order identifies the order of memory operations that are visible to all cores. Notice that program order is a per-core total order, and memory order is a total order over all operations from all cores; thus, some interleaving of all the memory operations.

Definition 3 (Memory order)**.** *A memory order,* $\xrightarrow{m}$*, is a total order on memory operations* $\mathbb{O}$.

Consider the example in Figure 1(a), which shows a multithreaded program. Recall that all initial values in memory are 0. Core c_1 executes one store and two load operations: (i1) writes the value 1 to memory address C, (i2) reads from memory address A into register r_1, and (i2) reads from B into r_2. Core c_2 executes two stores and a load: (i4) and (i5) write the value 1 to memory addresses B and A, respectively, and (i6) reads C into register r_3. Here, (i1)$\xrightarrow{p}$(i2)$\xrightarrow{p}$(i3) for c_1 and (i4)$\xrightarrow{p}$(i5)$\xrightarrow{p}$(i6) for c_2 illustrate the program orders. For the memory order, an execution where all operations of c_1 execute in program order followed by all operations of c_2, also in program order, results in (i1)$\xrightarrow{m}$(i2)$\xrightarrow{m}$(i3)$\xrightarrow{m}$(i4)$\xrightarrow{m}$(i5)$\xrightarrow{m}$(i6). This memory order returns the value 0 for r_1, r_2, and 1 for r_3. Another execution may execute c_2's operations in program order before c_1's resulting in the value 1 for r_1, r_2 and 0 for r_3. The memory order for the second possible execution is (i4)$\xrightarrow{m}$(i5)$\xrightarrow{m}$(i6)$\xrightarrow{m}$(i1)$\xrightarrow{m}$(i2)$\xrightarrow{m}$(i3). Both sets of values for the registers are possible depending on how memory operations are viewed by all cores. We compactly represent the operations in program order, and memory order using $\langle . \rangle_p$ and $\langle . \rangle_m$. For example, $\langle i1, i2, i3 \rangle_p$ is the program order of operations for core c_1, and $\langle i4, i5, i6, i1, i2, i3 \rangle_m$ is the memory order for the second possible execution.

A memory consistency model has a value function that identifies the values returned when a core performs a load. Therefore, the value function maps each load to the store that it obtains its value from. Defining the value function requires defining the *closest store*, which identifies the store that wrote the value that a load reads from. We formalize the *closest store* to a load operation in Definition 4. This definition takes a total order t and the load l as inputs, and identifies the store from which the load operation obtains its value. If such a store does not exist, then the load takes on the initial value. When we use a total order as a parameter or an argument, we refer to it without the underlined arrow. Thus, we use t for the total order $\xrightarrow{t}$, p for $\xrightarrow{p}$, $p(o)$ for $\xrightarrow{p(o)}$, and m for $\xrightarrow{m}$.

Definition 4 (Closest store)**.** *Given a total order $\xrightarrow{t}$ on memory operations, the **closest store** to a load operation $l \in \mathbb{L}$ is defined as:*

$$C(t,l) = \begin{cases} s, s \in \mathbb{S}|_l \wedge \forall s' \in \mathbb{S}|_l, s' \neq s \implies \neg(s \xrightarrow{t} s' \xrightarrow{t} l), \\ s_0^{l.addr}, \textit{otherwise} \end{cases} \tag{1}$$

Here, the closest store operation s to a load l is the store where $s \xrightarrow{t} l$, and there is no other store s' to the same address that occurs between s and l in the total order.

Using the closest store definition, we can define the value functions. However, different memory consistency models have different value functions based on the constraints imposed by the particular memory consistency model. In Definition 5, we show that for SC, the load returns the value from the most recent store in the memory order.

Definition 5 (Value function for SC)**.** *Given a program order $\xrightarrow{p}$, and a memory order on $\mathbb{O}$ of the program, $\xrightarrow{m}$, the value function for the sequential consistency memory consistency model is defined as:*

$$V_{SC}(p,m,l) = C(m,l)$$

For TSO, as described in Definition 6, a load gets its value from the most recent store to the same address. However, this store may be in the store buffer of the core that issued the store; thus, its value would not have been propagated to the memory hierarchy. This means that other cores would observe the older value, but loads to the same address from the core that issued the store would return the value written to by the store. Such store is captured by $C(p(l), l)$, which is the closest store to l in the program order of the core issuing l. Note that TSO enables the store buffer optimization, which is the reason for the difference between SC and TSO.

Definition 6 (Value function for TSO)**.** *Given a program order $\xrightarrow{p}$, and a memory order on $\mathbb{O}$ of the program, $\xrightarrow{m}$, its value function is defined as follows:*

$$V_{TSO}(p,m,l) = \begin{cases} C(p(l),l), & l \xrightarrow{m} C(p(l),l) \\ C(m,l), & \textit{otherwise} \end{cases}.$$

A multithreaded program can have multiple memory orders. This is because cores see values in the memory via loads; however, the order in which these loads occur can vary based on their executions by their respective cores. Even so, two or more memory orders can have loads return the same values for all load operations. We determine these memory orders to be equivalent. Definition 7 formalizes this insight as an equivalence relation. This requirement complies with SC and TSO, and complies with the architecture assumed in this work.

Definition 7 (Memory order equivalence)**.** *Given a memory consistency model $k \in \{SC, TSO\}$, a program order $\xrightarrow{p}$, and the value function, V_k, two memory orders $\xrightarrow{m_1}$ and $\xrightarrow{m_2}$ are **equivalent** iff they return the same values for all loads, and the orders of stores to the same address are the same. Formally, $m_1 \equiv_k m_2$ iff*

1) $\forall l \in \mathbb{L}$, $V_k(p, m_1, l) = V_k(p, m_2, l)$, and
2) $\forall s_1, s_2 \in \mathbb{S}$, $s_1 \xrightarrow{m_1} s_2 \iff s_1 \xrightarrow{m_2} s_2$.

We now define the meaning of a memory order honouring a particular memory consistency model. Definition 8 shows that a memory order is compliant with SC when operations in program order also appear in the same program order in the memory order. We also say the memory order is in SC to denote that it is compliant with SC. There are four possible program orderings of operations: Load followed by a Load (load-to-load), Load followed by a Store (load-to-store), Store followed by a Load (store-to-load), and Store followed by a Store (store-to-store). A memory order is in SC when values returned by the memory order are the same as an equivalent memory order where all four program orderings are preserved. Consequently, the interleaving of memory operations in the memory order preserves the orderings of each individual core's program, and the values returned are the same.

Definition 8 (Compliance with SC)**.** *Given memory operations* $\mathbb{O}$ *and a program order* $\xrightarrow{p}$ *, a memory order* $\xrightarrow{m}$ *of* $\mathbb{O}$ *complies with the SC iff*

$$\exists m', m \equiv_{SC} m', \forall o_1, o_2 \in \mathbb{O}, o_1 \xrightarrow{p} o_2 \implies o_1 \xrightarrow{m'} o_2.$$

Recall that TSO allows store buffer optimization, where a load operation obtains the value written by a store from the same core (to the same address) before the store's value is written to the memory hierarchy. Hence, unlike SC, TSO does not require the store-to-load order to be preserved in the memory order; instead, TSO ensures the orderings of load-to-load, load-to-store, and store-to-store.

