Predictable Cache Subsystem Design for Real-time Multicores

by

Xinzhe Wang

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

Waterloo, Ontario, Canada, 2024

© Xinzhe Wang 2024

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 4 uses the content of the following publication.

X. Wang, Z. Wu, R. Pellizzoni, and H. Patel. Exclusive Hierarchies for Predictable Sharing in Last-level Cache. In *2024 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2024 [1].

The use of its content in this thesis have been approved by all co-authors.

Abstract

This thesis focuses on the design of the cache subsystem for multicore real-time systems. Specifically, it looks at the two common contention sources in the cache subsystem that undermine the system's timing predictability: temporal contention (queuing delays resulting from simultaneous shared resource accesses) and spatial contention (conflicts in cache line allocations). Accordingly, this thesis proposes two solutions to address the two problems. First, to address temporal contention induced by the shared last-level cache (LLC), we propose PECC, a predictable exclusive cache coherence mechanism. Unlike the common choice of inclusive cache hierarchies, PECC incorporates an exclusive LLC free of back invalidation, which is a significant contributor to the per-request worst-case latency (PR-WCL) in inclusive hierarchies. As a result, PECC reduces the bound of PR-WCL by 6% and improves the average performance by 2.33× over the predictable solution with an inclusive LLC. Second, to address spatial contention in the shared LLC, we propose ParRP, a novel cache partitioning scheme that provides cache space isolation for shared data. Conventionally, achieving cache space isolation for shared data is challenging due to its intrinsic sharing nature, which violates the resource isolation principle. ParRP overcomes this by partitioning the replacement policy instead of cache entries. Consequently, ParRP guarantees isolation property even with shared data, enabling isolated worst-case execution time (WCET) analysis without interference from other cores. Our evaluation shows that this leads to a 2.4× reduction in the WCET of multi-threaded tasks with shared data at the cost of a 16.5% decrease in average performance.

Acknowledgments

I would like to express my foremost thanks to my two supervisors, Professor Hiren Patel and Professor Rodolfo Pellizzoni, for their mentorship and encouragement throughout my master journey. Their expertise and insight have greatly enlightened my thinking in my pursuit of computer architecture research, which made a significant impact on the completion of this thesis.

I would like to thank my research group members for their support. Special thanks to Zhuanhao for his prompt assistance, which greatly helped me adapt smoothly to graduate study.

Above all, I thank my family for your love and support. Heartfelt thanks to my love, Yijing, for her companionship and kindness over the past two years. The memories we share are a steady source of strength for me.

Table of Contents

Author's Declaration ii

Statement of Contributions iii

Abstract iv

Acknowledgments v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Thesis Contributions 3

1.2 Thesis Structure 4

2 Background 5

2.1 Cache Hierarchies 5

2.2 Cache Coherence 6

2.3 Cache Partitioning 8

3 Related Works 10

3.1 Related Works Targeting Temporal Contention 10

3.2 Related Works Targeting Spatial Contention 11

4 Exclusive Hierarchies for Predictable Sharing in Last-level Cache 13
4.1 Motivation . . . 14
4.2 System Model . . . 17
4.2.1 Processing Cores and Cache Hierarchy . . . 17
4.2.2 Coherent Interconnect . . . 19
4.2.3 Main Memory . . . 19
4.3 Predictable Exclusive Cache Coherence . . . 20
4.3.1 PECC Protocol . . . 20
4.3.2 Bus Arbitration . . . 27
4.4 Analytical Worst-case Bound Derivation . . . 29
4.4.1 Request Processing Model . . . 29
4.4.2 Analysis . . . 31
4.5 Evaluation . . . 34
4.5.1 WCL for Load/Store Instructions . . . 35
4.5.2 Average Performance . . . 36
4.6 Conclusions . . . 38

5 ParRP: Enabling Space Isolation in Caches with Shared Data 39
5.1 System Model . . . 40
5.1.1 Multicore Architecture . . . 40
5.1.2 Cache Semantics . . . 40
5.2 Interference in Shared LLC . . . 42
5.2.1 Goal of Isolation . . . 42
5.2.2 Violation of Isolation under Shared Data Interference . . . 42
5.2.3 Insights . . . 43
5.3 ParRP . . . 45
5.3.1 Replacement Policy Partitioning for Shared LLC . . . 46
5.3.2 Multi-level Cache Isolation under Cache Coherence . . . 50

5.4 ParRP's Example with MSI protocol 55
5.5 ParRP's Properties 59
5.6 Evaluation 61
5.6.1 Verification 62
5.6.2 WCET 64
5.6.3 Average Performance 65
5.7 Conclusions 68

6 Conclusion and Future Work **69**

References **70**

List of Figures

1.1 Overview of multicore cache subsystem. 2
1.2 Cache contention in different domains. Colors are used to distinguish cores. 3

2.1 MSI state transitions. 7
2.2 Overview of cache partitioning 8

4.1 Request processing timeline of $read(A)$ request of the core under analysis C_{ua} in a multicore system with N cores. 15
4.2 Worst-case traffic flow in ICH and ECH with operations related to main memory writes highlighted. 17
4.3 Overview of system model. 18
4.4 Stable state transitions in PECC protocol with differences compared to MOESI protocol highlighted in red. 23
4.5 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. 28
4.6 Example of request dependencies. 31
4.7 Latency components of r_{ua} in case 2 of Lemma 4.4. 33
4.8 Observed WCLs of synthetic benchmark. 35
4.9 Observed WCLs of Splash3 36
4.10 Average execution time speedup of Splash-3. Baseline is 256-KiB ZCLLC. . 37

5.1 Overview of system model. 41

5.2 Cache state update with LRU replacement policy 41
5.3 Violations of isolation property in conventional way-based partitioning. . . 43
5.4 Restore isolation property by replacement policy partitioning. 45
5.5 Overview of ParRP architecture. 49
5.6 Violations of cache space isolation due to coherence interference. 53
5.7 Example of coherence transactions under ParRP 54
5.8 Synthetic benchmark. 63
5.9 Percentage breakdown of access frequency in BOTS benchmark. 65
5.10 WCET for BOTS benchmarks. 66
5.11 Average Performance. 67

List of Tables

4.1 Cache line properties. 21

4.2 Stable states in PECC 24

4.3 Coherence requests in PECC 24

4.4 PECC protocol table. If multiple next states are listed, the correct one is determined by the cache line properties. 25

5.1 Function semantics. 47

5.2 MSI protocol table. 56

5.3 Updated MSI protocol table to enforce ParRP's rules. '*' indicates that the coherence request is state-preserving in terms of the LLC replacement view. 57

5.4 Replacement view of the updated MSI protocol table. 58

5.5 System configuration. 62

5.6 Per-request worst-case latency. 62

5.7 Guarantee in various setups. 62

Chapter 1

Introduction

Over the past two decades, real-time embedded computing systems have undergone significant developments and now spans many different application domains. Several examples include medical devices, avionics, automotive, and industrial automation [2]–[5]. In the context of real-time systems, correctness depends not only on the computed values but also on the time required to deliver those results (responsiveness). Typically, computations have timing constraints to meet with different criticality levels varying across systems. In hard real-time systems such as electronic control in aircraft, timing constraints are strict, and missing a deadline may result in severe consequences such as the whole system failure and loss of human life.

Many of these real-time systems are now deployed on multicore platforms [6]. This is because multicores promise significant performance improvements, and the opportunity to consolidate multiple functions onto a single platform helps reduce the size, weight, and power consumption. However, employing multicores also poses new challenges in the timing predictability of the real-time systems. Timing predictability refers to the ability to obtain a safe and tight bound on the worst-case execution time (WCET) of software tasks. This is crucial to ensure that tasks can always finish before their deadlines in real-time systems. The culprits behind the poor timing predictability of multicore platforms are the shared hardware resources, such as buses, caches, and memories [4], [5], [7]. These shared hardware resources introduce interference amongst multiple cores. Essentially, a core's access to a shared resource may affect another core's temporal behaviour. This inter-core interference makes it hard to determine a safe yet tight bound of a task's worst-case execution time, which is crucial in schedulability analyses to ensure that tasks always meet their deadlines.

One of the primary interference sources, which is the focus of this thesis, is the cache

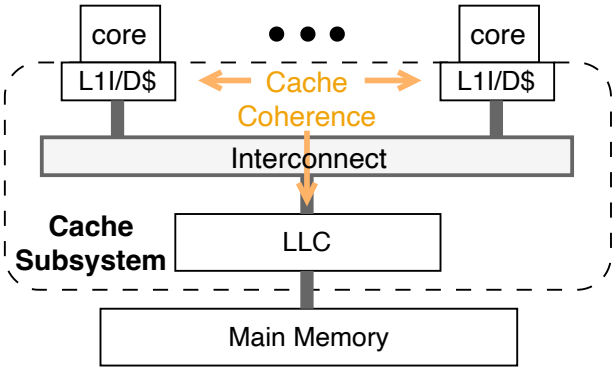
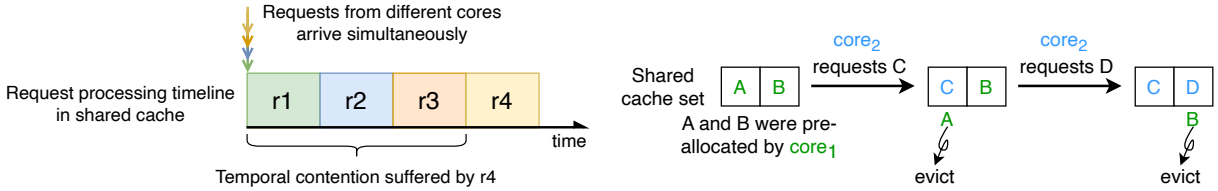


Figure 1.1: Overview of multicore cache subsystem.

subsystem. Modern commercial-off-the-shelves (COTS) multicore platforms often feature multi-level cache hierarchies interfaced before the main memory for high performance [8]–[12]. Typically, each core has its private split data and instruction cache at the level one (L1), and a much larger last-level cache (LLC) shared amongst all cores. This cache hierarchy is also equipped with hardware cache coherence to maintain a consistent view of privately cached data copies amongst all cores [13], which is crucial to enable efficient shared memory communication. Figure 1.1 shows a high-level diagram of the described cache subsystem. These high performance features often involve sophisticated interactions amongst multiple components such as caches and buses, which can lead to significant timing variability and make the system highly unpredictable when under contention [14], [15]. More specifically, the cause of unpredictability is attributed to contention in both temporal and spatial domains, as shown in Figure 1.2. Temporal contention directly impacts the timing of accessing shared resources when multiple cores attempt to access them simultaneously. Spatial contention is conflicts in the allocation of cache lines [16], which affects the presence of data within the cache hierarchy and, in turn, impacts the time to retrieve data. Although the focused platform of this thesis is real-time embedded platforms, it is noteworthy that these contention problems also exist in cloud settings [17]. For example, multiple virtual machines compete for the shared cache resources on a server platform, degrading the quality of service.

To mitigate these types of contention, common solutions use software techniques to impose constraints on task scheduling or data caching. An example using task scheduling constraints is the shared-data-aware scheduler [15]. In that work, the authors identified cache coherence as a main source of timing variability in applications with intensive data sharing. For instance, when a core updates a data, it must invalidate data copies cached by other cores. This delays the timing for other cores to access the data, as they must



(a) Temporal contention.

(b) Spatial contention.

Figure 1.2: Cache contention in different domains. Colors are used to distinguish cores.

wait for the most up-to-date data copy to be located and returned. To mitigate the timing effects due to cache coherence, the proposed scheduler identifies tasks with shared data and never schedules them at the same time. Examples of data caching constraints are bypassing the shared LLC to mitigate shared cache contention [7] and uncaching shared data in private levels to eliminate the need of cache coherence [18]. Clearly, these types of techniques greatly trade off the performance for predictability [19], which defeats the purpose of having caches in the system. We argue that, to achieve a more efficient solution, the hardware itself should be designed to control the contention and be more predictable. Hence, this thesis focuses on cache subsystem designs that enable the high performance features with less unpredictability, which is transparent to software stacks running above.

1.1 Thesis Contributions

This thesis presents two orthogonal solutions on the predictable cache subsystem design for real-time multicores, each aiming at addressing the contention in different domains. The main contributions are as follows.

- We first introduce PECC to bound the temporal contention when using a shared LLC in the cache hierarchy. A shared LLC is a common feature to offer high performance in modern multicore cache hierarchy designs. However, as it introduces an additional level of cache interaction, the request handling steps can become much more intricate, which often introduces a pathological scenario with an extremely high worst-case latency (WCL). In particular, an inclusive LLC design, which is a common design choice used by many prior works [19]–[24], suffers from the problem of back invalidation and is shown to significantly lengthen the per-request worst-case latency (PR-WCL) [23]. Motivated by this, we introduce PECC, a predictable exclusive cache coherence mechanism, as an alternative to setup the LLC as an exclusive cache,

which is naturally free of the timing interference sources presented in the inclusive cache. PECC is predictable in that it provides a bound of PR-WCL. Furthermore, its PR-WCL bound is low, which is linear in terms of the core counts. PECC is the first work that offers a predictable cache coherence solution with an exclusive LLC setup. Compared to all its inclusive counterparts, PECC is able to guarantee a linear PR-WCL bound while maintaining similar performance as general purpose designs.

- We introduce ParRP to resolve the spatial contention problem in the cache hierarchy. Although techniques [19]–[22] on predictable cache coherence such as PECC bound the temporal contention and ensure a safe WCL bound for accessing data, they still permits spatial contention by default. Spatial contention makes it hard to analyze the cache content. As a result, most memory references are unclassified regarding the hit status and treated as main memory access in the worst case. Clearly, this leads to rather pessimistic worst-case bound on the task level as cache hit rate is often high in average case. To cope with spatial contention, typical solutions require partitioning the shared cache amongst multiple cores to provide resource isolation. However, conventional cache partitioning assumes no data sharing because its sharing nature intrinsically violates the isolation principle. Motivated by this, we introduce ParRP, a novel cache partitioning approach that works with shared data. Different from all prior efforts, ParRP aims to provide spatial isolation for shared data, similar to how conventional partitioning provides isolation for private data.

1.2 Thesis Structure

Chapter 2 provides the background on multicore cache system design, covering cache hierarchies, cache coherence, and cache partitioning management schemes. Chapter 3 reviews the prior works on cache management for real-time multicores, classified according to their targeting of temporal and spatial contention. Chapter 4 presents our solution of PECC that primarily targets the temporal contention when using a shared LLC in the system. Chapter 5 presents our solution of ParRP that targets the spatial contention in the cache-coherent cache hierarchy with data sharing. Chapter 6 concludes this thesis and provides an outlook of future work.

Chapter 2

Background

Modern multicores feature multi-level cache hierarchies with cache coherence to bridge the performance gap between the high processor speed and slow main memory accesses. This chapter provides the background on these two common design features, cache hierarchies and cache coherence. It also provides background on cache partitioning, a common cache management technique to tackle the spatial contention problem in the shared LLC.

2.1 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 [25]. Consequently, modern commercial-off-the-shelf (COTS) multicores, such as Intel's Raptor Lake, AMD Zen, and ARM Cortex R82 [8], [9], [11], are moving towards using non-inclusive hierarchies [25].

