X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

Exclusive Hierarchies for Predictable Sharing in Last-level Cache

Xinzhe Wang, Zhuanhao Wu, Rodolfo Pellizzoni and Hiren Patel
{xinzhe.wang, zhuanhao.wu, rpellizz, hiren.patel}@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

***Abstract*—This work presents an approach to use a last-level cache (LLC) in a memory hierarchy for cache-coherent real-time multicores that delivers a low worst-case latency (WCL) and higher performance than all of its counterparts. Our approach relies on the key observation that an exclusive memory hierarchy, by definition, eliminates back invalidations, which are one of the largest contributors to the WCL when using inclusive memory hierarchies. However, to the best of our knowledge, there are no prior efforts that ensure the predictability of exclusive hierarchies for cache-coherent multicores. Consequently, in this work, we propose PECC, a predictable exclusive cache coherence mechanism, that achieves a lower average data access latency while providing a low WCL bound that scales linearly in the number of cores. Our evaluation shows that PECC reduces the bound by 6% and improves the average performance by $2.33\times$ over the predictable solution with an inclusive LLC.**

***Index Terms*—cache coherence, exclusive cache hierarchies, predictable architecture, shared last-level cache**

I. Introduction

Modern safety-critical systems are seeking to use multicores to deploy time-sensitive applications commonly seen in automotive [1] and avionics [2], [3]. These multicores offer the promise of high performance and an approach to consolidate multiple functions into a single platform. For safety-critical systems, they are typically designed to strike a balance between predictability and high performance. Predictability allows analysis methods to compute worst-case latency (WCL) bounds to ensure that certain tasks always execute within their temporal requirements. However, developing predictable multicores that deliver high performance continues to be an active topic of research as the requirements for predictability and performance often pull against each other [3]–[5].

Recent research efforts in the design of multicores for safety-critical systems have focused on improving the performance of the memory hierarchy by proposing predictable cache coherence schemes that allow multiple cores to cache data while providing a WCL bound on memory accesses [6]–[10]. While demonstrating promising results in improving average-case performance, most of these works[1] assume a shared inclusive last-level cache (LLC), where the interference poses significant challenges in ensuring a low WCL. For example, a recent analysis [11] shows that freely sharing an inclusive LLC can significantly degrade the WCL of memory requests, which can potentially make tasks unschedulable. Most existing solutions that employ a LLC [5], [12] eliminate this interference by partitioning the LLC among the different cores. This effectively isolates a portion of the LLC to each core exclusively. However, LLC partitioning can incur a significant performance penalty because it under-utilizes the available LLC capacity from a core’s perspective and makes coherent caching of shared data challenging [11]. To address this, zero-cost LLC (ZCLLC) [13] presents a solution that includes a shared LLC with a novel inclusive LLC architecture. Although ZCLLC lowered the WCL of a memory request, we find that there are further opportunities to improve on both the WCL and performance.

Motivated by this, we present an alternative approach to introduce a shared LLC to multicores for safety-critical systems. Our key insight is in employing an exclusive cache hierarchy rather than an inclusive one. Note that almost all prior works related to predictable cache coherence use an inclusive cache hierarchy making our contributions distinct from those. This brings us to the following main contributions of our work. (1) We present our solution, Predictable Exclusive Cache Coherence (PECC), which offers both high performance and a low WCL. (2) We prove that the WCL bound in PECC scales linearly with the number of cores. (3) We evaluate PECC in the Gem5 micro-architecture simulator and show that it reduces the WCL bound by 6% and provides substantially better average-case performance ($2.33\times$) when compared to ZCLLC.

II. Background

A. Hardware Cache Coherence

Modern multicores use hardware cache coherence [14] to deliver high performance by allowing caching of data while maintaining a consistent view of that data across all cores. Cache coherence operates at the granularity of a cache line, which is a fixed number of contiguous bytes of data. To maintain a consistent view of the same cache line, each cache controller implements a finite state machine encoding the rules of the prescribed cache coherence protocol. The state machine uses states to encode access permissions and other metadata (e.g. dirtiness) of the data, and the transitions to represent changes in states based on memory activities of other cores.

Figure 1 shows the modified-shared-invalid (MSI) cache coherence protocol [14]. Each transition is labelled with a list of events (separated with “;”) that trigger this transition, and

[1] Some of the works assume that the system has a perfect LLC where accesses are all hits and does not consider the interference on a LLC miss.

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

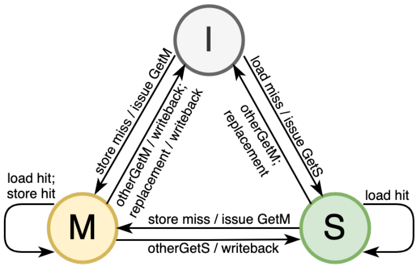


Fig. 1: MSI state transitions.

the event format is “event / action”. This protocol contains three states. (1) Modified (M): denotes a state that has read and write permissions on the cache line, and that the cache line has been modified; thus, dirty. Only one core can privately cache a line in the M state. (2) Shared (S): identifies a state with the read-only permission, and that the cache line data is clean. Multiple cores can privately cache a cache line in the S state to allow sharing of clean data. (3) Invalid (I): corresponds to a state where the cache line is invalid and has neither read nor write permissions. These three states serve as the base states for a multitude of other coherence protocols. For example, a popular protocol is MOESI [15] that introduces two additional states. (1) Exclusive (E): identifies a cache line as a read-only and exclusive copy, which means that it is not shared by any other core. (2) Owned (O): represents a read-only cache line that may be dirty, but not exclusive. This allows the owner of the cache line to reply to coherence requests by other cores for that cache line [14].

Example. A core issues a request to its private cache in the form of a load or a store. If the requested cache line is not present in the core’s private cache with the correct permission (i.e. cache miss), the cache controller generates coherence requests such as a GetS for a load and a GetM for a store, respectively. The GetS denotes a request to get the cache line with the read-only permission (cache line ends up being in the S), and the GetM with the read-write permission (cache line ends up in M). Suppose that a cache line is in the M state, and another core’s cache controller generates a GetM coherence request for the same cache line. Only one core is allowed to have the shared cache line in M; thus, the core with the cache line responds to the other core’s request, and then invalidates its own copy. Note that the other requesting core would transition the cache line to the M state. Now suppose that the cache line in the M state must be replaced in the other core’s cache, then its respective controller will generate a Put coherence request to writeback the dirty data.

In essence, the coherence protocol orchestrates the necessary communication between caches to achieve coherent sharing of data by maintaining single-writer-multiple-reader (SWMR) invariant and data-value invariant [14]. SWMR invariant ensures that only one copy of the data exists in the caches when the data is being written to or multiple copies of the data can be cached exclusively for reading. The data-value invariant requires the coherence protocol to ensure that the requestor of data always receives the most up-to-date data.

B. Inclusive and Exclusive Policies of Cache Hierarchies

An important design decision for multi-level caches involves determining whether the cache hierarchy is inclusive, exclusive or non-inclusive. We explain the differences between the three using an example where a core has two caches: a level-one (L1) cache being closest to the core, and a level-two (L2) cache being further away. An inclusive cache hierarchy (ICH) requires any cache line present in the L1 cache to also be present in the L2 cache. A non-inclusive cache hierarchy is one that is not inclusive. An exclusive cache hierarchy (ECH) requires a cache line to be present exclusively either in the L1 cache or the L2 cache, but never in both. An ECH is essentially a non-inclusive hierarchy with an additional constraint that data should not be duplicated across levels in the hierarchy. An ICH has less effective LLC capacity when compared to non-inclusive alternatives. This is because an ICH duplicates data. This affects the performance as the core count on multicores increases and each with larger private caches [16]. Consequently, modern commercial-off-the-shelf (COTS) multicores, such as Intel’s Raptor Lake, AMD Zen, and ARM Cortex R82 [17]–[19], are moving towards using non-inclusive hierarchies [16].

III. Related Work and Motivation

Prior works show that predictable versions of cache coherence [6]–[10] provide a WCL on memory requests while improving average-case performance. Similar to these prior efforts, our work uses a bus snooping protocol where the coherence request is broadcasted on the bus and monitored by all cores’ private cache controllers. We find that bus snooping protocols are appropriate for systems with core counts ranging up to 8 cores [14], which is a common setup in current real-time multicores [19]. Our detailed study of prior efforts [6]–[11], [13] revealed a common assumption amongst all these works: the cache hierarchy is inclusive. This assumption makes it difficult to include a shared LLC as it introduces interference between levels of the hierarchy that makes guaranteeing temporal requirements challenging [11]. We make three critical observations about such challenges.