Definition 9 (Compliance with TSO)**.** *Given memory operations* $\mathbb{O} = \mathbb{S} \cup \mathbb{L}$ *and a program order* $\xrightarrow{p}$ *, a memory order* $\xrightarrow{m}$ *of* $\mathbb{O}$ *complies with TSO iff*

$$\exists m', m \equiv_{TSO} m',$$
$$\forall o_1, o_2 \in \mathbb{O}, o_1 \xrightarrow{p} o_2 \implies o_1 \xrightarrow{m'} o_2 \vee (o_1 \in \mathbb{S} \wedge o_2 \in \mathbb{L}).$$

In Definitions 8 (rsp. 9), the compliance with SC (rsp. TSO) requires the existence of an equivalent memory order m' in which the loads are in program order. Hence, under memory order equivalence, load-to-load ordering is preserved in both SC and TSO. This specification allows us to study the execution when an out-of-order core reorders loads for higher performance; meanwhile, incorrect reordering of memory operations violating memory consistency models, i.e. memory orders producing non-consistent values, are excluded.

Example of TSO. Consider the example shown in Figure 1(b), where the *difference* with the program in Figure 1(a) is that (i2) in c_1 loads from C instead of A. Recall that TSO relaxes the ordering of store-to-load operations, and allows loads to read values from the store buffer to dependent loads from the same core. Let us assume that an execution of this example results in a memory order of $\langle i6, i2, i3, i1, i4, i5\rangle_m$ and the value for r_1 is 1 and 0 for r_2 and r_3. This memory order is TSO because all program orders except for store-to-load are preserved with the following execution. Suppose that core c_1 executes all its operations in program order, causing the store of C, (i1), to be inserted in the store buffer. Since (i2) loads C, it can read the value 1 from the store buffer. (i3) can then proceed before the store to C has propagated to the memory hierarchy. Similarly, core c_2's (i4) and (i5) are entered in the store buffer for core c_2. Then, (i6) can proceed before A, B get propagated to the memory hierarchy. Hence, (i3) and (i6) can proceed before the stores in the store buffers write their values to the memory, causing the loads to return the value 0. This memory order is TSO, but not SC. It is not SC because (i2) is older (comes before) in the memory order than (i1).

Disallowed load reordering in TSO and SC. We wish to allow reordering of loads in hardware, but only as long as the reorderings comply with the memory consistency model. When a pair of loads are reordered in a memory order, and such reordering violates the memory consistency model, we refer to this as a disallowed load reordering. To understand how a violation occurs when such reordering happens, consider again the example in Figure 1(a). We will explain how the value of 1 for r_1 and 0 for r_2 will result in non-compliance with TSO. The central observation is that (i2) and (i3), and (i4) and (i5) must execute in program order because store-to-store ordering is maintained in TSO. If (i2) returns the value 1, then (i5) must have written 1 to A. Also, since (i4) and (i5) must execute in program order, (i4) will write 1 to B. Thus, (i3) must return 1, but it does not. The contradiction is evident since (i3) executing after (i2) reads B, which should return the value 1. Note that this violation occurs irrespective of the value taken by r_3. One memory order leading to such a result is $\langle i3, i6, i4, i5, i1, i2\rangle_m$. Notice how (i2) and (i3) are reordered, leading to (i3) reading 0 from B. Hence, $r_1 = 1$ and $r_2 = 0$ are not allowed in TSO. This execution result triggers disallowed load reordering of (i2) and (i3) for TSO. A similar argument can be made for SC.

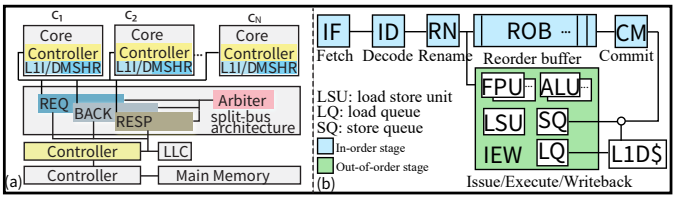


Fig. 2: System model for PPP assumes that N out-of-order cores and the LLC are interconnected with a split-bus.

C. Key takeaways

All state-of-the-art research on predictable hardware cache coherence for safety-critical embedded systems allow only one memory request per core. This disallows an architecture that exploits memory-level parallelism. This work contends that predictable hardware cache coherence can improve performance by exploiting memory-level parallelism exposed by out-of-order cores. However, the predictable hardware cache coherence must be aware of the memory consistency model; both in its implementation and the ensuing WCL analysis. Further, the selection technique must aim to reduce the WCL. Our work is a first step towards integrating out-of-order pipelines with predictable hardware cache coherence while guaranteeing both timing predictability requirements and performance.

III. System Model

Our work considers a multicore system with N cores denoted as $c_1, c_2, \ldots, c_N$ as shown in Figure 2(a). This multicore system uses the state-of-the-art cache coherence designed for safety-critical systems called PMSI* [6]. PMSI* uses a split-transaction bus architecture as done with prior state-of-the-art efforts [4], [6], [7], [12] to interconnect the multiple cores.

Core pipeline model. Figure 2(b) illustrates an out-of-order core pipeline. The execution of an instruction begins by fetching and decoding instructions in program order during the fetch and decode stages. Decoded instructions are then sent to the rename stage, where register renaming occurs. If hardware resources such as arithmetic logic units, floating

point units, or the load-store unit for memory instructions are unavailable, the pipeline stalls in the rename stage. The load-store unit handles memory instructions through its load and store queues. Load and store instructions are buffered in the load queue and store queue in program order, respectively, until they are ready to be sent to the cache. Renamed instructions enter the issue/execute/writeback stage, where out-of-order execution occurs once dependencies are met. An instruction entering issue/execute/writeback stage from rename stage is also added to the reorder buffer, a first-in-first-out queue that helps committing instructions in program order. For arithmetic instructions, operands must be ready, while memory instructions require address calculation and, for stores, the data to be stored. The load-store unit calculates addresses and handles cache requests for loads, while stores prepare the address and data but delay sending to the cache. After issue/execute/writeback, instructions wait in the reorder buffer in the commit stage. The pipeline commits instructions in program order. The core *retires* a store instruction by sending it to the cache *after* the store instruction is committed and all older store instructions in the SQ are retired. This means that stores are processed in their program order.

Memory hierarchy – Caches, LLC, main memory. We assume a memory hierarchy with L1 instruction and data caches, a shared last-level cache (LLC), and a main memory. Each core has a private L1 instruction and data cache. Both L1 caches are managed by an L1 cache controller, and the LLC is managed by an LLC controller. The L1 cache controller also manages the *miss status holding register (MSHR)* to allow non-blocking accesses from the core. We use L1 cache and private cache interchangeably. The LLC is exclusive to the L1; and the LLC is banked to exploit memory-level parallelism. We assume that access to main memory has a bounded WCL [15].

Split-transaction bus architecture. The cores and LLC communicate via a split-transaction bus, like prior works that deliver high performance [4], [7], [12], [14]. The baseline configuration of the split-transaction bus includes two buses: (1) a request bus (REQ) allowing cores to send requests to the LLC and to snoop other cores' requests, and (2) a response bus (RESP) transfering responses from LLC and cores. The order of requests seen by all cores is governed by the REQ bus. Therefore, the broadcast order of requests on REQ determines the ordering of requests for the entire system. Note that when a core transfers data over RESP, another core can snoop the transferred data, enabling cache-to-cache transfers. PPP includes a Broadcast Acknowledgement Bus (BACK) for the cores to communicate with the arbiter to enforce the memory consistency model.

Arbiter – Oldest-first global arbitration. The access to REQ, RESP and the LLC follows the oldest-first global arbitration scheme inspired by [12]. In this arbitration scheme, the arbiter uses a work-conserving round-robin to pick the oldest request from each core's MSHR. The access to the LLC and RESP follows the same ordering of requests as that on the REQ bus: a work-conserving oldest-first global arbitration scheme.

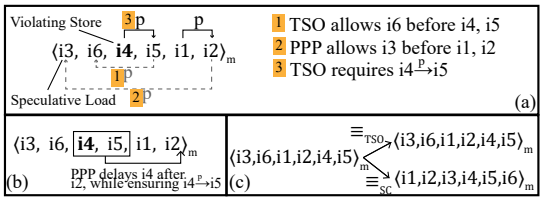


Fig. 3: (a) Illustration of DLRC. (b) PPP postpones violating store. (c) final memory order is equivalent to SC and TSO. Memory operations are from Figure 1(a).