2.2 Cache Coherence

Modern multicores use hardware cache coherence [13] to allow 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 2.1 shows the state machine for modified-shared-invalid (MSI) cache coherence protocol [13]. Each transition is labelled with a list of events (separated with ";") that trigger this transition, and the event format is "event / action". This protocol contains three fundamental states.

- 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.
- 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.
- Invalid (I): corresponds to a state where the cache line is invalid and has neither read nor write permissions.

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,

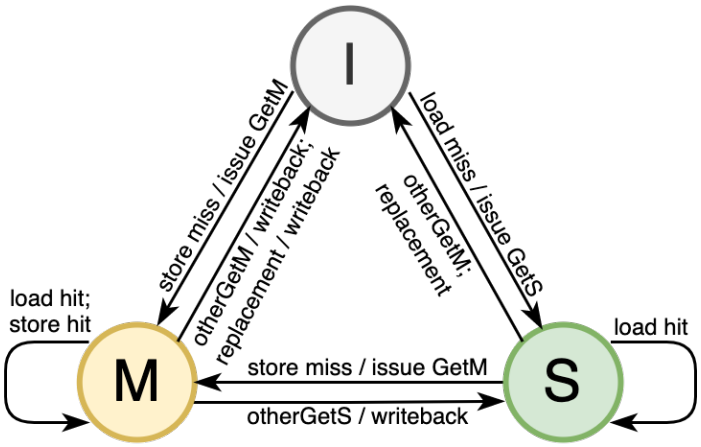


Figure 2.1: MSI state transitions.

then its respective controller will generate a PutM coherence request to writeback the dirty data.

These three states in MSI protocol serve as the base states for a multitude of other coherence protocols. For example, a popular protocol is MOESI [26] that introduces two additional states.

- Exclusive (E): identifies a cache line as a read-only and exclusive copy, which means that it is not shared by any other core. This optimizes the common write-after-read pattern by allowing a silent upgrade to the M state without generating a coherence request when there is no sharer.
- Owned (O): represents the owner of a read-only cache line that may be dirty, but not exclusive. This state allows the L1 cache controller to directly send data to requestors without the need to writeback the data first [13]. It exploits the common architectural feature of direct cache-to-cache transfer for higher performance [27].

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 [13]. 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 requester of data always receives the most up-to-date data. To maintain the two invariants, coherence protocols need to invalidate all private



(a) Set coloring.

(b) Way partitioning.

Figure 2.2: Overview of cache partitioning

copies cached by other cores when a write request occurs. For example, in MSI, cache lines in S state must transition to I state upon observing other cores' write requests. Such an invalidation triggered by other cores' write requests to maintain coherency is referred to as coherence invalidation. This ensures that a core that previously cached the data would have to request the up-to-date copy.

2.3 Cache Partitioning

Multiple cores competing for cache entries in the shared LLC is one key source of interference in the cache subsystem [28]. For example, a core may allocate a cache line for its data, which gets evicted by another core's access. A common solution to eliminate such contention is cache partitioning. Cache partitioning assigns a portion of the LLC to each core, thereby isolating accesses a core makes to its assigned partition. There are two techniques for cache partitioning: set coloring (partition cache sets) and way partitioning (partition cache ways) [16]. Figure 2.2 shows the overview of two cache partitioning approaches with partitions in different colors. Set coloring maps memory pages to different cache sets with support from the operating system and compiler. Specifically, set coloring controls the overlapped bits (i.e. color bits) of the physical page number, and the set index in the address during address translation. This means that the number of colors available depends on the cache organization. Further, modern multicores employ hidden hash functions for set mapping, which makes set coloring difficult to use in practice [29]. On the other hand, way partitioning is typically managed in the hardware that groups cache ways into different partitions. Under way partitioning, multiple cores can access the same cache set, but a core's request to a new cache line can only occupy the cache ways in its assigned partition. Modern commercial-off-the-shelf (COTS) multicore platforms use

way partitioning, such as ARM MPAM [30] and Intel RDT [31] technology, to isolate the contention. Its main constraint is the limited partition number due to the cache associativity. However, recent trends in embedded multicore architecture indicate a tendency of employing larger associativity in the LLC [32], which could mitigate the problem.

Chapter 3

Related Works

The challenge of poor timing predictability in multicore cache subsystem has driven many research efforts [7], [15], [16], [18]–[24], [33]–[35]. This chapter reviews a multitude of prior works on tackling the multicore interference in cache subsystems. At a high level, related works can be classified based on their focus on temporal [19]–[24] or spatial contention [7], [16], [33]–[35] to address the problem, and we follow this classification to introduce them. Also, we show the positioning of the two works in this thesis amongst these related works.

3.1 Related Works Targeting Temporal Contention

Temporal contention occurs when multiple cores simultaneously access the shared resource, such as shared caches, and buses in the cache subsystem. In applications with intensive data sharing, the timing overhead from cache coherence becomes non-negligible as shown by case studies in existing multicore platforms [14], [15]. This is because cache coherence involves complex communication between cores to synchronize data updates. The communication happens through shared resource, creating heavy competition on access when frequent data exchange occurs. For instance, when a core writes to a data value cached by other cores, it must send coherence messages to invalidate other data copies. Similarly, if a core wants to read a data that it does not cache, it must send coherence messages to locate the most up-to-date copy and wait for it to be returned. Besides the empirical case studies [14], [15], authors in [19] used a detailed theoretical model of a cache coherence protocol to study its timing effects. They found that the coherence protocol can lead to unbounded data access delays under unregulated contention caused by coherence messages.

To mitigate the timing effects of cache coherence, a class of works [7], [15], [18] focused on coherence avoidance by placing constraints on either data caching or task scheduling. Constraints on data caching include bypassing private cache levels for shared data [18] and even simply marking shared data as uncachable [7]. Scheduling constraints aim to achieve temporal isolation between tasks that share data. For example, [15] required custom scheduler that prevent tasks that share data from executing at the same time. Another similar technique adopted in [7] require co-allocating tasks with shared data on the same core. The obvious problem with this class of approaches is that they underutilize the hardware and suffers performance slowdown.

Another class of approaches aimed at addressing the sources of unpredictability in conventional cache coherence directly, and proposed to re-design a predictable version of hardware cache coherence [19]–[24]. Predictable hardware coherence aims to provide a guaranteed PR-WCL bound to access the data, which is transparent to the software stack, while preserving the most performance benefits of conventional cache coherence. PMSI [19] is the first predictable cache coherence protocol and demonstrated the performance benefits compared to the above approaches based on coherence avoidance. Since then, many different variants [20]–[24] of predictable cache coherence have been proposed to further lower the PR-WCL bound while improving the average performance. Recent research focused on designing a predictable cache coherence that works with a shared LLC to extract further performance [23], [24]. Authors in [23] studied the timing overhead due to the introduction of a shared LLC and demonstrated that it can significantly lengthen the PR-WCL under an inclusive cache hierarchy assumption. Later, they proposed ZCLLC [24], a novel inclusive LLC architecture that achieves a low PR-WCL bound that is linear in terms of the core counts. The first work of this thesis, PECC, falls into this category of work aiming to provide a low PR-WCL bound while enabling the performance benefits from having a shared LLC.

3.2 Related Works Targeting Spatial Contention

Spatial contention refers to the multicore competition on cache entries for data allocation. As a result, one core may evict other cache lines used by another core for its own data requests, incurring cache misses and delay data accesses for that other core. A common source of spatial contention is the shared LLC, where multiple cores contend for cache entries, frequently evicting cache lines allocated by each other. There have been several efforts to account for shared LLC interference in static cache analyses [33], [34], [36]. The prevailing methods simply assume that all possible shared cache contentions result in

memory references being *not classified*, i.e., they are treated as main memory accesses in the worst case [33], [36]. A more precise analysis has been recently proposed in [34]. This method limits shared LLC contention caused by one task on another by jointly analyzing all possible paths (without loops) in both programs. However, the approach requires knowledge of which tasks are co-running, and it is unclear whether it would scale to more than two cores. Hence, the summary is that direct analyses on the shared LLC with spatial contention is hard to realize in practice.

A more practical approach is to provide resource isolation without concerning other cores' actions. Cache partitioning is a well-known practice to provide such isolation in the shared LLC [16]. However, conventional cache partitioning schemes do not account for data sharing, which intrinsically violates the resource isolation principle [7], [18]. This is because data needs to be shared across multiple partitions and still suffer interference from multiple cores. As data sharing is of vital importance to enable multi-threaded workloads, a few research efforts have been proposed to extend partitioning methods to support it [7], [35]. A technique proposed in [7] assigns a shared buffer space for shared data communication and performs cache locking to lock the entire buffer space in the shared LLC. Although this technique ensures strong predictability, it effectively reduces the caching capacity and damages average performance. In light of the considerations of both performance and predictability, [35] proposed a novel shared LLC management scheme that classifies memory types into deterministic and best-effort to achieve the best out of both. The framework enables certain partitions to be shared while the rest are strictly dedicated to different cores. The deterministic memory is assigned to a core's private partition to provide guaranteed low access bound for critical data, while best-effort memory is assigned to a shared partition accessible by all cores for performance. To support data sharing, shared data can be assigned to the best-effort region while the private data for each core are kept in deterministic region. This effectively guards the private data against the interference caused by shared data. Meanwhile, the average performance benefit from caching shared data is obtained. Nonetheless, this approach does not address the interference on shared data itself. The second work of this thesis, ParRP, belongs to this category of works targeting resolving spatial contention occurred with data sharing. ParRP offers a different solution than all prior works targeting in that it provides strict resource isolation for both shared data and private data.

Chapter 4

Exclusive Hierarchies for Predictable Sharing in Last-level Cache

This chapter presents PECC, our solution targeting the temporal contention when sharing a LLC in real-time multicores. The shared LLC is essential for delivering high performance in many COTS multicore platforms [8]–[12]. However, the temporal contention that it introduces remains one of the major challenges impeding its adoption in real-time systems. In particular, an inclusive LLC design assumed by many prior works experiences significant resource contention, resulting in an extremely high PR-WCL. To tackle this challenge, our approach relies on the key observation that an exclusive cache hierarchy, by definition, eliminates back invalidation, which is one of the largest contributors to the high PR-WCL when using inclusive cache 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, we propose PECC, a novel predictable cache coherence mechanism with an exclusive LLC setup. PECC delivers a low WCL bound and higher performance than all of its inclusive counterparts. Our specific contributions in this chapter are as follows.

1. We present Predictable Exclusive Cache Coherence (PECC) as our solution to incorporate a shared LLC for real-time multicores, which offers both high performance and a low WCL bound.

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 perfor-

mance (2.33×) when compared to the state-of-the-art predictable inclusive counterpart ZCLLC.

4.1 Motivation

Prior works show that predictable versions of cache coherence [19]–[22], [37] 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 [13], which is a common setup in current real-time multicores [11]. Our detailed study of prior efforts [19]–[24], [37] revealed a common assumption amongst all these works: the cache hierarchy is inclusive[1]. 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 [23]. 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 [23], [24]. We illustrate the crux behind the reasons using the illustration in Figure 4.1. 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 $\mathsf{GetS(A)}$ request on the bus. The $\mathsf{GetS(A)}$ 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 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 4.1). Then, the LLC issues a write of B to the main memory, and waits for it to complete (window 2 marked in Figure 4.1). 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

[1]Some of the works assume that the system has a perfect LLC where accesses are all hits, which is essentially a shared memory instead of a realistic LLC where misses can happen.

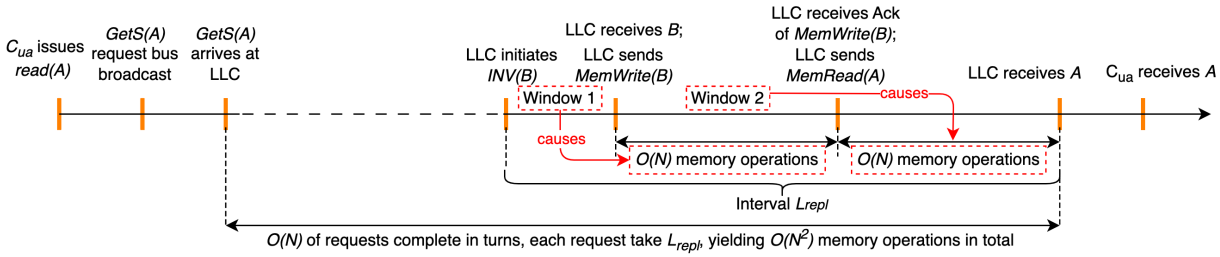


Figure 4.1: Request processing timeline of $read(A)$ request of the core under analysis C_{ua} in a multicore system with N cores.

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 [36], [38], 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 [36], [38]. 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

[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 [36], [37].

analysis circumvent this interference by requiring the shared LLC to be partitioned to different cores [36]. However, existing cache partitioning schemes [36] face challenges to isolate interference from shared data under cache coherence (see Chapter 5 for details). We wish to enable cache coherence for high performance implementations.

Observation 3: Limitations of the state-of-the-art. To address the challenges in the shared inclusive LLC, [24] 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 [19]–[24], [37] use an ICH. However, 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 4.2 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

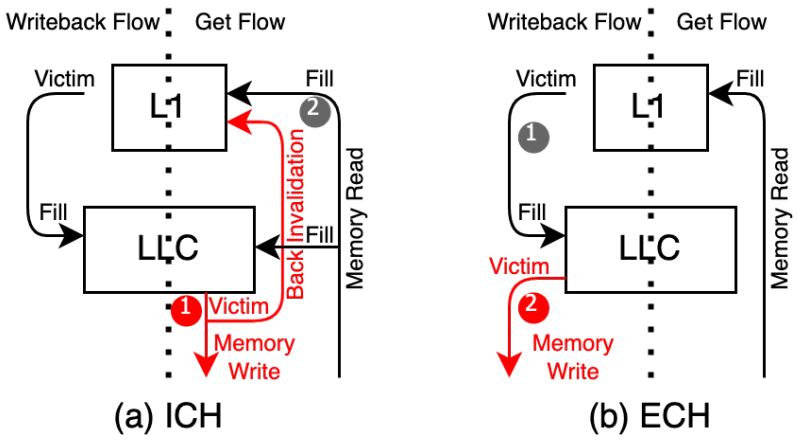


Figure 4.2: Worst-case traffic flow in ICH and ECH with operations related to main memory writes highlighted.

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 this chapter 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).

4.2 System Model

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

4.2.1 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 4.3. Each core implements an in-order pipeline that issues at most one outstanding memory

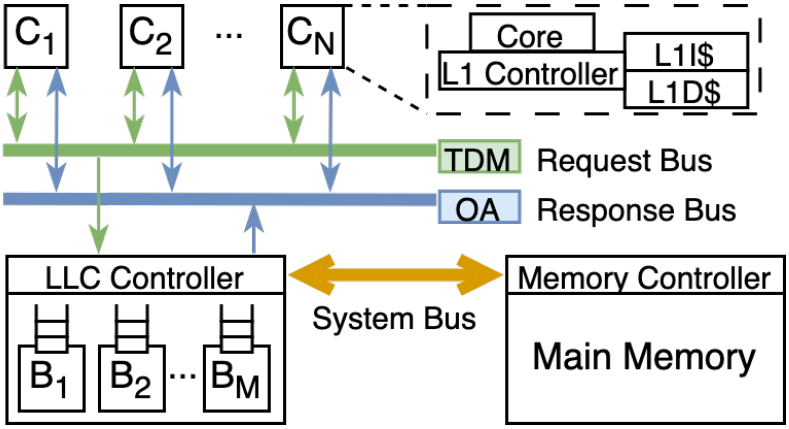


Figure 4.3: Overview of system model.

request at a time. This requirement of one outstanding memory request is consistent with prior related works [19]–[21], [23], [24]. 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 [39]. 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 $\text{bank}_{i\%M+1}$). 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 4.3.1.