Observation 1: Large per-request WCL. It has been reported that the interference caused by using a shared inclusive LLC can significantly lengthen the WCL of a single memory request when compared to a system without a LLC [11], [13]. We illustrate the crux behind the reasons using the illustration in Figure 2. Consider a read request to cache line A issued by the core under analysis C_{ua}. Suppose this request misses in C_{ua}’s private cache and broadcasts a GetS(A) request on the bus. The GetS(A) arrives at the LLC, and the LLC’s controller processes it. However, during processing, the controller discovers that the request for A is a miss, and the cache set that A maps to is full. This requires one of the cache lines to be evicted. Suppose that the chosen cache line is B, and it is in the M state. This means that a core has privately cached the line B, and modified it. To maintain an ICH, the privately

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, "Exclusive Hierarchies for Predictable Sharing in Last-level Cache," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

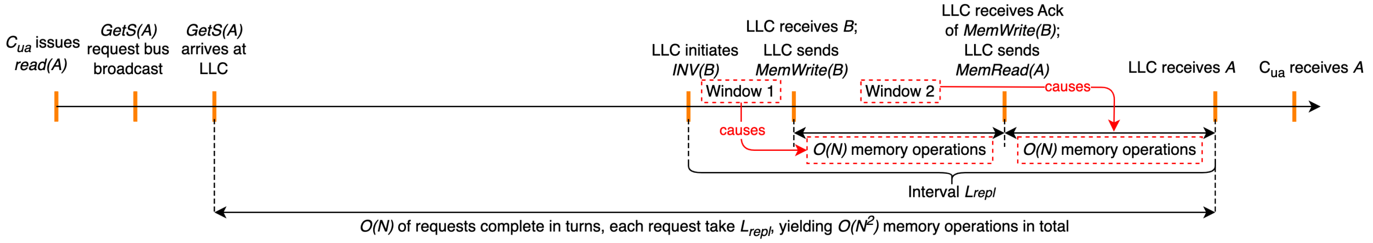

C_{ua} issues *read(A)*
GetS(A) request bus broadcast
GetS(A) arrives at LLC
LLC initiates *INV(B)*
LLC receives *B*; LLC sends *MemWrite(B)*
LLC receives Ack of *MemWrite(B)*; LLC sends *MemRead(A)*
LLC receives *A*
C_{ua} receives *A*
Window 1
Window 2
causes
causes
O(N) memory operations
O(N) memory operations
Interval *Lrepl*
O(N) of requests complete in turns, each request take L_{repl}, yielding $O(N^2)$ memory operations in total

Fig. 2: Request processing timeline of $read(A)$ request of the core under analysis C_{ua} in a multicore system with N cores.

cached copy of B must be written back and invalidated in the private cache prior to ensuring that the line B is no longer present in the LLC. This type of invalidation of private L1 copies due to the eviction of the same cache line from the LLC is referred to as back invalidations. After the back invalidation is initiated, the LLC waits for the dirty line B to be sent to the LLC (window 1 marked in Figure 2). Then, the LLC issues a write of B to the main memory, and waits for it to complete (window 2 marked in Figure 2). Finally, the LLC can issue a main memory read to fetch A following the completion of the main memory write of B. During these two waiting windows, other cores can proceed and insert main memory requests ahead of C_{ua}. As a result, at worst, when the main memory request of C_{ua} is issued, all requests from other cores are queued before it, taking $O(N)$ main memory operations to complete where N is the number of cores. Looking earlier into the request timeline when GetS(A) arrives at the LLC, GetS(A) may encounter resource contention with $O(N)$ other requests that need to complete one after another. For example, all requests may need to occupy the same cache entry as determined by the replacement policy. Since each request can experience the same scenario and takes $O(N)$ main memory operations to complete, overall, GetS(A) requires $O(N^2)$ main memory operations to complete.

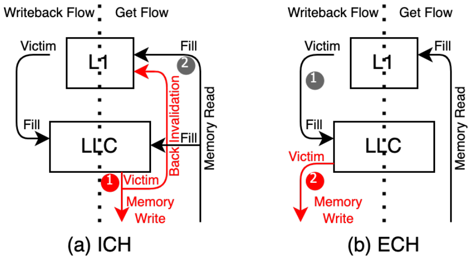
Observation 2: Imprecision of static cache analysis. The interference caused by back invalidations in a shared inclusive LLC can pose additional challenges for the precision of static cache timing analysis[2] on private data in L1 [20], [21], resulting in a pessimistic WCL bound for the task. When calculating the WCL at the task level, using the WCL of accessing the main memory for all memory requests is grossly pessimistic because the number of cache hits is typically higher than misses. Typically, a safe tighter bound for the task can be obtained by only multiplying the hit latency for the memory accesses that are guaranteed to hit inside caches as determined by static cache analysis. However, in a multicore system with a shared inclusive LLC, such analysis is difficult due to the interference of back invalidations [20], [21]. Specifically, private data cached in a core's L1 can be back invalidated by another core's actions to the shared LLC that triggers a LLC replacement. Due to the complexity of this type of multicore interference, for many memory accesses, the analysis tool is uncertain if the requested data is present inside the cache and has to assume the worst case where a cache miss is incurred to obtain a safe upper bound (i.e. degradation of analysis precision). Most prior works on static cache analysis circumvent this interference by requiring the shared LLC to be partitioned to different cores [20]. This requirement makes coherent caching of shared data challenging, which we wish to enable for high performance implementations.

Observation 3: Limitations of the state-of-the-art. To address the challenges in the shared inclusive LLC, [13] proposes zero-cost LLC (ZCLLC), a novel inclusive LLC architecture that does not incur additional penalty to the WCL of memory requests. ZCLLC achieves this by performing smart relocation of cache entries and maintaining a vacancy invariant that a clean victim line is always available for a LLC replacement. Guaranteed by the vacancy invariant, neither back invalidations nor main memory writes are triggered upon a LLC replacement. However, we observe that ZCLLC has the following two key limitations. (1) ZCLLC models all shared hardware (e.g. buses, the LLC, and the main memory) as one single entity for arbitration under time-division-multiplexing (TDM) scheme. This requires the length of the TDM slot to cover the WCL to transfer data to and from the main memory. We observe that this approach limits the benefits of the LLC. This is because, even when a request experiences a hit in the LLC, the entire TDM slot must expire before completing the transaction. (2) ZCLLC requires a custom architecture for the LLC. In particular, it modifies the LLC architecture to support cache line relocation, which incurs additional storage and logic overhead. For example, to perform reallocation of a cache line, the cache line needs to first be read from the original cache set and then written to the destination cache set. Meanwhile, the storage holding the reallocation information needs to be updated. To read a reallocated cache line, two sequential data array look-ups are also required. These additional accesses due to relocation can lengthen the LLC processing time. In our work, we do not modify the LLC architecture as it is a performance sensitive component of conventional memory hierarchies.

Proposal: Exclusive Cache Hierarchy (ECH). We find that we can address the challenge of including a LLC using an alternate approach. Note that all prior efforts on predictable cache coherence protocols [6]–[11], [13] use an ICH. However,

[2] Although no existing static cache analysis work is applicable to shared data under coherence effects, the WCL of a task can be tightened by applying the analysis to the private data portion [9], [20].

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.



(a) ICH (b) ECH

Fig. 3: Worst-case traffic flow in ICH and ECH with operations related to main memory writes highlighted.

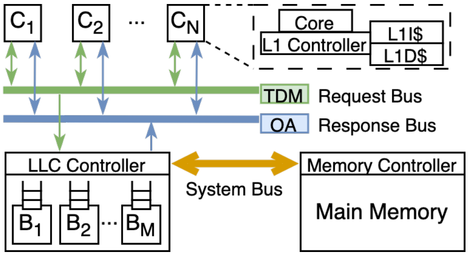


Fig. 4: Overview of system model.

we observe that an ECH for the LLC, by definition, removes the central issues such as back invalidations that make using LLCs with ICHs difficult. To illustrate this insight, Figure 3 compares the worst-case traffic flow between an ICH and an ECH. In an ICH, before filling new data into both the L1 cache and the LLC, the LLC must ensure a vacant entry is available, which potentially triggers a back invalidation and a main memory write when completing a Get request (Get flow). As explained earlier in observation 1, these are the crux that significantly lengthens the WCL in an ICH. However, in an ECH, new data from the main memory bypasses the LLC and directly fills the L1 cache. This immediately removes the back invalidation and main memory write in Get flow. Note that the main memory write is instead rescheduled to a core’s writeback request from the L1 cache to the LLC (writeback flow). Specifically, this happens when the LLC replaces a dirty line to make space for the data written back from the L1 cache. The rest of the paper discusses our solution in detail, which mainly comprises of two ingredients: (1) We develop a predictable cache coherence protocol that maintains an ECH. We contend that by maintaining an ECH, the LLC functions similar to caches in COTS platforms (resolving limitation 2 of ZCLLC). (2) We employ a split-transaction bus with fine-grained arbitration to promote parallelism available in the cache hardware (resolving limitation 1 of ZCLLC).

