

WCET Preserving Memory Management for High Performance Mixed Criticality Systems

by

Wafic Lawand

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, 2025

© Wafic Lawand 2025

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 Contribution

Chapter 4 uses the content of the following publication.

W. Lawand, R. Pellizzoni. Duration-based Instruction Cache Locking. 2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, 2024. [45]

The use of the paper’s content in this thesis has been approved by all co-authors.

Abstract

This thesis explores innovative memory and cache management mechanisms for mixed-criticality systems that aim to improve the performance of the system while maintaining its predictability. We consider mixed-criticality systems where requestors may issue latency-critical (*LTC*) and non-latency-critical (*NLTC*) requests. *LTC* requests must adhere to strict latency bounds imposed by safety-critical applications, but timely servicing *NLTC* requests is necessary to maximize overall system performance in the average case. In this thesis, we focus on addressing the challenges of processing *LTC* and *NLTC* memory requests in mixed-criticality systems, and the limitations of existing cache management mechanisms designed to improve the system’s average performance. Accordingly, we introduce two key contributions: one at the memory arbitration level and the other at the cache controller level.

In the first contribution, we propose DAMA, a dual arbitration mechanism that imposes an upper bound on the cumulative latency of *LTC* requests without unduly impacting the performance of *NLTC* requests. DAMA comprises a high-performance arbiter, a real-time arbiter, and a mechanism that constantly monitors the cumulative latency of requests suffered by each requestor. DAMA primarily executes in high-performance mode and only switches to real-time mode in the rare instances when its incorporated mechanism detects a violation of a task’s timing guarantee. We demonstrate the effectiveness of our arbitration scheme by adapting a predictable prefetcher that issues *NLTC* requests and attaching it to the L1 caches of our cores. We show both formally and experimentally that DAMA provides timing guarantees for *LTC* requests while processing other *NLTC* requests. Our evaluation demonstrates that with a negligible overhead of less than 1% on the cumulative latency bound of *LTC* requests, DAMA can achieve an equivalent average performance to a prefetcher that processes requests under a high-performance arbitration scheme.

In the second contribution, we introduce a novel hardware mechanism at the cache controller level that leverages the LRU age bits to perform duration-based cache locking. Our proposed mechanism dynamically locks and unlocks cache lines for different durations at run-time, without the need to modify the program’s code. We further devise a heuristic that analyzes a program’s loop structure and selects the set of addresses to be locked in a L1 instruction cache alongside their locking durations. Evaluation results show that our duration-based locking mechanism significantly reduces initialization overhead and eliminates the need for program code modifications. The proposed mechanism achieves a performance comparable to the dynamic approach, which adjusts locked program content at runtime.

Acknowledgements

First and foremost, I am profoundly grateful to Allah for granting me the strength, perseverance, and guidance throughout this journey.

I extend my deepest gratitude to my supervisor, Professor Rodolfo Pellizoni, for his invaluable support and guidance throughout my degree. His mentorship has been instrumental in shaping my academic and professional growth.

I am forever indebted to my parents, Nabila and Khodor, for their unwavering love, sacrifices, and prayers that have been the cornerstone of my success. To my sisters, Ghida and Ghinwa, thank you for your immense support in every way possible and for always believing in me and my decisions. For my little niece Talia, thank you for being a bundle of joy and filling our lives with happiness and laughter.

I am deeply grateful to my cousin Alaa, whose support made my transition to Canada smooth and whose delicious food made me feel at home, and to Jana, who, despite being in Lebanon, has always been there to cheer me on.

A heartfelt thank you to my friends, Mohamad El Iskandarani and Mohamad Mashaka, for always standing by my side and helping me navigate through the ups and downs I encountered during this time, as well as for the insightful conversations we shared, which greatly contributed to my personal growth.

Finally, I am grateful to my lab mates, Abdur Rahman and Mohamed Sajjad, for their collaboration in both my research and coursework, as well as their friendship, which has made my academic journey even more enriching.

Dedication

This is dedicated to my parents, Nabila and Khodor.

Table of Contents

Author's Declaration	ii
Statement of Contribution	iii
Abstract	iv
Acknowledgements	v
Dedication	vi
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Problem statement	2
1.2 Thesis Contributions	3
1.3 Thesis Structure	4
1.4 Acknowledgement	5
2 Background and Related Work	6
2.1 Memory Bandwidth Regulation	6
2.2 Predictable Arbitration Schemes	7

2.3	Prefetchers	10
2.3.1	Software Prefetching Approaches	11
2.3.2	Hardware Prefetching Approaches	12
2.4	Cache Locking	13
3	DAMA: A Dual Arbitration Mechanism for Mixed-Criticality Applications	15
3.1	DAMA: The Proposed Solution	15
3.1.1	System Model and Assumptions	16
3.1.2	High Level Architecture	18
	Queuing Structure	18
	High Performance and Real-Time Arbiters	18
	Latency Counters and Arbitration Mode Switcher	19
3.1.3	Latency Regulation Mechanism	19
3.1.4	Illustrative Example	20
3.1.5	Latency Guarantees	22
3.2	Predictable Prefetcher	25
3.2.1	Ensuring Predictability with a Prefetch Side Buffer	25
3.2.2	Features Integrated into our Predictable Prefetcher	26
3.2.3	Handling Prefetch Requests at the Cache Controller Side	27
3.3	Evaluation	29
3.3.1	Achieving Latency Bounds with DAMA	31
3.3.2	Impact of Prefetch Side Buffer Size on IPC	33
3.3.3	Handling Prefetch Requests with DAMA	34
3.4	Conclusions	38

4	Duration-based Instruction Cache Locking	39
4.1	Duration-based Locking Mechanism	39
4.1.1	Duration-based LRU Policy	40
4.1.2	Hardware Implementation	42
4.2	Region Selection Heuristic	42
4.2.1	Extracting and Analyzing Loops	43
4.2.2	Selecting Locked Cache Lines	44
4.2.3	Selecting Memory Regions and Durations	45
4.3	Evaluation	46
4.4	Conclusions	51
5	Conclusions and Future Work	52
	References	54

List of Figures

2.1	An example of how requests are handled under the FCFS arbitration scheme	8
2.2	An example of how requests are handled under the RR arbitration scheme	8
2.3	An example of how requests are handled under the TDMA arbitration scheme	9
2.4	An example demonstrating the impact of the prefetcher on the compute and memory load phases of a program	10
3.1	Processing and Queueing latencies under various scenarios	17
3.2	Block diagram for the proposed arbitration mechanism	18
3.3	The variation of the latency counter values and their effect on arbitration switching with respect to $t_{i,j}^0, t_{i,j}^a, t_{i,j}^f$	22
3.4	Per-request and cumulative latency bounds under FCFS, RR, and DAMA when the PF is disabled	32
3.5	Per-request and cumulative latency bounds under DAMA when the PF is enabled	33
3.6	Normalized IPC for a prefetcher in a single core setup with varying buffer and cache sizes	34
3.7	Average IPC as a function of $\mathcal{S}_i$ for different arbitration models and configurations	36
3.8	Percentage of time spent in FCFS mode as a function of $\mathcal{S}_i$	36
3.9	Normalized IPC relative to percentage increment of Δ_i with $\mathcal{S}_i = 2.25 \times \mathcal{K} \times \Delta_i \times 10^{-6}$	36
3.10	Normalized IPC for a prefetcher in a single core setup with varying buffer and cache sizes	37

4.1	Normalized miss rate for duration-based varying the age's bitwidth; 2KB I-cache with $K = 8$ memory regions	47
4.2	Normalized miss rate and area overhead for duration-based varying K ; 2KB I-cache with 10 age bits	48
4.3	Normalized I-cache miss rate for dynamic, static, and duration-based locking; $K = 8, 10$ age bits	49

List of Tables

4.1	Overheads for dynamic and duration-based locking	50
-----	--	----

Chapter 1

Introduction

The increasing complexity of workloads with varying criticalities has driven the evolution of computing platforms, particularly in high-performance and real-time multi-core embedded systems. Such systems, often constrained by size, weight, power, and cost (SWaPC), are required to meet performance and predictability objectives. Specifically, real-time systems executing mixed criticality workloads face a fundamental tradeoff between improving the overall system performance and adhering to strict timing guarantees for high-criticality tasks. In such systems, memory resources constitute a major bottleneck, as shared memory hierarchies must balance the competing demands of high-throughput and low-latency access for diverse workloads. Hence, meeting the requirements of high-performance and real-time embedded systems necessitates innovative solutions to enhance the utilization and efficiency of memory resources.

Mixed criticality systems (MCS) have gained prominence with the advent of multicore architectures, as they enable running applications with varying criticalities on separate cores within the same platform. In MCS, applications are categorized depending on their timing requirements and importance to system operation. High-criticality applications, such as those in medical devices, automotive, and avionics [28], [4], [65], [12], [3], impose strict constraints on worst-case execution time (WCET) and demand deterministic timing guarantees to ensure safety and reliability. In contrast, medium to low criticality tasks, which often include non-safety-critical applications like multimedia processing, prioritize improving average system performance over meeting hard timing constraints. To efficiently manage the diverse requirements of mixed-criticality applications, multicore systems often leverage virtualization technologies. Virtual machines (VMs) are deployed across cores to provide task isolation, resource sharing, and fault containment across VMs [53]. These virtualized environments are managed by hypervisors such as Bao [53], Jailhouse [72], Xen

[33], KVM [44], and many more. These hypervisors ensure strong temporal and spatial isolation, which is essential for maintaining the safety and security of MCS.

A critical aspect of addressing the challenges of multicore architectures in MCS lies in memory arbitration. High-criticality tasks require stringent WCET guarantees. Such guarantees depend on the latency of memory requests issued by hardware components used to execute the tasks, e.g. requests generated by a core following L1 misses. We name the requests that affect the WCET of a high-criticality task as latency-critical requests (*LTC*). Additionally, hardware components issue other, non-latency-critical (*NLTC*) requests that affect the average execution time of both high-criticality and low-criticality tasks. Although adopting multicore architectures offers significant computational benefits, they also introduce substantial challenges from a real-time perspective. These challenges arise from two factors: resource contention between *LTC* and *NLTC* requests in multicore systems, and the unpredictable behavior of advanced features (e.g. prefetching and out-of-order execution). Such challenges complicate the system analysis and make it difficult to provide timing guarantees for *LTC* requests. As a result, a tradeoff arises between performance and predictability, where promptly servicing *NLTC* requests can improve the overall system performance, but can potentially violate the latency bounds of *LTC* requests.

1.1 Problem statement

Previous works that aim to provide timing guarantees for memory requests have tackled this tradeoff in mixed-criticality systems by regulating the bandwidth of the memory resource [85],[89], enforcing space isolation [43],[84], or by delivering differentiated services for requests based on criticality [29, 21, 37, 26, 30]. Prior work has also addressed this tradeoff at the memory arbiter level by introducing a dual arbitration scheme called Duetto [54, 55, 56]. This scheme utilizes a high-performance arbiter (HPA) to maximize throughput until a timing violation is detected by a monitoring mechanism that constantly checks the state of the memory resource. When a violation occurs, the system transitions to a real-time arbiter (RTA) to enforce a strict latency bound on a per-request basis. Although this scheme is provably safe, providing a bound on a per-request basis is complex: the approach relies on non-trivial mathematical operations to check the maximum request latency at runtime based on the state of the resource and request queues. Since this process must happen in hardware on a clock cycle basis, we argue that it can make it challenging both to meet critical path constraints and to verify the hardware.

Beyond memory arbitration, efficient cache management can help in further enhancing

the overall system performance. Mechanisms such as cache partitioning, locking, and coloring have been extensively explored to improve cache efficiency, either through software or hardware implementations [24]. Cache locking is one such mechanism that entails the preservation of critical blocks in the cache, thereby avoiding the latency of fetching them from lower levels in the memory hierarchy. Locking mechanisms have been implemented both in static [83, 5, 47, 68] and dynamic [1, 5, 19, 87, 88] configurations. Existing static and dynamic cache-locking methods face tradeoffs between flexibility and overhead. While static locking simplifies implementation by preloading critical blocks at program startup, it sacrifices adaptability to the changing program behavior during runtime. Conversely, dynamic locking offers enhanced flexibility but at the cost of program modifications that can complicate compilation flows and limit its applicability.

1.2 Thesis Contributions

This thesis explores mechanisms to improve the predictability and performance of multi-core embedded systems. It focuses on addressing key challenges in managing memory resources by proposing two solutions: one at the memory controller level and another at the cache controller level. Together, these contributions pave the way for more predictable and high-performance multi-core embedded systems, addressing the evolving challenges of modern workloads.

At the memory arbitration level, we propose DAMA, a novel dual arbitration mechanism. Our key insight is that the WCET guarantee for high-criticality tasks depends on the cumulative latency of *LTC* requests. Hence, instead of the per-request bound provided by Duetto [54], DAMA offers a bound on the cumulative request latency. This enables the use of a much simpler monitoring mechanism that tracks the cumulative latencies experienced by a requestor and triggers the arbiter to switch between high-performance and real-time modes when a timing violation is detected. In high-performance mode, all *LTC* and *NLTC* requests are given equal priority to maximize performance. In contrast, in real-time mode, only *LTC* requests are considered to maintain their latency bounds. More in details, our contributions are summarized as follows:

- We introduce and discuss the DAMA model. DAMA maintains a set of latency counters, which track the accumulated latency slack by each requestor at run-time in a simple and efficient manner. By bounding the maximum value of the counters, we formally show that DAMA provides a guarantee on the cumulative latency of *LTC* requests issued by a task.

- We show how DAMA can incorporate *NLTC* requests by integrating a predictable prefetcher. We maintain cache analyzability using a similar approach to [23], where a prefetch side buffer is employed to avoid altering the cache state. Contrary to [23], we consider a prefetcher attached to the L1 data cache, which is more efficient [57] but requires careful management of *LTC* and *NLTC* requests issued by the same core.
- We implement DAMA within the gem5 simulation environment and evaluate it by running different sets of real-world and synthetic benchmarks. We also compare our work against [23].

At the cache controller level, we propose a novel duration-based locking mechanism designed to emulate the dynamic approach without requiring any program code modification. Our objective in duration-based locking is not to outperform the dynamic locking approach, but rather achieve comparable results without altering the program code and with a minimal area overhead. More in details, our contributions are summarized as follows:

- We present a novel hardware mechanism that utilizes the age bits of the LRU replacement policy. This mechanism allows cache lines to be locked at runtime for a configurable duration.
- We propose a heuristic to leverage our designed hardware mechanism. The heuristic analyzes program loop structures to identify the cache lines to lock and determine their locking durations.
- We implement our duration-based mechanism within the gem5 simulation environment [49] based on a L1 instruction cache, and compare it against the dynamic approach.

1.3 Thesis Structure

Chapter 2 provides background and examines related work on bandwidth regulation mechanisms, predictable arbitration schemes, hardware and software prefetching techniques, and different flavors of cache-locking. Chapter 3 introduces the proposed dual arbitration scheme, DAMA, and evaluates its effectiveness when a predictable prefetcher is attached

to the L1 data caches of cores to serve as a source of *NLTC* requests. Chapter 4 introduces a novel duration-based locking approach, which leverages a profiling-based software heuristic to select the locked content and their locking durations for a program. Chapter 5 concludes this thesis, it discusses the limitations of the proposed work and provides an outlook for potential future works.

1.4 Acknowledgement

I would like to thank Professor Rodolfo Pellizzoni for his contributions to Chapter 3, Section 3.1.5, where he provided the formal mathematical proofs.

Chapter 2

Background and Related Work

Efficient memory resource management is essential for optimizing the performance of multi-core systems. Key factors impacting system performance include bandwidth allocation and arbitration schemes, particularly in the context of shared memory resources. In this thesis, we explore previously proposed real-time oriented mechanisms for bandwidth regulation in Section 2.1, and examine memory arbitration schemes designed for high-performance systems, real-time systems, and mixed-criticality systems in Section 2.2.

Another integral resource in modern systems is cache memories due to their significant impact on the overall performance of the system. A considerable body of research has been dedicated to exploring different hardware and software mechanisms aimed at optimizing the efficiency and utilization of cache memories. In this thesis, we focus on two mechanisms, namely, prefetching and cache locking. Sections 2.3 and 2.4 delve into related research that has been conducted on prefetchers and cache locking respectively.