IV. Key Intuition: Memory Consistency Model Violation and Correction with PPP

Out-of-order pipelines execute loads out of program order for higher performance. This is known as speculatively executing loads. Loads delay the execution of later dependent instructions, causing the pipeline to stall. Instead, if other loads can execute before earlier loads, then this improves the pipeline's utilization [13], [16], [17]. An important consequence of enabling speculative load execution is that the pipeline can benefit from using non-blocking caches [18]. A non-blocking cache allows cores to proceed executing later operations while earlier operations experience cache misses. However, speculatively executing load operations can violate the memory consistency model resulting in non-compliance with the memory consistency model.

Speculative load execution causing memory consistency model violation. We show that speculatively executing loads can result in a memory consistency model violation. We revisit the example in Figure 1(a), assuming the pipeline in Section III with non-blocking caches. Suppose that B and C are the only cache lines privately cached by c_1 and c_2, respectively. This means that loads from c_1 to B and from c_2 to C will be cache hits. Assume that (i1) and (i2) are stalled due to unmet dependencies or resource constraints. For instance, (i1) can stall in issue/execute/writeback stage because the address computation for C may have earlier dependent operations, or stall in the store buffer waiting to be drained. Similarly, (i2) can stall due to a cache miss. However, by using a non-blocking cache, if (i3)'s dependencies are met, then (i3) can speculatively execute before (i1) and (i2). Since B is cached by c_1, (i3) is a cache hit and returns the value 0. We term a load such as (i3) that executes out of program order a *speculative load*. Then, c_1 executes (i1) and (i2) in order, after their dependencies and resource constraints are resolved. For c_2, let us assume that (i4) and (i5) are stalled due to dependencies in computing the addresses A and B, respectively. Since TSO allows store-to-load to be executed out of program order, (i6) executes before (i4) and (i5). As C is cached by c_2, (i6) is a cache hit and returns the value 0. Then, c_2 retires (i4) and (i5) in order. The resulting execution produces $\langle i3, i6, i4, i5, i1, i2\rangle_m$. Note that since (i2) comes after (i5) in memory order, $r_1 = 1$; and $r_2 = 0$ due to the cache hit on the speculative load (i3). As we explained earlier in section II, this memory order is not SC nor TSO. This shows that speculatively executing loads out-of-

order can cause memory consistency model violations when using SC and TSO. Hence, an implementation that promotes speculative execution of loads must prevent such inconsistent results from occurring, and ensure they are not propagated to the architectural state.

Identifying disallowed load reordering conditions. We need to identify conditions under which executing loads out of order violate the memory consistency model. We notice that there are three necessary conditions, as shown below, which we call the ***d**isallowed **l**oad **r**eordering **c**onditions* (DLRC). These DLRC guide our design of PPP in the following way. PPP ensures that DLRC never occur; therefore, preventing any memory consistency model violations. In the subsequent exposition, we assume memory operations $\mathbb{O}$ with its program order $\xrightarrow{p}$, and we also assume a memory order $\xrightarrow{m}$. Additionally, we will focus on two load operations $l_1, l_2 \in \mathbb{L}$ where they access different addresses: $l_1.addr \neq l_2.addr$.

Definition 10 (DLRC)**.** $\xrightarrow{m}$ *violates the memory consistency model due to speculative load execution only if there exists two loads l_1 and l_2, and these three conditions are met:*

1) $\langle l_1, l_2 \rangle_p \wedge \langle l_2, l_1 \rangle_m$,
2) $\exists s_1, s_2 \in \mathbb{S}, \langle l_2, s_1, s_2, l_1 \rangle_m \wedge s_1.core \neq c \wedge s_2.core \neq c$,
3) $s_1 \in \mathbb{S}|_{l_2} \wedge s_2 \in \mathbb{S}|_{l_1}$,

where core c refers to the core issuing l_1 and l_2.

We refer to conditions in DLRC as DLRC 1, DLRC 2, and DLRC 3. DLRC 1 states two loads in program order are reordered in the memory order due to speculatively executing loads. DLRC 2 ensures there are at least two stores from other cores between the reordered pair of loads in the memory order. DLRC 3 requires that the store older in memory order writes to the address of the speculative load l_2, while the second store writes to the address of the the other load l_1. We show the correctness of DLRC in Lemma 11 by contradiction. Our proof assumes that stores are processed in order; this means that the memory order of two stores from the same core complies with their program order, as assumed in our system model.

Lemma 11 (DLRC correctness)**.** *A memory order m violates the memory consistency model due to speculative loads only if there exists two loads that satisfy the DLRC conditions.*

Proof. We prove by contradiction. Suppose for all pairs of loads, DLRC do not hold, then either DLRC 1, DLRC 2, or DLRC 3 fails. If DLRC 1 fails, it means that all loads in m follow their program order; hence, the memory order is compliant with the consistency model. Otherwise, it suffices to show that *when DLRC 2 or DLRC 3 fails and DLRC 1 holds*, we can construct a new memory order equivalent to memory order m that is compliant with the consistency model. We focus on two reordered loads l_1 and l_2 (i.e. $\langle l_2, l_1 \rangle_m$ and $\langle l_1, l_2 \rangle_p$) where there are *no same-core loads between l_1 and l_2 in the memory order m*. Such a pair of loads must *exist* as follows. DLRC 1 implies that there exists a pair of reordered loads l_u and l_v such that $\langle l_u, l_v \rangle_m$ and $\langle l_v, l_u \rangle_p$. If there are no same-core loads between l_u and l_v in m, we can pick $l_1 = l_u$ and $l_2 = l_v$. Otherwise, there exists another load l_w such that $\langle l_u, l_w, l_v \rangle_m$. If $\langle l_v, l_w \rangle_p$, we can pick l_w and l_v, where there are less same-core loads between $\langle l_w, l_v \rangle_m$ than between $\langle l_u, l_v \rangle_m$; this applies equally when $\langle l_w, l_v \rangle_p$ where we can pick l_w and l_v instead. Repeating this process yields a pair of reordered loads l_1 and l_2 with no same-core loads in between. Next, we use case analysis to examine the memory operations between l_1 and l_2 in m, and show that we can rearrange l_1 and l_2 in the memory order to get a new memory order m' that is compliant with the consistency model. The new memory order m' preserves the order in m for all memory operations other than l_1 and l_2.

Case ► Intermediary operations to addresses other than that of the loads. Suppose all operations between l_2 and l_1 access addresses that are different than l_2's and l_1's, hence failing DLRC 2. These intermediary operations do not affect the value returned by l_2 and l_1. Hence, l_2 and l_1 obtain their values from stores older than l_2 in the memory order. Therefore, we can simply swap l_1 and l_2 to obtain m' such that $\langle l_1, l_2 \rangle_{m'}$; m and m' are equivalent.

► **Case** $\langle l_2, s_1, l_1 \rangle_m$**.** Since l_2 reads the value before s_1 writes to it, the write performed by s_1 does not change the value returned by any of the loads. Note that l_1 reads from a different address than s_1. Thus, the resulting values the loads return are the same as the memory order where $\langle l_1, l_2, s_1 \rangle_{m'}$. The new memory order m' does not change the value function for both SC and TSO because the relative ordering of store operations are unchanged. Hence, $m' \equiv_{TSO} m$ and $m' \equiv_{SC} m$.

► **Case** $\langle l_2, s_2, l_1 \rangle_m$**.** If the intermediate operation is a store s_2 that writes to l_1's address, the exact same reasoning applies. The order $m' \equiv_{TSO} m$ and $m' \equiv_{SC} m$ where $\langle s_2, l_1, l_2 \rangle_{m'}$.