IV. System Model

We present our system model and introduce the symbols we use in the remainder of the paper.

A. Processing Cores and Cache Hierarchy

Cores. We consider a multicore system with N cache-coherent cores connected to a memory hierarchy with two levels of caches and a main memory as shown in Figure 4. Each core implements an in-order pipeline that issues at most one outstanding memory request at a time. This requirement of one outstanding memory request is consistent with prior related works [6]–[8], [11], [13]. It limits the overall number of outstanding memory requests in the system to N, serving as the basis for conducting timing analysis in various system components (e.g. buses, caches, and the main memory).

Cache hierarchy. The cache hierarchy consists of two levels: a split instruction and data cache that is private to each core at level one (L1), and a shared last-level cache (LLC) for all cores at level two. The two-level caches maintain an exclusive hierarchy. All L1 and LLC caches are writeback caches, and they use write-allocate for write requests. The shared LLC employs an implementation that has M independent banks with each bank having a request queue at its input. We assume such an LLC implementation because banks are known to improve performance in COTS platforms [22]. A request that arrives at the shared LLC is inserted into one of these queues. We assume that the size of the queue allows each core to make one request. A bank processes requests from its queue in first-come-first-serve (FCFS) order. Note that the shared LLC’s cache sets are assigned to different banks in an interleaved manner (e.g. set$_i$ is assigned to bank$_{i\%M+1}$). This allows the LLC to process requests in parallel if the requests access cache sets that are mapped to different banks. We assume that the WCL to access a bank in the LLC is t_{BANK}, which includes checking the tag array and reading from or writing to the data array. A bank can issue a main memory request. In particular, it issues a main memory read when the data is not found in the bank, or a main memory write when a dirty cache line is replaced. Additionally, the bank contains miss status handling registers for temporarily holding the dirty data to be written to the main memory. For additional details on the LLC and bank operations, we refer the readers to the exclusive LLC controller design discussed in Section V-A.

B. Coherent Interconnect

Our system model uses separate request and response buses for the coherent interconnect between L1 caches and the LLC. The request and response buses together comprise a split-transaction bus, which improves system performance by interleaving processing of transactions from multiple requests [14]. The split-transaction bus design is commonly deployed in COTS platforms such as Arm Corelink [23] and Intel’s QPI [24]. The two buses are explained next.

Request bus. The request bus is responsible for broadcasting coherence requests from the cores’ L1 cache controllers. Note that this also includes the data payload contained inside the writeback request. Additionally, similar to common bus snooping protocols [14], [15], the request bus contains a shared wired-OR line to report the snoop result of the requested cache line. In our case, the shared wired-OR line is asserted high if any L1 cache has a copy of the requested cache line. We use

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

t_{REQ} to denote the WCL to complete a request broadcast on the request bus.

Response bus. The response bus transfers the response to coherence requests, which includes data response for Get type requests and acknowledgement (Ack) for writeback requests. The response bus also supports direct cache-to-cache transfer for data transfer between L1 caches. We denote the WCL for sending a response as t_{RESP}.

Bus arbitration. We deploy two bus arbiters to predictably manage accesses to the split-transaction bus while delivering high performance. We use a work-conserving time-division multiplexing (TDM) arbiter at the request bus and an oldest-age arbiter (OA) at the response bus. We present the details of the design and rationale for the arbiters in Section V-B.

C. *Main Memory*

The multicore system also has a main memory. We assume that all instructions and data required by the application are available in the main memory. The system bus connects the LLC and the main memory for transmitting requests from the LLC to the main memory, and responses from the main memory to the LLC. Note that the LLC contains M independent banks that can issue main memory request individually, and the maximum number of outstanding main memory requests in the system is N. The system bus and the main memory must employ buffering mechanisms to support this requirement. This is required to ensure that a bank can immediately issue a main memory request without being blocked by the back pressure from the system bus and main memory. Starting from the time when a bank issues a main memory request, we denote the WCL for completing this request as t_{MEM}. This accounts for the interference from M bank requestors, with a limit of at most N outstanding main memory requests across the system. For the evaluation (Section VII), we use a single-port SRAM model. The details of this model are provided next. The system bus has a global queue for M banks to insert their main memory requests. The global queue orders the requests based on FCFS order. For the main memory requests issued in the same clock cycle, they can be inserted into the global queue in any order. In addition, the insertion happens fast so that even if all requests from M banks arrive at the same time, all of them finish insertions before the next request is issued. The system bus and the SRAM process the requests in the order of this global queue. Also, the system bus supports simultaneous transmission of request from the LLC to the main memory, and the response from the main memory to the LLC. We denote the latency to service a single main memory request as t_{SRAM}, including the processing time on the global queue, system bus and the SRAM. Therefore, t_{MEM} is upper-bounded by $N \cdot t_{SRAM}$ in this model.

V. Predictable Exclusive Cache Coherence

We propose a Predictable Exclusive Cache Coherence (PECC) mechanism that offers both high performance and a low per-request WCL bound. This section is divided into two parts: (1) We present our protocol description for the exclusive cache coherence. (2) We discuss our bus arbitration policy that predictably manages coherence traffic while delivering high performance. The combination of the coherence protocol and the bus arbitration results in PECC.

TABLE I: Cache line properties.

Property	Definition
Dirtiness	A cache line is dirty if it is modified and holds the most up-to-date value compared to main memory.
Uniqueness	A cache line is unique if it is unshared (i.e. the only privately cached copy is itself).
Ownership	A cache line is owned if the entity (i.e. cache or memory) is responsible for replying to coherence requests for that cache line.

A. PECC *Protocol*

Although implementations for ECHs exist [25], public descriptions, documents, and open-source implementations for ECHs with a shared LLC are, to the best of our knowledge, not available. Consequently, we begin by exploring the design of a MOESI protocol for an ECH by articulating the challenges one would encounter when subsuming approaches taken in the design of an ICH-based MOESI protocol. Then, based on the observed challenges, we define invariants one must honour when designing coherence protocols for ECHs and propose the protocol for PECC. We provide three definitions in Table I that are essential in our discussion of the protocol design.

Challenge 1: Invalidation on writeback. In ECHs, since the contents of L1 and the LLC must be exclusive to each other, a writeback of a line from a L1 cache to the LLC would force all other copies of the cache line in other cores' L1 caches to also be invalidated. Consider an example where cores C_1 and C_2 cache the same dirty line in the O state and the S state, respectively. Now, suppose that C_1 evicts that cache line. According to MOESI, C_1 needs to writeback the dirty cache line to the LLC. In an ICH, C_2 can retain its shared copy after the writeback by C_1. However, in an ECH, C_2 must invalidate its shared copy since there can only be one copy of that cache line in either L1 or the LLC. Furthermore, in an ECH, this situation arises even when the cache line is clean since clean cache lines must also be written back to the LLC [16] on an eviction to populate the LLC content. Similar to the back invalidation problem in an ICH, this type of invalidations due to another core's action on the shared LLC complicates the analysis and degrades its precision. Specifically, this creates issues for the analysis of read-only data and instructions in ECHs. We propose the following invariant to remove the invalidation on writeback problem.

Invariant 1. *For an ECH, a cache line can be written back to the LLC from L1 if and only if it is unique.*

Challenge 2: Inefficient ownership assignment. An important design aspect of cache coherence protocols is the ownership assignment to identify the entity (i.e. cache or main memory) responsible for sending the requested data. In an

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

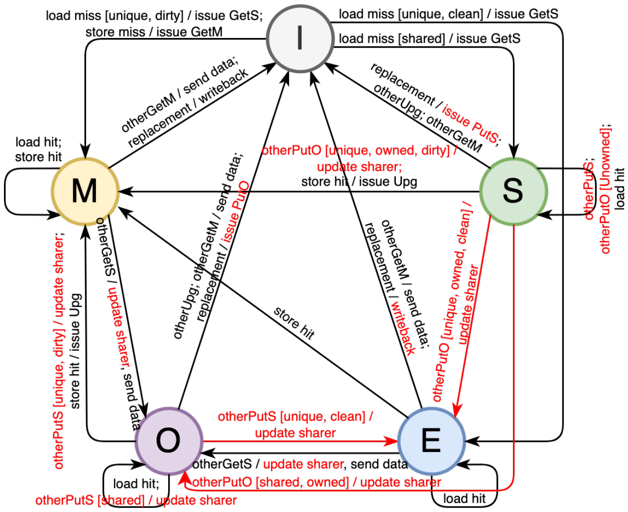

I
M
S
O
E
load miss [unique, dirty] / issue GetS;
store miss / issue GetM
load miss [unique, clean] / issue GetS
load miss [shared] / issue GetS
otherGetM / send data;
replacement / writeback
replacement / issue PutS;
otherUpg; otherGetM
load hit;
store hit
otherPutO [unique, owned, dirty] /
update sharer;
store hit / issue Upg
otherPutS;
otherPutO [Unowned];
load hit
otherUpg; otherGetM / send data;
replacement / issue PutO
otherGetM / send data;
replacement / writeback
otherGetS / update sharer, send data
store hit
otherPutO [unique, owned, clean] /
update sharer
otherPutS [unique, dirty] / update sharer;
store hit / issue Upg
otherPutS [unique, clean] /
update sharer
otherGetS / update sharer, send data
otherPutO [shared, owned] / update sharer
load hit;
otherPutS [shared] / update sharer
load hit

Fig. 5: Stable state transitions in PECC protocol with differences compared to MOESI protocol highlighted in red.