2.1 Memory Bandwidth Regulation

Bandwidth, defined as the amount of data that can be transferred per unit time, plays a pivotal role in the performance of multicore systems. When multiple processing cores share a resource, simultaneous access requests can lead to contention and interference. This issue is exacerbated when a core executes a bandwidth-intensive workload, potentially saturating the shared resource and causing significant delays for other requestors. Such unfair access patterns can degrade system performance and even result in starvation for cores executing workloads that are not bandwidth-intensive. To address these challenges,

substantial research has been devoted to developing interrupt-based [85],[86] and polling-based [89] mechanisms for regulating memory bandwidth across cores. In [85],[86] Yun et al. introduced MemGuard, an interrupt-based regulation mechanism designed to guarantee a minimum bandwidth allocation for each core. Once the reserved bandwidth for all cores is satisfied, the mechanism employs a reclamation strategy to enhance system performance by utilizing any residual bandwidth left after the initial allocations. This approach has been explored at the OS level in [8] and extended to enable distinct regulation strategies for reads and writes on a per-core basis. To avoid modifying the host OS, it has also been integrated at the hypervisor level in [53], [59], [16]. In [89], Zuepke et. al. argue that interrupt-based mechanisms introduce significant overheads, resulting in a pessimistic worst-case bandwidth threshold and coarse regulation periods. To address this, they propose Mempo, a bandwidth regulation approach that utilizes polling of performance counters. The authors demonstrate that polling considerably reduces the overhead of interrupt-based methods and enables fine-grained regulation at a microsecond scale.

2.2 Predictable Arbitration Schemes

Multi-core systems comprise shared resources, such as the bus interconnect, the last level cache, and main memory, which are accessed by multiple requestors at the same time. In such resources, an arbiter is needed to manage shared access through an arbitration scheme. Such schemes significantly affect the latency of memory requests and the delays suffered by tasks. We categorize arbitration schemes into three primary classes: high-performance arbiters, real-time arbiters, and mixed-criticality arbiters. The following subsections provide a brief overview of each class.

High-performance systems are designed to handle diverse workloads efficiently while ensuring a high quality of service for all requestors. To maximize average performance, these systems rely on high-performance arbiters that implement policies like First Come First Served (FCFS) policy [35], parallelism-aware batch policy [63], and out-of-order policy [73], among others. These policies exploit specific characteristics of the issued requests to enhance throughput, thereby improving the system’s average performance. For example, as shown in Figure 2.1, the FCFS policy prioritizes requests based on arrival time, ensuring that older requests are always processed before newer ones. In [54], the authors show that under FCFS arbitration memory requests issued by real-world benchmarks can experience significant latency spikes, especially when faced with adversarial, bandwidth-intensive traffic. This stems from the fact that FCFS prioritizes requests based on their arrival time. Such prioritization can lead to severe performance degradation for real-world

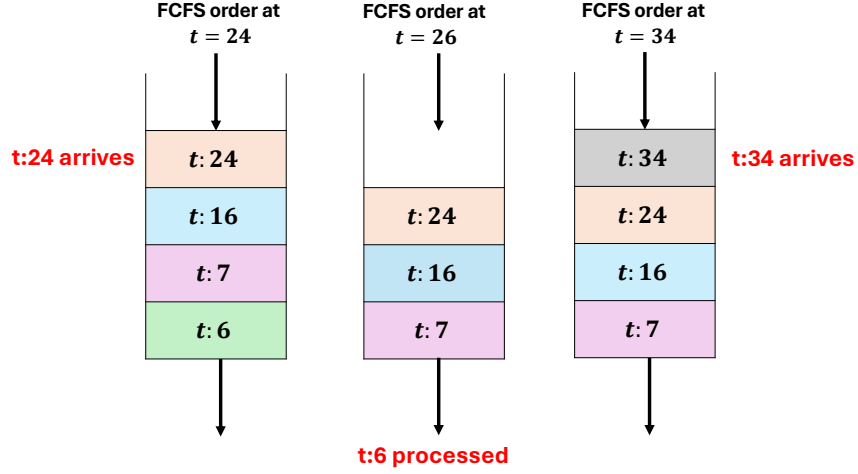


Figure 2.1: An example of how requests are handled under the FCFS arbitration scheme

benchmarks, as their requests may be arbitrarily delayed by the high volume of adversarial traffic. While these arbitration schemes significantly boost average performance, they do not guarantee a tight bound on the time required to process a single request, making them unsuitable for real-time systems.

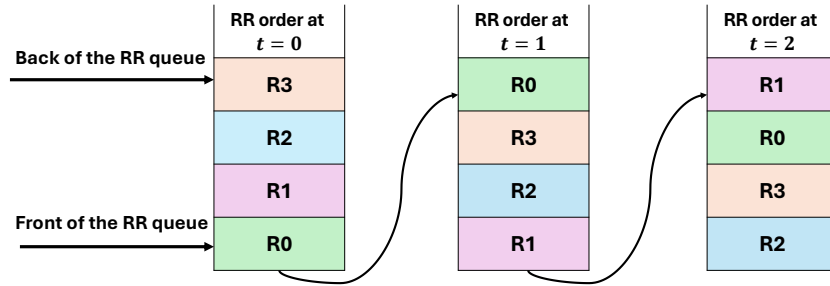


Figure 2.2: An example of how requests are handled under the RR arbitration scheme

In contrast, real-time systems prioritize worst-case execution time (WCET) over average performance. Therefore, it is essential to have an arbitration policy that ensures a tight bound on the processing latency of requests, as failing to do so would result in a highly pessimistic WCET estimate. Consequently, extensive research has focused on developing predictable arbitration policies, such as round-robin (RR) policy and its various adaptations [40], time division multiple access (TDMA) policy [32], and other similar approaches.

For example, RR arbitration in memory controllers ensures that each requestor is allocated a single slot to issue a memory request during its turn in a cyclic manner as illustrated in Figure 2.2. By rotating access among requestors, round robin guarantees that no requestor is starved, even under heavy contention. Similarly, the TDMA policy operates like RR but

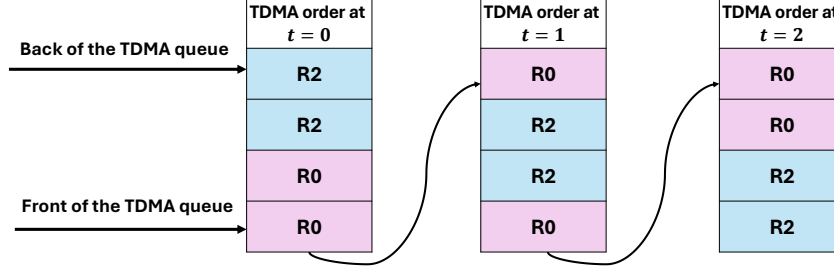


Figure 2.3: An example of how requests are handled under the TDMA arbitration scheme

allocates multiple slots to the same requestor, as illustrated in Figure 2.3. These arbiters are designed to ensure fairness and provide timing guarantees, making them well-suited for real-time applications. However, their rigid scheduling policy can result in significant overall performance degradation.

Many studies have addressed the tradeoff between predictability and average performance by handling requests with varying levels of criticality. A significant body of work has explored redesigning the memory controllers to support mixed-criticality systems, offering differentiated services to critical and non-critical requests [29, 21, 37, 26, 30]. In [29],[30] Hassan et. al. propose a framework that constructs optimal schedules for different tasks with varying criticalities. They also modified the memory controller to be programmable, allowing it to load the optimal schedules at boot time. The work most related to ours is the Duetto model [54], which combines the performance benefits of high-performance arbiters with bounded request latency. Duetto associates a relative deadline (maximum per-request latency) to each requestor and uses it to derive an absolute deadline for each request at runtime. It then uses a monitoring component to dynamically compute the worst-case finish time of the request, assuming that the system continues to operate in high-performance mode for the current clock cycle. If the computed finish time exceeds the deadline, Duetto switches to a real-time arbiter to preserve the per-request latency guarantee. Duetto is safe but relies on complex computations to enforce the per-request bounds, which complicates meeting timing constraints and verifying the hardware. This arbitration scheme has been applied to a banked SRAM model in [54], a DRAM model in [55], and to a coherent shared cache in [56].

2.3 Prefetchers

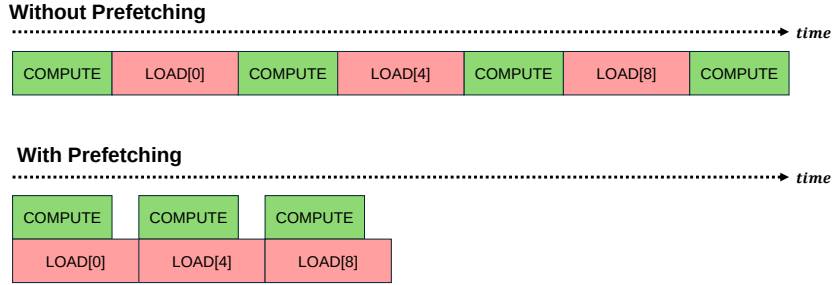


Figure 2.4: An example demonstrating the impact of the prefetcher on the compute and memory load phases of a program

Prefetching is a commonly used technique that hides memory latency by anticipating future memory accesses and proactively loading data into the cache before the processor requests it. Figure 2.4 illustrates how in an in-order processor, computation phases are often significantly stalled by load requests. When a prefetcher is enabled, these load requests can be overlapped with computation phases, ensuring that data is available in the cache when needed during compute stages. This has proven to be an effective technique that reduces the likelihood of stalling the core by reducing the number of cache misses, thereby improving the system’s overall performance.

Despite the effectiveness of this method, determining precisely when to prefetch the data is a very challenging task [22]. Prefetching too early, even when the prefetched data is accurate, can cause useful data to be evicted from the cache before it gets accessed. This could potentially waste memory bandwidth and cause cache pollution [57]. Conversely, prefetching too late fails to hide memory latency, diminishing the potential benefits of overlap between computation and memory access. Another major challenge in prefetching is identifying access patterns and selecting data that will be useful to the program.

In real-time embedded systems, these challenges are further exacerbated by the inherent complexity of prefetchers, which makes their behavior difficult to analyze. This often results in their exclusion from analysis, leading to overly pessimistic execution time bounds for *LTC* requests. This thesis addresses the two primary factors contributing to this limitation:

- Prefetchers can **induce memory reordering**, potentially leading to pathological scenarios where *LTC* requests are delayed by prefetch requests fetching non-essential data. We address this limitation in Chapter 3.1.

- A prefetcher can **alter the state of cache lines** by replacing existing blocks with prefetched blocks, or by modifying the ages of cache lines within a particular associative set. We address this limitation in Chapter 3.2.

For such reasons, disabling hardware prefetchers is a common practice in real-time systems[25]. The following subsections examine prior work on software and hardware prefetching approaches that aim to address these challenges.

2.3.1 Software Prefetching Approaches

Software prefetching involves techniques where the compiler or programmer directly inserts prefetch instructions into the code, bringing data into the cache before it is required. Unlike hardware prefetching, which relies on hardware to predict future memory accesses, software prefetching offers developers greater control, allowing them to tailor prefetching strategies to specific applications and workloads. A common software approach is compiler-assisted prefetching [61],[50],[14]. The compiler may analyze the code to pinpoint optimal prefetch points and insert prefetch instructions, or it can apply transformations that make the code more prefetch-friendly. For example, in [61], Mowry et. al. proposed a compiler algorithm that examines loops to predict which cache lines will be accessed in upcoming iterations. The algorithm modifies the code to insert prefetch instructions at the loop’s entry, loading data into the cache in advance. In [50], Luk et. al. propose a compiler-based prefetching approach for pointer-based recursive data structures. The compiler analyzes the program to identify points where pointers are recurrently updated during traversals and inserts prefetch instructions accordingly.

Researchers in the real-time community have also proposed various software-based prefetching approaches that mitigate their unpredictability and capitalize on their performance gains. The work in [67, 51, 20] addresses this by introducing a software model called PREM that segments tasks into computational and memory phases, with data and code prefetched into the cache at the start of each memory phase. While the original PREM model in [67] requires manual modifications to the program, Mancuso et al. [51] extend this model by proposing a framework that automates the refactoring process needed to convert legacy software into PREM-compliant code. Other software-based approaches have investigated prefetching data into a scratchpad memory rather than the cache in a predictable manner [6, 76, 81, 75, 18]. In these approaches, data is loaded into the scratchpad memory either statically at compilation time [6, 76] or dynamically during runtime [81, 75, 18]. For instance, in [75], Soliman et al. propose a compiler-directed prefetching mechanism

designed to optimize the WCET of a program. This approach involves traversing the program’s CFG and inserting prefetch instructions that load data into the SPM memory at the beginning of program regions. A genetic algorithm is employed to determine the optimal placement of these instructions, ensuring the WCET is minimized. Software-based approaches, while effective, often require modifying the code and potentially the compiler to incorporate prefetching instructions or directives.

2.3.2 Hardware Prefetching Approaches

Unlike software prefetching, hardware prefetching autonomously predicts and fetches data using microarchitectural components, avoiding program or compiler modifications. Among the most widely adopted hardware-based prefetching techniques in systems are sequential [17] and stride [7] prefetchers. Sequential prefetchers issue fetch requests for consecutive memory addresses, making them ideal for applications with linear memory access patterns. On the other hand, stride prefetchers can identify recurring patterns for accesses that are not necessarily separated with constant strides. These prefetchers are particularly effective for dense matrix operations and multi-dimensional array traversals [22]. Conventionally, stride prefetchers place prefetched blocks directly into the cache, which can lead to cache pollution by displacing useful datasets especially if the prefetcher is very aggressive. To that end, Joupi et al. [39] proposed placing prefetched blocks into a separate buffer, known as a stream buffer, and migrating the blocks to the cache only when requested by the core. The paper also explores the use of multiple stream buffers, each prefetching with a different stride, which proves highly effective for accessing multiple arrays within loops [57].

Stride prefetchers are not well-suited for irregular data access patterns, such as those found in pointer-based data structures, where memory accesses are not easily predictable [22]. In these structures, memory addresses are determined by pointer values that can change at runtime, making accurate future memory access prediction challenging. To address this, numerous studies have explored address-correlating prefetchers [13, 15] designed to handle pointer-chasing access patterns. A notable example is the Markov prefetcher [38] which predicts future memory accesses by modeling the probability of transitions between states, where each state represents a sequence of recent memory accesses. Other more advanced prefetchers, such as those proposed in [10, 9], utilize low-cost perceptron networks to predict future memory access patterns.

Previous work proposing hardware-oriented predictable prefetching solutions is scarcer. The most relevant solution is [23], whose authors propose a WCET-preserving hardware prefetcher for many-core systems. Their solution involves a memory-level prefetcher con-

nected to a predictable interconnect with a fixed-priority scheme, which bounds the blocking periods for high and low-priority packets. However, as noted in [57], attaching a prefetcher at the memory side may not always be optimal, as the memory can only detect misses and lacks accurate knowledge of the core’s reference patterns and whether the prefetched blocks are hitting or missing in the cache. In their paper, they utilize idle system slots to issue prefetch requests and employ a variant of the stream buffer to temporarily store prefetched blocks. These blocks are only moved to the cache upon demand, helping to preserve the cache state and maintain its analyzability.

2.4 Cache Locking

Cache locking is a mechanism that entails the preservation of critical blocks in the cache, thereby avoiding the latency of fetching them from lower levels in the memory hierarchy. Cache locking has been implemented in a variety of commercial embedded processors (e.g., ARM Cortex A8, IBM PowerPC [58]), whereby a set of cache lines can be locked by the software using special locking instructions. To exploit this feature, much work has been done on exploring various algorithmic approaches to identify the most suitable set of cache lines for locking [58]. Prior works have employed cache-locking strategies for optimizing worst-case execution time (WCET) [19, 87, 88, 70, 68], while others have considered it for improving the average program performance [1, 83, 5, 47]. Locking mechanisms have been implemented both in static [83, 5, 47, 68] and dynamic [1, 5, 19, 87, 88] configurations. Static locking entails loading and locking memory blocks at program startup without subsequent modification throughout the entire program execution. Conversely, dynamic locking allows for modifying the cache’s locked content during run-time by adding locking and unlocking instructions at different locations in the program code. According to [19], the dynamic locking method exhibited a performance enhancement of up to 40% when compared with static locking.

Dynamic locking requires identifying specific points in the program where locking and unlocking instructions must be added. In previous works, various program structures such as single-entry-single-exit regions [70], loop structures [19], and persistent program phases [1] were considered to identify those points. While WCET approaches rely on static analysis to assess the costs and benefits of performing cache locking, other average case approaches [83, 5] rely on iterative analysis methods that invoke a cache simulator whenever the cost and benefit for a given locking configuration need to be assessed. In [47] Liang et. al. argue that relying on an iteratively invoked cache simulator to run the analysis constrains the approach’s scalability as the size of the program increases. To address this they proposed

a single-pass heuristic that utilizes the temporal reuse profile metric to pick the cache lines to lock. In [83], Xue et. al. argue that applying data cache locking for enhancing average case performances yields marginal benefits due to the inherent complex access patterns of memory blocks in data caches. To address this limitation, they proposed a compiler-based data reallocation scheme that makes the memory map more locking-friendly.