4.2.2 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 [13]. The split-transaction bus design is commonly deployed in COTS platforms such as Arm Corelink [40] and Intel's QPI [41]. 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 [13], [26], 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 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 4.3.2.

4.2.3 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 4.5), 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.

4.3 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.

4.3.1 PECC Protocol

Although implementations for ECHs exist [27], 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

Table 4.1: 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.

propose the protocol for PECC. We provide three definitions in Table 4.1 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 [25] 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 4.1. *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 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 [27]. 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 4.2. *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 4.1. 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 4.2. 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.

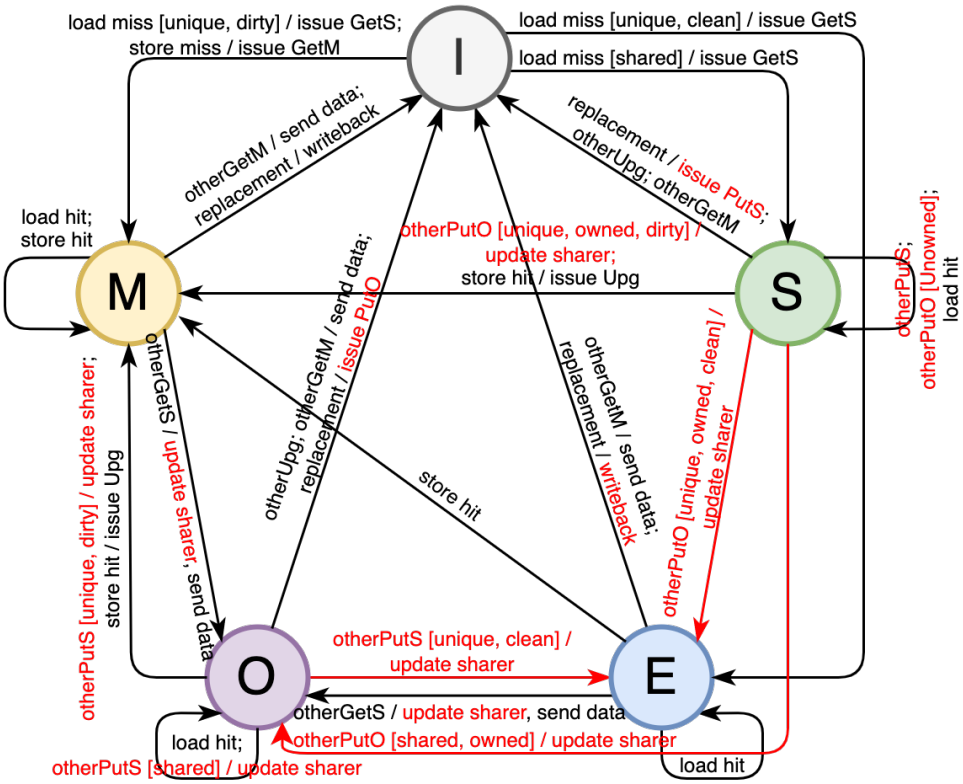


Figure 4.4: Stable state transitions in PECC protocol with differences compared to MOESI protocol highlighted in red.

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 [13]. The definitions of stable states and coherence requests in PECC are summarized in Table 4.2 and Table 4.3, respectively. Figure 4.4 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

Table 4.2: 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

Table 4.3: 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.

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 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 [13]. 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 4.4 shows the complete protocol table with the transient

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^dI	AR, US	AR / IS^dI	US / -	CR, US / IO^d, IS^d	X
IO^d	X	X	X	LH, SD, CR / E, M, O	X	- / IO^dI	AR, US	AR / IO^dI	US / -	X	X
IM^d	X	X	X	SH / M	X	X	AR, US / IM^dO	AR / IM^dI	X	X	X
IS^dI	X	X	X	LH, SD, CR / I	X	X	-	-	X	-	-
IO^dI	X	X	X	LH, SD, CR / I	X	X	-	-	X	-	-
IM^dI	X	X	X	SH, SD, CR / I	X	X	-	-	X	X	X
IM^dO	X	X	X	SH, SD, CR / O	X	X	AR, US	AR / IM^dI	X	X	X
I^A	X	X	X	X	- / I	-	-	-	-	-	-

Format: "action / next state". Notes: 1. Next state is ignored if remains the same. 2. "-" indicates no action or transition is required. 3."X" indicates impossible transitions.
Action acronyms: 1. LH: Load hit. 2. SH: Store hit. 3. SD: Send data if owned and requestors exist. 4. US: Update sharer info. 5. AR: Add requestor. 6. CR: Clear requestor info.

Table 4.4: PECC protocol table. If multiple next states are listed, the correct one is determined by the cache line properties.

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^d once C_1 is granted its slot to broadcast the request. The state name IM^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^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^dO. 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 4.5), 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.

4.3.2 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 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 [19], [20], [23], [24]. 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 4.4.

OA arbiter at response bus. One approach to ensure low WCL bounds for a split-transaction bus is to use a pipelined bus (PIPE) [21] 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 4.5 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

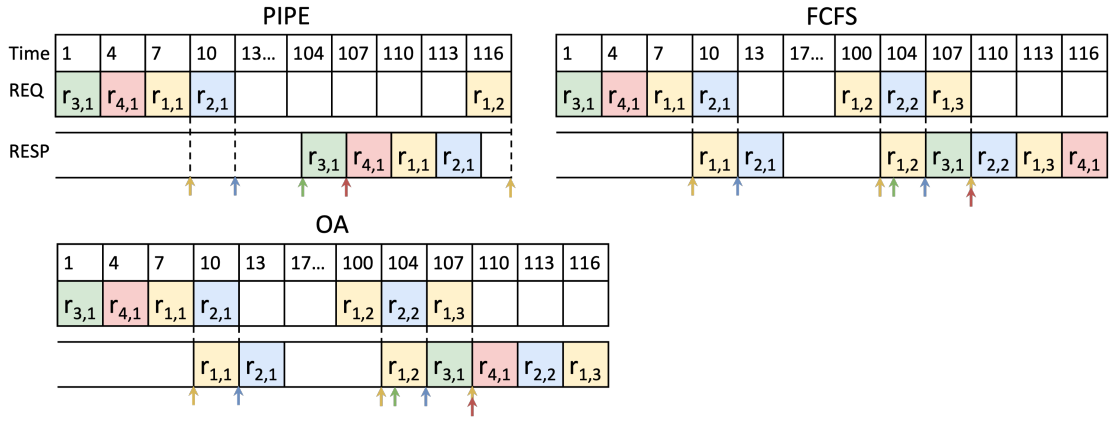


Figure 4.5: Comparison of response bus arbiters. Events of processing requests from the same core use the same color. ↑ marks the ready time of the response. The FCFS timeline is drawn assuming the worst-case arbitration decision for $r_{4,1}$'s response.

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 $r_{1,2}$.

4.4 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.

4.4.1 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 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 4.3.1: 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

[3]In this thesis, "WCL is ..." refers to an upper bound of the actual WCL.

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 4.1 (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 4.2 (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 4.6 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 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.

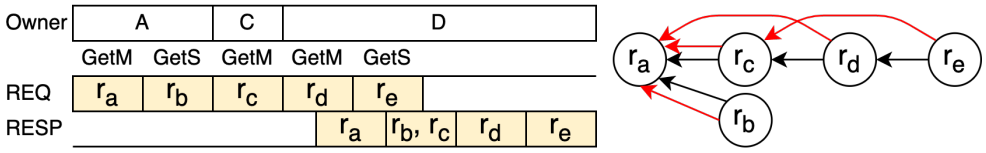


Figure 4.6: Example of request dependencies.

4.4.2 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 4.1. *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 4.2. *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 4.3. *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 4.4. *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 4.1-4.3 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 4.7: $\mathcal{L}_1$ 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

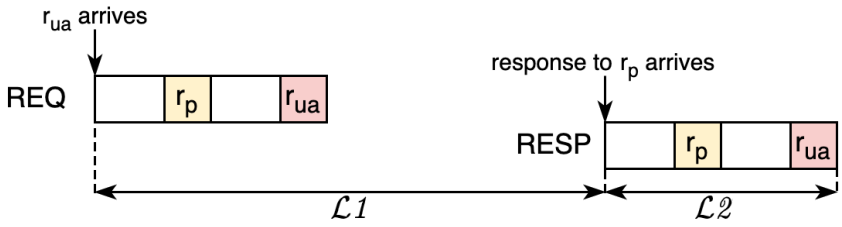


Figure 4.7: Latency components of r_{ua} in case 2 of Lemma 4.4.

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 4.1. **(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 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 4.5. *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 4.4. In the worst case, r_{ua} belongs to type $\mathcal{T}_4$; summing the WCL in each resource based on Lemmas 4.1-4.3 and the definition of t_{MEM} yields the result. □

Theorem 4.1. *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 4.5) and the WCL of a GetS/GetM request (Lemma 4.4). □

4.5 Evaluation

Experimental setup. We implement PECC in the Gem5 simulator [42]; 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 [10], 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 4.2.3). 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 [24]. 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 [43]. 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 4.1. We also use the Splash-3 benchmark suite as a representative

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

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 [44]. 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.

4.5.1 WCL for Load/Store Instructions

We first evaluate the WCL for load and store instructions over various setups. Figure 4.8 and Figure 4.9 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.

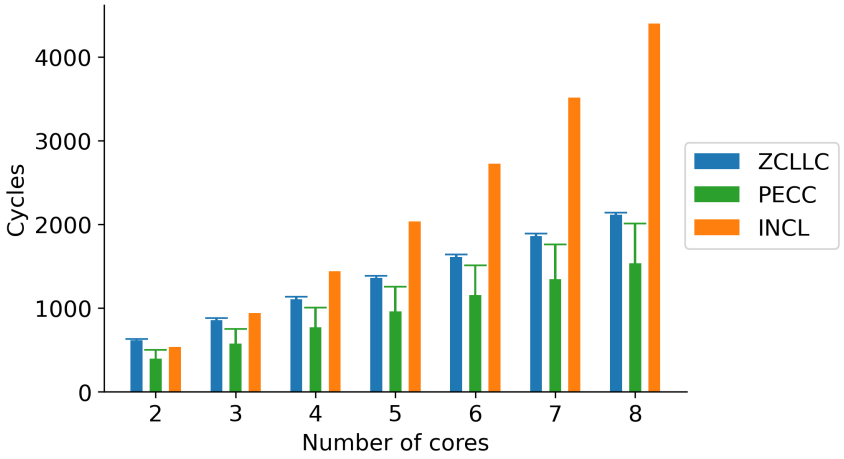


Figure 4.8: Observed WCLs of synthetic benchmark.

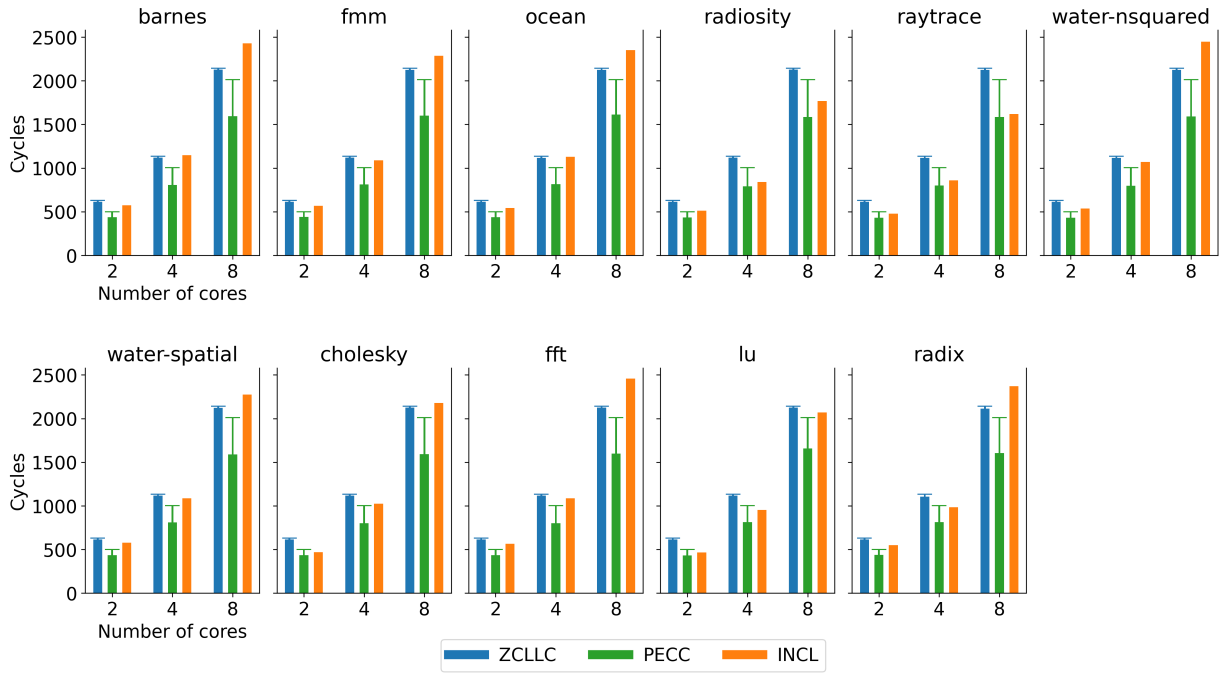


Figure 4.9: Observed WCLs of Splash3

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 4.1, 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.

4.5.2 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

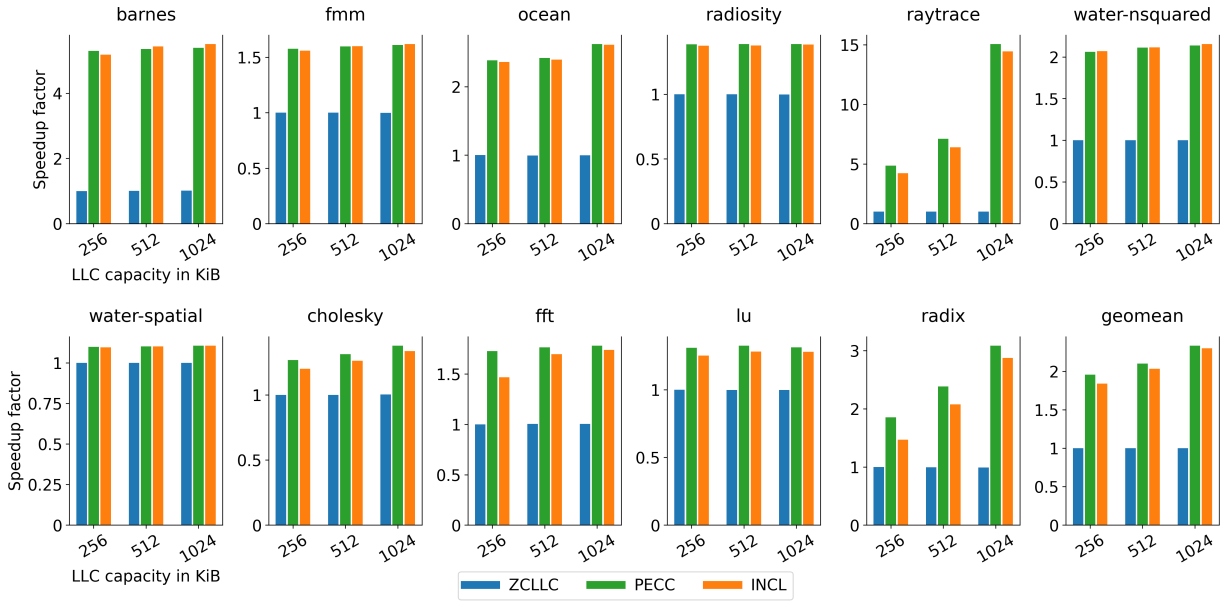


Figure 4.10: Average execution time speedup of Splash-3. Baseline is 256-KiB ZCLLC.