TABLE II: Stable states in PECC

State	Permission	Dirtiness	Uniqueness	Ownership
M	Read-write	Dirty	Unique	Owned
O	Read-only	Clean/Dirty	Shared	Owned
E	Read-only	Clean	Unique	Owned
S	Read-only	Clean/Dirty	Shared	Unowned
I	No permission	N/A	N/A	Unowned

ICH, in the case when no L1 controller holds the ownership of privately cached data, the LLC is the owner because it also holds the consistent copy of the data. For example, suppose that C_1 has a cache line in S. Then, when another core, C_2, makes a request to that same line cached in C_1's L1, it is the LLC that responds to the GetS or GetM request if there is no owner in L1. We observe that the same situation becomes problematic for ECH. Due to the exclusive property, it is not possible for the LLC to hold a valid copy of privately cached data. As a result, the main memory, which also has a consistent copy of the cache line, is deemed the owner. In summary, in an ECH, accesses to private cached data in L1 would necessitate communication with the main memory, thereby degrading the performance [25]. To address this issue, we propose the second invariant to ensure that any coherence request to privately cached data in L1 can be directly supplied by L1 caches.

Invariant 2. *In an ECH, the ownership of any privately cached data must be retained in L1.*

Based on these invariants, we propose PECC protocol for the ECH described in the system model by extending the conventional MOESI protocol. The key reason why MOESI fails to satisfy these invariants is that writeback is issued upon replacing a cache line in the O state where sharers exist. If the data is written back to the LLC, cache line copies in the S state are invalidated, violating Invariant 1. Also, if one chooses to write back the data directly to the main memory without invalidating sharers in L1, the ownership is downgraded to the main memory, breaking Invariant 2. To maintain both invariants, PECC employs sharer tracking that records the cache line sharers and performs ownership transfer to one of the sharers upon evicting cache line in the O state.

Sharer tracking. To track the cores sharing a cache line, PECC maintains a bit vector of size $N - 1$ to store the sharers in L1. The owner of each cache line updates the sharer information by observing GetS and PutS messages on the bus. Note that this requires that the replacement of a cache line in the S state generates an explicit PutS, whereas a cache line in the S state in MOESI protocol can be silently evicted.

Ownership transfer. When a cache line in the O state gets evicted, PECC uses a PutO message to transfer the ownership to one of the other remaining sharers. Note that when a cache line is in the O state, we are guaranteed to have at least one other core that has a privately cached copy of the same cache line in the S state. Since we extend the cache line with sharer tracking information, each privately cached line knows other sharers of the same cache line. Thus, the cache controller evicting the cache line in the O state can select one of the existing sharers as the next new owner of the cache line. After selecting the new owner, the L1 cache controller generates a PutO message on the bus with the new owner identifier and the recent sharer information. All cores' cache controllers receive this PutO message, and the cache controller associated with the new owner claims itself as the owner of the cache line by copying the sharer information and moving the cache line to the O state. Note that this ownership transfer does not writeback data to the LLC.

In summary, PECC protocol satisfies the invariants by ensuring the ownership and data remain in L1 unless a replacement of a unique cache line happens (i.e. M or E state). Note that the M and E state already hold the ownership for a unique cache line that is dirty and clean respectively [14]. The definitions of stable states and coherence requests in PECC are summarized in Table II and Table III, respectively. Figure 5 shows the stable state transitions in PECC. Each transition is labelled with a list of events (separated with “;”) that trigger this transition. The event format is “event [condition] / action”. The condition specifies the cache line properties after the transition. Take the transitions for a load miss in state I as an example, where the possible next states are S, E, and M. In this case, the state properties are inferred by the metadata associated with the data response. For instance, if the data response is sent through cache-to-cache transfer by another core, it means that the data is shared, so the target state is S. On the other hand, if the data response comes from the lower levels (i.e. the LLC or the main memory), it means that the data is unique, and the target state is further determined by the dirtiness of the data (i.e. M if dirty otherwise E).

Transient states. For PECC to deliver high performance, we use transient states in the protocol. A transient state is an intermediary state between two stable states. Starting from one

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, "Exclusive Hierarchies for Predictable Sharing in Last-level Cache," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

TABLE III: Coherence requests in PECC

Request	Description
GetS	Obtain a cache line with read-only permission.
GetM	Obtain a cache line with read-write permission.
Upg	Upgrade a cache line from read-only to read-write permission.
PutS	Evict a cache line in the S state. The request is used to notify other cores that the requestor is no longer a sharer of the cache line.
PutO	Evict a cache line in the O state. The request specifies one of the sharers as the new owner. In our case, we select the sharer with the smallest ID as the new owner. Also, the sharer information for the cache line is passed along with the request.
PutD	Evict and **writeback** a cache line in the E or the M state to the LLC. The writeback data is transmitted along with the request.

of the stable states, it is possible that a cache line's state may transition through multiple transient states before reaching the final stable state. This is essential in allowing for interleaved memory requests [14]. PECC uses a TDM arbiter with a slot width that only covers the time required to broadcast the request without waiting for the entire transaction to complete. This means that although a core can guarantee that no request interleaving from other cores can be observed when it issues a request within its assigned TDM slot, it is possible that the core observes other cores' requests while it is waiting for the response. Table IV shows the complete protocol table with the transient states to handle such request interleaving. To illustrate their use, consider an example with two cores C_1 and C_2. Initially, C_1 issues a store that results in a miss in its private L1 cache. Hence, C_1's cache controller generates a GetM and enters a transient state IM$^{\mathrm{d}}$ once C_1 is granted its slot to broadcast the request. The state name IM$^{\mathrm{d}}$ indicates that between stable states I and M there is a transient state where d indicates that the L1 cache controller is awaiting a data response. Suppose that while C_1 holds the requested cache line in the IM$^{\mathrm{d}}$ state, C_2 encounters a load miss to the same cache line and issues a GetS request. Instead of blocking the request until the data response arrives, C_1 processes it by updating the sharer info (action US) and add it as a pending requestor (action AR), then moves to another transient state IM$^{\mathrm{d}}$O. After receiving the data response, C_1 completes its own store request (action SH) and then send the modified cache line to C_2 (action SD). Finally, C_1 clears the requestor information (CR) and transitions to state O. Recall that C_1 enters the O because C_2 has the same cache line data in the S state. Other interleaving of requests are handled in a similar way.

Exclusive LLC controller. In PECC, the LLC behaves as a victim cache for the L1 caches. This means that the LLC gets populated by writeback data from PutD requests from the L1 caches. The LLC controller observes the events on the request bus and inserts them in the corresponding bank's input queue. This only occurs when the LLC controller notices that no other L1 caches have the requested cache line. We identify this absence of cache line in the L1 caches using the shared wired-OR line on the request bus. Once the bank is idle, it picks up the request at the head of the queue and becomes busy for a duration determined by the request type and cache state. The operations the bank must do for the different request types are described next.

- GetS/GetM: If the requested data is found in the bank (cache hit), then the bank reads the data and deallocates the cache entry. The data from the bank is sent to the requesting core. However, if the requested data is not found in the bank (cache miss), the bank generates a main memory read request and becomes free to process the next request in its queue while the main memory read is being processed. This allows parallel processing of the memory read and bank access. In the worst case, accessing the bank takes t_{BANK} to process GetS/GetM requests, irrespective of the request hit status.
- PutD: If the targeted cache entry CE for a PutD request is empty or contains a clean cache line, the bank writes PutD's data to CE and sends an acknowledgement (Ack) to the requestor. This also takes the bank t_{BANK} to process at worst. However, if the data in CE is dirty, the bank first reads the dirty cache line and places it in a miss status handling register, taking t_{BANK}. Then, the bank writes PutD's data to CE, taking another t_{BANK}. After the bank completes the write, it issues a main memory write of the dirty data in the register and completes the processing without waiting for the main memory response. In this case, since two sequential accesses to the bank are required (i.e. a read followed by a write), the processing time is $2 \cdot t_{BANK}$.