► **Case** $\langle l_2, s_2, s_1, l_1 \rangle_m$**.** Next, another possible memory order is $\langle l_2, s_2, s_1, l_1 \rangle_m$. l_2 loads a value from an older store than s_1 for TSO and SC, or the cloest store in program order for TSO; and the same applies to l_1, which loads a value from s_2 or the cloest store in program order for TSO. Hence, l_1 and l_2 still loads the same values in the equivalent memory order $\langle s_2, l_1, l_2, s_1 \rangle_{m'}$. The reason is that such movement does not change the relative order of stores, and hence the values returned by the loads remain.

► **Case** $\langle l_2, s_1, s_2, l_1 \rangle_m$**.** In this memory order, l_2 loads a value from an older store before s_1, while l_1 loads a value from s_2. Note that if this hold for m, it means *all DLRC hold*. Intuitively, l_2 must occur before s_1 for l_2 to return the value before the write of s_1; similarly, l_1 must be after s_2 for l_1 to return the value of the write by s_2. Therefore, there does not exist an equivalent memory order where l_1 and l_2 are in program order while returning the same values. This memory order is not in SC and TSO.

The $\xrightarrow{m'}$, as constructed with the case analysis, preserves all m' order for operations other than l_1 and l_2, all load values, and still fails DLRC 2 or DLRC 3. In addition, this $\xrightarrow{m'}$ has *one fewer* load reordering than $\xrightarrow{m}$. Note that altering the positions of l_1 and l_2 does not change the values returned

by other loads. Hence, after repeating this process, we can find an equivalent memory order with no reordered load by repeatedly reducing the number of reordered loads to zero. This contradicts the initial violation assumption, proving the lemma by contradiction. □

Our approach: Correcting DLRC in PPP. We allow load reordering even if the memory consistency model disallows it, but, when we recognize a potential violation, we correct it at runtime. Our key insight is that multiple memory orders conform to the memory consistency model, and if we are able to produce one of those equivalent and conforming memory orders, then the result also conforms. Previously, in section II, we showed that $\langle i3, i6, i4, i5, i1, i2\rangle_m$ is not TSO. This memory order reorders loads ((i3) before (i1) and (i2)), and allows store-to-load reorders with (i6) before (i4) and (i5). While the latter reordering is allowed under TSO, the load reordering is not allowed. Suppose that we are able to detect such a violating memory order before it actually happens, and in response, we produce a corrected memory order $\langle i3, i6, i1, i2, i4, i5\rangle_m$ instead. A key change in the corrected memory order is that the stores to B and A ((i4) and (i5)) occur after their loads ((i2) and (i3)). Hence, the loads return values that are not updated by stores (i4) and (i5). The corrected memory order is equivalent to $\langle i6, i1, i2, i3, i4, i5\rangle_m$ where loads are in program order, and only the store-to-load reordering is permitted; thus, complying with TSO.

PPP's main technique is in providing corrected memory orders when potential memory consistency model violations are identified. We present the main high-level steps that PPP uses to correct potential memory consistency model violations that arise by speculatively executing loads.

Step 1: At runtime, PPP speculatively detects if a possible memory consistency model violation may occur due to load reordering. This detection is based on the DLRC conditions. When the detection confirms a possible memory consistency model violation, there exists a store, called the *violating store*, that changes the value being returned by a *speculative load*.

Step 2: PPP checks if there are any loads prior to the speculative load in program order that have not yet returned their values. If this check returns true, then PPP delays the violating store until all loads older than and including the speculative load are visible to all cores.

We revisit the memory order $\langle i3, i6, i4, i5, i1, i2\rangle_m$ that violates TSO and show it in Figure 3. We mark the violating store as the store to B (i4) and the speculative load as the load to B (i3). (i3) is the speculative load because it appears before both (i1) and (i2) in the memory order. When (i4) is visible to all cores, each core executes step 1, and detects if a memory consistency model violation may occur. Since (i4) writes to the same address B and (i3) is a speculative load, it is possible to have a memory consistency model violation. Then, in step 2, the core with the speculative load discovers if there are loads in program order before the speculative load that have not returned their values. For this example, (i2) has not returned its value. Thus, the violating store, (i4), is delayed until both (i2) and (i3) are visible to all cores. A natural way to visualize the effect of PPP's delaying technique is shown in Figure 3, where (i4) and (i5) move in the memory order after (i2). This is the corrected memory order made visible to all cores with PPP at runtime.

V. PPP: Hardware Approach to Correct Potential Memory Consistency Model Violations with DLRC

We present details of the hardware for PPP to correct potential memory consistency model violations due to speculatively executing loads. A key novelty of our technique is that PPP maintains predictability when correcting potential memory consistency model violations, unlike all prior efforts. PPP introduces two key hardware innovations: (1) a *broadcast acknowledgement bus (BACK)* to detect a violating store, and (2) an orchestrator of operations over REQ, RESP, and BACK to delay the violating store until all speculative loads are completed. We describe the hardware that enables us to detect and delay violating stores, and then with a concrete example, we show how the hardware is used to correct potential memory consistency model violations. Finally, we introduce a technique that ensures the values returned to loads are correct during the delay of the violating store.

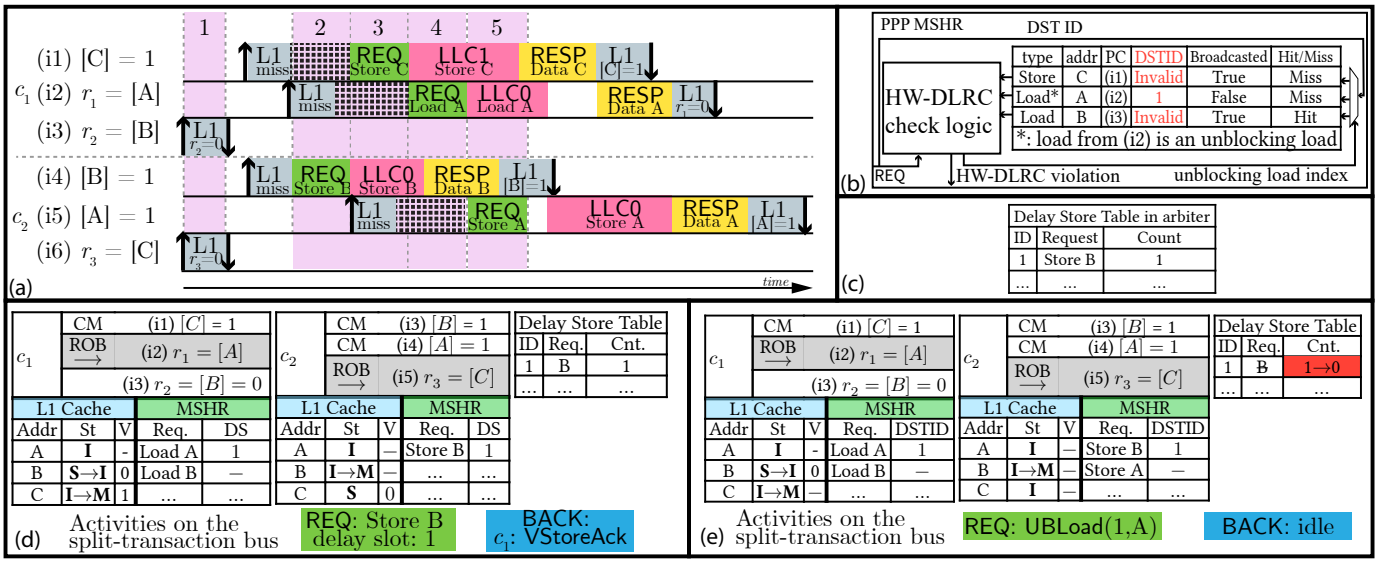
A. Hardware implementation of DLRC correction

The DLRC conditions in Section IV determine when a load reordering violates the memory consistency model. However, implementing these conditions directly in hardware is impractical, as they require prior knowledge of the memory order, necessitating visibility of all memory operations across cores beforehand. By then, stores would have already updated memory, and loads would have returned inconsistent values, rendering the identification of violations moot. Thus, we propose a runtime approach to detect potential memory consistency violations without the complete memory order.