Much work has been done on exploiting the cache line’s recency information to prioritize or de-prioritize their eviction at run-time. In [36], Jaleel et. al. introduced a metric termed re-reference predicted value (RRPV), which determines the re-reference distance for each cache line, allowing for the prioritization of evictions based on these distances. Expanding on this idea, J. Wu et al. in [80] implemented a learning mechanism that uses a history table to precisely predict RRPV for each cache line. Similarly, in [41] Kharbuti et. al., employed a set of counters and a large prediction table to learn and predict instances where cache lines that are no longer in use can be evicted, thereby freeing up cache space. The previously mentioned methods incur a huge area overhead to perform online learning and prediction (1.8KB in [80] and 40 KB in [41]).

Chapter 3

DAMA: A Dual Arbitration Mechanism for Mixed-Criticality Applications

In this Chapter, we introduce DAMA, a dual arbitration mechanism that imposes a deterministic upper bound on the cumulative latency of *LTC* requests while optimizing the average performance of both *LTC* and non-latency-critical (*NLTC*) requests (Section 3.1). We attach a predictable prefetcher to our L1 data caches as a source of *NLTC* requests and leverage the properties of DAMA to further enhance the average performance of the system without impacting the bounds (Section 3.2). We then assess the effectiveness of our proposed solution by running synthetic and real-world benchmarks under various setups (Section 3.3).

3.1 DAMA: The Proposed Solution

We begin by describing the system model and underlying assumptions (Section 3.1.1). Following this, we outline the high-level architecture of the arbitration mechanism and explain the roles of its main modules (Section 3.1.2). We then provide a detailed description of how the entire regulation mechanism operates (Section 3.1.3). Finally, we present an illustrative example (Section 3.1.4) and formally discuss provided latency guarantees (Section 3.1.5).

3.1.1 System Model and Assumptions

We consider a system comprising M distinct requestors $\{R_1, \dots, R_M\}$, which may include cores, DMA engines, accelerators, bus masters, and more. We assume that a requestor R_i can either issue *LTC* requests, *NLTC* requests, or both, targeting a shared resource. For the sake of simplicity, we present our solution without considering data sharing between requestors. We denote $r_{i,j}$ to represent *LTC* requests, originating from requestor R_i and indexed by j (e.g. $r_{i,1}, r_{i,2}, \dots, r_{i,j}$) according to their arrival time $t_{i,j}^a$ at the resource. We assume that a request is enqueued at the shared resource upon its arrival, and remains outstanding in the queue until it finishes being serviced; we denote $t_{i,j}^f$ the finishing time for *LTC* request $r_{i,j}$. We consider a discrete model of time, expressed in multiples of the clock period at the resource. We assume that it takes at least one clock cycle to service a request, hence $t_{i,j}^f > t_{i,j}^a$, and that no more than one request can finish in any given clock cycle. We further assume that requests can be serviced in an out-of-order fashion and that a single requestor can have multiple outstanding requests. To precisely define the latency for *LTC*, we lay out the following definitions, similar to previous work [54, 55, 56]:

Definition 1. *At clock cycle t , the oldest *LTC* request of R_i (if any) is the earliest arrived request of R_i that is outstanding. We define $t_{i,j}^o$ as the time at which a request $r_{i,j}$ becomes oldest.*

Given that in our model requests do not need to be serviced in the same order they arrive, a later arriving request can finish before an earlier arriving request. In this case, the later arriving request never becomes oldest. Note that if a request $r_{i,j}$ does become oldest, then $t_{i,j}^o$ is equal to either the time $t_{i,\underline{j}}^f$ at which a previously arrived request $r_{i,\underline{j}}$ finishes, or to $t_{i,j}^a$, in case all previous *LTC* requests of R_i have already finished by that time. Since at most one request can finish in a clock cycle, two requests of R_i cannot become oldest at the same time.

Definition 2. *We define $\delta_{i,j}$ as the queuing latency and $\beta_{i,j}$ as the processing latency of a request $r_{i,j}$.*

- *If $r_{i,j}$ is serviced before becoming the oldest request: then $\delta_{i,j}$ is equal to $t_{i,j}^f - t_{i,j}^a$ and $\beta_{i,j}$ to 0.*
- *If $r_{i,j}$ becomes the oldest request before finishing being serviced: then $\delta_{i,j}$ is equal to $t_{i,j}^o - t_{i,j}^a$ and $\beta_{i,j}$ to $t_{i,j}^f - t_{i,j}^o$.*

Figure 3.1 depicts various scenarios for the processing and queueing latencies of 4 consecutive *LTC* requests. In this example, we assume that initially, no request of R_i is outstanding. This means that all previous requests issued by R_i have finished, and thus $r_{i,j}$ becomes oldest as soon as it arrives. For that reason, the queueing latency for request $r_{i,j}$ is equal to zero. As outlined in Definition 2, and illustrated in Figure 3.1, our processing latency is the time the request spends in the queue after it becomes the oldest. Given that in our model requests do not need to be serviced in the same way they arrive, their processing time can be completely overlapped. Request $r_{i,j+2}$ exhibits this behavior by completing prior to $r_{i,j+1}$, thereby incurring a processing latency of 0.

As discussed in much related work [42, 31, 54, 55, 56], we argue that the delay suffered by a real-time task under analysis depends on the processing latency of its requests. Specifically, a core can stall while waiting for its demand requests to finish. If we consider demand requests from the core to be *LTC*, then whenever the core issues multiple *LTC* requests and stalls waiting for them to finish, the stall time is equal to the sum of their processing latencies. Therefore, we assume that the WCET of the task can be obtained by summing a base execution time (which might have to account for the effect of processor timing anomalies) plus an upper bound to the cumulative processing time of *LTC* requests. A final concern is related to the ordering of requests issued by the core. To prevent requests issued by the OS or other tasks to be serviced before earlier requests by the task under analysis, we assume that a memory barrier is inserted whenever the task is preempted and upon its completion.

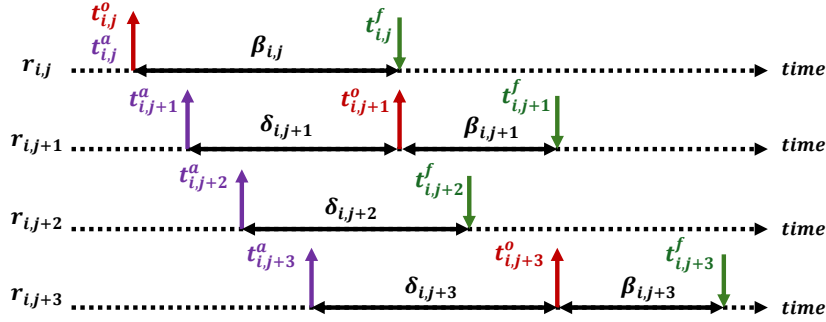


Figure 3.1: Processing and Queueing latencies under various scenarios

3.1.2 High Level Architecture

Queuing Structure

Our arbitration scheme features two queues for each requestor: $\mathcal{L}_i$ that handles *LTC* requests, and $\mathcal{N}_i$ that handles *NLTC* requests. This separation of queues ensures that the guarantees for *LTC* requests are maintained, preventing *NLTC* requests from compromising them.

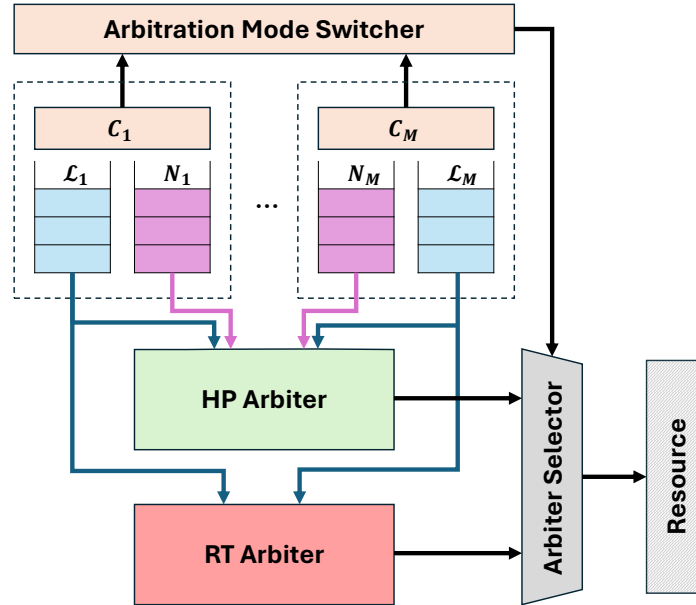


Figure 3.2: Block diagram for the proposed arbitration mechanism

High Performance and Real-Time Arbiters

DAMA comprises two arbiters: a high-performance arbiter (HPA) which is optimal for maximizing average performances and a real-time arbiter (RTA) which provides tight latency bounds. The HPA services both *LTC* and *NLTC* requests flowing from the respective queues $\mathcal{L}_i$ and $\mathcal{N}_i$, as illustrated in Figure 3.2. When operating in high-performance mode, both types of requests are given equal priorities, and the arbitration mechanism dictates the order in which they get serviced. For instance, if the HPA operates on a first-come first-served basis (FCFS), it services requests based on their arrival time. On the

contrary, when operating in real-time mode, RTA only services *LTC* requests originating from $\mathcal{L}_i$. The rationale behind blocking out *NLTC* requests in RTA is to provide a tight bound on the cumulative latency of requests while minimizing the time spent in this mode. Specifically, the RTA must provide a latency bounds of $\mathcal{B}_i > 0$ to oldest requests, in the sense that upon switching from HPA to RTA, regardless of the resource's state, the oldest *LTC* request $r_{i,j}$ of R_i residing in $\mathcal{L}_i$ must finish in at most $\mathcal{B}_i$ clock cycles.

Latency Counters and Arbitration Mode Switcher

As part of the system configuration, each requestor is assigned a value Δ_i , which represents the target average per-request latency. In Section 3.1.5, we prove that the bound on the cumulative processing latency of *LTC* requests is proportional to the total number of requests and Δ_i . For each requestor, at run-time we maintain a counter $\mathcal{C}_i$ that tracks the *slack* - that is, the difference between Δ_i and the actual latency - accumulated by *LTC* requests issued by requester i . To prevent the latency counters from potentially growing indefinitely in the negative range, we impose the constraint $\Delta_i \geq \mathcal{B}_i$. The Arbitration Mode Switcher continuously monitors the value of the counters and forces the system to operate in RTA mode whenever the value of any counter is less than or equal to zero. At the start of the system ($t = 0$), each requestor's latency counter $\mathcal{C}_i$ is initialized to a maximum slack value $\mathcal{S}_i \geq 0$ as described in Algorithm 1 (lines 5-8). Intuitively, the higher the value of the initial slack and the higher the value of Δ_i , the longer the system can remain in HPA mode, albeit at the cost of a larger latency bound for *LTC* requests; we explore such tradeoff in Section 3.3.

3.1.3 Latency Regulation Mechanism

In this subsection, we describe the latency regulation mechanism outlined in Algorithm 1. After performing the system initialization at $t = 0$, the memory controller conducts a series of checks on a per-cycle basis. At every cycle, the queues $\{\mathcal{L}_1, \dots, \mathcal{L}_M\}$ are checked to determine whether outstanding requests are still waiting to be serviced (line 12). If this is the case, the counter $\mathcal{C}_i$ of requestor i is decremented by 1, indicating the passage of time and the consumption of 1 cycle from its latency budget (line 13). It is important to note that the latency counters are only updated when there are outstanding *LTC* requests residing in $\{\mathcal{L}_1, \dots, \mathcal{L}_M\}$ and when these requests are serviced. This is because our mechanism focuses solely on regulating the latency of *LTC* requests, with no consideration for regulating *NLTC* requests. Let $\Theta(\mathcal{L}_i, t)$ be a function that returns a request $r_{i,j}$ that finishes at time t . If such a request does not exist, the function returns NULL. At every

cycle, the memory controller checks whether any of the oldest requests residing in queues $\{\mathcal{L}_1, \dots, \mathcal{L}_M\}$ have been serviced (line 15). If such a request exists, the latency counter $\mathcal{C}_i$ for the corresponding requestor i is incremented by the minimum between the maximum slack value $\mathcal{S}_i$, and the sum of the current counter value $\mathcal{C}_i$ with Δ_i (line 16). The purpose of this minimum operation is to bound the accumulated counter value, ensuring it does not exceed $\mathcal{S}_i$. After updating the value of the counters, we push the newly arrived requests into their respective queues and pop the ones that finished (line 18). At the end of each processing round, the status of the latency counters is checked. If any counter $\mathcal{C}_i \in \{\mathcal{C}_1, \dots, \mathcal{C}_M\}$ is less than or equal to zero (line 20), this indicates that requester i has exhausted its allocated latency budget, and the arbiter executes in real-time mode (line 21). Transitioning to the RTA when any of the counters gets depleted is essential as this mode imposes a bound $\mathcal{B}_i$ on the maximum service time of a request issued by requestor R_i . Conversely, if all counters return to values above zero, the arbiter transitions back to high-performance mode (line 23).

3.1.4 Illustrative Example

Figure 3.3 provides an illustrative example of how the counter value changes throughout the program's execution, demonstrating how switches are performed accordingly. For this example, we assume $\Delta_i = 3$, $\mathcal{S}_i = 4$, and $\mathcal{B}_i = 3$. The counter begins with a value initialized to $\mathcal{S}_i$, allowing the arbiter to operate in HPA mode, as the counter value is positive. At $t = 1$ the first memory request from requestor 1 becomes the oldest. As it can be seen in the figure, the arrival time $t_{i,j}^a$ overlaps $t_{i,j}^o$ the time the request becomes oldest. This indicates that queue $\mathcal{L}_1$ was empty and did not have any outstanding requests when $r_{1,1}$ arrived, which makes queuing time for $r_{i,j}$ equal to zero. After the request becomes oldest the counter starts decrementing by 1 with each cycle until the request gets serviced. At $t = 3$, request $r_{1,1}$ completes, prompting the controller to add Δ_i to the counter. Although the counter would have reached 5 at $t = 3$, it is capped at $\mathcal{S}_i$ to keep the value bounded. At $t = 4$ and $t = 11$ two new requests $r_{1,2}$ and $r_{1,3}$ become the oldest and the same behavior with the counters repeats. Notice that $r_{1,3}$ arrives at $t = 6$ but does not become the oldest until $t = 11$ since at $t = 6$ $r_{1,2}$ is the oldest outstanding request that is still waiting to get serviced. At $t = 8$ the counter reaches 0, prompting the controller to switch to RTA. In the worst-case scenario, the oldest request will spend at most $\mathcal{B}$ cycles in RTA mode. The constraint $\Delta \geq \mathcal{B}$ ensures that when a request $r_{i,j}$ finishes, its counter $\mathcal{C}_i$ will be greater than or equal to zero, as demonstrated at cycles 11 to 13.