If a main memory request (read/write) is issued when processing a request, the response to the requestor (data for read/Ack for write) is sent when the main memory request completes. In this case, the LLC controller forwards the response from the main memory to the requestor without accessing the bank.

Hardware overhead. In PECC protocol, the primary hardware overhead is the L1 tag storage overhead required to store the sharer list, which adds an additional $N-1$ bits in tag storage per L1 cache line. Note that PECC protocol has the same number of stable states as MOESI protocol, thus incurring no overhead in state encoding. We argue that the storage overhead for sharer tracking is small for a bus-based system with a small core count. For example, in the eight-core system used in our evaluation (Section VII), the overhead is 18% in the tag storage or only 1.3% in the total storage (combining the tag and data). While this can potentially lengthen the access time of the tag array, its impact on the critical path of a cache hit is minimal, which is dominated by the data array access.

B. Bus Arbitration

This section describes the details of the TDM-OA arbiter pair that predictably manage the split-transaction bus while promoting parallelism in the cache resource.

TDM arbiter at request bus. The request bus arbiter is a work-conserving time-division-multiplexing (TDM) arbiter. The TDM arbiter assigns cores to a periodic time slot in a round-robin order. Work-conserving means that the TDM arbiter skips a core if it has no work to do (i.e. no coherence

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

State	Core Events			Bus Events							
	load	store	replacement	OwnData	OwnAck	OtherUpg	OtherGetS	OtherGetM	OtherPutS	OtherPutO	OtherPutD
I	issue GetS / IS^d	issue GetM / IM^d	X	X	X	-	-	-	-	-	-
S	LH	issue Upg, SH / M	issue PutS / I	X	X	- / I	-	- / I	-	US / E, M, O, S	X
M	LH	SH	issue PutD / I^A	X	X	X	US, SD / O	SD / I	X	X	X
E	LH	SH / M	issue PutD / I^A	X	X	X	US, SD / O	SD / I	X	X	X
O	LH	issue Upg, SH / M	issue PutO / I	X	X	- / I	US, SD	SD / I	US / E, M, O	X	X
IS^d	X	X	X	LH, SD, CR / E, M, S	X	- / IS^{d}I	AR, US	AR / IS^{d}I	US / -	CR, US / IO^d, IS^d	X
IO^d	X	X	X	LH, SD, CR / E, M, O	X	- / IO^{d}I	AR, US	AR / IO^{d}I	US / -	X	X
IM^d	X	X	X	SH / M	X	X	AR, US / IM^{d}O	AR / IM^{d}I	X	X	X
IS^{d}I	X	X	X	LH, SD, CR / I	X	X	-	-	X	-	-
IO^{d}I	X	X	X	LH, SD, CR / I	X	X	-	-	X	-	-
IM^{d}I	X	X	X	SH, SD, CR / I	X	X	-	-	X	X	X
IM^{d}O	X	X	X	SH, SD, CR / O	X	X	AR, US	AR / IM^{d}I	X	X	X
I^A	X	X	X	X	- / I	-	-	-	-	-	-

Format: “action / next state”.
- Next state is ignored if remains the same.
- “-” indicates no action or transition is required.
- “X” indicates impossible transitions.

Action acronyms

LH	Load hit
SH	Store hit
SD	Send data if owned and requestors exist
US	Update sharer info
AR	Add requestor
CR	Clear requestor info

TABLE IV: PECC protocol table. If multiple next states are listed, the correct one is determined by the cache line properties.

request is required) and directly assigns its slot to the next core which needs the request bus following the round-robin order. Notice that we only require the slot width to be large enough to transmit a request over the request bus (latency of t_{REQ}) rather than the worst-case latency of a memory transaction as assumed in prior works [6], [7], [11], [13]. Another difference is that the TDM bus implements a counter that increments whenever a core is granted access to the request bus. Every request is associated with the counter value as its request idientifier (ID) when being broadcasted on the request bus. The request ID is used to compare the request broadcast time (i.e. the request age) in the response bus arbiter. We say that a request is older if it has a smaller request ID compared to another request. To account for counter overflow, the counter size is required to be at least twice as large as the maximum difference between the IDs of two pending broadcasted requests in the system, which is upper bounded by the WCL of requests divided by t_{REQ}. We show how to derive a request WCL bound in Section VI.

OA arbiter at response bus. One approach to ensure low WCL bounds for a split-transaction bus is to use a pipelined bus (PIPE) [8] where responses are serviced in the same order of requests being broadcasted on the request bus. However, PIPE delays sending the response of a younger request until the older requests receive their responses. This problem gets aggravated in multicore systems supporting cache-to-cache transfer and a shared LLC because a low-latency cache-to-cache response or a LLC hit response may be blocked by a main memory response taking up a longer latency. This degrades the system throughput. One way to resolve this throughput issue is to use a FCFS arbiter at the response bus. However, we note that using FCFS can lead to a larger WCL. To resolve the WCL issue without significantly impacting system throughput, we propose to employ an oldest-age (OA) arbiter: among all ready responses (i.e. responses that can be issued on the bus), the OA arbiter prioritizes sending the response of the oldest request.

Figure 6 gives an example of the operation of the OA arbiter and compares it against PIPE and FCFS. In the example, $r_{i,j}$ stands for the j^{th} request issued by core C_i. $r_{3,1}$ and $r_{4,1}$ are GetM requests to the same cache line, and $r_{4,1}$ depends on $r_{3,1}$. It means that $r_{3,1}$ must complete first, and the response to $r_{4,1}$ only becomes ready and is forwarded by C_3 after C_3 updates the cache line. In this case, $r_{3,1}$ requires a main memory read. Notice that in the PIPE approach, the long-latency response of $r_{3,1}$ effectively blocks the low-latency responses of $r_{1,1}$ and $r_{2,1}$ (e.g. LLC hit). In both FCFS and OA, $r_{1,1}$ and $r_{1,2}$ can complete without waiting for $r_{3,1}$, and C_1 and C_2 can insert new requests once the previous ones complete. Within the same interval of time, 7 requests are completed in FCFS and OA compared to only 4 requests in PIPE. However, in FCFS, the response to $r_{4,1}$ is delayed by almost $3 \cdot t_{RESP}$, compared to PIPE. Recall that $r_{4,1}$ depends on $r_{3,1}$ and requires $r_{3,1}$ to complete first before its response is ready. While waiting for $r_{3,1}$ to complete, the system can generate new requests (e.g. $r_{1,2}$, $r_{2,2}$, and $r_{1,3}$) and have their responses ready to interfere with $r_{4,1}$. Note that this pattern can be extended if a longer chain of dependency is established in a system with more cores, incurring an even larger latency than the example scenario. Since OA always prioritizes sending the response to the oldest request, the response is at most delayed by only one t_{RESP} compared to PIPE. The one t_{RESP} delay is caused by the fact that the response bus is busy under service when the response to the oldest request becomes ready. For example, when the response of $r_{3,1}$ becomes ready, the response bus is currently busy servicing $r_{1,2}$.

VI. Analytical Worst-case Bound Derivation

In this section, we derive the WCL[3] of coherence requests for loads and stores issued by a core. We first define the request processing model, acting as a basis for the derivation of WCL bounds. Then, we prove the WCL bound of PECC for loads and stores to be asymptotically linear in the number of cores.

A. Request Processing Model

When a coherence request enters the system (i.e. generated by the L1 controller for a core’s load or store that misses in

[3] In this paper, “WCL is ...” refers to an upper bound of the actual WCL.

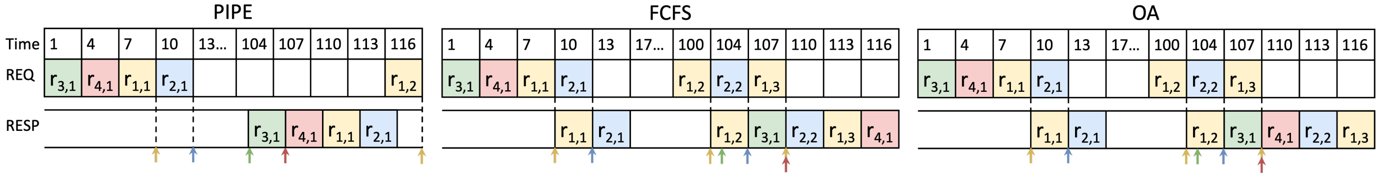

PIPE
Time 1 4 7 10 13… 104 107 110 113 116
REQ $r_{3,1}$ $r_{4,1}$ $r_{1,1}$ $r_{2,1}$ $r_{1,2}$
RESP $r_{3,1}$ $r_{4,1}$ $r_{1,1}$ $r_{2,1}$
FCFS
Time 1 4 7 10 13 17… 100 104 107 110 113 116
REQ $r_{3,1}$ $r_{4,1}$ $r_{1,1}$ $r_{2,1}$ $r_{1,2}$ $r_{2,2}$ $r_{1,3}$
RESP $r_{1,1}$ $r_{2,1}$ $r_{1,2}$ $r_{3,1}$ $r_{2,2}$ $r_{1,3}$ $r_{4,1}$
OA
Time 1 4 7 10 13 17… 100 104 107 110 113 116
REQ $r_{3,1}$ $r_{4,1}$ $r_{1,1}$ $r_{2,1}$ $r_{1,2}$ $r_{2,2}$ $r_{1,3}$
RESP $r_{1,1}$ $r_{2,1}$ $r_{1,2}$ $r_{3,1}$ $r_{4,1}$ $r_{2,2}$ $r_{1,3}$

Fig. 6: Comparison of response bus arbiters. Events of processing requests from the same core use the same color. ↑ marks the ready time of the response. The FCFS timeline is drawn assuming the worst-case arbitration decision for $r_{4,1}$'s response.