Detecting DLRC violations. We detect *potential* memory consistency model violations due to load reordering with only one *violating store*, outlined in HW-DLRC.

Definition 12 (HW-DLRC)**.** *Given two load memory operations l_1, l_2 from core c_1, and a store operation s from another core, the hardware disallowed load reordering violation conditions (HW-DLRC) are as follows. (1) $\langle l_1, l_2\rangle_p$ and l_2 is inserted into memory order, but l_1 is not. (2) $\langle l_2, s\rangle_m$ and (3) $s \in \mathbb{S}|_{l_2}$.*

HW-DLRC 1 indicates l_2 executes speculatively before l_1. HW-DLRC 2 states that store s executes after l_2, and is visible to all cores. HW-DLRC 3 means that s writes to the same address as l_2. These conditions are stronger than DLRC conditions because it only uses one store to identify a potential memory consistency model violation. Hence, HW-DLRC could identify a possible memory consistency model violation when there is none. Nonetheless, this technique is essential for a practical hardware implementation. The soundness of PPP follows in the following lemma and theorem.



* Table head in L1 Cache: Addr - address, St - State, V - Value ; Table head in MSHR: Req. - Request, DS - Dependent Stores

Fig. 4: The working of PPP. (a) A concrete execution of the program in Figure 1(a). (b) MSHR is extended with the delay store table index field (DST ID). (c) The delay store table in the arbiter split-transaction bus arbiter. (d) The operation of PPP when an invalidating store broadcasts on REQ. (e) The operation of PPP when the invalidating store is no longer blocked.

Lemma 13. *PPP ensures that three conditions DLRC will not hold in a memory order.*

Proof. Proof by contradiction. Suppose all DLRC hold in a memory order, this means that there is a pair of load l_1 and l_2 and two stores s_1, s_2 such that DLRC 1, DLRC 2, and DLRC 3 hold. However, for DLRC 1 and DLRC 3 to hold, both HW-DLRC 1 and HW-DLRC 2 must hold for l_1, l_2, and s_2. As such, the PPP mechanism engages and prevents s_2 from being broadcasted until l_1 is broadcasted, preventing DLRC 2 and DLRC 3 from holding, contradicting that DLRC hold. □

Theorem 14 (soundness of PPP)**.** *When PPP is deployed, it ensures no memory order violation.*

Proof. Lemma 13 shows that DLRC do not hold with PPP. With contraposition of Lemma 11, PPP ensures no memory order violation. □

B. Hardware implementation to delay violating stores

PPP requires three hardware components to support delaying violating stores, and returning correct values to loads dependent on such stores. A violating store begins as a cache miss, prompting the cache controller to broadcast a request to fetch the accessed line on REQ. All cores see the REQ broadcast, and all except the issuing core check for any speculative loads reading from the same address. If such a load exists, then the cache controller identifies if there are *other* unbroadcasted loads that are older in program order than the speculative load. When older loads exist, the store is a violating store, and all cores and the arbiter coordinate to delay the violating store. PPP needs hardware support to implement this ability to delay stores and to enforce the memory consistency model.

Delay store table (DST) and REQ bus. PPP implements an N entry delay store table (DST) in the arbiter that records information about stores identified as violating stores. One entry corresponds to delaying one violating store. An entry records an identifier (ID), the address targeted by the violating store, and a Count of the number of cores that identified a violating store to be delayed. There are N entries because each core can delay at most one store at the same time. This is because the core must wait until the violating store completes before proceeding to the next memory operation. Figure 4(c) shows an example of the DST. The DST shows a violating store to address B with one core identifying it as a violating store. When the arbiter grants access to a core to broadcast a store on REQ, the arbiter checks the DST for a vacant entry. The ID of the DST entry is broadcast together with the store.

MSHR modifications. Non-blocking caches use MSHR to buffer memory requests that miss in the core's private cache. The MSHR records the type, address (addr), the broadcast status of the operation, and instruction (instr) as shown in Figure 4(b). The type and address indicate the memory operation type (load or store) and the targeted address. The instr field denotes the instructions that bind to the memory operation. The broadcast field is true when the request is placed on REQ. This means all cores have seen this request. PPP extends the MSHR with a field to represent DST ID. The DST ID field is a list of identifiers from the arbiter's DST of the violating stores in the DST. Figure 4(b) shows an example of an MSHR with three memory operations. Only the load to A has its DST ID marked with a 1, and its broadcast to false. This 1 corresponds to the ID 1 in the DST in Figure 4(c) for the store to B signifying that the store to B may cause the speculative load to A to return an inconsistent value for the memory consistency

Message Type	Channel	Explanation
Store (A, s)	REQ	Store request has address A and free delay table entry s
UBLoad (s, A)	REQ	Unblocking load to address A with delay table entry s
VStoreAck	BACK	Acknowledging that the store broadcast on REQ is a violating store
LoadOV	REQ	The Load request that the LLC should return the data value before the violating store

Fig. 5: PPP extends coherence messages to manage the delaying of violating store and to ensure correct value forwarding.

model. We annotate this load as an unblocking load.

The HW-DLRC check logic component implements the detection of a possible memory consistency model violation (HW-DLRC conditions) and updating the DST ID entry in the MSHR. Our implementation does the following. When the check logic observes a store on REQ, it initiates the check for HW-DLRC. The first step checks the MSHR for loads that have the same address as the store broadcasted on REQ (HW-DLRC 2 and 3). This is the load to B for (i3). Since this load is in the MSHR, and it has been broadcasted, HW-DLRC 2 is satisfied as the store broadcasted after the load. Then, the check logic searches loads older than the load to B in program order that have not been broadcasted. This identifies speculative loads older than the load of B (HW-DLRC 1), which we call the unblocking load. When there are multiple speculative loads, the hardware selects the youngest of these loads and marks it as an unblocking load. The load to A is a speculative load as it has not been broadcasted. The address matching process, done in parallel, is feasible due to the MSHR's small size; similar techniques are also used in silicon-proven processors [19] for efficient store-to-load forwarding within a single clock cycle.

New Coherence Messages. PPP introduces four new messages to coordinate the delaying of violating stores, as shown in Figure 5. The use of these messages follows while detailing required hardware structures.

Broadcast Acknowledgement Bus (BACK). PPP extends the interconnect with a BACK bus that private cache controllers use to communicate with the arbiter that a violating store should be delayed. Cores use the VStoreAck message to inform the arbiter that the most recent store is a violating store.

Cache controller modifications. Figure 5 lists the new coherence messages used in PPP. When a core observes a violating store, it sends VStoreAck over the BACK bus to the arbiter, indicating that the store broadcast on REQ is a violating store; hence, a DST entry should be allocated.

Recall that the load to A is the unblocking load. When an unblocking load is ready to be broadcasted on REQ, the cache controller sends an UBLoad message with the address and the DST ID on REQ instead of a regular load request. UBLoad notifies the arbiter to decrement the count field in the DST entry of the violating store. When the count field reaches zero, the violating store is no longer blocked and the arbiter allows the store to proceed. Note that this means that the arbiter now grants the access of the BANK and RESP to the store.