Algorithm 1 Latency Regulation Mechanism

```
1:  $M$ : Total number of requestors
2:  $\mathcal{S}_i$ : Maximum slack value for requestor  $i$ 
3:  $\mathcal{C}_i$ : Latency counter for requestor  $i$ 
4:  $\mathcal{L}_i$ : Queue that stores LTC requests for requestor  $i$ 
5:  $\mathcal{N}_i$ : Queue that stores NLTC requests for requestor  $i$ 
6: if  $t == 0$  then ▷ When the system starts
7:   for  $i \leftarrow 0$  to  $M$  do
8:      $\mathcal{C}_i = \mathcal{S}_i$  ▷ Initialize counters with  $\mathcal{S}$ 
9:   end for
10: else
11:   for  $i \leftarrow 0$  to  $M$  do
12:     if  $len(\mathcal{L}_i) \neq 0$  then ▷ Queue is not empty
13:        $\mathcal{C}_i = \mathcal{C}_i - 1$  ▷ Decrement counter
14:     end if
15:     if  $\Theta(\mathcal{L}_i, t) \neq NULL$  then
16:        $\mathcal{C}_i = \min(\mathcal{S}_i, \mathcal{C}_i + \Delta_i)$ 
17:     end if
18:      $updateQueues(\mathcal{L}_i, \mathcal{N}_i)$ 
19:   end for
20:   if  $|\mathcal{C}_i \leq 0 \forall i < M| > 0$  then ▷ A counter equals 0
21:      $MODE(RTA)$  ▷ Execute in real-time mode
22:   else ▷ All counters are greater than 0
23:      $MODE(HPA)$  ▷ Execute in high-perf mode
24:   end if
25: end if
```

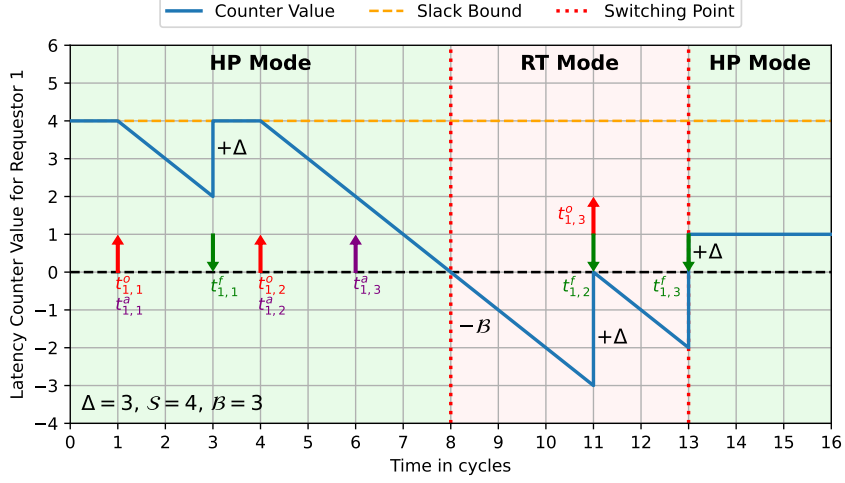


Figure 3.3: The variation of the latency counter values and their effect on arbitration switching with respect to $t_{i,j}^o, t_{i,j}^a, t_{i,j}^f$

3.1.5 Latency Guarantees

We next formally prove the timing guarantee provided by DAMA. In what follows, let $\mathcal{C}_i(t)$ to denote the value of counter $\mathcal{C}_i$ at clock cycle t (after executing Algorithm 1 for that clock cycle). Also, let $\text{next}(j)$ to denote the index of the first *LTC* request of R_i after $r_{i,j}$ that becomes oldest. First, in Lemmas 1 and 2, we constrain the value of the counter whenever a *LTC* request becomes oldest. Then, Theorem 1 uses the lemmas to compute a latency bound for a sequence of *LTC* requests. Finally, Theorem 2 extends the bound to a task under analysis considering the effects of preemptions.

Lemma 1. *For all i, j such that $r_{i,j}$ becomes oldest at $t_{i,j}^o$: $0 \leq \mathcal{C}_i(t_{i,j}^o) \leq \mathcal{S}_i$.*

Proof. By induction on the index j of *LTC* requests of R_i that become oldest.

Base case: since there are no previous *LTC* requests, the first request $r_{i,1}$ must become oldest when it arrives. By line 8 of Algorithm 1, $\mathcal{C}_i$ is initialized to $\mathcal{S}_i$ when the system starts, and furthermore, $\mathcal{C}_i$ cannot be decremented until the clock cycle after the first request arrives (line 13). Hence: $\mathcal{C}_i(t_{i,1}^o) = \mathcal{S}_i$, and since we assume $\mathcal{S}_i \geq 0$, the base case holds.

Induction case: assuming that $0 \leq \mathcal{C}_i(t_{i,j}^o) \leq \mathcal{S}_i$ holds, we need to prove that $0 \leq \mathcal{C}_i(t_{i,\text{next}(j)}^o) \leq \mathcal{S}_i$ also holds. Note that based on Algorithm 1, $\mathcal{C}_i$ can only be incremented

at line 16, where the current counter value is incremented by Δ_i ; however, at the same line, the counter cannot be set to a value greater than $\mathcal{S}_i$. Hence, from $\mathcal{C}_i(t_{i,j}^o) \leq \mathcal{S}_i$, it immediately follows that it must also hold $\mathcal{C}_i(t_{i,\text{next}(j)}^o) \leq \mathcal{S}_i$.

Next, consider the value of $\mathcal{C}_i$ computed by the algorithm at time $t_{i,j}^f$ after subtracting 1 at line 13 (as $r_{i,j}$ itself is in the queue) but before increasing it at line 16. If such value is greater than or equal to 0, from $\Delta_i \geq \mathcal{B}_i > 0$ it follows: $t_{i,j}^f > 0$. Otherwise, from $\mathcal{C}_i(t_{i,j}^o) \geq 0$ and the fact that the counter is decremented by at most 1 in a cycle, there must exist a previous cycle $t_0 \in [t_{i,j}^o, t_{i,j}^f)$ such that $\mathcal{C}_i(t_0) = 0$ and $\mathcal{C}_i(t) \leq 0$ for all $t \in [t_0, t_{i,j}^f)$. Because arbitration switches to RTA when the counter reaches 0 and from the definition of $\mathcal{B}_i$, it follows that $t_{i,j}^f - t_0 \leq \mathcal{B}_i$. Therefore, the value of $\mathcal{C}_i$ at $t_{i,j}^f$ after line 13 cannot be smaller than $-\mathcal{B}_i$, and since $\Delta_i \geq \mathcal{B}_i$ and $\mathcal{S}_i \geq 0$, after line 16, it must be $\mathcal{C}_i(t_{i,j}^f) \geq 0$.

In summary, in any case it holds: $\mathcal{C}_i(t_{i,j}^f) \geq 0$. Finally, note that either $r_{i,\text{next}(j)}$ arrives at or before $t_{i,j}^f$, and then $t_{i,\text{next}(j)}^o = t_{i,j}^f$; or $t_{i,\text{next}(j)}^a = t_{i,\text{next}(j)}^o > t_{i,j}^f$, in which case no *LTC* request of R_i can be present in the queue at line 13 in $(t_{i,j}^f, t_{i,\text{next}(j)}^a]$, and thus the counter value is not modified. In either case, we obtain $\mathcal{C}_i(t_{i,\text{next}(j)}^o) = \mathcal{C}_i(t_{i,j}^f) \geq 0$, concluding the proof. $\square$

Lemma 2. *For all i, j such that $r_{i,j}$ becomes oldest at $t_{i,j}^o$:*

$$\mathcal{C}_i(t_{i,\text{next}(j)}^o) \leq \mathcal{C}_i(t_{i,j}^o) + F_{i,j} \cdot \Delta_i - \beta_{i,j}, \quad (3.1)$$

where $F_{i,j}$ is the number of *LTC* requests of R_i (including $r_{i,j}$) that finish in $(t_{i,j}^o, t_{i,\text{next}(j)}^o]$.

Proof. Based on line 16 of Algorithm 1 and definition of $F_{i,j}$, counter $\mathcal{C}_i$ is incremented by at most $F_{i,j} \cdot \Delta_i$ in $(t_{i,j}^o, t_{i,\text{next}(j)}^o]$. Furthermore, note that the number of clock cycles in the same interval when there is a request in the queue at line 13 is equal to $t_{i,j}^f - t_{i,j}^o = \beta_{i,j}$; hence, the counter is decremented by exactly $\beta_{i,j}$. The lemma follows. $\square$

Theorem 1. *Let $1 \leq f \leq l$ and assume that no *LTC* request of R_i with index greater than l finish before requests with index less than or equal to l . Then:*

$$\sum_{j=f}^l \beta_{i,j} \leq \mathcal{S}_i + (l - f + 1) \cdot \Delta_i \quad (3.2)$$

Proof. Let $\bar{f}$ and $\bar{l}$ to be the first and last oldest *LTC* request of R_i such that $f \leq \bar{f} \leq \bar{l} \leq l$. If neither exist, then no request $r_{i,j}$ with $j \in [f, l]$ becomes oldest: therefore, $\sum_{j=f}^l \beta_{i,j} = 0$

and the theorem trivially holds. Otherwise, both must exist (note it may be $\bar{f} = \underline{l}$) and furthermore, $\sum_{j=f}^l \beta_{i,j} = \sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} \beta_{i,j}$.

Next note that since $r_{i,\bar{f}}$ is oldest, by definition no request $r_{i,j}$ with $j < \bar{f}$ can finish after $t_{i,\bar{f}}^o$. By assumption, no request $r_{i,j}$ with $j > l$ can finish before $t_{i,\underline{l}}^f$. Since $r_{i,\text{next}(\underline{l})}$ becomes oldest either at $t_{i,\underline{l}}^f$, or when it arrives - in case there is no outstanding *LTC* request of R_i after $t_{i,\underline{l}}^f$ - it follows that no request $r_{i,j}$ with $j > l$ can finish before $t_{i,\text{next}(\underline{l})}^o$ either. Therefore, a request $r_{i,j}$ can complete in the interval $(t_{i,\bar{f}}^o, t_{i,\text{next}(\underline{l})}^o]$ only if $\bar{f} \leq j \leq l$, yielding:

$$\sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} F_{i,j} \leq l - \bar{f} + 1 \leq l - f + 1. \quad (3.3)$$

Next, by summing Equation 3.1 over all oldest requests $r_{i,j}$ with $\bar{f} \leq j \leq \underline{l}$ and simplifying we obtain:

$$\begin{aligned} \sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} \beta_{i,j} &\leq \mathcal{C}_i(t_{i,\bar{f}}^o) - \mathcal{C}_i(t_{i,\text{next}(\underline{l})}^o) \\ &+ \sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} F_{i,j} \cdot \Delta_i. \end{aligned} \quad (3.4)$$

Using Lemma 1 to upper / under bound $\mathcal{C}_i(t_{i,\bar{f}}^o)$ and $\mathcal{C}_i(t_{i,\text{next}(\underline{l})}^o)$ and Equation 3.3 to upper bound $\sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} F_{i,j}$, we obtain:

$$\sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} \beta_{i,j} \leq \mathcal{S}_i - 0 + (l - f + 1) \cdot \Delta_i, \quad (3.5)$$

and since $\sum_{j=f}^l \beta_{i,j} = \sum_{j=\bar{f} \dots \underline{l}, r_{i,j} \text{ is oldest}} \beta_{i,j}$, the theorem holds. $\square$

Theorem 2. *Consider a job of a task under analysis performing $\mathcal{K}$ demand requests and suffering CS context switches. Assume that every time the task is preempted and when it finishes executing, a barrier is inserted to prevent successive demand requests of the same core to be serviced before earlier requests of the task. Then the cumulative processing latency of the $\mathcal{K}$ demand requests of the job is bounded by $CS \cdot \mathcal{S}_i + \mathcal{K} \cdot \Delta_i$.*

Proof. Note that the number of context switches is equal to one (when the job starts) plus the number of times it is preempted. Since the task suffers CS context switches, it must issue *LTC* requests with indexes in CS intervals $[f_1, l_1], \dots, [f_{CS}, l_{CS}]$, such that the sum of

the requests in all intervals is equal to $\mathcal{K}$. Based on the barrier assumption, no request with index greater than l_1 can finish before a request with index in $[f_1, l_1]$; the same holds for the other intervals. Hence, we can apply Theorem 1 to each interval. Summing Equation 3.2 over the CS intervals yields a cumulative latency bound over all LTC requests of the task:

$$\begin{aligned} & S_i + (l_1 - f_1 + 1) \cdot \Delta_i + \dots \\ & + S_i + (l_{CS} - f_{CS} + 1) \cdot \Delta_i = CS \cdot S_i + \mathcal{K} \cdot \Delta_i. \end{aligned} \quad (3.6)$$

□

3.2 Predictable Prefetcher

In this section, we describe a predictable prefetcher that does not compromise the cumulative processing latency bound of LTC requests described in Section 3.1.5 and preserves the WCET analyzability of the cache. As discussed in Section 3.1, we consider demand requests from the core to be LTC ; while prefetch requests from the predictable prefetcher are $NLTC$.

3.2.1 Ensuring Predictability with a Prefetch Side Buffer

In this model, Our goal is to implement a predictable prefetcher that does not alter the state of cache lines, preserves the WCET analyzability of the cache, and maintains the cumulative processing latency bound for LTC requests. Conventionally, prefetchers place data fetched from main memory into the cache, thus altering the state of its cache lines. This alteration can involve replacing existing blocks with prefetched data or updating the ages of cache lines within a particular associative set. Incorporating this behavior into static WCET analysis is generally avoided due to its complexity. In our approach, we maintain WCET analyzability of the cache by using a prefetch side buffer, a concept first introduced in [39] and later adapted for real-time systems in [23]. This side buffer prevents cache pollution and avoids modifying the cache state by storing prefetched blocks separately and transferring them to the cache only upon a demand hit. However, it is important to note that although a prefetch side buffer prevents changes to the cache line state, it does impact the time needed to service a miss request. This is because, before introducing the prefetch side buffer, servicing a cache miss required accessing the main memory. However, with the side buffer, servicing a cache miss can either be handled by fetching the data from the main memory or by accessing the side buffer, which takes less time. Due to this difference

in the time required to handle a cache miss, any static analysis tool used to determine the state of cache lines (e.g., always hit, always miss, etc.) must be able to account for these timing variabilities.

Note that whenever the core misses in the L1 data cache, we assume it generates a *LTC* demand request $r_{i,j}$. However, if $r_{i,j}$ finds a matching address for a block in the prefetch side buffer, it is not propagated to the memory controller; instead, the block is moved directly into the cache. Therefore, from the core’s perspective, the request’s processing latency $\beta_{i,j}$ is zero. By including the request in the sequence $\{r_{i,j}\}$ of *LTC* requests issued by R_i , we ensure that the total number of issued *LTC* requests remains unchanged, with or without the predictable prefetcher. This also means that the previously derived bound on the cumulative latency of *LTC* requests in Section 3.1.5 still holds.

3.2.2 Features Integrated into our Predictable Prefetcher

When a prefetch request is issued by requestor R_i , an *NLTC* request is created and added to its corresponding queue $\mathcal{N}_i$ at the memory controller. Before servicing this request, the requester may issue a demand request (considered to be *LTC*) to the same address. Under normal circumstances, this *LTC* request would be inserted into its designated queue $\mathcal{L}_i$ at its intended arrival time. However, this is problematic, as two requests targetting the same address would exist in $\mathcal{L}_i$ and $\mathcal{N}_i$, both to be processed by the memory controller. To address this, we incorporate a request-merging mechanism. This mechanism involves retaining the *LTC* request in $\mathcal{L}_i$ and removing the *NLTC* request from $\mathcal{N}_i$. Before discarding the *NLTC* request, we merge any relevant information from it into the *LTC* request, as required by the HPA. For instance, if the HPA operates on a FCFS basis, the merged information would be the arrival time of the *NLTC* request, which arrived before the *LTC* request¹. This process ensures that the system behaves as though the earlier-arrived *NLTC* request was maintained, without affecting the cumulative processing latency bound of *LTC* requests.

Another feature we incorporated into our system is incrementing the latency counter C_i for requestor i when it accesses a prefetched block that is already present in the prefetch side buffer. Conventionally, upon processing a demand request issued by requestor R_i , the value Δ_i would be added to its counter upon completion, as outlined in Algorithm 1.

¹Note that from a hardware implementation perspective, FCFS is typically implemented through a queue of pointers to requests, ordered by arrival time. Hence, the merging operation effectively consists in overwriting the queue position previously occupied by the *NLTC* request with a pointer to the *LTC* request.

However, as discussed in Section 3.2.1, locating a matching block in the prefetch side buffer indicates that the request has already been prefetched, so there is no need to send a new request to the memory controller. This allows us to instantly increment the latency counter C_i for the requestor R_i that issued the demand request. Incrementing the counter in this way enables us to maintain HPA mode for a longer period, thereby avoiding a transition to RTA mode.

3.2.3 Handling Prefetch Requests at the Cache Controller Side

Implementing the predictable prefetcher described in the previous subsections requires modifying the cache controller to handle prefetched blocks differently. In our model, we assume a cache controller that includes a prefetch side buffer, implemented as a fully associative cache, along with two separate miss status holding register (MSHR) queues: one for demand requests *DMSHR* and another for prefetch requests *PFMSHR*. An MSHR queue is used to track outstanding cache misses, storing information such as the address of the missed request. As discussed in [77], MSHRs can be a significant contention point and therefore require partitioning based on request criticality. In our model, we split the MSHR queues to prevent the prefetcher, which issues *NLTC* requests, from overwhelming the MSHR queue handling *LTC* requests. This approach limits the number of prefetch requests that can be issued while ensuring that requestors issuing *LTC* requests are never blocked by the prefetcher. We now describe how our cache controller handles different cases in which request packets are sent from the core and response packets are received from the memory controller.