L1), it needs to be serviced by one or more shared hardware resource(s) in a specific order. The resources we consider include the request bus (REQ), the response bus (RESP), the LLC banks (BANK), and the main memory (MEM). Based on the required hardware resources and the access order, we classify the coherence requests into the following four types: ($\mathcal{T}_1$) →REQ. The first category represents the request type that only requires a request bus broadcast to mark its completion without the need for any explicit response message. The requests belonging to this type are Upg, PutS and PutO. ($\mathcal{T}_2$) →REQ→RESP. Requests in this category must be serviced by the request bus and the response bus in sequence. This category includes GetS/GetM requests that hit in another core's L1 cache and trigger cache-to-cache transfers without accessing the LLC bank. ($\mathcal{T}_3$) →REQ→BANK→RESP. This category includes any request that needs to be processed by the request bus, the LLC bank, and finally the response bus in order. Requests falling in this category can be either a GetS/GetM request hitting in the LLC or a PutD request that does not trigger extra eviction from the LLC bank to the main memory. ($\mathcal{T}_4$) →REQ→BANK→MEM→RESP. The final category represents any request that requires access to the main memory. Such a request is either a GetS/GetM missing in all L1 caches and the LLC or a PutD request that triggers writeback from the LLC bank to the main memory.

Any request arrives at REQ immediately upon entering the system. When a request arrives at a resource, it experiences a latency consisting of the time waiting to be arbitrated and then serviced by that resource. After a request finishes its processing in the current resource, it immediately moves to the next resource except for the case where it depends on another request. Recall the example of transient states in Section V-A: at the time C_2 broadcasts its GetS request, the GetM request issued by C_1 is still pending, but C_1 already holds the ownership of the cache line. As a result, after the GetS of C_2 finishes at REQ, it cannot move to the next resource (i.e. RESP) immediately. Instead, it must wait until C_1 completes its GetM request; thus, finishing the store operation at which point C_1 can respond to the GetS request of C_2. Note that this kind of request blocking due to request dependencies is only possible between GetS/GetM requests, but not other requests: $\mathcal{T}_1$ requests (Upg/PutS/PutO) only transmit control information and immediately finish after REQ; while for PutD requests, the writeback data from L1 to the LLC is transmitted together with the request itself on REQ, meaning that data is available in the LLC without waiting for the request completion. We now formally define request dependency to capture the notion of request blocking.

Definition 1 (Request Dependency)**.** *Given two* GetS/GetM *requests, r_a and r_b issued by different L1 controllers, we say that r_b directly depends on r_a if the following conditions are satisfied. (1) Both r_a and r_b are requests to the same cache line. (2) r_a finishes at REQ before r_b. (3) At the time when r_b finishes at REQ, the requestor of r_a holds the ownership of the cache line, but r_a has not completed yet (i.e. r_a has not finished at RESP).*

When one or more requests directly depend on a request r_a, all the dependent requests wait for r_a's completion and then move to RESP. At the same time, the requestor of r_a associates the response with the earliest request ID among dependent requests. When the OA arbiter selects the response based on such ID, the response bus broadcasts the data to all dependent requests supported by cache-to-cache transfer and taking t_{RESP} time. This also implies that the dependent requests must belong to $\mathcal{T}_2$.

Additionally, multiple requests can form a dependency chain, where the earliest request in the chain blocks the processing of all other requests. We extend the concept of dependency to eventual dependency to capture the earliest request that causes such blocking.

Definition 2 (Eventual Dependency)**.** *Given two* GetS/GetM *requests, r_a and r_b, we say that r_b eventually depends on r_a if the following conditions are satisfied. (1) r_b directly depends on r_a, or there are k ($k \geq 1$) intermediate requests such that r_1 directly depends on r_a, r_i directly depends on r_{i-1} for ($2 \leq i \leq k$) and r_b directly depends on r_k. (2) At the time when r_b finishes at REQ, r_a has not completed yet. (3) r_a is the oldest request that satisfies (1) and (2).*

Example of request dependency. Figure 7 shows the request dependencies for an example timeline in the form of a directed acyclic graph. In this case, each of the five L1 controllers issue a coherence request. For example, L1 controller A issues a coherence request r_a of GetM. All requests are made to the same cache line. The timeline also indicates the owner of the target cache line at each specific time. A black arrow represents a direct dependency, while a red arrow shows an eventual dependency. Note that r_b and r_c share the same response as they directly depend on the same request r_a. r_c

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

directly depends on r_a, but not r_b because the requestor of r_b does not hold the ownership of the cache line when r_c is broadcasted on the request bus. Additionally, r_e eventually depends on r_c, but not r_a. This is because r_a has finished by the time r_e finishes at REQ, and r_c is the oldest pending request that blocks r_e from entering RESP.

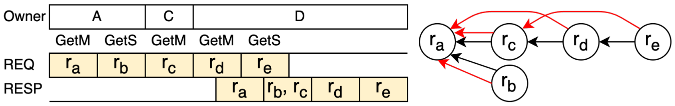


Fig. 7: Example of request dependencies.

B. Analysis

After providing necessary definitions and formalizing our processing model, we can now proceed by deriving the WCL bound for a coherence request under analysis, denoted as r_{ua}.

Lemma 1. *The WCL in REQ is* $(N+1)\cdot t_{REQ}$.

Proof. The work-conserving TDM arbiter at the request bus assigns a TDM slot to each core in a round-robin manner. In the worst case, the request under analysis, r_{ua} just misses its assigned slot, and all other cores issue requests to occupy the remaining slots in the TDM period. As a result, r_{ua} waits no longer than one full TDM period to win arbitration and then t_{REQ} to be serviced by the request bus, yielding a latency bound of $(N+1)\cdot t_{REQ}$. □

Lemma 2. *The WCL in BANK is* $(2N-1)\cdot t_{BANK}$ *if* r_{ua} *is* GetS/GetM *or* $2\cdot N\cdot t_{BANK}$ *if* r_{ua} *is* PutD.

Proof. Note that only requests in $\mathcal{T}_3$ or $\mathcal{T}_4$ are processed by the LLC bank, which is one of GetS, GetM or PutD. The WCL for the bank to process a single request is $2\cdot t_{BANK}$. This happens when the request is a PutD of type $\mathcal{T}_4$ that requires a replacement in the bank, in which case a bank read and a bank write are required to happen in sequence. For a GetS/GetM, the latency to process it is t_{BANK} as only one bank read is required. For a PutD of type $\mathcal{T}_3$, only a bank write is required, which also takes t_{BANK} to complete. When a request arrives at its corresponding bank, it needs to wait for at most $N-1$ other requests. This is because the bank services the requests in FCFS order, and there is at most one request from each core. Thus, at worst, a request needs to wait for $(N-1)\cdot 2\cdot t_{BANK}$ before it is serviced when the bank is busy processing other $N-1$ PutD of $\mathcal{T}_4$. Combined with the processing time of r_{ua} itself, the WCL in BANK is $(2N-1)\cdot t_{BANK}$ for GetS/GetM and $2\cdot N\cdot t_{BANK}$ for PutD. □

Lemma 3. *The WCL in RESP is* $N\cdot t_{RESP}$.

Proof. The OA arbiter guarantees that a ready response to the oldest request is sent first. There can be at most N ready responses in the system since each core issues at most one request at a time. At worst, the response of r_{ua} is associated with the largest request ID among the N ready responses. In this case, the response to r_{ua} needs to first wait for the other $N-1$ responses to be sent, taking $(N-1)\cdot t_{RESP}$, then it is serviced, taking t_{RESP}. This WCL guarantee holds because any newly generated response must be associated with a larger request ID and cannot delay sending the response to r_{ua}. In some cases, multiple requests are serviced by the same response if all of them directly depend on the same request. When this happens, this clearly only shortens the latency to send the response to r_{ua} and does not change the bound. □

Lemma 4. *If* r_{ua} *is a* Get *request (*GetS/GetM*), its WCL is* $(N+1)\cdot t_{REQ}+(2N-1)\cdot t_{BANK}+t_{MEM}+N\cdot t_{RESP}$.