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 4.10 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.

4.6 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 [24].

Chapter 5

ParRP: Enabling Space Isolation in Caches with Shared Data

In this chapter, we presents our solution, ParRP, targeting the spatial contention in caches with shared data. Spatial contention makes the cache content highly unpredictable and in turn results in great timing variability of the task. Cache partitioning is a well-known practice for achieving resource isolation to eliminate spatial contention. However, existing partitioning techniques [16] face challenges to support data sharing. This is because the partition that allocates shared data needs to be shared across cores and still suffers interference. Motivated by this challenge, we propose ParRP, a novel hardware cache partitioning scheme with shared data support. Different from all prior efforts, we aim to provide cache space isolation for shared data, similar to how conventional partitioning provides isolation for private data. The key idea behind ParRP is that shared data isolation can be achieved by partitioning the replacement information rather than the data itself. This brings us to the following contributions in this chapter.

1. We identify the sources of interference from shared data accesses that break the isolation property.
2. We propose a novel cache partitioning scheme, ParRP, which guards against interference from shared data accesses and achieves the isolation property. The key idea is to partition the replacement policies so that each core maintains its own copy of replacement information attached to the same shared cache line. Furthermore, we devise additional rules imposed on the cache controller behaviours to account for interference from cache coherence activities.

3. We analytically prove that ParRP upholds the isolation property.

4. We evaluate ParRP using the gem5 simulator [42]. The evaluation shows that, on average, ParRP tightens the WCET bounds for multi-threaded tasks by 2.4×, compared to the best result achieved with conventional cache partitioning. The isolation property comes at the expense of average performance: under an extreme configuration with the smallest possible partition size, ParRP results in 16.5% performance slow down compared to freely sharing the LLC.

5.1 System Model

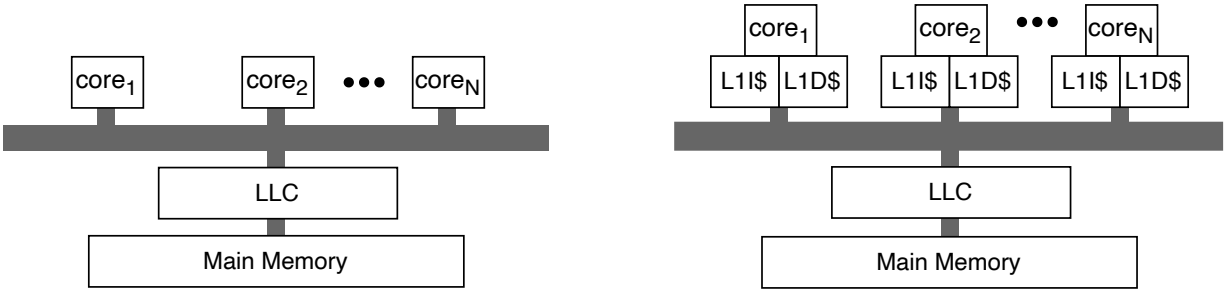
In this section, we present our system model. We discuss our assumptions on the multicore architecture, and the cache semantics used in our architecture.

5.1.1 Multicore Architecture

Our system model consists of a multicore with N cores and a two-level cache hierarchy as shown in Figure 5.1b. Each core issues at most one outstanding memory request. A core has separate instruction and data level-one (L1) caches that are private. On level two, there is a shared LLC that all cores can access. The LLC is connected to the main memory. There is a bus connecting all the cores and the LLC for transmitting requests and data amongst them. There is a separate bus that connects the LLC and the main memory. The two-level of caches maintain an inclusive hierarchy where the cache lines allocated in L1 must also be present in the LLC. Due to coherence complexity, we wish to start our discussion with a simplified system model as shown in Figure 5.1a without worrying cache coherence. This allows us to solely focus on the interference experienced by the shared LLC first. Then, in (Section 5.3.2), we extend our solution to the full system model and start addressing coherence impacts.

5.1.2 Cache Semantics

We assume set-associative writeback caches. To understand the cache behaviour for timing predictability, similar to prior cache analysis works [36], we define the cache state to capture the observable behaviour of allocation status without modelling the physical implementation details. In particular, the cache state is the aggregated state of the sets in



(a) Simplified system model (b) Full system model

Figure 5.1: Overview of system model.

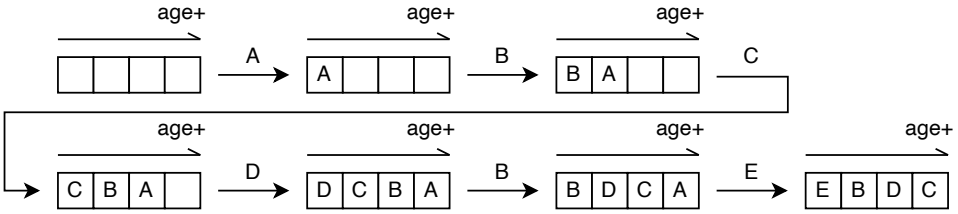


Figure 5.2: Cache state update with LRU replacement policy

the cache, and the state of a set is further defined to contain the following two aspects. (1) Allocation status: A cache way is either empty or occupied as denoted uniquely by the cache line address. (2) Replacement information: The replacement policy uses this information to determine the victim to replace when a set is full, thus shaping how the allocation status evolves as accesses are made. For the least-recently-used (LRU) policy, the replacement information is the relative age order of allocated cache lines. We assume that the cache uses LRU replacement policy. It is worth mentioning that our proposed mechanism can work with other replacement policies; the assumption of LRU is made only for the purpose of discussion in this chapter. For a w-way set-associative LRU-cache, the state of a cache set is represented by a w-entry ordered list with the address of the youngest (i.e. most-recently used) cache line starting from the left. Empty cache lines are denoted by '$-$' following the allocated cache lines. For example, for a 4-way set with the allocated cache line addresses A, B, and C from youngest to oldest, its state is represented by $[A, B, C, -]$. Figure 5.2 shows the state update of a LRU-cache set under a sequence of memory accesses.

5.2 Interference in Shared LLC

In this section, we first formally define the isolation property that provides space isolation in caches amongst multiple cores. Then, we show how conventional partitioning schemes fail to achieve isolation property with data sharing. Finally, we provide our insights to the problem and outline our solution.

5.2.1 Goal of Isolation

Inter-core interference on cache space makes the cache content highly unpredictable. A practical solution is to strictly partition the cache subsystem among multiple cores to achieve isolation property, which we formally defined as follows.

Definition 5.1 (Isolation Property)**.** *Assuming the same initial cache state and the same sequence of memory accesses performed by a core, a cache subsystem satisfies the isolation property under partitioning if requests that are cache hits when the core runs in isolation are still cache hits when it suffers contention from other cores.*

The isolation property essentially allows a core in the multicore system to be treated as a single core running alone when determining cache hits. For example, to determine cache hits for a core's memory requests, people can either use a measurement-based approach or statically analyzing the cache content without concerning other cores' activities. At the end, the cache content becomes more predictable, and we can determine more accesses to be cache hits, thereby tightening the WCET bound for a task.

5.2.2 Violation of Isolation under Shared Data Interference

We observe that existing partitioning schemes rely on a common assumption for this property to hold true: there is no data sharing amongst multiple cores. In the following, we use way partitioning as an example to show that shared data accesses can compromise the property.

Example. Figure 5.3 compares the state changes when two cores run in contention against their isolation references under way partitioning. Each core is assigned with two ways in 4-way shared sets, and the superscript indicates the partition that a cache line is placed in. Under such partitioning, the replacement is triggered when the target partition is full, where it selects the least-recently-used cache line *in the partition* as the victim. The state

Access Sequence		Isolation Reference		Contention
		4-way Shared Set with Two 2-way partitions		4-way Shared Set with Two 2-way partitions
1	$core_1$: A	**Miss [A^1, -, -, -]**	[-, -, -, -]	**Miss [A^1, -, -, -]**
2	$core_1$: B	**Miss [B^1, A^1, -, -]**	[-, -, -, -]	**Miss [B^1, A^1, -, -]**
3	$core_2$: A	[B^1, A^1, -, -]	**Miss [A^2, -, -, -]**	**Hit [A^1, B^1, -, -]**
4	$core_1$: C	**Miss [C^1, B^1, -, -]**	[A^2, -, -, -]	**Miss [C^1, A^1, -, -]**
5	$core_1$: B	**Hit [B^1, C^1, -, -]**	[A^2, -, -, -]	**Miss [B^1, C^1, -, -]**
6	$core_2$: A	[B^1, C^1, -, -]	Hit [A^2, -, -, -]	**Miss [A^2, B^1, C^1, -]**

Both accesses are hits under isolation but become misses under contention.

Figure 5.3: Violations of isolation property in conventional way-based partitioning.

in bold indicates the change due to the current request. In this example, requests 5 and 6, which originally hit under isolation, become cache misses under contention. This violates the isolation property, and the cause is the accesses to shared line A. From the perspective of $core_1$, the replacement information is altered by $core_2$'s access to line A in its partition (request 3). As a result, line B becomes the least-recently-used entry and is replaced by new line C accessed by $core_1$ later. This results in a miss for $core_1$'s second access to line B (request 5), which would instead be a hit under isolation. Note that the problem cannot be simply resolved by preventing a core from updating the replacement information if the accessed line is not in its partition, which is illustrated from the perspective of $core_2$. When $core_2$ accesses line A (request 3), A is already placed in $core_1$'s partition, and $core_2$ cannot reallocate the line to its own partition. Therefore, $core_1$ has the right to evict line A to make space for its next request (request 4). At the end, $core_2$'s second access (request 6) to A becomes a miss. In contrast, in $core_2$'s isolation reference, $core_2$'s is able to place line A in its partition until its second access (request 6) hits it.

It is important to note that the isolation property is also violated for set-based partitioning schemes. This is because the set partition that allocates shared data needs to be shared among multiple cores and suffers interference.

5.2.3 Insights

We observe that the current strategy of partitioning only targets the cache entries for cache line allocation [16], [45], [46]. This design mostly aims to mitigate shared LLC contention for average performance benefits [16], [47], [48] rather than achieving strict isolation for hard real-time systems. Therefore, it fails to satisfy isolation property under

certain scenarios (e.g., data sharing is enabled). In particular, for shared data that is allocated to one entry but is accessed by multiple cores, we cannot strictly partition shared data under this strategy.

To resolve the issue, our main idea is to change the partition target to replacement policy. Instead of completely partitioning the cache ways, we decouple the replacement information from the cache line storage and partition only the replacement part. Each partition now only holds the replacement information associated with the cache lines owned by a core (i.e. in a core's partition), and we call it as replacement partition. The cache line storage is referred as physical ways, which can be shared by multiple partitions while each maintains its own replacement information for the shared line. Essentially, the replacement partition mimics the normal set-associative cache with the same number of ways as the partition size. This means that the state of replacement partition (partition state) is exactly the same as the cache state as defined in Section 5.1.2. When the physical ways are full, a core cannot replace cache lines allocated in other partitions to achieve isolation. We use the following example to illustrate the core idea of replacement policy partitioning.

Example. Figure 5.4 shows the state change for a sequence of memory accesses under replacement policy partitioning. Each core maintains a 2-way replacement partition to keep the replacement information of cache lines it owns. The occupancy of the physical ways in a set is represented by an unordered list surrounded by curly brackets, and the superscript of a cache line lists the owner cores. The example starts with the same access sequence (request 1 to 6) in Figure 5.3 to show how the isolation property is restored. At request 1, core_1 makes an access to a new line A and allocates it into a free physical entry. Also, it adds line A into its 2-way replacement partition following LRU rule and claims itself as an owner of A. Next, at request 2, core_1 allocates line B following a similar procedure. When core_2 accesses cache line A (request 3), since A is already present in the cache, it only updates its own partition state and adds itself as another owner of line A. At this point, A have both the cores as its owners but have distinct replacement information recorded in each core's partition state. At request 4, core_1 accesses new line C, but its partition is full. According to LRU rule, line A is selected as the victim to be removed from the partition, and A is no longer owned by core_1. Meanwhile, core_2 still owns line A, and A safely remains in the cache set. Therefore, Request 5 and 6 preserve the hit statuses, restoring the isolation property.

We further extend the request sequence (request 7 to 9) to provide more details on the replacement handling. At request 7, core_1 requires replacing line C in its partition for the new line D. Same as request 4, replacement in partition only requires relinquishing the ownership of the victim line without deallocating it from the cache set. This allows line

Access Sequence		4-way Shared Set under ParRP		
		$core_1$'s 2-way Replacement Partition	$core_2$'s 2-way Replacement Partition	Shared Physical Way Entries
1	$core_1$: A [Miss]	**[A, -]**	[-, -]	**{A^1, -, -, -}**
2	$core_1$: B [Miss]	**[B, A]**	[-, -]	**{A^1, B^1, - , -}**
3	$core_2$: A [Hit]	[B, A]	**[A, -]**	**{$A^{1,2}$, B^1, -, -}**
4	$core_1$: C [Miss]	**[C, B]**	[A, -]	**{A^2, B^1, C^1, -}**
5	$core_1$: B [Hit]	**[B, C]**	[A, -]	**{A^2, B^1, C^1, -}**
6	$core_2$: A [Hit]	[B, C]	[A, -]	**{A^2, B^1, C^1, -}**
7	$core_1$: D [Miss]	**[D, B]**	[A, -]	**{A^2, B^1, C, D^1}**
8	$core_2$: C [Hit]	[D, B]	**[C, A]**	**{A^2, B^1, C^2, D^1}**
9	$core_1$: E [Miss]	**[E, D]**	[C, A]	**{A^2, C^2, D^1, E^1}**

Figure 5.4: Restore isolation property by replacement policy partitioning.

C to remain in the cache set for the next request (request 8). This is in clear contrast compared to conventional partitioning, where deallocations happen whenever individual partitions reach their capacity as shown in Figure 5.3. In our approach, since physical ways are shared, deallocations are only necessary if whole set is full, as shown at request 9. At request 9, $core_1$ makes an access to the new line E, and B becomes unowned following the $core_1$'s partition state update. At this time, all physical cache ways are occupied, and a victim line must be deallocated to make space for E. Since B is only the unowned entry according to the resultant partition states, it is selected to be replaced by E in the physical way.