When a request arrives from the core, we first check if the requested data is available in the cache or the prefetch side buffer (line 5). If the data is not found and the request does not match a PFMSHR or DMSHR entry (line 6), a new entry is created in the appropriate MSHR queue (lines 7-10). Subsequently, the request is forwarded to the next level in the memory hierarchy (line 12). On the other hand, if a cache hit is encountered, the ages of the cache lines within the accessed set are adjusted (line 25). If a request packet from the core is received and is identified as being present in either the PFMSHR or the prefetch side buffer, there are two cases to consider:

- **Case 1: The received request packet corresponds to a demand request and matches the address of a prefetch request in the prefetch MSHR (line 14).** This indicates that a demand request is accessing the same address as a prefetch request that has been issued but not yet serviced. As discussed in Section 3.2.2, in

Algorithm 2 Cache Controller

```
1: PFMSHR: Stores MSHR entries for prefetch requests
2: DMSHR: Stores MSHR entries for demand requests
3: req_pkt: Request packet coming from the core
4: resp_pkt: Response packet coming from memory resource
5: if accessCache(req_pkt) == MISS AND req_pkt ∉ pf_side_buffer then
6:   if req_pkt ∉ (PFMSHR||DMSHR) then
7:     if isPrefetch(req_pkt) then
8:       PFMSHR.push(req_pkt)
9:     else
10:      DMSHR.push(req_pkt)
11:    end if
12:    sendNextLevel(req_pkt)
13:  else
14:    if req_pkt ∈ PFMSHR AND isDemand(req_pkt) then
15:      sendMergePacket()
16:    end if
17:    if req_pkt ∈ pf_side_buffer AND isDemand(req_pkt) then
18:      blk = pf_side_buffer[phys_addr]
19:      cache.handleFill(blk)
20:      pf_side_buffer.evict(phys_addr)
21:      sendIncrementPacket()
22:    end if
23:  end if
24: else
25:   cache.updateAges() ▷ Cache hit
26: end if
27: if isPrefetch(resp_pkt) then
28:   if len(pf_side_buffer) == buffer_size then
29:     old_blk = pf_side_buffer.getOldest()
30:     pf_side_buffer.evict(old_blk)
31:   end if
32:   new_blk = createBlk(resp_pkt)
33:   phys_addr = resp_pkt.getAddress()
34:   pf_side_buffer[phys_addr] = new_blk
35:   PFMSHR.remove(phys_addr)
36: else
37:   new_blk = createBlk(resp_pkt)
38:   cache.handleFill(new_blk)
39: end if
```

this case, the memory controller merges both requests by discarding the prefetch packet and transferring its information to the demand packet, which is then placed into the $\mathcal{L}_i$ queue. To achieve this, the cache controller is modified to send a merge packet with all the required information, which the memory controller must receive and respond to accordingly (line 15).

- **Case 2: The received request packet corresponds to a demand request that hits in the prefetch side buffer (line 17).** This indicates that the demand request is accessing a block that was previously prefetched and stored in the prefetch side buffer. In this case, the block is copied to the cache and evicted from the prefetch side buffer (lines 18-20). Additionally, as outlined in Section 3.2.2, when a demand request hits in the prefetch side buffer, the latency counter of the corresponding requestor is immediately incremented by Δ_i . To facilitate this, we modify the cache controller to send a counter increment packet, which the memory controller must receive and handle accordingly (line 21).

Upon receiving a response packet from the memory controller, we first verify if it corresponds to a prefetch request (line 27). If so, this suggests that data for a prefetch request, which initially missed in the cache, has now arrived. We then check if there is space in the prefetch side buffer to accommodate this request (line 28). In the event that the buffer is full, the oldest prefetched block is evicted to make room for the incoming data (lines 29-30). Once sufficient space is available, the new prefetched block is inserted into the prefetch side buffer, and the corresponding entry in the prefetch MSHR queue is subsequently freed (lines 32-35). However, if the response packet corresponds to a demand request, we allocate a new cache block, perform the necessary evictions, and insert the new block into the cache (line 38).

3.3 Evaluation

We implemented and evaluated our proposed solution, DAMA, using the gem5 simulation environment [49], operating in syscall emulation mode. Our setup consists of eight 8-wide superscalar out-of-order RISC-V cores running at 2 GHz. Each core has a 4-way set-associative 32 KB instruction cache and a 4-way set-associative 16 KB data cache, with the sizes chosen to align with the tested benchmarks’ working set sizes. All instruction and data caches are connected to a shared memory through a crossbar interconnect. A crossbar is considered since our work does not focus on handling interconnect contention. For this implementation, we employ the gem5’s simple memory model with a fixed latency of 30ns.

Note that this model makes an arbitration decision every 30ns; therefore, from a model perspective, it is equivalent to a resource with a clock period of 30ns and a service time of $\mathcal{P} = 1$ clock cycle per request. The caches include 32-entry MSHR queues. In experiments using a predictable stride prefetcher attached to the L1 data cache, we maintain two separate MSHR queues: one with 8 entries for prefetch requests and another with 24 entries for demand requests. We selected the stride prefetcher [7] for our experiments; however, our proposed method is independent of the prefetcher type and can be applied to any. We adopt FCFS arbitration as the HPA, while as RTA we employ RR arbitration among the M requestors, prioritizing the oldest *LTC* request of each requestor. Note that both arbiters service requests of a given requestor in the order they arrive at the memory controller.

To evaluate different aspects of our proposed solution, we conduct various experiments using combinations of the San Diego Vision (SDV) [78], PolyBench [69], and synthetic IsolBench [77] benchmark suites. In most experiments, we run IsolBench on the background cores to demonstrate how DAMA responds to adversarial cores attempting to saturate the shared memory resource. In one set of experiments, we use SDV as a real-world benchmark to demonstrate how DAMA ensures *LTC* requests meet their cumulative and per-request bounds. In another set of experiments, we run the linear algebra kernels in Polybench to demonstrate DAMA’s effectiveness when paired with a stride prefetcher and compare it against the predictable prefetch-based arbitration policy proposed in [23]. Polybench was chosen for its stride-based memory access patterns, which frequently trigger our attached prefetcher.

In this section, we begin by demonstrating how DAMA achieves latency bounds. We then attach a predictable prefetcher, examine how the size of the prefetch side buffer impacts the core’s IPC, and discuss how DAMA manages prefetch requests. Our comparison is against a variation of the TDMA arbitration mechanism proposed in [23]. Because our focus is not on contention at the interconnect level, we chose to abstract away the predictable interconnect proposed in the original work. For this reason, we employ a similar arbitration scheme, with the slight variation being that our approach allocates a single memory slot per core in a round-robin fashion, while their interconnect setup allows for multiple slots per core. Due to the limitations of the simulation environment, which does not support attaching a prefetcher to the memory side, we attached multiple prefetchers to the L1 data caches of the cores.

3.3.1 Achieving Latency Bounds with DAMA

In this section, we experimentally demonstrate DAMA’s latency guarantees by running a subset of real-world SDV benchmarks on the foreground core and the synthetic bandwidth-intensive IsolBench benchmark on the background cores. Our goal is to show that, under DAMA, the foreground core meets its derived per-request and cumulative latency bounds even when background cores are acting in an adversarial manner. In this set of experiments, we regulate the latency of requests originating from the foreground and background cores. This means that if any counter $\mathcal{C}_i$ for any $i \leq M$ gets depleted a switch to RTA is prompted. Figure 3.4 delineates the processing latency of requests for the foreground core executing SDV benchmarks under FCFS (HPA), RR (RTA), and DAMA. It also shows the per-request RR and DAMA bounds, the cumulative processing latency bound scaled by the number of requests $\mathcal{K}$, and the average latencies of requests. We scale the cumulative bound by $\mathcal{K}$ and display the average latency instead of the total latency for clearer visualization. For this set of experiments, we configure DAMA’s parameters as follows: $\Delta_i = \mathcal{B}_i = 8$, $\mathcal{S} = 16$, $\mathcal{P} = 1$. Under RR, the worst-case scenario occurs when a switch to RR happens after starting to process an *NLTC* request, causing the next *LTC* request in RR order to suffer a waiting time of $\mathcal{P} - 1$ plus the maximum waiting time in RR which is $M \cdot \mathcal{P}$. Hence, the RR per-request bound is computed as $\mathcal{B}_i = M \cdot \mathcal{P} + \mathcal{P} - 1$. Since in our case $\mathcal{P} = 1$, our per-request RR bound becomes $\mathcal{B}_i = 8$. Under DAMA, the worst-case processing latency of an *LTC* request $r_{i,j}$ occurs when the counter $\mathcal{C}_i$ starts at its maximum value $\mathcal{S}_i$ when $r_{i,j}$ becomes oldest. Because our employed arbiters process requests of R_i at the memory controller in order, no other *LTC* request of R_i can finish while $r_{i,j}$ is oldest. Therefore, in the worst case the counter keeps decrementing until it reaches 0 and the arbiter switches to RTA mode. Then, the *LTC* requests takes an additional $\mathcal{B}_i$ cycles to get serviced. Consequently, the maximum latency that could be experienced by a single request under DAMA in this experiment cannot exceed $\mathcal{S}_i + \mathcal{B}_i = 24$.

Figure 3.4 illustrates how requests executing under FCFS experience substantial latency spikes. This is because FCFS prioritizes requests based on their arrival time, and the high volume of background core requests overloads the shared memory resource. As a result, the requests issued by the foreground core face significant delays, leading to the average latency exceeding the scaled cumulative latency bound. Under RR, however, the maximum latency experienced by a foreground core request remains within the bound imposed by RR. Consequently, the average latency of requests saturates at this bound and never exceeds it. The DAMA model, on the other hand, keeps executing requests in FCFS mode until the counter $\mathcal{C}_i$ is depleted, at which point it switches to RR mode. This explains the latency spikes that surpass the RR bound but stay within the per-request bound imposed

by DAMA in both benchmarks *tracking* and *localization*. It is important to note that the foreground core does not always trigger the switch to RTA mode. As a result, requests on the foreground core may not always experience latency spikes, since this depends on the core’s access pattern and the possibility of other cores triggering a switch. This helps explain why *tracking* has fewer latency spikes than *localization*. As for the average latency of requests, we can see that it consistently remains below the scaled cumulative bound and eventually saturates at the RR bound. This saturation occurs because scaling the cumulative bound by $1/\mathcal{K}$ results in the expression $\mathcal{S}/\mathcal{K} + \Delta$ which corresponds to the cumulative bound with $CS = 1$ in Theorem 2. As the number of requests increases, the first term approaches zero, leaving only Δ which is equal to the RR bound $\mathcal{B}$ in this experiment.

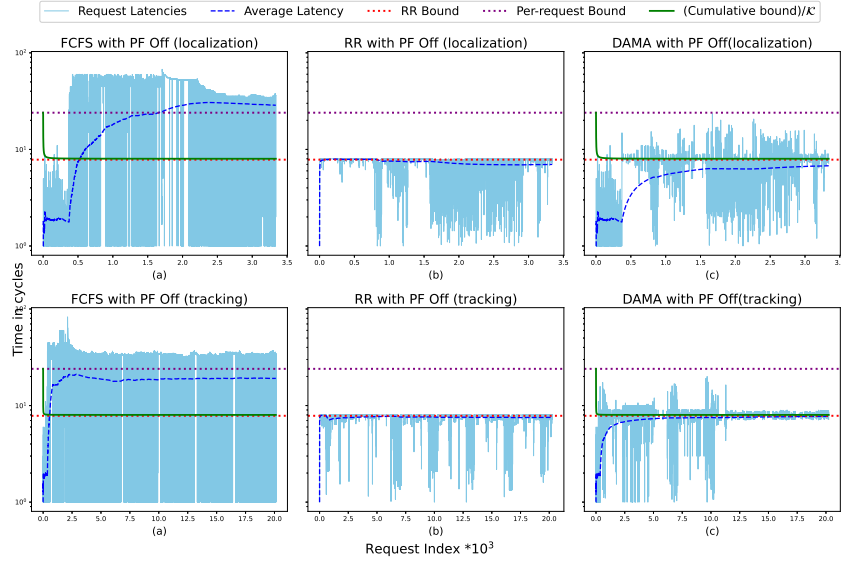


Figure 3.4: Per-request and cumulative latency bounds under FCFS, RR, and DAMA when the PF is disabled

In Figure 3.5, we experimentally demonstrate that after enabling the prefetcher on all cores the cumulative processing latency bound is still honored for *tracking* and *localization* benchmarks. However, the *tracking* benchmark exhibits per-request latency spikes, suggesting that the previous per-request latency bound is violated upon enabling the prefetcher. This happens because, with the prefetcher enabled, DAMA allows requests to be serviced out of order in the specific case where a demand request hits the prefetch side buffer while the memory controller is still servicing an earlier demand request. In this scenario, the

latency counter is incremented by Δ_i , giving the request being serviced more time to continue executing in HPA mode, thereby exceeding the previous per-request latency bound. We remark that the focus of our approach is on bounding the cumulative processing latency of *LTC* requests, allowing flexibility with the latency of individual requests to maintain the system in HPA mode as long as possible.

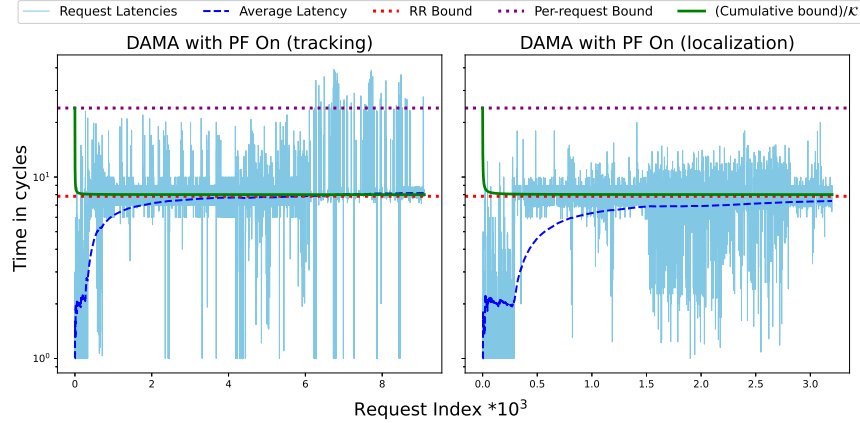


Figure 3.5: Per-request and cumulative latency bounds under DAMA when the PF is enabled

3.3.2 Impact of Prefetch Side Buffer Size on IPC

To evaluate the influence of prefetch buffer size on average performance, we executed a subset of linear algebra kernels from PolyBench on a single core. Figure 3.6 depicts the normalized Instructions Per Cycle (IPC) of the core for various cache and prefetch buffer sizes, relative to a configuration without a prefetcher. As observed, most benchmarks exhibit a performance increase with larger cache sizes. This is attributed to the larger cache’s ability to accommodate more prefetch blocks, potentially reducing cache evictions. For example, benchmarks *bicg* and *mvt* experienced an approximate 15% IPC boost when the cache size was increased from 2KB to 16KB.

As Figure 3.6 illustrates, for small cache sizes the prefetch side buffer can perform even better than a conventional (not pred) prefetcher. For example, with a 2KB cache, benchmarks *bicg*, *mvt*, and *doitgen* experienced IPC increases of 12%, 5%, and 21%, respectively, with a 16-entry buffer. This improvement occurs because the prefetch side buffer stores prefetched blocks until they are needed, reducing cache pollution, especially

when the prefetcher is aggressive. However, if the buffer is too small, performance can degrade significantly. In this case, prefetched data may be evicted before it is needed, leading to wasted prefetch effort. This behavior is exhibited with a prefetch buffer size of 2 and 4 for all benchmarks. Based on these results, we chose an 8-entry prefetch buffer, as Figure 3.6 indicates that this configuration achieves near-optimal performance with a 16KB cache. Additionally, the extra memory overhead for this configuration is minimal, amounting to 3%.