When a violating store is delayed, other cores can still issue loads to access the cache line written by the violating store. Note that since the violating store is delayed, and not yet visible to all cores (not in the memory order), these loads should appear to complete prior to the violating store. Hence, the loads must return the value of the cache line before the violating store writes to it. Since the violating store is a private cache miss, the value *before* the violating store is in the main memory or is cached in the LLC; the value written by the violating store is in the store buffer of the core that issues the store. When TSO is deployed, younger loads can return the value before the violating store with store-to-load forwarding. PPP ensures that such loads receive the old value by introducing a new request message named LoadOV (*load old value*). When a load is issued on REQ, and the cache line is accessed by a delayed violating store, instead of broadcasting an ordinary load on REQ, the arbiter intercepts the load request and broadcasts a LoadOV request instead.

C. Concrete execution

We re-examine the high-level example presented in Figure 3. Initially, B is cached in c_1, and C is cached in c_2. Other cache lines are not privately cached by any cores. Since c_1 and c_2 executes loads speculatively, (i3) and (i6) are sent to the cache out of program order. Both loads hit in the cache. This concludes the execution during 1. Next, c_1 sends (i1) and (i2) to the cache, while c_2 sends (i4) to the cache. PPP requires that younger stores must be sent after all older stores are visible to all cores, c_2 hence stalls on (i5) until (i4) broadcasts. Commercial processors implementing TSO also sends younger stores after all older stores are visible to all cores [20]. (i1), (i2), and (i4) miss in the cache.

Next, during 2, the arbiter grants access to c_2 and broadcasts (i4) on REQ. When c_1 receives (i4) on REQ, it observes that (i3) is a speculative load hit; in addition, (i3) is not yet broadcasted on REQ. This leads c_1 to send a VStoreAck on BACK, indicating that (i4) is a *violating store*. Recognizing this fact, the arbiter allocates a DST entry with ID 1 for (i4). Additionally, c_1's MSHR entry corresponding to (i2) records that the DST entry ID of the invalidating store is 1; this marks (i2) as an unblocking load. This indicates that when (i2) broadcasts, a UBLoad is sent instead of an ordinary load. This is shown in Figure 4(d). At the end of 2, since (i4) is broadcasted, c_2 can now issue (i5) to the cache, which misses.

During 3, the arbiter grants access to c_1 due to round-robin arbitration; c_1's (i1) is c_1's oldest request, and c_1 broadcasts (i1) on REQ. Although C is cached in c_2, and (i6) is a speculative load, there is no older load in c_2 that is not broadcasted; hence, (i1) is not a violating store and can proceed. If the retry mechanism was used, (i6) must be retried because C is invalidated by (i1).

Next at 4, the round-robin arbiter on REQ should grant access to c_2. However, since (i4) is a violating store, c_1's younger memory operations are delayed, until (i2), the unblocking load, broadcasts. Hence, PPP's arbiter grants access to REQ again to c_1, whose oldest unbroadcast request is now (i2). When (i2) broadcasts on REQ the arbiter recognizes that

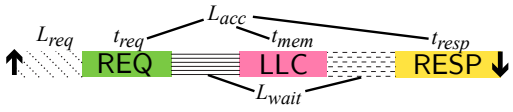


Fig. 6: Timeline of a request.

it is an unblocking load for delay slot 1 because its delay slots field is not empty. Hence the arbiter reduces the count field for DST entry from 1 to 0. Since now the count field for delay slot 1 becomes 0, the arbiter clears the delay slot and allows (i3) to proceed in 5. Without PPP, the arbiter grants access to c_2 and allows (i5) to proceed before (i2), causing r_1 to load 1, exhibiting DLRC.

VI. Worst-case Latency Analysis

Our worst-case latency analysis computes the latency from when a request becomes the oldest among the inflight requests to when this oldest request receives its response from the memory hierarchy. As such, the worst-case latency of multiple requests can be upper-bounded by the sum of the worst-case latency of individual requests [12].

Our analysis divides the worst-case latency of a request into three parts, which naturally follows the life cycle of a request. Figure 6 shows the life cycle of a request that begins with a miss in the private cache and needs to retrieve the data from the LLC. Before this request can be broadcasted over REQ, it suffers L_{req} in the worst case. This is because the round-robin arbitration controls the access to REQ. Next, before the request can access the LLC, and before the request can access RESP, it can be delayed. This is because requests from other cores can be broadcast on REQ before the request under analysis. These older requests access BANK and RESP before the request under analysis. The total latency the request must wait to access the LLC and RESP in the worst-case is bounded by the worst-case resource latency L_{wait}. Accessing the LLC takes t_{mem} in the worst case while accessing RESP takes t_{resp} in the worst case. This means that t_{mem} bounds the worst-case latency of accessing the LLC *and* fetching data from the main memory. The total latency of accessing the resources in the worst case is $L_{acc} = t_{req} + t_{mem} + t_{resp}$.

We identify the critical instance where the request under analysis, r_{ua}, suffers the worst-case latency. We start with the worst-case request latency, which is the worst-case latency it takes for r_{ua} to be broadcasted on the bus.

Lemma 15. *In the worst case, r_{ua} experiences the worst-case request latency when it is ready to be broadcasted on REQ and it is the last among all N cores to broadcast requests. The worst-case request latency is $L_{req} = (N-1)t_{req}$.*

Proof. In the worst-case, when r_{ua} from c_{ua} is ready to be broadcast on REQ, it is the last in the round-robin order to broadcast; hence it must wait until its turn. Also note that there are $(N-1)$ cores in the worst-case that must broadcast their requests before c_{ua} can broadcast r_{ua}, where each such broadcast costs t_{req}. □

A request r *delays* r_{ua}'s access to the LLC or RESP if after r_{ua} broadcasts on REQ, r_{ua} must wait for r's access to any of the resources to complete (REQ, the LLC, or RESP) before r_{ua} can access the resource.

Lemma 16. *A request r delays r_{ua}'s access to the LLC or RESP if and only if: r is in the MSHR of a core other than c_{ua}, and any of the following conditions is true: (1) r broadcasts before r_{ua} on REQ, and, (2) r is a load request that is older than a speculative load to cache line A.*

Proof. We first prove the *if* part: If a request completes when r_{ua} broadcasts, since this request will not access the LLC or RESP; hence if a request delays r_{ua}, it must be inflight when r_{ua} broadcasts on REQ, that is, in the MSHR. Because the arbiter maintains the broadcast order of requests, any requests that broadcast before r_{ua} must access the LLC or RESP before r_{ua}, hence a request satisfying (1) would be processed before r_{ua}. Finally, PPP ensures that when r_{ua} invalidates A and there is a speculative load that hits on A, r_{ua} must complete after all prior requests of the speculative load, hence (2). The *only if* part follows from the implementation of the arbiter. Condition (1) follows when r_{ua} is not in the delay slot, the arbiter prioritizes the access to the LLC and RESP for requests that are broadcast before r_{ua}. When r_{ua} is in the delay slot, it must wait until all unblocking load to broadcast on REQ, which incurs condition (2). □

The worst-case resource latency captures the latency that r_{ua} must wait before accessing LLC and RESP.

Lemma 17. *The worst-case resource latency L_{wait} due to waiting for accessing the LLC and RESP is $L_{wait} = (N-1) \times M \times (t_{mem} + t_{resp}) + (M-1) \times (N-1) \times t_{req}$.*

Proof. We can count the number of requests that delay r_{ua} using Lemma 16. In the worst case, the requests that satisfy condition (1) in Lemma 16 is $(N-1) \times M$. Note that M is the maximal number of outstanding requests. This is because each of the other $N-1$ cores can broadcast at most M inflight requests when r_{ua} broadcasts its request on REQ. These requests can delay r_{ua} by at most $(t_{mem} + t_{resp})$. In the worst case, the requests that satisfy condition (2) are $(M-1)(N-1)$. This is because all requests from all other $N-1$ cores can have at most $(M-1)$ requests (excluding the speculative load) that must finish before r_{ua}. Note that in the worst case, all these requests are not broadcast. Hence, these requests can delay r_{ua} by at most t_{req}. □

Theorem 18. *The worst-case latency of a request under analysis r_{ua} is $WCL = L_{req} + L_{wait} + L_{acc}$, where $L_{acc} = t_{req} + t_{resp} + t_{mem}$ is the latency for REQ, RESP, and the LLC to process the request.*

Theorem 18 follows Lemma 15 and Lemma 17.