With replacement policy partitioning, each core maintains its distinct partition view of lines that it owns, even for shared lines. The concept of partitioning is now well defined as clearly represented by the partition states. Further, the partition states evolve in the same way as under isolation for different cores, satisfying the isolation property.

5.3 ParRP

In this section, we present our solution, ParRP, that achieves isolation property for cache-coherent multicores. We introduce our two-fold solutions in the order as follows. (1) We start by focusing on addressing shared LLC interference without considering coherence impacts. Therefore, at this step, we assume a simplified system model as shown in Figure 5.1a, where the only cache in the system is a shared LLC. To address shared LLC

interference, our solution lies in employing a novel cache partitioning scheme that partitions the replacement policy information instead of physical ways, as previously outlined in Section 5.2.3. (2) Next, we consider the full system model with private L1 caches and a shared LLC (Figure 5.1b), where cache coherence emerges. For the complete two-level hierarchy under partitioning, we inspect sources of isolation property violations due to the added coherence complexities. Based on these sources, we devise additional rules on cache controller behaviours to maintain the isolation property under the full system model.

5.3.1 Replacement Policy Partitioning for Shared LLC

The core idea of replacement partitioning is to maintain a distinct partition view for different cores, regardless whether the owned lines are shared or not. This is achieved by only partitioning the replacement information and associate it with shared physical ways, as outlined in in Section 5.2.3. Algorithm 1 formalizes the partitioning mechanism, and Table 5.1 explains the function semantics used by the algorithms in this chapter. The function calling convention follows the syntax We use the syntax "⟨*context*⟩.⟨function name⟩(*paramters*)"". For example, "*set*.tagCheck(*addr*)" means performing the tag check of the address *addr* in this particular cache set denoted as *set*. The context and parameters can be ignored if they are not required. At the start, the algorithm first gathers the initial status about the request. Line 2 and line 3 get the target set and target partition for the request, respectively. Then, it obtains the allocation statuses of the target cache line in both the set and replacement partition (line 4 to line 9). If the cache line is present in the set, *set_entry* is the physical way storing the cache line otherwise null. The same assignment applies to *partition_entry*. Depending on the allocation statuses and space availability, the algorithm performs the partition eviction (line 10 to line 15), shared set (i.e. physical ways) eviction (line 18 to line 21), shared set allocation (line 22 to line 24), and partition allocation (line 25 to line 28) as necessary. When selecting victim to be replaced in physical ways, only unowned cache lines are selected on line 19. We ensure that there is always at least one unowned cache line valid for replacement because the sum of the partition sizes is equal to the number of physical ways in the shared set. In details, if all cache lines are owned, it means that all cache lines are exclusively owned by a core, and the requestor's partition is full. Consequently, when there is a request to allocate new cache line, one of the cache line in the requestor's replacement partition must be removed from the partition (line 10 to line 15) and becomes unowned for physical replacement. After the requested cache line is allocated in both the set and partition, it performs a replacement information update in the requestor's replacement partition given that the requested line is most-recently accessed (line 29). Next, we present the actual hardware organization and

the underlying request processing steps to realize the partitioning mechanism.

Hardware organization. Figure 5.5 shows the cache set organization of ParRP. The shared physical ways are normal cache entries consisting of tag and data arrays, similar to normal set-associative cache organizations. The primary difference is that, instead of storing the replacement information alongside each way, the replacement information is now decoupled and stored separately in replacement information entries. The total number of replacement information entries are equal to the number of physical ways, and they are statically partitioned amongst all cores. Therefore, we also refer the replacement information entry simply as the partition entry in the context of ParRP. Figure 5.5 shows an an example partition configuration with an even split of ways. Each replacement partition mirrors the structure of an associative set, with the number of ways equal to the partition size. Further, each partition entry mimics the normal cache way, but the data now stores

Table 5.1: Function semantics.

Function	Semantics
LLCSetMap	Given a cache line address, return the mapped LLC set.
L1SetMap	Given a cache line address and a core index, return the mapped set in the core's private L1 cache.
getRPPar	For a LLC set, given a core index, return the core's replacement partition in that set.
tagCheck	For a L1 or LLC set, given a cache line address, return the cache entry storing the cache line if it exists. Otherwise, return null.
wayCheck	For a replacement partition, given a way index, return the partition entry storing the replacement information associated with the cache line stored in that way.
getAddr	For a set, given a way index, return the allocated cache line address.
free	For a set or a replacement partition, mark the given entry as free.
touch	For a set or a replacement partition, given an entry in it that is most-recently accessed, update the replacement information according to replacement policy.
updateOwner	For a set, given the core and the way index, set the core's ownership of the way as true or false.
getVictim	For a L1 set, get the victim according to replacement policy.
getParVictim	For a replacement partition, get the victim according to replacement policy.
getPhyVictim	For a LLC set, select an unowned cache line in shared physical ways as the victim.

Algorithm 1: LLCUpdate

Input: request *req*
Output: cache line address of the partition victim

```
1  addr ← req.addr, core_idx ← req.requestor;
2  set ← LLCSetMap(addr);
3  par ← set.getRPPar(core_idx);
4  set_entry ← set.tagCheck(addr);
5  if set_entry ≠ null then
6  |  par_entry ← par.wayCheck(set_entry.way_idx);
7  else
8  |  par_entry ← null;
9  end
10 if par is full and par_entry = null then
   |  // Make free entry in the partition
11 |  par_victim ← par.getParVictim();
12 |  set.updateOwner(core_idx, par_victim.way_idx, false);
13 |  par.free(par_victim);
14 |  par_victim_addr ← set.getAddr(par_victim.way_idx);
15 else
16 |  par_victim_addr ← null
17 end
18 if set is full and set_entry = null then
   |  // Make free entry in the set
19 |  set_victim ← set.getPhyVictim();
20 |  set.free(set_victim);
21 end
22 if set_entry = null then
23 |  set_entry ← set.allocate(addr);
24 end
25 if par_entry = null then
26 |  par_entry ← par.allocate(set_entry.way_idx);
27 |  set.updateOwner(core_idx, par_entry, true);
28 end
   // Update replacement information
29 par.touch(par_entry);
30 return par_victim_addr
```

the replacement information, and the tag is the way index of the physical way storing the

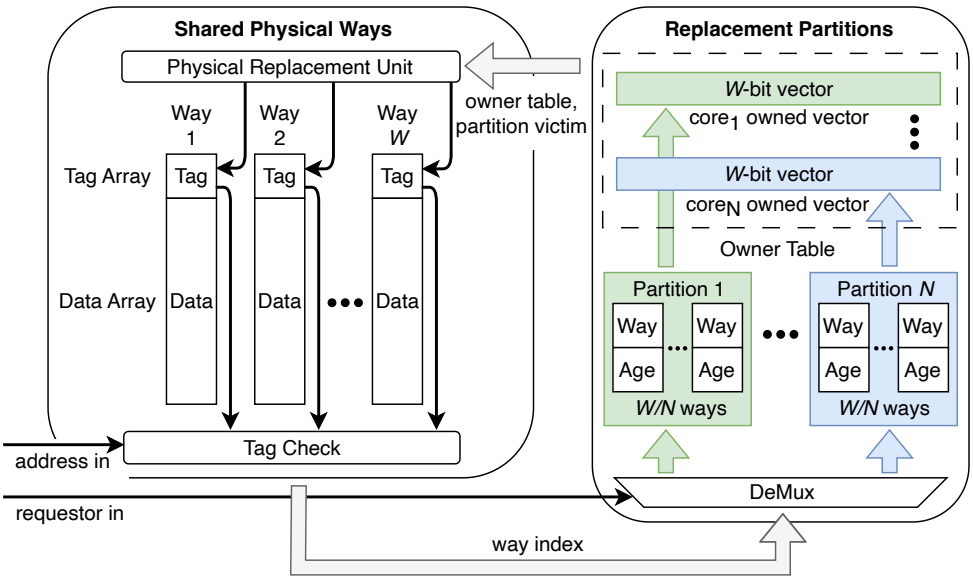


Figure 5.5: Overview of ParRP architecture.

corresponding cache line. The set also maintains an owner table that records the owner information of each physical way. The owner table consists of N W-bit owner vectors, with each vector recording the ownership status of W physical ways for a different core. It is used by the physical replacement unit to efficiently determine unowned physical ways for physical replacement.

Request processing. The request processing steps of the described cache set organization are detailed as follows. The set first performs a tag check to determine the presence of the requested cache line. If the requested cache line is not present, a physical cache way needs to be allocated for the new cache line. When all physical ways are occupied, a victim is selected by the physical replacement unit for the replacement. The physical replacement unit uses the owner table to determine unowned entries. If there are unowned cache entries, it randomly selects one of them for replacement. Otherwise, it uses the requesting core's partition victim based on the replacement policy. This corresponds to the procedure of first removing the victim from the partition and selects it as the only unowned entry for physical replacement in Algorithm 1. After determining the physical way for storing the requested cache line, the physical way index is passed to the core's replacement partition. The replacement partition follows a similar process to update the replacement information as a normal associative set in the cache. The primary difference is that the partition uses the physical way index as the tag to determine the presence of cache line in the partition.

After every partition state update, the owner table must be updated accordingly. For example, if a replacement happens in a partition, the victim cache way must be marked as unowned, and the requested cache way must be marked as owned in the owner vector of the requesting core. The updated owner table is again used in the physical replacement step of the next request processing.

Overhead. The ParRP architecture retains most of the structure present in the normal set-associative set. The additional hardware complexities are in the implementation of the partitioned replacement policy. We discusses the overheads in terms of hardware and timing. (1) Hardware overhead. The primary hardware overhead is the storage overhead for the owner table and the replacement partitions. The owner table costs $N \cdot W$ bits, which is often required in existing way partitioning techniques to compute the replacement mask [46]. The storage overhead of replacement partitions depend on the chosen replacement policy. For LRU, it requires basically the same amount of storage for the replacement information (i.e. age bits), plus additional $W \cdot \log(W)$ for way index storage. Regarding the logic overhead, it is the additional physical replacement unit because all partitions can share the same LRU replacement logic. The physical replacement unit is basically a random replacement policy with masking, which is cheap to implement. (2) Timing overhead. ParRP involves extra steps in request processing, such as way checks in the partition and ownership table update, which could potentially lengthen the access time. However, we argue that with proper pipelining, the latency overhead is minimal in most cases when no physical replacement is triggered. In cases where physical replacement is necessary, the physical replacement unit must wait for the updated partition state and ownership table to correctly determine the victim.

5.3.2 Multi-level Cache Isolation under Cache Coherence

We now consider the full system model with the two-level cache hierarchy as shown in Figure 5.1b. Under the complete cache hierarchy, memory request processing involves both the two levels of caches and update their cache states as formalized in Algorithm 2. At the start, a core first issues its request to its L1 private cache. If the requested cache line is allocated inside the L1 private cache (line 1 to line 7), the request hits in the L1 cache, and the processing concludes. Otherwise, the L1 cache performs cache line replacement as necessary (line 8 to line 11) and forwards the request to be processed by the LLC (line 13). If the replaced cache line is dirty, it is written back to the LLC. The writeback is not shown in the Algorithm 2 because it does not affect the partition state as explained below. Recall that the cache hierarchy is inclusive; furthermore, cache inclusion should be

Algorithm 2: fullCacheUpdate

```
   Input: request req
 1 addr ← req.addr, core_idx ← req.requestor;
 2 set ← L1SetMap(core_idx, addr);
 3 set_entry ← set.tagCheck(addr);
 4 if set_entry ≠ null then
 5 |   set.touch(set_entry);
 6 |   return;
 7 end
 8 if set is full then
   |   // Make free entry in the set
 9 |   set_victim ← set.getVictim();
10 |   set.free(set_victim);
11 end
12 set_entry ← set.allocate(addr);
13 BI_addr ← LLCUpdate(req);
   // Back invalidation
14 if BI_addr ≠ null then
15 |   BI_set ← L1SetMap(core_idx, BI_addr);
16 |   BI_victim ← BI_set.tagCheck(BI_addr);
17 |   if BI_victim ≠ null then
18 |   |   BI_set.free(BI_victim);
19 |   end
20 end
   // Update replacement information
21 set.touch(set_entry);
```

maintained for a core's private cache and its replacement partition to achieve isolation. It means that cache line written back from the L1 cache must already be allocated in the LLC. Thus, the writeback action only updates the value of the cache line while preserving the same partition state. During request processing in the LLC (Algorithm 1), if any cache line is removed from the a core's replacement partition, to maintain the cache inclusion requirement, the same line in the core's L1 cache must be deallocated if it exists (line 14 to line 20). This process of removing L1 copy as a result of the removal of the same line in the replacement partition is referred as back invalidation. Finally, the core updates the L1 cache state given that the requested cache line is the most recently accessed one (line 21).

After the addition of private caches, the cache hierarchy requires cache coherence to

maintain data consistency across different cores, which can compromise the isolation property. We examine the sources of property violations and devise rules on cache controller behaviours to guard against them.

L1 cache isolation under coherence actions. In a cache-coherent multicore, the L1 cache not only serves the requests coming from the local core but also listens to remote requests from other cores in the system. A core's L1 cache state can be modified by other cores' remote requests if there are no proper guards against coherence activities. The first interference source is that a core's L1 cache may update the replacement information when servicing a remote request that hits in its L1 cache. For example, a core caches the most up-to-date copy of a cache line and needs to respond to a remote request for it. It is worthy to mention that not all architectures allow updating replacement information when servicing remote requests and suffer this issue. However, since the exact implementation is normally hidden, we devise the following rule as a general design guideline to eliminate this possible interference source.

Rule 5.1. *When a remote request hits in a core's private L1 cache, the cache controller simply responds with the data if required without updating the replacement information.*

The second source of violations is the coherence invalidation, where sharer copies are invalidated due to a write request. This is also widely known as multiple-reader-single-writer invariant in coherence protocol design [13]. If a cache line is invalidated, the cache controller first replaces any invalidated line for performance optimization instead of following a normal victim selection procedure in the replacement policy (e.g., selecting the oldest victim in the LRU policy). This affects the replacement decision and breaks the isolation property. We devise the second rule to eliminate this interference source.

Rule 5.2. *When a cache line in a core's private L1 cache experiences a coherence invalidation due to a remote write access, the cache line is only marked as stale and retains the same replacement information without being deallocated. Furthermore, the cache controller always follows the normal victim selection process as defined in the replacement policy and disregards the data staleness of cache lines.*

It is crucial to highlight that, under ParRP, a request is considered a cache hit if it targets a cache line allocated in a L1 cache or a replacement partition, even if the data is stale. This is valid because, ensured by the coherence protocol and ParRP architecture together, for any cache line in the caches, there is at least one valid copy of cache line inside the cache hierarchy. We will prove this later in Section 5.5.

Access Sequence		core$_1$'s L1	Request to LLC	core$_1$'s 2-way Replacement Partition
1	core$_1$: Read A	**[A, -]**	A	**[A, -]**
2	core$_1$: Read B	**[B, A]**	B	**[B, A]**
3	core$_1$: Read A	**[A, B]**		**[B, A]**

(a) Isolation reference of multi-level cache state.