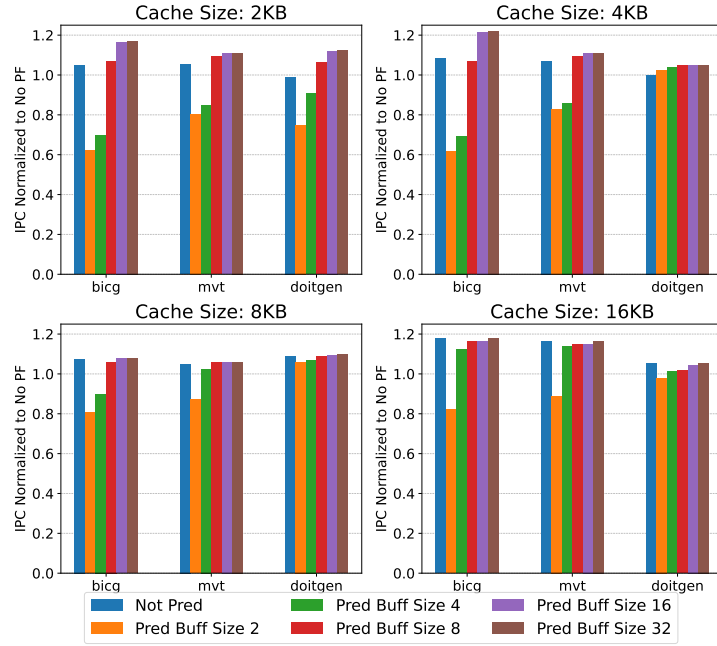


Figure 3.6: Normalized IPC for a prefetcher in a single core setup with varying buffer and cache sizes

3.3.3 Handling Prefetch Requests with DAMA

In this series of experiments, we evaluate the effectiveness of DAMA when we attach a predictable stride prefetcher to the L1 data caches of the system. To evaluate how DAMA performs when the prefetcher is enabled we use two different benchmark setups described as follows:

- **Setup 1:** The foreground core executes PolyBench benchmarks, while the background cores run bandwidth-intensive workloads.
- **Setup 2:** We execute eight distinct PolyBench benchmarks, each assigned to a separate core.

Figure 3.7 illustrates the normalized average IPC for DAMA with and without the prefetcher across various slack values, relative to the FCFS configuration without a prefetcher under setup 1.

Performance under FCFS and RR (setup 1): When the prefetcher is disabled, Figure 3.7 shows that the RR scheme performs poorly compared to FCFS. While RR may benefit the core running PolyBench benchmarks by giving it a fair opportunity to send requests, it severely harms the performance of background cores running IsolBench, which are more bandwidth-intensive. In contrast, FCFS prioritizes requests based on arrival time, resulting in a notable increase in average IPC. Figure 3.7 also illustrates that enabling the predictable prefetcher leads to a significant performance boost across all benchmarks under FCFS, with a maximum improvement of 22% compared to the configuration without a prefetcher.

Performance under DAMA while varying $\mathcal{S}_i$ (setup 1 and setup 2): Figure 3.7 illustrates the impact of $\mathcal{S}_i$ on the average IPC of the system under DAMA in setup 1. We scale the x-axis by $\mathcal{K} \cdot \Delta_i \cdot 10^{-6}$ to represent the additional latency overhead introduced by the slack relative to the total number of requests multiplied by Δ_i . This provides insight into how increasing $\mathcal{S}_i$ affects the cumulative latency bound. As $\mathcal{S}_i$ increases, DAMA’s performance converges to FCFS performance, both with and without the prefetcher. This demonstrates that larger slack values enable DAMA to sustain HPA mode until one of the counters gets depleted. Notably, a minimal increase of $1640.25 \cdot \mathcal{K} \cdot \Delta_i \cdot 10^{-6}$ in slack is sufficient for most benchmarks to achieve optimal FCFS performance with the prefetcher enabled, while a larger increase of $14762.25 \cdot \mathcal{K} \cdot \Delta_i \cdot 10^{-6}$ is required when the prefetcher is disabled. DAMA with the prefetcher enabled converges to the optimal solution with a smaller $\mathcal{S}_i$ value relative to the configuration where the prefetcher is disabled. This is attributed to the counter adjustment feature that we incorporated into our model, as detailed in Section 3.2.2. This feature involves incrementing the counter $\mathcal{C}_i$ by Δ_i whenever a demand request finds a matching block in the prefetch side buffer. As a result, DAMA can remain in HPA mode for a longer duration even with a small slack value. Under the same setup, Figure 3.8 illustrates the percentage of time spent in FCFS as a function of $\mathcal{S}_i$ for the *atax* benchmark. This plot exhibits the same trend as Figure 3.7 since a higher IPC is associated with more time spent in HPA mode by the arbiter compared to RTA mode.

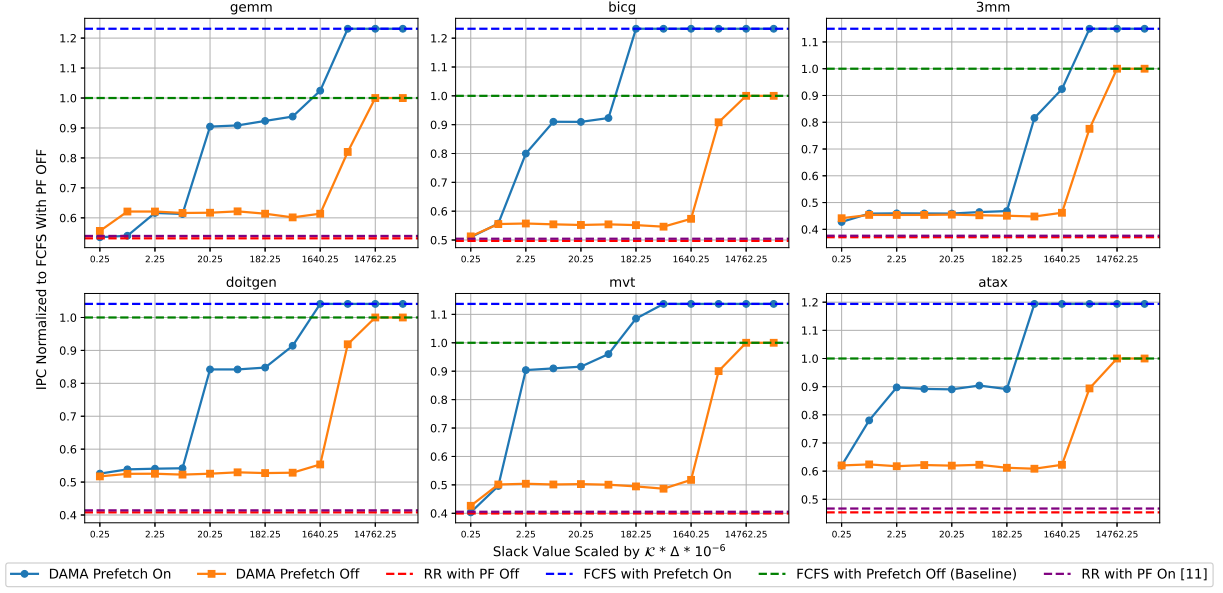


Figure 3.7: Average IPC as a function of $\mathcal{S}_i$ for different arbitration models and configurations

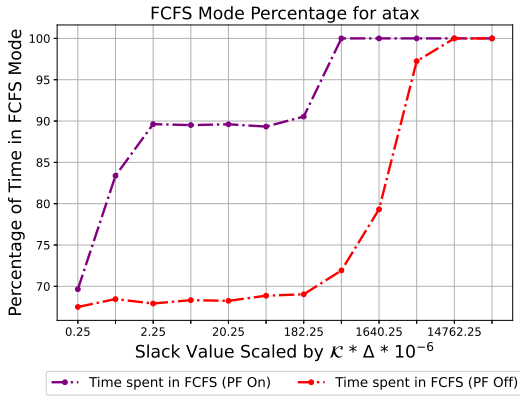


Figure 3.8: Percentage of time spent in FCFS mode as a function of $\mathcal{S}_i$

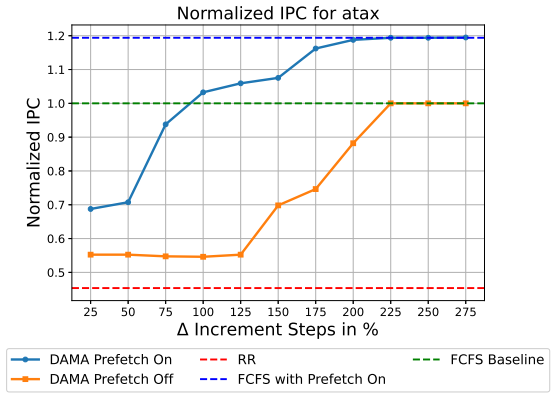


Figure 3.9: Normalized IPC relative to percentage increment of Δ_i with $\mathcal{S}_i = 2.25 \times \mathcal{K} \times \Delta_i \times 10^{-6}$

In setup 2, we run a distinct linear algebra benchmark on each core. Figure 3.10 illustrates the normalized IPC relative to the scaled $\mathcal{S}_i$. While the average IPC trend remains

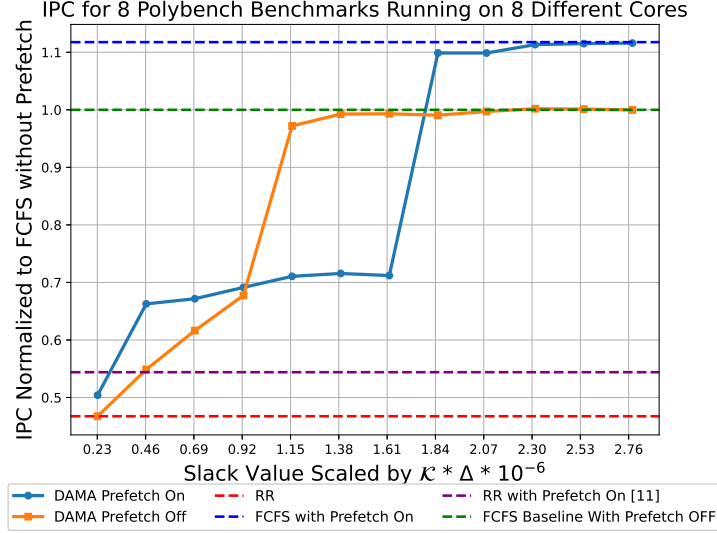


Figure 3.10: Normalized IPC for a prefetcher in a single core setup with varying buffer and cache sizes

consistent with the previous setup, a notable observation emerges: DAMA converges to the optimal FCFS mode for both configurations (prefetcher on and off) with a remarkably small slack value of $2.3 \cdot \mathcal{K} \cdot \Delta_i \cdot 10^{-6}$. This suggests that when cores execute real-world workloads that do not excessively overwhelm the shared memory resource, optimal performance can be achieved with minimal slack, resulting in negligible overhead.

Performance under DAMA while varying Δ_i (setup 1): Figure 3.9 shows how varying Δ_i affects the average IPC under DAMA for a setup running the *atax* benchmark on the foreground core and a bandwidth-intensive benchmark on the background cores. In this experiment, the slack value is fixed to $\mathcal{S}_i = 2.25 \cdot \mathcal{K} \cdot \Delta_i \cdot 10^{-6}$, while Δ_i increases incrementally by 25%. The results indicate that a 225% increment in Δ_i allows DAMA to reach optimal performance for both prefetcher-enabled and prefetcher-disabled configurations. Since the cumulative latency bound is given by $\mathcal{S}_i + \Delta_i \cdot \mathcal{K}$, increasing Δ_i by 225% causes a proportional increase in the bound. On the other hand, the slack value ($14762.25 \cdot \mathcal{K} \cdot \Delta_i \cdot 10^{-6}$) that achieves optimal performance results in an increase in the bound of only 1.4%.

Hence, upon attempting to balance between tight bounds and performance, it is advisable to vary the maximum slack value $\mathcal{S}_i$ rather than Δ_i . Note that unlike DAMA, Duetto [54] can only approach the optimal average performance by increasing Δ_i , which significantly affects its latency bound.

Comparison with [23] (setup 1 and setup 2): We evaluate our approach against the arbitration scheme proposed in [23] using setup 1 and setup 2. In setup 1, our experiments indicate that our adaptation of the arbitration scheme proposed in [23] performs poorly in this setup, yielding results similar to the RR arbitration scheme as can be seen in Figure 3.7. This outcome arises because the background cores generate a high volume of requests to the shared resource, leaving no slots available for issuing prefetch requests. However, in setup 2, the scheme proposed in [23] demonstrated an 8% improvement over RR as can be seen in Figure 3.10. This advantage is due to the fact that, with real-world benchmarks, memory resources are not always fully saturated, allowing the arbitration scheme to find opportunities to issue prefetch requests.

In both setups, our approach shows a performance advantage over the work presented in [23], though it is important to note that our simulation environment differs from theirs, which imposes some limitations on direct comparisons.

3.4 Conclusions

In this chapter, we introduced DAMA, a dual arbitration mechanism that provides an upper bound on the cumulative processing latency of *LTC* requests without unduly impacting the performance of *NLTC* requests. Our model employs an efficient monitoring mechanism that informs the memory controller when a switch in arbitration mode is needed. We attach a predictable prefetcher to our L1 data caches as a source of *NLTC* requests and leverage the properties of this mechanism to further enhance the average performance of the system without impacting the bounds. Our experiments demonstrate that, with a minimal overhead of 1.4% on the cumulative latency bound, DAMA without a prefetcher achieves up to a 60% improvement in IPC over RR. Furthermore, we demonstrated that DAMA can fully leverage the potential of a prefetcher, achieving an average performance increase of 22% compared to the configuration with the prefetcher disabled.

Chapter 4

Duration-based Instruction Cache Locking

In Chapter 2.4, we discussed the various related works on static and dynamic cache locking. Although dynamic locking provides the flexibility that the static approach lacks, modifying the program’s binary can be undesirable. Such modifications have implications for both the program’s size and memory map [58] and thus require complex adjustments to the compilation flow. Furthermore, in instances where recompilation is not possible because the application’s source code is not available, applying dynamic locking can become unfeasible. In this Chapter, we introduce a novel duration-based locking approach that emulates how dynamic locking works without incurring significant area overheads (Section 4.1). We propose a profiling-based software heuristic that utilizes the proposed hardware mechanism to identify the locked content of a program and determine their corresponding locking durations (Section 4.2). We then assess the effectiveness of the proposed solution by running various embedded benchmarks within the gem5 simulation environment (Section 4.3).

4.1 Duration-based Locking Mechanism

We begin by discussing how our duration-based locking mechanism is realized in the cache replacement policy. Our cache line selection heuristic is tailored to blocks that contain program code; hence, we discuss our mechanism based on a private L1 instruction cache. However, given a suitable selection algorithm, our hardware mechanism could be applied to any cache.

Specifically, we consider a system that runs single-threaded tasks without preemption and virtual memory, featuring an L1 set-associative cache with block size $\mathcal{B}$, number of cache sets $\mathcal{S}$, and number of ways $\mathcal{W}$. Hence, the total capacity of the cache is equal to $\mathcal{B} \cdot \mathcal{S} \cdot \mathcal{W}$. In set-associative caches, a memory block c can reside in any vacant way $w \in \{0, \dots, \mathcal{W} - 1\}$ within the cache set $s = \text{index}(c)$, $s \in \{0, \dots, \mathcal{S} - 1\}$ determined by its address's index bits. We propose to perform locking at the granularity of a *memory region*. We consider a set $\mathcal{R} = \{r_1, \dots, r_j, \dots, r_K\}$ of K memory regions. We define a memory region r_j to be a range of continuous physical addresses such that all cache lines within the range have the same locking duration $d_j \in \mathbb{N}$.

4.1.1 Duration-based LRU Policy

We adopt a modified version of the Least Recently Used (LRU) replacement policy, incorporating our duration-based locking mechanism. The conventional LRU replacement policy employs age bits to track the recency of cache lines, ensuring that the least recently accessed blocks are chosen for eviction when all the lines in a set are fully occupied [66]. Our cache-locking mechanism operates on a per-line basis and leverages the LRU age bits to implement locks with different durations. More formally, at any given time, let $\beta_{s,w} \in \mathbb{Z}$ represent the age of the block allocated in cache in set s and way w . We have repurposed the LRU ages to convey two sets of information: the age of the block and its lock status. The lock status of a block is determined by the sign of $\beta_{s,w}$. Negative ages indicate locked blocks, while positive ages indicate unlocked blocks that are considered for eviction. Upon assigning a negative age to a given block, the replacement policy excludes it from the list of potential candidates during eviction. Determining the lock status of incoming addresses to the cache is done through a lookup table $DLUT : \mathcal{R} \rightarrow \mathbb{N}$ that maps memory regions $\mathcal{R}$ to locking durations. The logic of the Duration-Based LRU policy is depicted in Algorithm 3.

Upon accessing an address, we first verify its hit status by checking the presence of the address's tag in the tag store (line 4). If the accessed address results in a cache hit its age's signedness is checked to determine whether the block is already locked or not (line 5). If the age is negative, this indicates an existing lock, and the algorithm advances the negative and positive ages of all blocks within the accessed set (lines 6, 7). The age of the locked address keeps getting incremented whenever its set gets accessed until it reaches zero, signaling its eligibility for eviction. If the age is positive on the other hand, the algorithm sets the accessed block's age to zero and increments the age of all other blocks within the set by one (lines 9, 10). Upon fetching a block that does not reside in the cache, we first evict the block that has the oldest age among others in the same set (lines 13, 14).