VII. Empirical Evaluation

We evaluate our proposed approach using the gem5 [11] micro-architectural simulator. We simulate configurations with

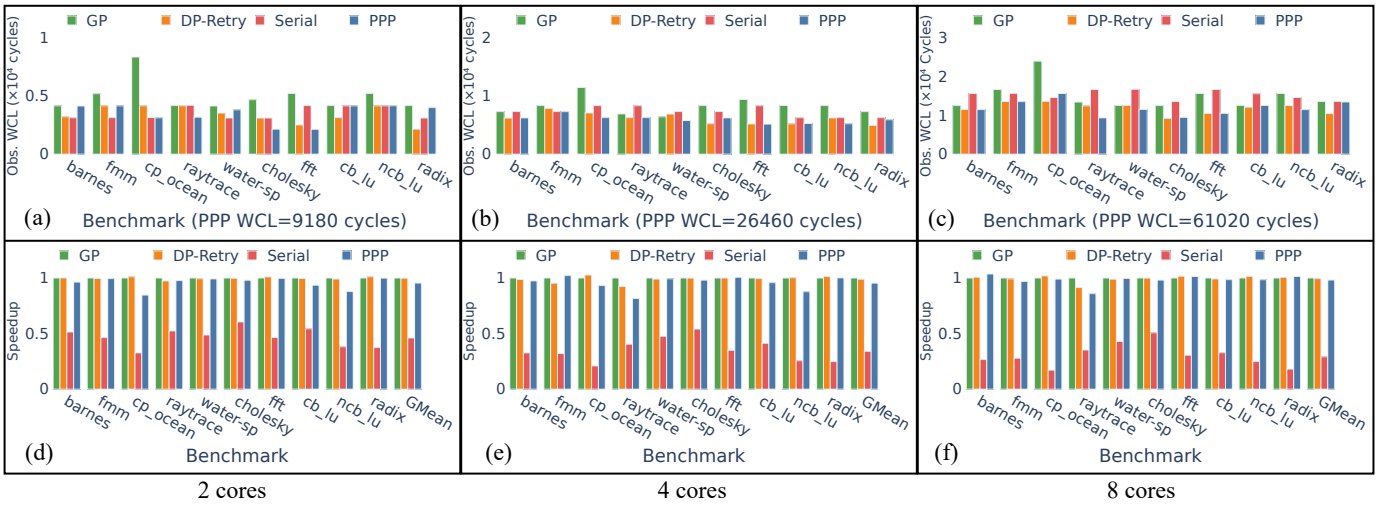


Fig. 7: Observed worst-case latencies of Serial, DUEPCO, GP, and PPP for 2, 4, and 8 cores are shown in (a), (b), and (c) respectively. Performance of 2, 4, and 8 cores configurations are shown in (d), (e), and (f) respectively. Baseline configuration is GP. The GMean indicates geometric mean.

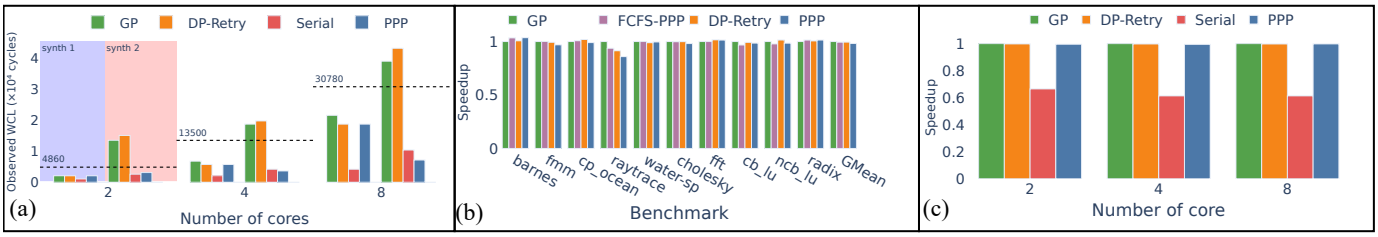


Fig. 8: (a) Observed WCL of 2 to 8 cores configurations with synthetic benchmarks. The dashed lines tag the WCL of PPP. (b) Speedups of 8-core PPP, FCFS-PPP, and DP-Retry normalized to GP. (c)Running SPLASH-3 benchmarks on 2, 4, and 8 cores configurations with single-threaded execution. Speedups are normalized to GP.

various number of cores. Each core implements an out-of-order pipeline with an issue width of 8 operating at 2Ghz, and all L1 caches are non-blocking.

We implement five configurations: (1) the Serial configuration that serializes all memory requests, (2) GP that reflects the **g**eneral-**p**urpose approach of handling memory requests for high performance, where retry mechanism ensures consistency and a first-come-first-serve arbiter arbitrates the memory requests on the split-bus, (3) the prior work on predictable cache coherence that supports *multiple outstanding requests*, DUEPCO, where no special mechanism ensures memory consistency, (4) DP-Retry that deploys the retry mechanism in DUEPCO to ensure memory consistency model, and (5) our proposed approach, PPP. Since each core in Serial has only one request, the L1 functions as a blocking cache. Note that this design reflects the handling of memory requests in state-of-the-art out-of-order pipelines [21]. We assume that each core is equipped with 16kB L1 private instruction and data caches. The private caches are backed by a shared 2MB LLC. The private caches and the shared LLC communicate through split buses as presented in the system model. We assume that $t_{req} = 20$ cycles, $t_{resp} = 10$ cycles, and $t_{mem} = 500$ cycles.

A. *Verification of correctness*

We exercise our implementation with manually crafted litmus test [22] memory access traces and do not observe TSO or SC violation in our implementation. We further observe that DUEPCO violates SC and TSO, when exercised with the **mp** and **iriw** litmus tests [22]; hence we do not include DUEPCO for further evaluation on performance and predictability.

B. *Worst-case latency*

Synthetic benchmarks. Our synthetic benchmarks attempt to stress the different configurations. In all configurations except for Serial, we use an MSHR of size 8 ($M = 8$), for efficient creation of the worst-case scenario. Serial has an MSHR of size 1 ($M = 1$) to reflect the blocking nature of the cache. The benchmark **synth 1** exercises the WCL of the system by generating memory accesses to the same cache lines. Note that **synth 1** does not exercise the retry mechanism. **synth 2** exercises the memory consistency enforcement mechanism by timing the requests such that store requests write to cache lines read by speculative loads, causing the retry mechanism to be invoked in both DP-Retry and GP. This triggers both the retry mechanism in DP-Retry and the delay mechanism in PPP.

The results are shown in Figure 8(a), where dashed lines and numbers above indicate the WCL of PPP. We observe in **synth 1** that all predictable configurations, including DP-Retry have a WCL that is within the analytical WCL bound. Specifically for PPP, the observed WCL are 2080, 5720, and 18720 cycles for 2, 4, and 8 cores configurations, respectively. In **synth 2**, PPP has a WCL of 3120, 3640, 7163 cycles for 2, 4, and 8 cores, respectively, which are lower than the corresponding analytical WCL bound of PPP of 4860, 13500, and 30780 cycles. We observe that both DP-Retry and GP have a WCL that is higher than the analytical WCL bound of PPP. For example, for 8 core configuration, DP-Retry and GP has a WCL of 43160 cycles and 39000 cycles respectively, higher than PPP's WCL. In addition, we observe that the total execution time of **synth 2** of DP-Retry and GP is higher than PPP; this is an indication that PPP has a better performance than DP-Retry and GP when the retry mechanism is exercised. Serial shows the lowest WCLs of 4240, 8480, and 16960 cycles compared to PPP and Retry. Despite its low WCL, we will present in following sections Serial's subpar performance compared to GP and PPP.