Access Sequence		core$_1$'s L1	Request to LLC	core$_1$'s 2-way Replacement Partition
1	core$_1$: Read A	**[A, -]**	A	**[A, -]**
2	core$_1$: Read B	**[B, A]**	B	**[B, A]**
3	core$_2$: Write A	[B, A]		[B, A]
4	core$_1$: Read A	**[A, B]**	A	**[A, B]**

(b) Interference due to cache coherence in multi-level cache state.

Figure 5.6: Violations of cache space isolation due to coherence interference.

Replacement partition isolation under coherence actions. Similar to L1 caches, the cache lines stored in the LLC also suffer coherence invalidation, and Rule 5.2 is used to eliminate the interference source. In particular, under the partition structure as described in Section 5.3.1, enforcing Rule 5.2 means that the cache controller only marks the physical way that suffers coherence invalidation as stale without affecting the partition information and owner table. Therefore, the partitioning mechanism remains intact upon enforcing Rule 5.2.

However, a core may indirectly affect another core's replacement partition by causing it to generate extra requests to the LLC. This happens when a core's privately cached copy in L1 becomes stale due to coherence invalidation, and it needs to broadcast a coherence request to fetch the up-to-date data. Figure 5.6b shows this type of indirect interference, and the isolation reference is provided in Figure 5.6a. In this example, core$_1$'s partition state deviates from its isolation reference due to core$_2$'s write access to line A cached by core$_1$ (request 3 in Figure 5.6b). Although core$_2$'s write access does not alter core$_1$'s L1 cache state directly as protected by Rule 5.1 and Rule 5.2, core$_1$'s copy of A becomes stale. As a result, when core$_1$ makes its second request to A, A is stale in its private cache, and it sends a coherence request listened by the LLC. This coherence request is an extra request compared to the isolation reference and alters the partition state unexpectedly. The state difference can lead to different hit statuses for future requests, breaking the

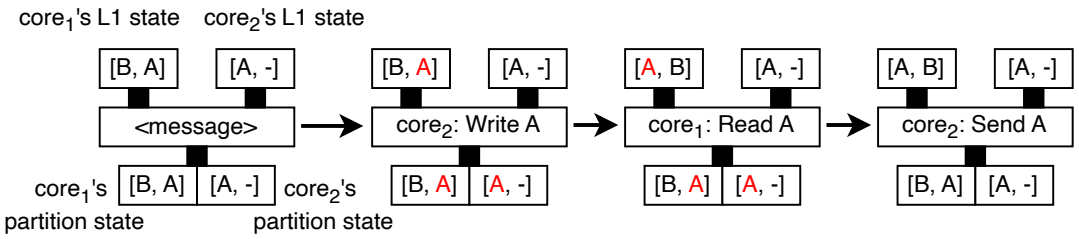


Figure 5.7: Example of coherence transactions under ParRP

isolation property.

To summarize, coherence invalidation incurs additional coherence requests for cache line that were previously cached in the L1, thereby altering the partition state. Clearly, restricting these coherence requests is not feasible as required for maintaining cache coherence. Instead, the solution is to identify these extra requests and inform the LLC not to update the partition state for them (i.e. state-preserving). To achieve that, we utilize the coherence state information embedded in existing coherence protocol implementations. For example, if a cache line is allocated but the data is stale, the line is previously invalidated by other cores. We devise Rule 5.3 to eliminate this indirect interference source.

Rule 5.3. *When a core accesses a stale copy of cache line allocated in its private L1 and needs to issue a coherence request to retrieve the valid data, this coherence request is marked as state-preserving. When processing the state-preserving request, the LLC simply process it* ***without*** *updating the replacement information.*

Example. Figure 5.7 shows an example of coherence transactions under the three rules imposed in ParRP. Staring with the initial state as shown, core_2 issues a write request to line A. Although core_2 already cached line A in its private cache, it still needs to broadcast a coherence request to invalidate the other copies cached by core_1's L1 and the LLC. According to rule 5.2, core_1 and the LLC only marks line A as stale (indicated in red) while preserving the same cache state. Later when core_1 needs to read A, it discovers that its copy of A is stale, so it broadcasts a state-preserving coherence request (rule 5.3). Therefore, when the LLC processes the request, it does not update core_1's partition state. Note that core_1 still needs to update its L1 cache state as normal. In this case, core_2 caches the most up-to-date data of A due to its prior write request, so it provides the valid data while preserving the same state (rule 5.1). Overall, the three rules together isolate a core's cache state from the coherence activities.

5.4 ParRP's Example with MSI protocol

This section presents the details of ParRP's rule enforcement upon a specific cache coherence protocol. We use the MSI protocol as an example to illustrate the core idea, and the design can be extended in a similar way to work with other protocols such as MESI and MOESI. In this section, we specifically refer to the state used in the cache coherence protocol as the coherence state to clearly distinguish it from the cache state we defined earlier in Section 5.1.2. It is important to first understand their different roles before the discussion of design details. Recall that we define the cache state from the perspective of replacement policy. In other words, the cache state concerns with which cache line addresses are allocated and how they are replaced and changed in response to requests. For this reason, we also refer to the cache state as the replacement view of the cache. For other more concrete information on the cache line storage (e.g., data value), we refer to it as the physical view of the cache. On the other hand, the coherence state is a property associated with each cache line for operating the cache coherence protocol. Unlike the cache state, the coherence state is used to maintain data consistency, which is closely tied to the physical view of the cache, rather than managing how cache lines are replaced. The operation of the coherence protocol changes the physical view of the cache, but it should leave the replacement view update process intact as described in Algorithm 2 to achieve the goal of isolation.

Table 5.2 shows the original state machine of the MSI protocol. Specifically, Table 5.2a shows the state machine at the L1 cache, which is already extensively explained in Chapter 2 and Chapter 4. We briefly explain the state machine at the LLC of the MSI protocol as shown in Table 5.2b. The coherence state MinL1 means that the cache line is held in the M coherence state in a L1 cache. Under an inclusive cache hierarchy as assumed in the system model, the line in MinL1 must be physically allocated in the LLC to satisfy the cache inclusion requirement. However, the data content is stale because it was modified by a core before. Therefore, when the LLC observes a request to this state, it needs to wait for the up-to-date data to be written back from the core that modified it and then send it to the requestor. For the other two coherence state S and I, they are the same as in the L1 state machine.

Now, we show the modification to the MSI protocol to enforce the three rules of ParRP. As emphasized previously, the coherence protocol operates on a per-cache-line basis and does not concern with the replacement decision. This is evident from the protocol table, which does not contain any action directly relating to updating replacement information. In particular, the coherence protocol does not decide which line to replace if the cache set is full. Instead, the replacement event is an input from the replacement policy running in

State	Core Events			Bus Events		
	load	store	replacement	OtherGetS	OtherGetM	OtherUpg
M	load hit	store hit	writeback / I	writeback / S	writeback / I	X
S	load hit	issue Upg / M	- / I	-	- / I	- / I
I	issue GetS / S	issue GetM / M	X	-	-	-

(a) State machine at L1.

State	Bus Events			
	OtherGetS	OtherGetM	OtherUpg	Data Writeback
MinL1	send data after written back from L1 / S	send data after written back from L1	X	update data value / S
S	send data	send data / MinL1	- / MinL1	X
I	read data from memory and send it / S	read data from memory and send it / MinL1	X	X

(b) State machine at the LLC.

Table 5.2: MSI protocol table.

Algorithm 2. This means that Rule 5.1 is automatically enforced for MSI. For example, when a core sees another core's GetS coherence request in the coherence state M, it only writebacks the data without updating the replacement information. The corner case is that the coherence protocol can implicitly affect the replacement view by deallocating the cache line. The problem occurs when lines in M or S are transitioning to I due to requests from other cores, which are referred to as the coherence invalidation in the previous section. This changes the replacement view because the I coherence state means that the cache line is invalid and needs to be removed, changing the allocation status. Recall that we devise Rule 5.2 to resolve this problem. To enforce Rule 5.2, we extend the MSI protocol with an additional coherence state called STALE as shown in Table 5.3. If a cache line is in STALE, it means that it is allocated but its data is stale. The key difference between STALE and I is that the cache line is allocated for STALE but not present for I. The coherence state STALE is added to preserve the replacement view when a valid cache line suffers coherence invalidation. In the new state machine, cache lines in M or S move to STALE instead of I when observing other cores' GetM or Upg coherence requests. The transition to I now only occurs during a replacement event triggered from Algorithm 2. This modification enforces Rule 5.2. When processing a core's load or store request, the coherence state STALE has

State	Core Events			Bus Events		
	load	store	replacement	OtherGetS or OtherGetS*	OtherGetM or OtherGetM*	OtherUpg
M	load hit	store hit	writeback / I	writeback / S	writeback / STALE	X
S	load hit	issue Upg / M	- / I	-	- / STALE	- / STALE
STALE	issue GetS* / S	issue GetM* / M	- / I	-	-	-
I	issue GetS / S	issue GetM / M	X	-	-	-

(a) State machine at L1.

State	Bus Events			
	OtherGetS or OtherGetS*	OtherGetM or OtherGetM*	OtherUpg	Data Writeback
MinL1	send data after written back from L1 / S	send data after written back from L1	X	update data value / S
S	send data	send data / MinL1	- / MinL1	X
I	read data from memory and send it / S	read data from memory and send it / M	X	X

(b) State machine at the LLC.

Table 5.3: Updated MSI protocol table to enforce ParRP's rules. '*' indicates that the coherence request is state-preserving in terms of the LLC replacement view.

the same transitions as the I coherence state. The only difference is that, if the line is in STALE, it marks the coherence request as state-preserving (SP), indicated by the '*' mark in the protocol table. This identifies additional requests caused by coherence invalidation and instruct the LLC to preserve the same replacement view of the LLC when processing them, as required by Rule 5.3. Note that the term "state-preserving" is described from the replacement view standpoint but not the coherence state. From the coherence standpoint, the state-preserving version of requests do not have any difference compared to normal requests and trigger the same coherence state transition. For the LLC's protocol state machine, the MinL1 coherence state already serves as the STALE coherence state, so no modification is required.

Under the complete solution with cache coherence, the request processing of caches now consists of two parts: First, each cache performs the coherence actions and coherence state transition according to the protocol's state machine as described in Table 5.3. At the same time, the cache follows the same replacement procedure for each core as if they were

State	Core Events			Bus Events	
	load	store	replacement	OtherGetS or OtherGetS* or OtherGetM or OtherGetM*	OtherUpg
M	Directly touch the line (Algorithm 2: line 4-7)	Directly touch the line (Algorithm 2: line 4-7)	Deallocate cache line (Algorithm 2: 8-11)	No replacement update	X
S	Directly touch the line (Algorithm 2: line 4-7)	Directly touch the line (Algorithm 2: line 4-7)	Deallocate cache line (Algorithm 2: 8-11)	No replacement update	No replacement update
STALE	Directly touch the line (Algorithm 2: line 4-7)	Directly touch the line (Algorithm 2: line 4-7)	Deallocate cache line (Algorithm 2: 8-11)	No replacement update	No replacement update
I	Trigger replacement if full, may receive back invalidation from the LLC to replace another line, allocate and touch the line (Algorithm 2: line 8-21)	Trigger replacement if full, may receive back invalidation from the LLC to replace another line, allocate and touch the line (Algorithm 2: line 8-21)	X	No replacement update	No replacement update

(a) Replacement view of L1's state machine.

State	Bus Events			
	OtherGetS or OtherGetM	OtherGetS* or OtherGetM*	OtherUpg	Data Writeback
MinL1	Follow Algorithm 1 to update replacement info: trigger partition replacement and back invalidation if the target partition is full, touch the line in the partition	No replacement update	X	No replacement update
S	Follow Algorithm 1 to update replacement info: trigger partition replacement and back invalidation if the target partition is full, touch the line in the partition	No replacement update	No replacement update	X
I	Follow Algorithm 1 to update replacement info: trigger physical replacement if physical ways are full, trigger partition replacement and back invalidation if the target partition is full, touch the line in the partition	X	X	X

(b) Replacement view of the LLC's state machine.

Table 5.4: Replacement view of the updated MSI protocol table.

running in isolation as described in Algorithm 2. To clarify how the cache replacement procedure operates in parallel with the protocol's state machine, Table 5.4 relates the cache processing from the replacement view standpoint for the same protocol's state machine in Table 5.3. Table 5.4a shows the replacement view at the L1 controller specifically. When a L1 cache processes a load or store request from its owner core, if the cache line is found in

the L1 (i.e., the line is in the M, S, or STALE coherence states), it simply touches the line (i.e., marks it as the most recently used in the replacement policy). If the line is not present (i.e., in the I coherence state), the cache needs to allocate it and then touches it, which may trigger replacement in L1, the LLC replacement and back invalidation for the LLC victim when the LLC processes GetS or GetM requests issued by the L1 (see Table 5.4b). Note that the table does not include the event for back invalidation, but it is handled in the same way as the replacement event, which requires deallocating the cache line. For bus events coming from other cores, there is no replacement state update, which is crucial for ensuring isolation. The LLC's replacement view under the protocol's state machine is shown in Table 5.4b. The LLC processes OtherGetS or OtherGetM coherence requests as normal data access requests from the L1 and update the partition's replacement state accordingly. These are the same set of requests that would be issued when the respective core were running in isolation. For other types of coherence requests (OtherGetS*, OtherGetM* and OtherUpg), which are caused by coherence actions amongst cores, the LLC does not update replacement view.

In summary, the new protocol state machine ensures that the coherence activity does not affect the replacement view. In particular, each core maintains its own replacement view of the cache subsystem under partitioning and updates it in the same way as it would in isolation. However, the coherence protocol does affect the physical view of the cache, where the valid data moves around between different caches as cores make their requests. Although the exact location of valid data depends on the physical view, the coherence protocol guarantees that for any cache line present in the replacement view, at least one cache holds the most up-to-date data of it. This effectively avoids the worst-case assumption of accessing the much slower main memory.

5.5 ParRP's Properties

This section details the hardware guarantees provided by ParRP. We first show that ParRP satisfies the isolation property, then we elaborate the architectural guarantee for cache hits under ParRP.

Theorem 5.1. ParRP *satisfies the isolation property for both a core's private L1 cache and its replacement partition, where cache hits mean accessing a cache line in either the core's L1 cache (L1 hit) or its replacement partition (partition hit).*

Proof. The implementation of ParRP realizes the mechanism described in Algorithm 2, which is the isolation reference of cache state update. Algorithm 2 calls Algorithm 1 to

update the partition state, which is deterministic and only affected by the core's actions. The rest of Algorithm 2 also updates the L1 cache state in a deterministic manner without dependence on other cores' actions. Therefore, for a core, starting with the same initial L1 cache state and partition state, the state evolution is exactly the same as if the core runs in isolation. Given that cache hits are a function of cache states (i.e. a request to a cache line is a hit if the line is present in the cache state), requests that are cache hits under isolation are still hits regardless of contentions from other cores. Thus, the isolation property is satisfied. □

Beyond the isolation property, it is noteworthy that ParRP also retains the same state update process as a normal LRU cache. This means that it is possible to analyze the replacement partition by treating it as a normal LRU cache with the same number of ways as the partition size, thus enabling the use of existing static cache analysis methods [36], [49]. Now, we elaborate the hardware guarantee ensured by ParRP for cache hits. ParRP provides different guarantees depending on whether cache lines suffer coherence invalidation.