It is important to note that the cache's hit path remains unaffected by the *DLUT* lookup delays, as the *DLUT* checks only occur on cache misses, where most time is spent fetching data from the next memory level. Then, the *DLUT* is checked to ascertain whether the fetched address belongs to a locked memory region (line 15). If the address exists in the *DLUT* (line 14) and the total number of locked blocks in a set does not exceed the locking threshold defined by locking percentage $\mathcal{P}$ multiplied by the number of ways $\mathcal{W}$ (line 15), the algorithm locks the cache block by setting its age to the negative value of the locking duration, indicating how long it will remain locked (line 16). Otherwise, the algorithm sets the block's age to zero (line 18) and increments the ages of all other blocks within the same set by one (line 21).

Algorithm 3 Duration-based LRU policy

```

1: phys_addr: Physical address of the accessed block
2: addr_tag: The physical address's tag portion
3: tag_store: Cache's structure that stores tags
4: s: The physical address's index portion
5: if  $\exists w \in \{0, \dots, \mathcal{W} - 1\} : \text{tag\_store}[s, w] = \text{addr\_tag}$  then ▷ Cache hit
6:   if  $\beta_{s,w} < 0$  then ▷ Cache line is already locked
7:      $\beta_{s,j} = \beta_{s,j} + 1$  for all  $j \in \{0, \dots, \mathcal{W} - 1\}$ 
8:   else
9:      $\beta_{s,w} = 0$ 
10:     $\beta_{s,j} = \beta_{s,j} + 1$  for all  $j \in \{0, \dots, \mathcal{W} - 1\}, j \neq w$ 
11:  end if
12: else ▷ Cache Miss
13:    $w = \text{oldest}(\text{tag\_store}[s, 0], \dots, \text{tag\_store}[s, \mathcal{W} - 1])$ 
14:   if  $|\{j \mid 0 \leq j < \mathcal{W}, \beta_{s,j} < 0\}| < \mathcal{P} \cdot \mathcal{W}$  AND
15:      $\text{DLUT}[\text{phys\_addr}] \neq \text{NULL}$  then
16:        $\beta_{s,w} = -(\text{DLUT}[\text{phys\_addr}])$ 
17:     else
18:        $\beta_{s,w} = 0$ 
19:     end if
20:      $\beta_{s,j} = \beta_{s,j} + 1$  for all  $j \in \{0, \dots, \mathcal{W} - 1\}, j \neq w$ 
21: end if

```

4.1.2 Hardware Implementation

To implement the proposed duration-based LRU policy, some hardware modifications are required at the level of the cache controller. These modifications include extending the bitwidths of ages in order to support the required ranges of negative durations in 2's complement form. Additionally, a range addressable lookup table (RALUT) is needed to store address ranges of memory regions and their corresponding locking durations, with invalid entries initialized to zero. RALUT is a special LUT type that utilizes an addressing method reliant on range matching as opposed to precise matching. This is accomplished by performing parallel comparisons and using a priority encoder to aggregate the comparator outputs, thereby determining the range to which the incoming addresses map. Additionally, to count the number of locked lines per set we extract the most significant bit from each cache line's age. This is made possible by our use of negative ages.

4.2 Region Selection Heuristic

In this section, we present a heuristic designed to leverage our proposed duration-based locking mechanism. This heuristic analyzes the behavior of a given program to determine which instructions to lock in the cache. It builds upon the loop-based approach introduced in [19], which focuses on analyzing loop structures. The rationale for analyzing loop structures lies in the expectation that the greatest benefit can be achieved by locking memory blocks with the highest access frequencies, which are typically found within loops. However, unlike the approach in [19], which primarily aims to achieve a tighter WCET bound, our cache line selection algorithm is optimized to enhance average-case performance. Our proposed heuristic operates as a single-pass method, selecting addresses to lock after just one profiling iteration. The heuristic takes as input the program's control flow graph (CFG) and outputs a set of memory regions along with their respective locking durations. We begin by providing an overview of the heuristic, followed by detailed discussions of each step in subsequent subsections.

By adopting the loop-based approach proposed in [19], the addresses that get accessed within loops are the only ones considered for locking. We start by extracting and analyzing outermost loops to collect information about cache lines that conflict on the same set (Section 4.2.1). We pass this information to an ILP that picks the locked content of the cache (Section 4.2.2). Finally, we cluster the picked locked lines into memory regions with continuous addresses and we select a suitable duration for every region (Section 4.2.3).

4.2.1 Extracting and Analyzing Loops

Algorithm 4 outlines the steps required to collect the necessary information for the ILP, which in turn determines the cache lines to be locked. Let C be the set of all cache lines containing code executed by the program. The algorithm takes a program CFG as input and produces a dictionary $\mathcal{H}$ that holds the global hotness of every cache line accessed in the program, a data structure σ that keeps track of all the addresses that conflict at every cache set, and a dictionary $\mathcal{A}$ that holds the duration for each locked cache line. Our heuristic starts by traversing the CFG of the program to extract the entry and exit addresses of each outermost loop. Let $\mathcal{L}$ be the set of all outermost loops in a given program. We start by initializing our σ data structure to hold an empty list for every cache set (line 8), and the $\mathcal{A}$ dictionary to hold an infinity value for all cache lines accessed in the program (line 7). We iterate through the nodes within each outermost loop $\ell \in \mathcal{L}$ and compute the set (line 12) for every cache line. Then, we populate the σ data structure which keeps track of all the cache lines that get mapped to the same cache set for every outermost loop (line 13). For every accessed cache line c in outermost loop ℓ , we maintain a record of its overall access frequency across the entire program by accumulating the values returned by a function called *get_freq* which returns how many times a cache line gets accessed within a given outermost loop (line 14). We assume that the set of cache lines accessed within a program and their frequency in each outermost loop, including any function or inner loop within, can either be determined by profiling the program off-line, or by static program analysis.

In addition to keeping track of overall access frequencies of cache lines, our heuristic also keeps track of the number of times a cache set gets accessed within a loop (line 10, 15). Our algorithm sets the locking duration for a given cache line based on the access frequency of the set to which the cache line belongs (line 17, 18). The rationale behind picking locking durations in this manner is that our duration-based LRU policy presented in Section 4.1 increments the negative ages for locked blocks whenever the cache set gets accessed. Thus, we set the duration in this manner to account for non-locked cache lines belonging to the same set as the locked line. Given that our mechanism locks addresses on access, the same address may be accessed in different loops, each with different frequencies. To avoid propagating residual locked blocks from earlier loops, we set the locking duration to be the minimum set access frequency across all loops.

Algorithm 4 Extracting and Analyzing Loops

```
1: prog_cfg : A graph representing the program CFG
2:  $\mathcal{L}$  : Set of outermost loops
3:  $\mathcal{H}$  : Stores global hotness of all cache lines
4:  $\mathcal{A}$  : Stores minimum access frequencies of cache lines
5:  $\sigma$  : Stores cache lines that conflict on the same set
6:  $\mathcal{L} = \text{extract\_loops}(\text{prog\_cfg})$ 
7:  $\mathcal{H}(c) = 0, \mathcal{A}(c) = \infty \quad \forall c \in C$ 
8:  $\sigma(\ell, s) = \emptyset \quad \forall \ell \in \mathcal{L}, \forall s \in \{0, \dots, \mathcal{S} - 1\}$ 
9: for each  $\ell$  in  $\mathcal{L}$  do
10:    $\text{set\_freq}(s) = 0 \quad \forall s \in \{0, \dots, \mathcal{S} - 1\}$ 
11:   for each  $c \in \ell$  do
12:      $s = \text{index}(c)$ 
13:      $\sigma(\ell, s) = \sigma(\ell, s) \cup c$ 
14:      $\mathcal{H}(c) + = \text{get\_freq}(c, \ell)$ 
15:      $\text{set\_freq}(s) + = \text{get\_freq}(c, \ell)$ 
16:   end for
17:    $\mathcal{A}(c) = \min(\text{set\_freq}(s), \mathcal{A}(c))$ 
18:    $\forall s \in \{0, \dots, \mathcal{S} - 1\}, \forall c \in \sigma(\ell, s)$ 
19: end for
```

4.2.2 Selecting Locked Cache Lines

To select the locked content of the cache, we propose an ILP formulation designed to maximize the gain of locking under a set of constraints. Let $x_c \in \{0, 1\}$ be a binary decision variable that specifies if cache line c gets selected for locking or not. Let $\mathcal{P}$ be the percentage of the cache that needs to be locked. We define our ILP objective function $\mathcal{G}(s)$ for a set s along with its constraints as follows:

$$\mathcal{G}(s) = \sum_{c, s=\text{index}(c)} x_c \cdot \mathcal{H}(c) \quad (4.1)$$

$$\text{s.t.} \quad \sum_{c \in \sigma(\ell, s)} x_c \leq \mathcal{P} \cdot \mathcal{W} \quad \forall \ell \in \mathcal{L} \quad (4.2)$$

- x_c : binary variable for cache line c
- $\mathcal{H}(c)$: Global hotness of cache line c

- $\sigma(\ell, s)$: cache lines that conflict in loop ℓ on set s
- $\mathcal{P}$: Locking percentage
- $\mathcal{W}$: Total number of ways

The objective function depicted in Equation 4.1 locks a subset of the cache lines in set s across all outermost loops such that $\mathcal{G}$ is maximized. We apply the same objective function on every cache set independently due to the fact that they are mutually exclusive: locking a block in a given cache set will never affect the ages or contents of other sets. For that reason, we can linearly decompose our objective function and apply it per set. The objective function in Equation 4.1 is subject to the set of constraints outlined in Equation 4.2. For every loop, the corresponding constraint dictates that the total number of locked addresses does not exceed a locking threshold defined by $\mathcal{P} \cdot \mathcal{W}$. Adjusting $\mathcal{P}$ to a value below 1 ensures the availability of a designated number of unoccupied slots per cache set, where unlocked data can reside. For instance, in a 4-way set-associative cache, setting $\mathcal{P}$ to 0.5 will leave 2 free slots per cache set for unlocked cache lines. In our approach we avoid setting $\mathcal{P}$ to 1.0 as this would require bypassing the cache and suffering the latency of fetching unlocked data from main memory which would dramatically affect the performance of the system. Finally, we note that our ILP formulation is equivalent to the multiple knapsack problem, which is known to be NP-Hard [52].

4.2.3 Selecting Memory Regions and Durations

The ILP formulation discussed in the previous section can choose to lock an arbitrary number of addresses. The ILP only constrains the number of ways that can be locked per set but does not impose any constraint on the number of *DLUT* entries. This is crucial as the size of the *DLUT* can have severe implications on latency and area. To that end, we employ a KMeans algorithm that groups locked addresses into clusters that constitute memory regions each containing a set of continuous addresses. KMeans is an iterative unsupervised learning algorithm that assigns points of a given dataset to K clusters by minimizing the sum of squared distances between points and their assigned clusters [48]. Let $\mathcal{Z}$ be the set of locked cache lines picked by the ILP, i.e.:

$$\mathcal{Z} = \{c \mid \forall c \in \mathcal{C}, x_c = 1\}. \quad (4.3)$$

Let $\mathcal{J}$ be the set of clusters such that $|\mathcal{J}| \leq K$. The cost function $\mathcal{F}$ of the employed KMeans algorithm is defined as:

$$\mathcal{F} = \sum_{j=1}^{|\mathcal{J}|} \sum_{i=1}^{|\mathcal{Z}|} \|z_i^{(j)} - \mu_j\|^2. \quad (4.4)$$

Here, $z_i^{(j)} \in \mathcal{Z}$ is the datapoint of the locked line z in cluster j , and μ_j denotes the centroid of cluster j .

After placing the locked addresses into clusters we define our memory regions based on the boundaries of every cluster. Let $\zeta(j)$ be a function that returns all locked addresses $c \in \mathcal{Z}$ that belong to the j -th cluster. Then, each memory region $r_j \in \mathcal{R}$ simply comprises all cache lines between $\min(\zeta(j))$ and $\max(\zeta(j))$. Note that configuring regions in this manner may result in the locking of cache lines that were not selected by the heuristic. The impact this has on performance is discussed in Section ???. Defining regions in this manner allows us to use a RALUT, with each entry storing the starting and ending addresses of the specified region. After determining the boundaries of memory regions, we assign their corresponding durations. Given that locked lines were clustered based on their address proximity, different addresses within the same cluster might have a different locking duration. For this reason, we estimate the locking duration of d_j for every region $r_j \in \mathcal{R}$ by averaging the durations of all locked addresses that reside within the same cluster as follows:

$$d_j = \sum_{c \in \zeta(j)} \frac{\mathcal{A}(c)}{|\zeta(j)|} \quad \forall j \in \{1, \dots, |\mathcal{J}|\}. \quad (4.5)$$

Finally, we ensure that the picked durations fall within the ranges dictated by the age's bitwidths by applying a clamping function that saturates the duration after exceeding the maximum threshold.

4.3 Evaluation

We implemented and evaluated the proposed duration-based instruction cache locking mechanism in the gem5 [49] simulation environment running in syscall emulation mode. The simulation setup consists of a single RISC-V core hosting an L1 I-cache and L1 D-cache connected through a crossbar to the main memory. Throughout our experiments, we use a 4-ways L1 I-cache with a size of either 2kB or 4kB.

We evaluated the effectiveness of our approach by running various benchmarks from the MiBench [27] and San Diego Vision [79] benchmark suites. We pick benchmarks that are most representative of embedded applications out of which we include those with more than 100 million dynamic instructions. Through experimentation, it was determined that performing locking on benchmarks below this threshold is ineffective as these benchmarks exhibit low I-cache footprints. We construct our CFGs and collect profiling information by analyzing the instruction traces generated by gem5. The time it took to generate the CFG, profile the program, and solve the ILP varied between 2 minutes for small benchmarks like *fft* and *sha* to around 40 minutes for benchmarks with large traces like *basicmath*.

In this section, we first discuss how to configure the number of lockable memory regions and the maximum representable duration, using a subset of the evaluated benchmarks. Those are hardware implementation parameters that affect our heuristic. Then, we compare our duration-based approach to the loop-based dynamic approach proposed in [19]. As discussed in Section ??, for the dynamic approach we employ a modified version of our heuristic; specifically, for each loop l and set s , we lock the $\mathcal{P} \cdot \mathcal{W}$ cache lines with the highest access frequencies. We set the value of $\mathcal{P}$ to 75% (we lock 3 ways) as this value performed best for both approaches over the selected cache sizes and benchmarks. A similar methodology was employed to assess the performance of the static approach, the findings of which are illustrated in Figure 4.3.

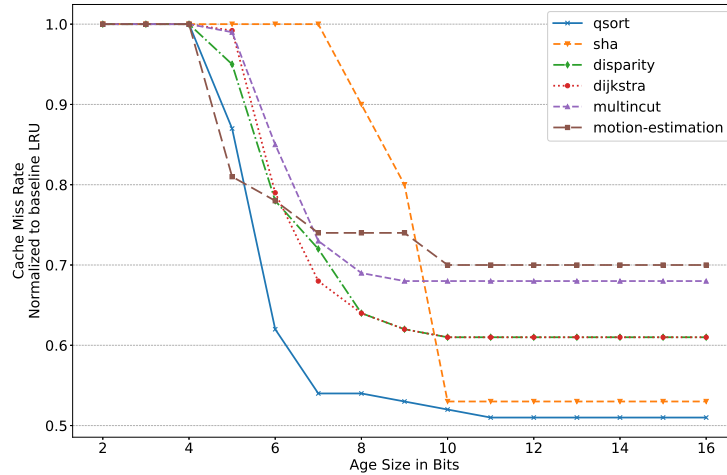


Figure 4.1: Normalized miss rate for duration-based varying the age’s bitwidth; 2KB I-cache with $K = 8$ memory regions

Age’s Bitwidth. Figure 4.1 depicts the I-cache miss rate for our duration-based approach as a function of the number of age bits, normalized by the one of the baseline counter-based LRU policy [66] with $\log_2(\mathcal{W}) = 2$ bits. As discussed in Section 4.2.3, small

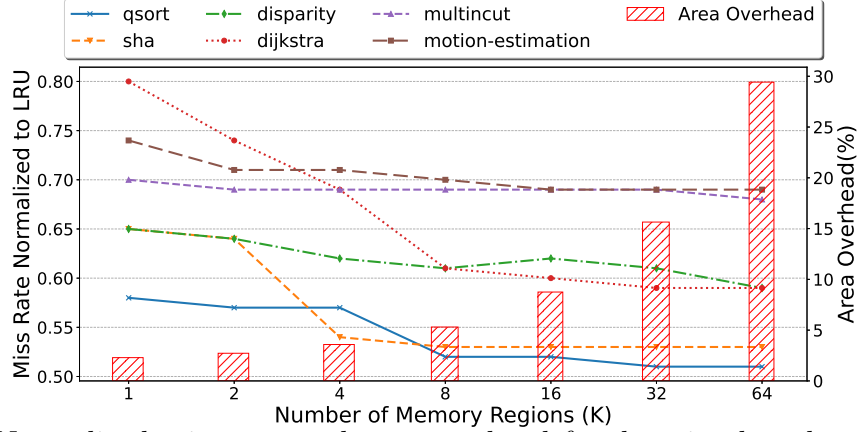


Figure 4.2: Normalized miss rate and area overhead for duration-based varying K ; 2KB I-cache with 10 age bits

bitwidths may result in clamped, suboptimal duration values: we observe a consistent pattern of miss rate enhancement as the number of bits increases across all benchmarks, followed by a plateau once a specific bitwidth is reached. Based on Figure 4.1, we select an age bitwidth of 10 bits, as larger values show marginal miss rate improvements.