Splash-3. We also evaluate the observed WCL with the SPLASH-3 [23] benchmarks. SPLASH-3 consists of a wide variety of complex parallel multi-threaded applications that exercise data sharing and synchronization across cores. We use SPLASH-3 to study the impact of these inter-core interactions on performance and WCL when a system deploys PPP. Figure 7(a)-(c) show the results for 2, 4, and 8 cores configurations. Our results show that the observed WCLs of PPP are within the analytical WCL.

C. Performance

We evaluate the performance of PPP using SPLASH-3 [23]. Figure 7(d)-(f) show the results when running SPLASH-3 benchmarks on 2, 4, and 8 cores configurations. In Figure 7(d)-(f), we report the *speedup* metric, which is the ratio of the execution time of the baseline configuration, GP, to the execution time of the configuration under evaluation. When compared to the GP mechanism, Serial exhibits slowdown of $2.17\times$, $2.94\times$, and $3.45\times$, for 2, 4, and 8 cores respectively. Hence, enforcing TSO by serializing memory requests incurs a significant performance penalty. On the other hand, PPP has similar performance as GP, where the speedup of PPP over GP is $0.95\times$, $0.95\times$, and $0.98\times$ for 2, 4, and 8 cores respectively. Thus, PPP offers similar performance to a general-purpose scheme to ensure memory consistency while guaranteeing predictability. This is a positive result of this work. We also notice that DP-Retry has a marginally better performance than PPP. It exhibits a speedup of $0.99\times$ for 2, 4, and 8 cores when compared to GP, such performance comes at the cost of a higher WCL as shown in the previous section. This performance gap is a result of the first-come-first-serve (FCFS) arbiter deployed in DP-Retry. To validate this, we integrate PPP with the FCFS arbiter (denoted FCFS-PPP), and evaluate the performance. Figure 8(b) shows that in an 8-core configuration when PPP is integrated with the FCFS arbiter, PPP achieves a speedup of $0.99\times$ over GP. We plan to investigate the impact of integrating PPP with the FCFS arbiter in future work.

Finally, we evaluate PPP's performance with SPLASH-3 benchmarks when no inter-core interference is present, by running the benchmarks using a single thread for 2, 4, and 8 cores configurations. Figure 8(c) shows that PPP has less than 1% performance difference for 2, 4, and 8 cores configurations, respectively, when compared to GP. The close performance between PPP and other mechanisms such as DP-Retry and GP is because in PPP, the delay mechanism is not triggered when there is no inter-core interference. Note that Serial has a slowdown of up to $1.4\times$ compared to GP due to its blocking nature when handling memory requests.

D. Discussion

This work reveals that PPP guarantees the worst-case latency and maintains SC and TSO while foregoing marginal performance. Unlike DUEPCO, which supports multiple outstanding requests for safety-critical systems but risks producing *logically incorrect results* for memory accesses, PPP ensures *both timing predictability and logical correctness*. Furthermore, PPP delivers performance comparable to GP and DP-Retry in typical execution scenarios, such as those illustrated in Figure 7(d)-(f). PPP offers a distinct advantage over others by possessing a worst-case latency bound that is *known* and lower than the observed worst-case latencies of GP and DP-Retry. We envision PPP's WCL bound could be incorporated into static analyses [21], [24], [25] to provide safe worst-case execution time estimates. Further, PPP's performance benefit improves the quality of service of applications, and it encourages the deployment of performance demanding applications in safety-critical systems. Also, for systems such as mixed criticality, the slack provided by the performance benefits can be used for other noncritical tasks [26], [27]. As core count increases, PPP's analytical WCL bounds exceed those of Serial due to increased contention for shared resources. This is an inherent consequence of the parallelism exploited by PPP and does not represent a limitation of the approach; rather, PPP's WCL remains a safe upper bound.

VIII. Conclusion

We present PPP, a set of micro-architectural augmentations to enable the use of non-blocking caches, and cache coherence for out-of-order pipelines in safety-critical embedded systems. PPP allows multiple outstanding requests, exposed by both the out-of-order pipeline and the non-blocking caches, for significant performance gains over state-of-the-art techniques that serialize memory accesses by $2.07\times$, $2.79\times$, and $3.38\times$ for 2, 4, and 8 cores. To the best of our knowledge, this is the first work to provide WCL guarantees for memory accesses in safety-critical embedded systems that allow for multiple outstanding requests, while enforcing the memory consistency model.

References

[1] R. Pujol, H. Tabani, J. Abella, M. Hassan, and F. J. Cazorla, "Empirical evidence for mpsocs in critical systems: The case of nxp's t2080 cache coherence," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1162–1165.

[2] B. D. de Dinechin, "Invited consolidating high-integrity, high-performance, and cyber-security functions on a manycore," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–4.

[3] Renesas, *R-Car-V4H - Best-in-Class Deep Learning at Very Low Power, System-on-Chip for Automated Driving Level 2+/Level 3*, 2024.

[4] A. M. Kaushik, M. Hassan, and H. Patel, "Designing predictable cache coherence protocols for multi-core real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2098–2111, 2021.

[5] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "Carp: A data communication mechanism for multi-core mixed-criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 419–432.

[6] A. M. Kaushik and H. Patel, "A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 105–117.

[7] S. Hessien and M. Hassan, "Piscot: A pipelined split-transaction cots-coherent bus for multi-core real-time systems," *ACM Trans. Embed. Comput. Syst.*, Jul. 2022, Just Accepted.

[8] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A comparative study of predictable dram controllers," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, Feb. 2018.

[9] S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitert: An efficient fpga noc for real-time applications," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 64–71.

[10] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, Feb. 2020.

[11] N. Binkert, B. Beckmann, G. Black, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[12] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, M. Maggio, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 231, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 16:1–16:27.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[14] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence, second edition," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, Feb. 2020.

[15] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *2013 IEEE 34th Real-Time Systems Symposium*, 2013, pp. 372–383.

[16] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *ICPP'91*, 1991.

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

[19] I. Parulkar, A. Wood, J. C. Hoe, *et al.*, "Opensparc: An open platform for hardware reliability experimentation," in *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*, Citeseer, 2008, pp. 1–6.

[20] J.-M. Frailong, P. Sindhu, M. Cekleov, M. Powell, and E. Jensen, *Method and apparatus for providing total and partial store ordering for a memory in multiprocessor system*, US Patent 5,265,233, Nov. 1993.

[21] A. Gruin, T. Carle, C. Rochange, H. Casse, and P. Sainrat, "Minotaur: A timing predictable risc-v core featuring speculative execution," *IEEE Transactions on Computers*, vol. 72, no. 01, pp. 183–195, 2023.

[22] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Litmus tests for comparing memory consistency models: How long do they need to be?" In *Proceedings of the 48th Design Automation Conference*, ser. DAC '11, San Diego, California: Association for Computing Machinery, 2011, pp. 504–509.

[23] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[24] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A Survey on Static Cache Analysis for Real-Time Systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, 05:1–05:48, 2016.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, May 2008.

[26] H.-K. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources," in *2011 International Conference on Parallel Processing*, 2011, pp. 753–762.

[27] A. Syed, G. Fohler, and D. G. Pérez, "Online admission of non-preemptive aperiodic mixed-critical tasks in hierarchic schedules," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–10.