Theorem 5.2. ParRP *provides the following hit guarantees as devised from the cache state. (1) For cache line that does not suffer coherence invalidation (e.g., instruction and private data), a L1 hit means that valid data is present in the L1 cache, and a partition hit means that valid data is present in the LLC. (2) For cache line that suffers coherence invalidation (e.g., shared data), both a L1 hit and a partition hit mean that valid data is present somewhere within the cache hierarchy.*

Proof. The data staleness is caused by coherence invalidation. Therefore, for cache lines storing private data or instructions that do not suffer coherence invalidation, their values can never be stale. Under this scenario, the presence of the cache line in the cache state directly means that the valid data is allocated in the corresponding cache. This completes the proof of the guarantee in case (1). For cache lines storing shared data that suffer coherence invalidation, the cache line may have stale copies. However, there is a least one valid copy in the cache hierarchy. We do a further case analysis to show that it is true. **Case (2.1).** If a request is a partition hit, by construction of Algorithm 1, the cache line must be allocated in a physical cache way. If the cache line is valid, the proof directly follows. If the cache line is stale, the cache line must be dirty (i.e. the valid copy) in at least one L1 cache due to a prior write request. This must be true because a dirty line needs to be written back to the LLC upon eviction, which makes the copy in the LLC valid again and contradicts the assumption.

Case (2.2). If a request is a L1 hit, it must also be a partition hit according to the inclusive requirement by construction of Algorithm 2. Therefore, the proof follows Case (2.1).

This completes the proof of the guarantee in Case (2). □

The two theorems only focus on the aspect of cache hits. For requests that are originally cache misses under isolation, they may find the cache lines already inside the cache when under contention. For example, a shared line may be fetched by another core earlier. However, those requests may still end up as misses like in isolation, and we assume the main memory access is required in the worst case.

5.6 Evaluation

We use gem5 micro-architecture simulator [42] to evaluate ParRP. Table 5.5 shows the default configuration of the system. To exploit the benefits of the isolation property provided by ParRP, we compare it against two other setups: (1) Share. LLC is simply shared without any isolation mechanisms. (2) Color. Perfect cache coloring is used to completely isolates private data from shared data. Essentially, private data and shared data are stored separately in different sets in both levels of caches. Further, private data for different cores is also stored in different sets in the LLC to isolate from each other.

Table 5.6 summarizes the per-request worst-case latency (PR-WCL) to retrieve the data with different guarantees of the valid cache line location in the system with different core counts. We obtain the values by measurement and applying a 20% safety margin on the observed worst-case value. The short names for different guarantees are highlighted in bold, which are used in Table 5.7 that summarizes the guarantees of all setups. Table 5.7 shows the guarantee given the request hit status in the isolation mode. Note that for ParRP, LLC hit means partition hit under isolation. Similar to the discussion of the guarantee provided by ParRP in Section 5.5, different guarantees are derived for different access types. There are three access types: "private" (private instruction or data), "shared inst" (shared instruction), and "shared data". We briefly explain the guarantees provided by the three setups. For Share, we do not have any guarantee that the valid data is in the cache hierarchy for all access categories due to inter-core interference (e.g., back invalidation and shared LLC contention). Therefore, in the worst case, we assume all accesses go to the main memory. For Color, since private data is isolated from shared data, the guarantees of private data locations are preserved. However, shared data lost all guarantees for the same reason as in Share. For ParRP, according to Theorem 5.2 in Section 5.5, most accesses preserves the same guarantee. The exception is the shared data access that is a L1 hit in isolation, for which we can only guarantee that the valid data is located somewhere within

Table 5.5: System configuration.

Parameters	Configuration
Private L1 I-cache	4-way, 16 KiB, 1 cycle latency
Private L1 D-cache	4-way, 16 KiB, 1 cycle latency
Shared LLC	32-way, 1 MiB, 10 cycles latency, inclusive of L1
Interconnect	Snooping bus, 3 cycles latency
Main memory	Single-port SRAM model, 100 cycles latency
Number of cores	2, 4, 8 (default)

Table 5.6: Per-request worst-case latency.

Number of cores	**L1** Access	Anywhere in **Cache**	**Mem** Access
2	1	87	568
4	1	175	1063
8	1	431	2065

Table 5.7: Guarantee in various setups.

Access type	private		shared inst		shared data	
Hit status	L1	LLC	L1	LLC	L1	LLC
Share	Mem	Mem	Mem	Mem	Mem	Mem
Color	L1	Cache	Mem	Mem	Mem	Mem
ParRP	L1	Cache	L1	Cache	Cache	Cache

the cache hierarchy due to coherence invalidation. However, this guarantee is still much better than assuming accessing the main memory as compared to the other two setups.

5.6.1 Verification

We use synthetic memory traffic generator for verifying the isolation property and guarantees provided by ParRP. The verification is performed as follows. The isolation reference is first obtained by running a core alone and collecting the hit statuses for its memory accesses. Next, we rerun the same memory trace for the same core and collect the hit statuses again while other cores are injecting interfering memory traffic. We verify the

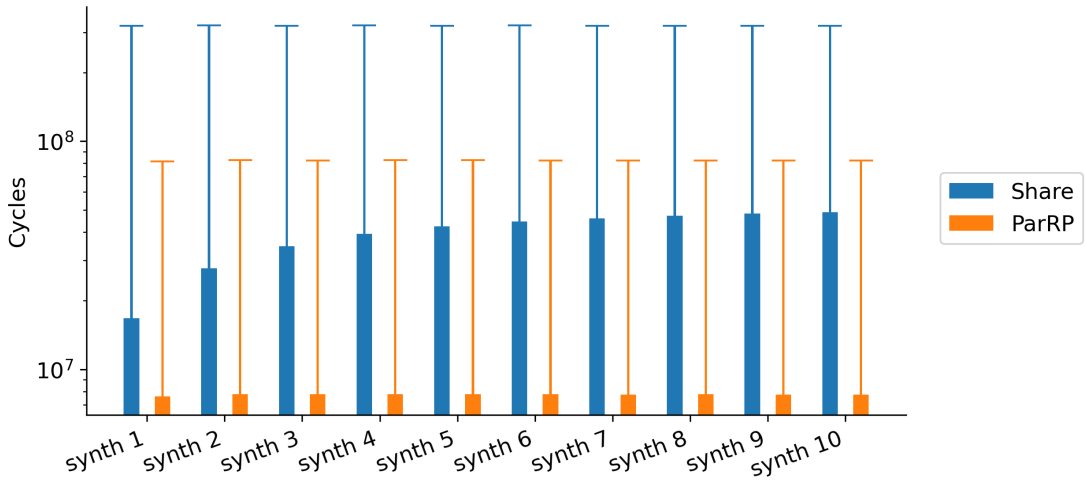


Figure 5.8: Synthetic benchmark.

correctness by checking that the guarantee is met for each access as indicated in Table 5.7 for ParRP. The synthetic workloads generate a mixture of read and write accesses to predefined memory regions, where some of them are private to a core, and some are shared. This allows us to test the guarantee for each category. Figure 5.8 shows the total WCL of all memory accesses in the synthetic workloads under ParRP and Share. We classify the synthetic workloads based on the amount of interference from other cores. For example, if a workload has more interference, it uses smaller time interval for other cores to inject a request. The index in the synthetic workloads denotes the intensity of the interference with one as baseline. The solid bar represents the actual WCL, and the "T" bar is the analytical bound. The analytical bound is computed by summing the PR-WCLs in Table 5.6 corresponding to the guarantees, which are in turn determined based on the recorded hit statuses according to Table 5.7.

Observations. No property violation is observed for ParRP for all synthetic workloads. All observed WCLs are within the analytical bounds for both setups. The analytical bound of Share is significantly larger than ParRP because it loses all hit guarantees when in contentions, while ParRP preserves most of them because of the isolation property. With increasing contention, the observed bound for Share increases significantly as compared to ParRP. This emphasizes the effectiveness of ParRP in isolating multicore interference.

5.6.2 WCET

We use BOTS [50] benchmark to evaluate the WCET of the task parallelized on the multicore platform. Benchmark programs in BOTS are written with OpenMP task framework that can be modelled as a directed acyclic graph (DAG) [51] for the WCET analysis of parallel programs. Basically, OpenMP task pragmas serve as synchronization points, and between the two consecutive points is the schedulable code segment that corresponds to a node in the DAG. We refer the readers to [52], [53] for more detailed discussion on the conversion from OpenMP task-based programs to their DAG models. In our case, we capture a program's DAG by code instrumentation and program tracing. Specifically, we instrument the OpenMP task pragmas to trace the nodes and their dependencies in runtime. For each DAG node, we record the time spent in computation (i.e. non-memory instructions) and the hit statuses of memory accesses. We compute the WCET of a DAG node as the sum of the recorded computation time and analytical total memory latency. Similar to synthetic workloads, the analytical total memory latency of a node is obtained using the guarantees in Table 5.7 based on the recorded hit statuses. After obtaining the WCET of each node, the WCET of the whole task (whole DAG) is computed using Graham's bound [54]. When computing the bound for different setups, Share and ParRP use their own version of memory traces as collected in the simulation. For Color, we do not implement it in simulation because it requires a full stack of supports from both the compiler and operating system. For example, the compiler needs to first annotate memory region as shared or private, and the operating system needs to provide a memory allocator that performs cache coloring to isolate private region from shared region. Instead, we estimate the result of Color by applying the guarantee of Color on the trace of Share. Figure 5.9 shows the access frequency breakdown based on the access types in BOTS programs. Finally, Figure 5.10 shows the normalized WCET of the BOTS programs with ParRP as the baseline. We also vary the core count to study the sensitivity.

Observations. All programs have a significant portion of shared instruction accesses because of the shared function call. Because Share and Color do not have hit guarantee for shared accesses, their WCET are two orders of magnitude larger than ParRP as shown in Figure 5.10a. For a fairer comparison, we assume that there are OS and compiler supports to duplicate the instructions for different cores, where duplicated copies are further partitioned for different cores. As a result, we treat the shared instruction simply as private instruction and update the WCET in Figure 5.10b. As expected, the benefits closely correlate with the amount of shared data accesses in the program. For example, compared to Color, ParRP tightens the bound by about an order of magnitude for sparselu, which has the most shared data accesses. sort and strassen have little shared accesses,

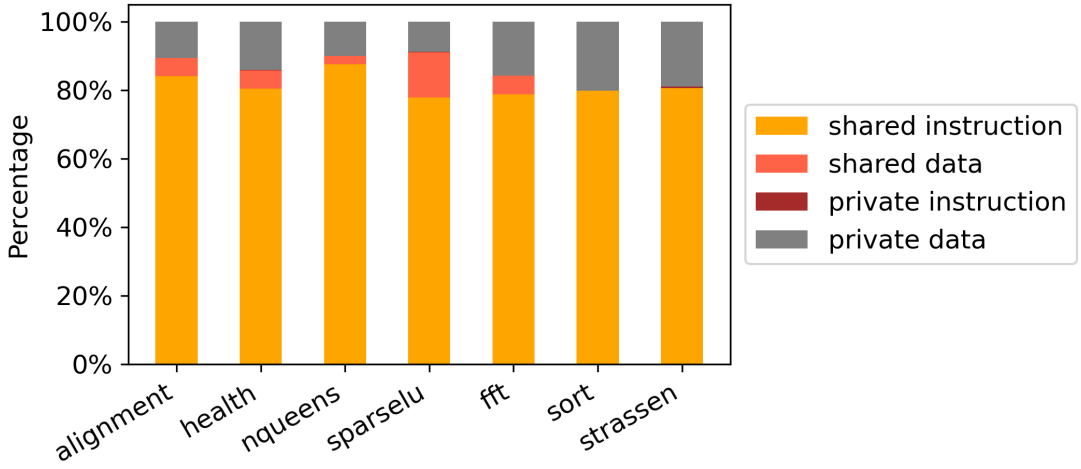


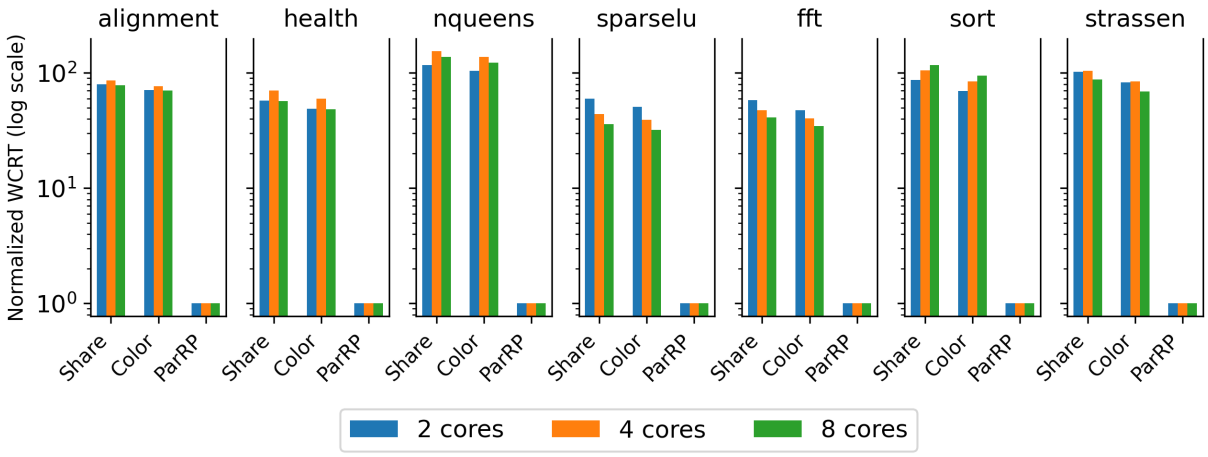
Figure 5.9: Percentage breakdown of access frequency in BOTS benchmark.

so no benefit is observed over Color. For strassen, when the core count is 8, Color has a tighter bound than ParRP because ParRP experiences less L1 hits due to more back invalidations when the partition is small. The impact by back invalidations are studied further in the next experiment on average performance. Overall, the results are consistent over different core counts, and on average, ParRP tightens the bound by 2.4× compared to Color with instruction duplication. It is worth noting that we are favoring the Color approach in this comparison, as achieving perfect cache coloring is challenging in practice. Additionally, it requires instruction duplication to achieve the performance number as shown in Figure 5.10b, which is not necessary for ParRP.