Proof. By cases, if r_{ua} directly depends another request.

Case (1): r_{ua} does not directly depend on any other request. In this case, r_{ua} immediately moves to the next resource once it finishes processing on the current one. Therefore, we can sum the WCL in each resource to obtain a safe upper bound on the overall WCL. Note that Lemmas 1-3 compute the WCL of REQ, BANK, and RESP, while the WCL in MEM is t_{MEM} as defined in the system model. The worst case happens when r_{ua} belongs to type $\mathcal{T}_4$ which requires all four types of resource, yielding the bound in the lemma.

Case (2): r_{ua} directly depends on another request. Note that since r_{ua} directly depends on a request, by definition there must exist a request, call it r_p, such that r_{ua} eventually depends on r_p. We divide the latency components of r_{ua} into two parts as shown in Figure 8: $\mathcal{L}_1$ is the interval between the time r_{ua} arrives at REQ to the time r_p arrives at the response bus, and $\mathcal{L}_2$ is the interval between the time r_p arrives at RESP to the finish time of r_{ua}.

Bounding $\mathcal{L}_1$: We do a case analysis whether r_p directly depends on another request. **(2.1)**: r_p directly depends on another request r_q. Note that by definition, r_{ua} does not eventually depend on r_q. It means that r_q must have already completed when r_{ua} finishes at REQ, which further implies that the response to r_p must be ready by the time r_{ua} finishes at REQ. Therefore, $\mathcal{L}_1$ is bounded by the request bus latency of r_{ua}, which is $(N+1)\cdot t_{REQ}$ as shown in Lemma 1. **(2.2)**: r_p does not directly depend on another request. By the definition of request dependency, r_p must have finished on REQ before r_{ua}. Therefore, $\mathcal{L}_1$ can be bounded by summing the WCL of r_{ua} in REQ, and the WCL of r_p in BANK and MEM, which is $(N+1)\cdot t_{REQ}+(2N-1)\cdot t_{BANK}+t_{MEM}$.

Bounding $\mathcal{L}_2$: At the start of $\mathcal{L}_2$, there can be at most $N-1$ requests older than r_{ua} waiting to be completed at RESP since each core issues at most one request at a time. Among the $N-1$ requests, there are requests that form the dependency chain from r_p to r_{ua} and requests that do not belong to this dependency chain. For the requests in the dependency chain, once a prior request finishes at RESP, the response to the next request in the dependency chain becomes ready. For the requests that do not belong to the dependency chain, in the worst case all of them can be ready at RESP and arbitrated before r_{ua}. Therefore, based on the operation of the OA arbiter, responses to requests that are younger than r_{ua} cannot be sent during interval $\mathcal{L}_2$. It follows that $\mathcal{L}_2$ must be bounded

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

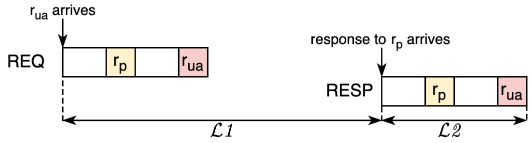


Fig. 8: Latency components of r_{ua} in case 2 of Lemma 5.

by $N \cdot t_{RESP}$, the time required to service the $N-1$ older requests first and then r_{ua}.

Finally, summing the bounds for $\mathcal{L}_1$ and $\mathcal{L}_2$ yields the same bound as in Case (1), thus completing the lemma. □

Lemma 5. *If r_{ua} is a* PutD *request, its WCL is* $(N+1) \cdot t_{REQ} + 2N \cdot t_{BANK} + t_{MEM} + N \cdot t_{RESP}$.

Proof. Since a PutD request cannot depend on another request, the proof follows the same reasoning as in Case (1) of Lemma 4. In the worst case, r_{ua} belongs to type $\mathcal{T}_4$; summing the WCL in each resource based on Lemmas 1-3 and the definition of t_{MEM} yields the result. □

Theorem 1. *The WCL of coherence requests generated by a load or store instruction is* $(2N+2) \cdot t_{REQ} + (4N-1) \cdot t_{BANK} + 2 \cdot t_{MEM} + 2N \cdot t_{RESP}$.

Proof. At worst, a core load or store instruction requires two coherence requests to complete in sequence. This happens for a load/store miss when the L1 cache does not have space for the new cache line. In this case, the cache controller needs to issue a Put request first to evict a victim line and then issue a Get request for the cache line requested by the core. Note that requests belonging to $\mathcal{T}_1$ (Upg/PutS/PutO) have a smaller latency than requests of other types as $\mathcal{T}_1$ requests require only accessing REQ. Therefore, the WCL can be bounded by summing the WCL of a PutD request (Lemma 5) and the WCL of a GetS/GetM request (Lemma 4). □

VII. Evaluation

Experimental setup. We implement PECC in the Gem5 simulator [26]; the source code is publicly available [4]. We simulate a multicore system with up to 8 cores running at 2 GHz. Similar to ARM Cortex R8 [27], the system uses a private instruction and private data cache per core, both configured as 16 KiB 2-way set-associative. For the L2 shared cache, the system employs an 8-way set-associative LLC with 8 internal banks. The default size of the LLC is configured as 1 MiB. All caches use the least-recently-used (LRU) replacement policy. For the main memory, we use a single-port SRAM module that is described in the system model (Section IV-C). We configure L1 cache hit latency to be 1 clock cycle, t_{BANK} = 10 clock cycles, and t_{SRAM} to be 100 clock cycles. Note that t_{MEM} is $100N$ cycles accounting for the interference with at most N outstanding requests in the system. We perform our evaluation using three protocols as follows. (1) PECC with t_{REQ} and t_{RESP} both configured as 3 cycles. (2) ZCLLC from [13]. The TDM slot of ZCLLC is configured to be the maximum transaction time, which is $t_{REQ} + 2 \cdot t_{BANK} + t_{SRAM} + t_{RESP} = 126$ cycles. Note that LLC replacement takes $2 \cdot t_{BANK}$. (3) MOESI protocol with a conventional inclusive LLC and the same split-bus architecture as PECC. This setup is denoted as INCL. We configure INCL to use the same latency parameters as PECC.

Benchmarks. Our evaluation uses a synthetic benchmark and the Splash-3 benchmark suite [28]. The synthetic benchmark stresses the protocol behaviour to explore the WCL using crafted memory traces. In particular, it constructs memory traces following the pattern in Figure 2. We also use the Splash-3 benchmark suite as a representative workload of multicore programming on shared data. The protocol correctness is verified by the internal data consistency check of the synthetic benchmark, as well as ensuring that correct outputs are computed in Splash-3 benchmarks. Note that some Splash-3 benchmarks (barnes, fmm, radiosity, water-nsquared, water-spatial and lu) are designed to have the working set fit in L1 regardless of the program input size [29]. We call these benchmarks as LLC-insensitive because they do not apply enough memory pressure to the LLC; thus, showing limited performance improvement when increasing the LLC size in all setups. On the other hand, the rest of the benchmarks (ocean, raytrace, cholesky, fft and radix) do not have their working sets fit in L1 and are LLC-sensitive. Splash-3 benchmarks also contain atomic instructions for fast synchronization. To support that, in our Gem5 implementation of PECC, the atomic instruction is treated as a special store instruction that keeps the cache line in the M state until it completes. The simulation setup allows the atomic instruction to finish quickly once the cache line is acquired in the M state. Therefore, its timing effect is subsumed in the case of store instructions.

A. WCL for Load/Store Instructions

We first evaluate the WCL for load and store instructions over various setups. Figure 9 and Figure 10 report the observed WCL when varying the number of cores for the synthetic workload and Splash-3 workload, respectively. A solid bar denotes the observed WCL, while the T bar marks the analytical bound. We do not provide the analytical bound for INCL since this is a conventional design without predictability in mind.

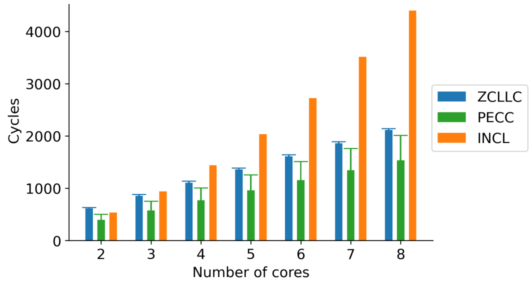


Fig. 9: Observed WCLs of synthetic benchmark.

[4] https://github.com/caesr-uwaterloo/pecc

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel, “Exclusive Hierarchies for Predictable Sharing in Last-level Cache,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2024, pp. 1–11.