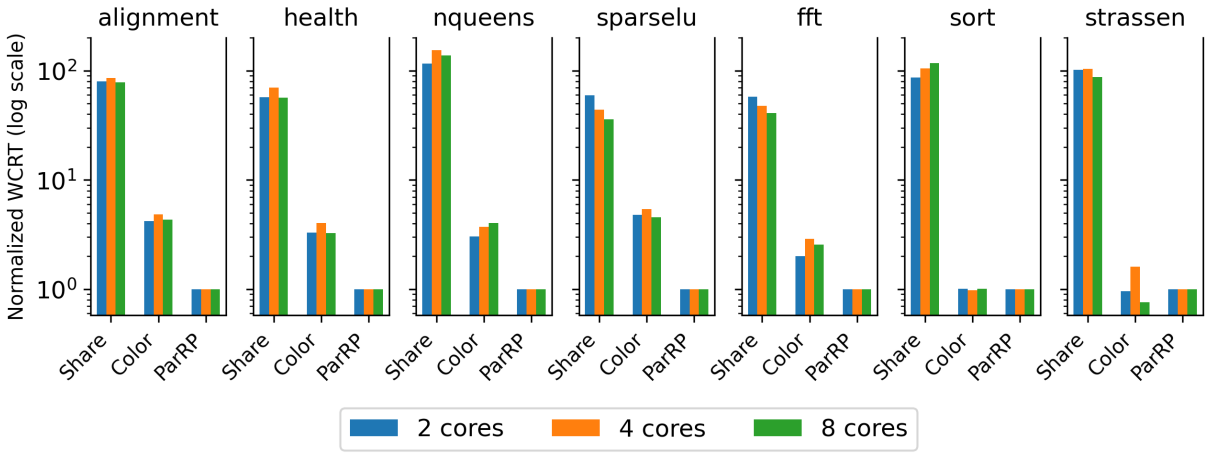
5.6.3 Average Performance

Figure 5.11 shows the normalized execution time of ParRP against Share. We vary the number of cache ways in the LLC while keeping the same total capacity to observe the influence of associativity. Note that 8-way is the minimum way configuration for an 8-core system where each core only gets assigned one cache way after partitioning the LLC. Besides BOTS benchmark, we also use Splash-3 benchmark to study the impact on average performance, and its result is shown in Figure 5.11b. Different from BOTS benchmark, Splash-3 is written for high-performance implementation, enabling us to exercise enough pressure to the memory system to reveal the average performance impact more consistently.

Observations. ParRP experiences the performance degradation with a maximum at



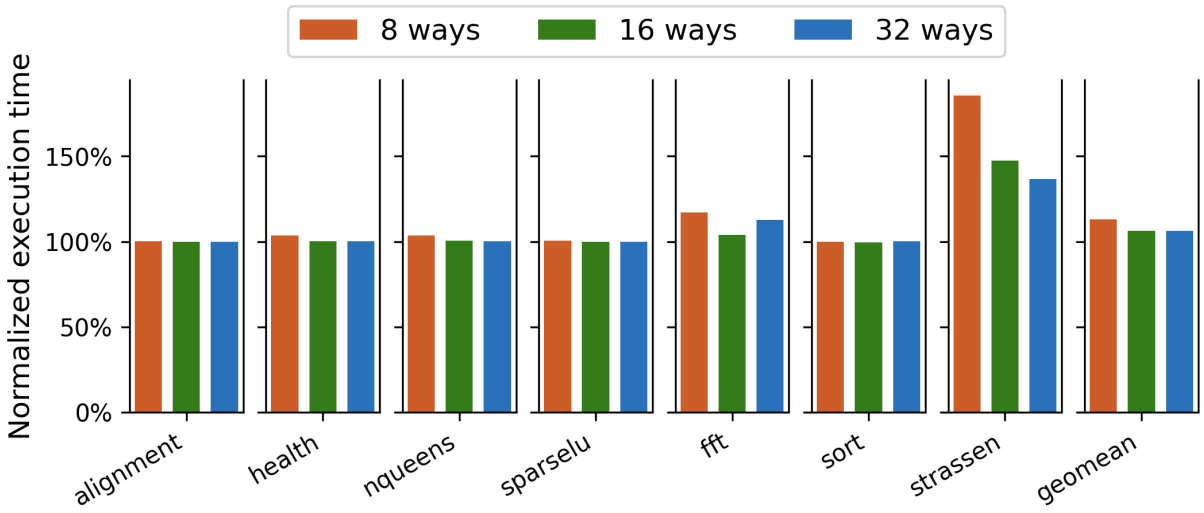
(a) default WCET without instruction duplication



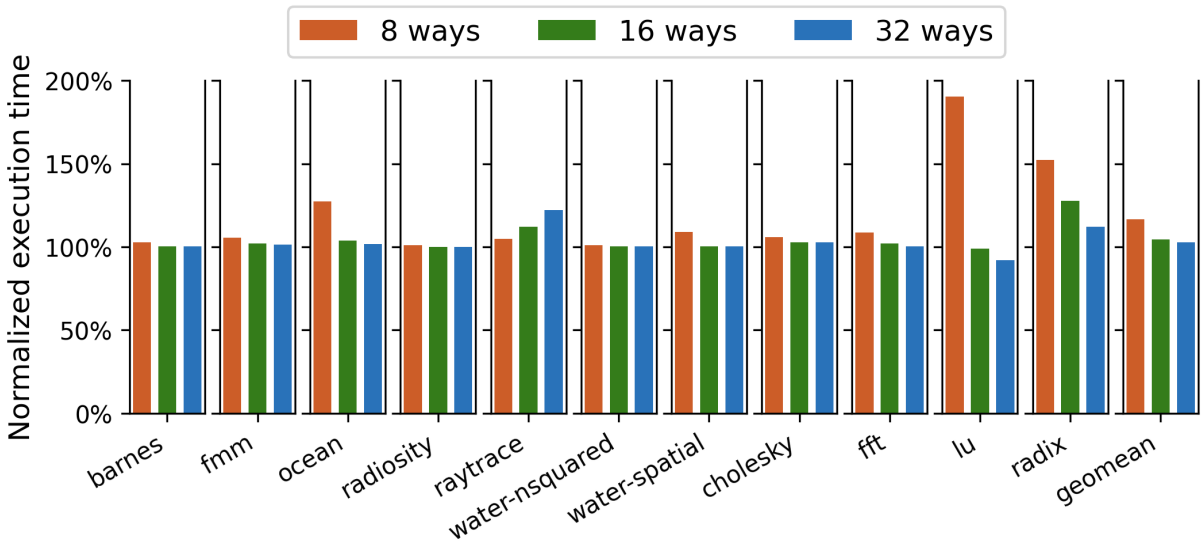
(b) WCET with instruction duplication

Figure 5.10: WCET for BOTS benchmarks.

16.5% under the extreme configuration of 8 ways cache set. The performance degradation becomes smaller as more ways are available. We observe that there are two factors of the performance degradation. The primary factor is that more back invalidations are generated with smaller ways, causing reduced L1 hit rate. This is consistent with ParRP requirement of generating back invalidation when the cache line is removed from the replacement partition. The secondary factor is that the LLC hit rate is reduced due to more conflict misses in the shared set. However, there are some outliers like raytrace where the performance



(a) Average Performance (BOTS)



(b) Average Performance (Splash-3)

Figure 5.11: Average Performance.

degrades with more ways. The reason for this is that there are more LLC misses compared to Share as more ways are available. We attribute this to the non-uniform access pattern in raytrace. Specifically, after changing the associativity of the cache organization, many

more accesses are mapped to the same set compared to the increase in the number of ways. As a result, more conflict misses happen in the LLC. In general, more ways are in fact beneficial in minimizing the performance degradation except a few outliers due to access patterns.

5.7 Conclusions

In this chapter, we present ParRP, a novel partitioning approach to achieve cache space isolation for cache-coherent multicores. The cache space isolation holds in a general sense for any data type including private data and shared data. Furthermore, ParRP is a hardware partitioning mechanism, which is transparent to the software stacks running atop. We evaluate the effectiveness of ParRP on parallel tasks following the DAG model, which shows that ParRP can tighten the bound by 2.4× compared with conventional partitioning.

Chapter 6

Conclusion and Future Work

Multicores are steadily being deployed in real-time embedded systems driven by their high performance, as well as their ability to consolidate multiple functions into a single platform in a cost-effective manner. However, the inter-core interference resulting from shared resources in multicores continues to be a challenge hindering multicore deployment on hard real-time systems. The focus of this thesis is on addressing the interference in the cache subsystem, which is one of the major sources of timing unpredictability in modern multicore architectures. In particular, this thesis presents two solutions addressing the contention in temporal and spatial domains respectively.

- A predictable exclusive cache coherence mechanism (PECC) that achieves a low PR-WCL while enabling the performance benefits by incorporating a shared LLC.
- A hardware cache partitioning techniques (ParRP) that offers strong space isolation for cache-coherent architectures.

The proposed solutions also suggest the following future extension. As our two solutions address contentions in orthogonal domains, it is expected to integrate them into a comprehensive solution. Currently, we demonstrate ParRP's working principles and effectiveness through an inclusive hierarchy based on measurements. However, it lacks discussion of extending this approach to an exclusive cache hierarchy, which is a crucial design aspect in PECC to address the contention in temporal domain. Given that exclusive hierarchies exhibit distinct characteristics in data allocations and coherence mechanisms compared to inclusive caches, future works will need to carefully study ParRP's extension to exclusive hierarchies and compare the overall benefit.

References

[1] X. Wang, Z. Wu, R. Pellizoni, and H. Patel, "Exclusive Hierarchies for Predictable Sharing in Last-level Cache," in *2024 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2024.

[2] 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)*, M. Bertogna, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:20, ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.10.

[3] 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. DOI: 10.1109/AERO.2007.352639.

[4] J. Nowotsch and M. Paulitsch, "Leveraging Multi-core Computing Architectures in Avionics," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 132–143. DOI: 10.1109/EDCC.2012.27.

[5] J. P. Cerrolaza, R. Obermaisser, J. Abella, *et al.*, "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, Aug. 2020, ISSN: 0360-0300. DOI: 10.1145/3398665.

[6] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "A comprehensive survey of industry practice in real-time systems," *Real-Time Syst.*, vol. 58, no. 3, pp. 358–398, Sep. 2022, ISSN: 0922-6443. DOI: 10.1007/s11241-021-09376-1.

[7] 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. DOI: 10.1109/RTSS.2016.015.

[8] *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.

[9] *Software Optimization Guide for the AMD Zen4 Microarchitecture*, 1.00, AMD, 2023.

[10] *Arm Cortex-R8 MPCore Processor*, r0p3, ARM, 2019.

[11] *Arm Cortex-R82 Processor Technical Reference Manual*, r0p2, ARM, 2022.

[12] *Cortex-A9 Technical Reference Manual*, r4p1, ARM, 2012.

[13] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger, *A Primer on Memory Consistency and Cache Coherence*, 2nd. Morgan Claypool Publishers, 2020, ISBN: 1681737094.

[14] J. Bin, S. Girbal, D. Gracia Pérez, A. Grasset, and A. Mérigot, "Studying co-running avionic real-time applications on multi-core COTS architectures," in *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, Feb. 2014.

[15] G. Gracioli and A. A. Fröhlich, "On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 2, pp. 2–16, Jan. 2016, ISSN: 0163-5980. DOI: 10.1145/2883591.2883594.

[16] 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, ISSN: 0360-0300. DOI: 10.1145/2830555.

[17] L. Chen, H. Shen, and S. Platt, "Cache contention aware Virtual Machine placement and migration in cloud datacenters," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, 2016, pp. 1–10. DOI: 10.1109/ICNP.2016.7784447.

[18] A. Bansal, J. Singh, Y. Hao, J. Wen, R. Mancuso, and M. Caccamo, "Cache Where you Want! Reconciling Predictability and Coherent Caching," *CoRR*, vol. abs/1909.05349, 2019. arXiv: 1909.05349.

[19] 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. DOI: 10.1109/RTAS.2017.13.

[20] 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. DOI: 10.1109/RTAS52030.2021.00017.

[21] 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, ISSN: 1539-9087. DOI: 10.1145/3556975.

[22] 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.

[23] 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, San Francisco, California: Association for Computing Machinery, 2022, pp. 1273–1278, ISBN: 9781450391429. DOI: 10.1145/3489517.3530614.

[24] 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.

[25] 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, pp. 321–332, Jun. 2012, ISSN: 0163-5964. DOI: 10.1145/2366231.2337196.

[26] 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, pp. 414–423, May 1986, ISSN: 0163-5964. DOI: 10.1145/17356.17404.

[27] 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, New York: Association for Computing Machinery, 2009, pp. 413–422, ISBN: 9781605587981. DOI: 10.1145/1669112.1669165.

[28] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 45–54. DOI: 10.1109/RTAS.2013.6531078.

[29] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions,*

and Defenses - Volume 9404, ser. RAID 2015, Kyoto, Japan: Springer-Verlag, 2015, pp. 48–65, ISBN: 9783319263618. DOI: 10.1007/978-3-319-26362-5_3.

[30] *Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring(MPAM), for Armv8-A*. A.a, ARM, 2019.

[31] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A Closer Look at Intel Resource Director Technology (RDT)," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS '22, Paris, France: Association for Computing Machinery, 2022, pp. 127–139, ISBN: 9781450396509. DOI: 10.1145/3534879.3534882.

[32] *NVIDIA Orin Series System-on-Chip Technical Reference Manual*, 1.2, Nvidia, 2023.

[33] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 57–67. DOI: 10.1109/RTSS.2009.32.

[34] Zhang, Wei and Lv, Mingsong and Chang, Wanli and Ju, Lei, "Precise and scalable shared cache contention analysis for WCET estimation," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22, San Francisco, California: Association for Computing Machinery, 2022, pp. 1267–1272, ISBN: 9781450391429. DOI: 10.1145/3489517.3530613.

[35] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun, *Deterministic Memory Abstraction and Supporting Multicore System Architecture*, 2018. arXiv: 1707.05260 [cs.AR].

[36] 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, Jun. 2016. DOI: 10.4230/LITES-v003-i001-a005.

[37] 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)*, M. Maggio, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 231, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 17:1–17:23, ISBN: 978-3-95977-239-6. DOI: 10.4230/LIPIcs.ECRTS.2022.17.

[38] 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, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2010.08.007.

[39] 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.

[40] *ARM CoreLink CCI-550 Cache Coherent Interconnect, Technical Reference Manual*, r1p0, ARM, 2018.

[41] 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. DOI: 10.1109/HOTI.2010.24.

[42] N. Binkert, B. Beckmann, G. Black, *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[43] 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. DOI: 10.1109/ISPASS.2016.7482078.

[44] 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. DOI: 10.1109/ISCA.1995.524546.

[45] D. T. Chiou, Arvind, and L. Rudolph, "Extending the reach of microprocessors: Column and curious caching," AAI0801277, Ph.D. dissertation, Massachusetts Institute of Technology, USA, 1999.

[46] K. Kędzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470352.

[47] Y. Chen, W. Li, C. Kim, and Z. Tang, "Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161016.

[48] K. T. Sundararajan, T. M. Jones, and N. P. Topham, "RECAP: Region-Aware Cache Partitioning," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 294–301. DOI: 10.1109/ICCD.2013.6657056.

[49] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, vol. 18, pp. 131–181, 1999. DOI: https://doi.org/10.1023/A:1008186323068.

[50] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *2009 International Conference on Parallel Processing*, 2009, pp. 124–131. DOI: 10.1109/ICPP.2009.64.

[51] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A Generalized Parallel Task Model for Recurrent Real-time Processes," in *2012 IEEE 33rd Real-Time Systems Symposium*, 2012, pp. 63–72. DOI: 10.1109/RTSS.2012.59.

[52] R. Vargas, E. Quinones, and A. Marongiu, "OpenMP and timing predictability: A possible union?" In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 617–620. DOI: 10.7873/DATE.2015.0778.

[53] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of OpenMP4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015, pp. 157–166. DOI: 10.1109/CASES.2015.7324556.

[54] R. L. Graham, "Bounds for certain multiprocessing anomalies," *The Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966. DOI: 10.1002/j.1538-7305.1966.tb01709.x.