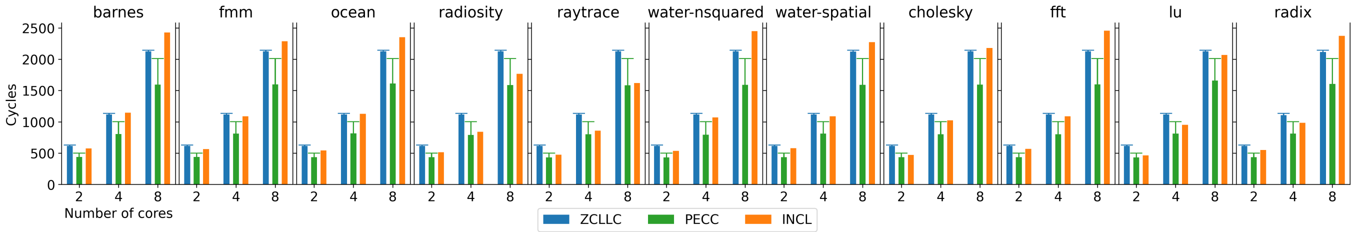


Fig. 10: Observed WCLs of Splash3

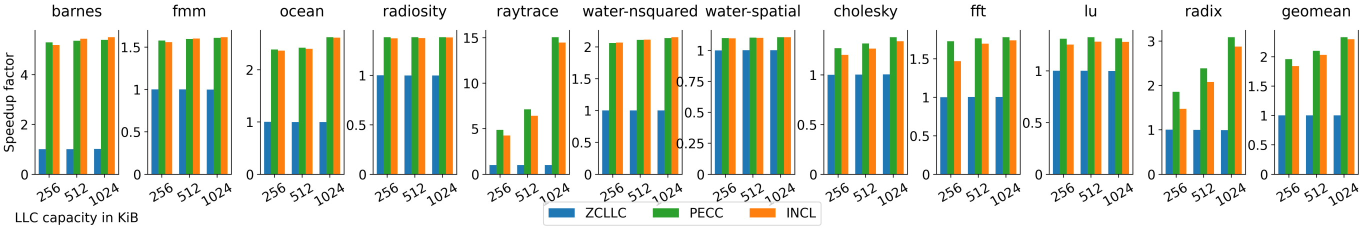


Fig. 11: Average execution time speedup of Splash-3. Baseline is ZCLLC with 256 KiB capacity.

Observations. In the synthetic workload, the observed WCL of INCL grows faster than the analytical bounds of PECC and ZCLLC, both of which scale linearly in the number of cores. However, the observed WCL of INCL under synthetic workload is much larger than the one under Splash-3. The reason is that the synthetic workload is designed to induce the worst-case request pattern described in Section III, which results in quadratic-scaling WCL. However, this scenario was not exercised in Splash-3. Both Splash-3 and synthetic results confirm that the observed WCLs of PECC and ZCLLC fall safely within their analytical bounds. Note that, in PECC, the gap between the observed WCL and the analytical WCL is caused by the pessimism in the analysis. Nonetheless, the analytical bound of PECC for an 8-core system is still 6% smaller than ZCLLC.

B. Average Performance

We use Splash-3 to study average performance benefits from employing an LLC of varying size. We fix the core count to 8 and set the LLC size to either 256 KiB, 512 KiB or 1024 KiB. Note that for inclusive LLC, the effective LLC size is the LLC capacity minus the total private cache capacity. Therefore, compared to PECC, for both ZCLLC and INCL, the effective LLC size is 0 (100% reduction), 256 KiB (50% reduction), 768 KiB (25% reduction) respectively for the LLC size of 256 KiB, 512 KiB, and 1024 KiB. Figure 11 reports the execution time speedup of Splash-3 benchmarks with the baseline of 256-KiB ZCLLC.

Observations. When comparing against ZCLLC, the average performance of INCL and PECC are similar and both outperform ZCLLC (geomean speedup of 2.30× for INCL and 2.33× for PECC with a 1MiB LLC). Although not reported in the figure, we confirmed that ZCLLC’s slowdown is due to the higher average miss latency rather than the L1 hit rate difference. The high average miss latency in ZCLLC is because of the unified arbitration for all shared resources resulting in a large TDM slot. This prevents any shared resource to be utilized even though the current slot owner completes a transaction much sooner (e.g. LLC hit). For instance, consider the benchmark raytrace, the most memory-intensive benchmark, as an extreme example. With a 1MiB LLC, the average miss latency of ZCLLC is 1957 cycles, which is close to its analytical bound of 2142 cycles. In stark contrast, the average miss latencies of PECC and INCL, where hardware parallelism is not limited by the bus architecture and arbitration design, are only 77 and 83 cycles, respectively. Also, the result that the performance of ZCLLC does not improve with a larger LLC for LLC-sensitive benchmarks further emphasizes this observation. Different from ZCLLC, in LLC-sensitive benchmarks, both INCL and PECC show performance improvements with increased LLC capacity. Compared to INCL, the performance gain of PECC has a positive correlation with the extra effective LLC capacity that PECC has over INCL, which reaches the maximum when the LLC size is the smallest (i.e. 256 KiB) and decreases with greater LLC capacity.

VIII. Conclusions

We present an alternative to using an inclusive hierarchy with a shared LLC for multicores used in safety-critical systems. This alternative, PECC, uses an exclusive cache hierarchy with a split-transaction bus, and a novel MOESI-based cache coherence protocol that lowers the WCL bound by 6% and improves the average-case performance by 2.33× when compared to the state-of-the-art approach [13].

Acknowledgments

This work has been supported in part by NSERC. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 10:1–10:20.

[2] R. L. Alena, J. P. Ossenfort, K. I. Laws, A. Goforth, and F. Figueroa, "Communications for Integrated Modular Avionics," in *2007 IEEE Aerospace Conference*, 2007, pp. 1–18.

[3] J. Nowotsch and M. Paulitsch, "Leveraging Multi-core Computing Architectures in Avionics," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[4] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, aug 2020.

[5] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, "Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 57–68.

[6] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable Cache Coherence for Multi-core Real-Time Systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 235–246.

[7] 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.

[8] S. Hessien and M. Hassan, "PISCOT: A Pipelined Split-Transaction COTS-Coherent Bus for Multi-Core Real-Time Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 1, oct 2022.

[9] M. Hossam and M. Hassan, "Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Maggio, Ed., vol. 231. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 17:1–17:23.

[10] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[11] Z. Wu and H. Patel, "Predictable Sharing of Last-Level Cache Partitions for Multi-Core Safety-Critical Systems," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1273–1278.

[12] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Comput. Surv.*, vol. 48, no. 2, nov 2015.

[13] Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 1–1.

[14] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed. Morgan Claypool Publishers, 2020.

[15] P. Sweazey and A. J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *SIGARCH Comput. Archit. News*, vol. 14, no. 2, p. 414–423, may 1986.

[16] J. Sim, J. Lee, M. K. Qureshi, and H. Kim, "FLEXclusion: Balancing Cache Capacity and on-Chip Bandwidth via Flexible Exclusion," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, p. 321–332, jun 2012.

[17] "Intel Core i9-13900 Processor," https://www.intel.ca/content/www/ca/en/products/sku/230499/intel-core-i913900-processor-36m-cache-up-to-5-60-ghz/specifications.html, accessed on July 5, 2023.

[18] *Software Optimization Guide for the AMD Zen4 Microarchitecture*, 1st ed., https://www.amd.com/en/support/tech-docs/software-optimization-guide-for-the-amd-zen4-microarchitecture, AMD, 2023.

[19] *Arm Cortex-R82 Processor Technical Reference Manual*, r0p2 ed., https://developer.arm.com/documentation/101548/0002/The-Cortex-R82-processor, ARM, 2022.

[20] 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, p. 05:1–05:48, Jun. 2016.

[21] D. Hardy and I. Puaut, "WCET analysis of instruction cache hierarchies," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 677–694, 2011, special Issue on Worst-Case Execution-Time Analysis.

[22] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," Cryptology ePrint Archive, Paper 2015/905, 2015.

[23] *ARM CoreLink CCI-550 Cache Coherent Interconnect, Technical Reference Manual*, r1p0 ed., https://developer.arm.com/documentation/100282/0100, ARM, 2018.

[24] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures," in *2010 18th IEEE Symposium on High Performance Interconnects*, 2010, pp. 1–6.

[25] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing Cache Architectures and Coherency Protocols on X86-64 Multicore SMP Systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 413–422.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[27] *Arm Cortex-R8 MPCore Processor*, r0p3 ed., https://developer.arm.com/documentation/100400/0003/revisions, ARM, 2019.

[28] 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.

[29] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.