Number of Memory Regions. In Figure 4.2, we again plot the normalized cache miss rate for the duration-based approach, this time based on the number of memory regions K . The figure also shows the memory overhead in terms of added memory bits due to the *DLUT* and the increased age’s bitwidth, compared to the total number of memory bits in the baseline LRU L1 I-cache. As seen in Figure 4.2, our approach performs poorly for some benchmarks at low number of regions: when K is small, the clusters selected by the heuristic cover large ranges, which may include addresses that were not picked by the ILP, and the selected durations are less accurate. At the same time, the overhead increases linearly in K . Based on the obtained results, we select a value of $K = 8$, which results in significant improvements for all benchmarks with a reasonable area overhead of 5.6%.

Area Overhead. For a 4-way 2kB instruction cache with a 64B line size, increasing the age bitwidth to 10 bits adds an extra memory overhead of 40 bytes. Furthermore, the proposed *DLUT* structure includes storing starting and ending addresses for each region, along with the associated duration. In our experiments, the *DLUT* we utilized had 8 entries, resulting in an additional memory overhead of 74 bytes. Thus, the combined area overhead of adopting the duration-based locking approach is 114 bytes, which amounts to a total area overhead of 5.6% as can be seen in Figure 4.2.

Duration-based vs Dynamic. Figure 4.3 shows the normalized I-cache miss rate for the duration-based and dynamic approaches over all benchmarks and cache sizes of 2kB

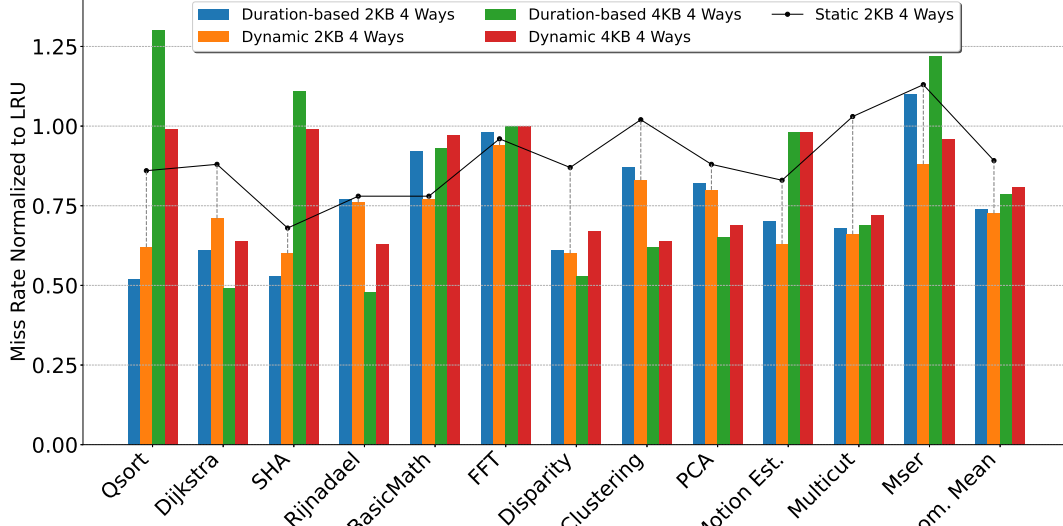


Figure 4.3: Normalized I-cache miss rate for dynamic, static, and duration-based locking; $K = 8, 10$ age bits

and 4kB. For a 2kB I-cache size, our proposed approach achieves a maximum miss rate improvement of 47% for *qsort*, and an average miss rate improvement of 26% across all benchmarks, which is nearly equivalent to the 27% improvement achieved by the dynamic approach. It performs better than the dynamic approach for the *qsort*, *dijkstra*, and *sha* benchmarks. Such benchmarks exhibit large outermost loops with intricate access patterns; locking/unlocking at loop entry/exit points proves to be less flexible in comparison to the duration-based approach which has the potential of adapting to how the program is behaving by selecting different durations for different cache lines. For a 4kB I-cache size, our proposed approach achieves a maximum miss rate improvement of 52% for *rij*, and an average miss rate improvement of 21% across all benchmarks, which is slightly better than the dynamic approach at 19%. Note that both the duration-based and the dynamic approach achieve no significant benefit, or even perform worse compared to the baseline for some benchmarks, in particular *qsort*, *sha*, *fft*, *motion – estimation*, and *mser*: at 4kB, the code of such benchmarks can fit entirely in the I-cache, removing any benefit from locking.

Execution Overhead. Both the duration-based and the dynamic approach suffer execution time overheads compared to the baseline LRU, as summarized in Table 4.1. For duration-based, the *DLUT* must be configured before the program is executed; assuming 3 load instructions are needed to read the values from memory and 3 store instructions are needed to write the starting address, end address, and duration for each region, this equates

Table 4.1: Overheads for dynamic and duration-based locking

Benchmark	Dyn. Locking Lock/Unlock Instr.	Dyn. Locking Overhead Impact	Dur. Locking Overhead Impact
qsort	292	1%	1%
dijkstra	220	1%	1%
sha	194	6%	1%
rij	170	1%	1%
b.math	212	0%	1%
fft	412	22.8%	8%
disparity	2134	3%	1%
clustering	778	1%	1%
pca	712	1%	1%
mot.est	462	2%	1%
multicut	1616	11.9%	1%
mser	802	71.3%	-

to $6K = 48$ instructions for any benchmark. We employ this abstract performance model because implementing and evaluating the entire pipeline for a dynamic locking approach necessitates complex compiler adjustments. For the dynamic approach, two instructions must be added at loop entry/exit points to lock/unlock each selected cache line; as shown in the table, the resulting number of instructions is in all cases larger than for duration-based and varies depending on the benchmark size and the number of outermost loops it exhibits. Finally, we evaluate the impact of the overhead on the performance gains achieved by each method. We assume a platform with an L1 miss penalty of 10 cycles and otherwise one cycle per instruction; the table thus reports the ratio of the overhead in number of instructions divided by the performance gain in number of saved cache misses multiplied 10. In the duration-based approach, we observe an 8% reduction in performance for *fft* (note that *fft* shows marginal benefits in Figure 4.3), while for all other benchmarks the decrease is negligible. For dynamic-based locking, a more significant performance reduction of 6%, 22.8%, 3%, 2%, 11.9%, and 71.3% was encountered for *sha*, *fft*, *motion – estimation*, *multicut*, and *mser* respectively.

4.4 Conclusions

In this chapter, we introduced a novel duration-based locking mechanism that achieves comparable results to dynamic locking without requiring program modifications. Our approach employs a hardware-based mechanism that repurposes LRU age bits to determine their lock status, alongside a heuristic for selecting the locked lines and their locking durations. Evaluation results based on program profiling have shown the potential of our approach which on average improved the miss rate of a 4-way 2kB L1 I-cache by 26% while paying a memory overhead of 5.6%.

Chapter 5

Conclusions and Future Work

In this thesis, we explore the challenges of maximizing average performance while honoring the timing requirements imposed by high-criticality tasks. Accordingly, we propose two contributions, one at the memory arbitration level and another one at the cache controller level.

At the memory arbitration level, this thesis proposes DAMA, a dual arbitration mechanism that provides an upper bound on the cumulative processing latency of *LTC* requests without degrading the performance of tasks issuing *NLTC* requests. The need to enforce a per-request latency bound in Duetto [54] requires a complex mechanism that monitors the resource state and performs intricate mathematical operations to determine whether a request meets its deadline. This complexity makes hardware implementation challenging due to critical path constraints and verification concerns. In contrast, our proposed DAMA provides a bound on the cumulative latency of *LTC* requests rather than enforcing per-request bounds. Under this scheme, checking for violations is significantly simpler and involves only basic logic elements, such as a counter and a comparator for each requestor. As part of our future work, we aim to integrate this arbitration scheme into a DRAM or SRAM memory controller and implement it in RTL.

Our proposed approach addresses the challenge of maintaining a latency bound for *LTC* requests in mixed criticality systems which is crucial, especially for high-criticality tasks. However, once latency guarantees are met, it becomes equally important to regulate the bandwidth budget allocated to each core. This is because the amount of memory requests that are sent out by each requestor can significantly impact both the overall performance and fairness of the system. Such issues may arise with *NLTC* requests that are not accounted for when DAMA operates in RTA mode. Although these requests are not

latency-critical, they may still originate from high-criticality tasks via DMAs or accelerators. Without controlling how frequently memory requests are issued, certain cores may unfairly dominate the available bandwidth, leading to inefficiencies and potential starvation for high-criticality tasks. As part of our future work, we plan to extend our approach by not only maintaining latency bounds but also regulating the bandwidth allocation for cores executing workloads with varying latency and throughput requirements. This will help ensure a more balanced and efficient use of system resources.

At the cache controller level, this thesis proposes a novel duration-based cache-locking mechanism. This mechanism achieves comparable results to the dynamic locking approach without requiring program binary modifications and with minimal area overhead. Our proposed approach relies primarily on profiling to understand the program’s behavior prior to runtime. This is well-suited for embedded systems, where workloads are typically simple and exhibit predictable behavior. However, in general-purpose systems, workloads tend to be less predictable as they are designed to handle a diverse range of tasks simultaneously. The reliance on offline profiling in our heuristic limits the applicability of the locking approach in such systems. Addressing this limitation may require adopting online profiling techniques that leverage run-time information provided by performance monitoring units such as the one described in [60].

In our preliminary experiments, we applied the same duration-based locking mechanism to data caches. The results demonstrated only marginal performance benefits. This is primarily due to the complex access patterns of data caches, as highlighted in [83]. Specifically, programs often access multiple conflicting data blocks within short time intervals, rendering cache line locking ineffective and, in some cases, detrimental to system performance. As part of our future work, we plan to explore program transformation techniques, such as those proposed in [83], to make programs more locking-friendly. Once transformed, we aim to apply our duration-based approach to evaluate its impact on L1 data caches. Additionally, similar techniques could be extended to the L2 cache and applied to alternative replacement policies, such as pseudo-LRU. We also intend to integrate our proposed mechanism into a static analysis tool to assess its effects on worst-case execution time (WCET).

References

- [1] Tosiron Adegbija and Ann Gordon-Ross. Phase-based cache locking for embedded systems. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 115–120, 2015.
- [2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.
- [3] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 58(3):358–398, 2022.
- [4] Richard L Alena, John P Ossenfort, Kenneth I Laws, Andre Goforth, and Fernando Figueroa. Communications for integrated modular avionics. In *2007 IEEE Aerospace Conference*, pages 1–18. IEEE, 2007.
- [5] Kapil Anand and Rajeev Barua. Instruction cache locking inside a binary rewriter. In *Proceedings of the 2009 Conference on Compilers, architecture, and synthesis for embedded systems*, pages 185–194, 2009.
- [6] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1):6–26, 2002.
- [7] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, 1991.
- [8] Michael Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367. IEEE, 2019.

- [9] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–18. IEEE, 2022.
- [10] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1121–1137, 2021.
- [11] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8. IEEE, 2015.
- [12] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020.
- [13] Mark Jay Charney. *Correlation-based hardware prefetching*. Cornell University, 1995.
- [14] William Y Chen, Scott A Mahlke, Pohua P Chang, and Wen-mei W Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 69–73, 1991.
- [15] Trishul M Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, 2002.
- [16] Nicolas Dagieu, Alexander Spyridakis, and Daniel Raho. Memguard: A memory bandwidth management in mixed criticality virtualized systems memguard kvm scheduling. In *10th Int. Conf. on Mobile Ubiquitous Comput., Syst., Services and Technologies (UBICOMM)*, pages 21–27, 2016.
- [17] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

- [18] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Adonios Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(3):279–291, 2006.
- [19] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.
- [20] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS’14)*, 2014.
- [21] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.
- [22] Babak Falsafi and Thomas F Wenisch. *A primer on hardware prefetching*. Springer Nature, 2022.
- [23] Jamie Garside and Neil C Audsley. Wcet preserving hardware prefetch for many-core real-time systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, pages 193–202, 2014.
- [24] Giovanni Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Surveys (CSUR)*, (2):1–36, 2015.
- [25] Giovanni Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):1–36, 2015.
- [26] Danlu Guo and Rodolfo Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 247–258. IEEE, 2017.
- [27] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of fourth IEEE international workshop on workload characterization. WWC-4*, pages 3–14. IEEE, 2001.

- [28] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [29] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316. IEEE, 2015.
- [30] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. Pmc: A requirement-aware dram controller for multicore mixed criticality systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):1–28, 2017.
- [31] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsoes. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.
- [32] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468. IEEE, 2018.
- [33] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 257–261. IEEE, 2008.
- [34] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.
- [35] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.
- [36] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM computer architecture news*, 38(3):60–71, 2010.
- [37] Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *2014 IEEE Real-Time Systems Symposium*, pages 207–217. IEEE, 2014.

- [38] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, 1997.
- [39] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI):364–373, 1990.
- [40] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on selected Areas in Communications*, 9(8):1265–1279, 1991.
- [41] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 2008.
- [42] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragnathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.
- [43] Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H Anderson, and F Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53:709–759, 2017.
- [44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.
- [45] Wafic Lawand and Rodolfo Pellizzoni. Duration-based instruction cache locking. In *2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 85–90. IEEE, 2024.
- [46] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. Prefetch-aware memory controllers. *IEEE Transactions on computers*, 60(10):1406–1430, 2011.
- [47] Yun Liang and Tulika Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference*, pages 344–349, 2010.
- [48] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.

- [49] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikanth Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.
- [50] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, 1996.
- [51] Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.
- [52] Silvano Martello and Michele Monaci. Algorithmic approaches to the multiple knapsack assignment problem. *Omega*, 90:102004, 2020.
- [53] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on next generation real-time embedded systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [54] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1136–1141. IEEE, 2021.
- [55] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duomc: Tight dram latency bounds with shared banks and near-cots performance. In *Proceedings of the International Symposium on Memory Systems*, pages 1–16, 2021.
- [56] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Parallelism-aware high-performance cache coherence with tight latency bounds. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- [57] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.
- [58] Sparsh Mittal. A survey of techniques for cache locking. *ACM Trans. on Design Automation of Electronic Systems*, 21(3):1–24, 2016.

- [59] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657. IEEE, 2018.
- [60] Mohammed Sajjad Jafri Mohammed Sajjad Jafri. A centralized system performance monitoring infrastructure. Master’s thesis, University of Waterloo, 2024.
- [61] Todd C Mowry, Monica S Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices*, 27(9):62–73, 1992.
- [62] Roberto Muscedere, Vassil Dimitrov, Graham A Jullien, and William C Miller. Efficient techniques for binary-to-multidigit multidimensional logarithmic number system conversion using range-addressable look-up tables. *IEEE Transactions on Computers*, 54(3):257–271, 2005.
- [63] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. *ACM SIGARCH Computer Architecture News*, 36(3):63–74, 2008.
- [64] Fan Ni, Xiang Long, Han Wan, and Xiaopeng Gao. Using basic block based instruction prefetching to optimize wcet analysis for real-time applications. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 459–466. IEEE, 2012.
- [65] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143. IEEE, 2012.
- [66] Parag Panda, Geeta Patil, and Biju Raveendran. A survey on replacement strategies in cache memory for embedded systems. *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pages 12–17, 2016.
- [67] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.
- [68] Sascha Plazar, Jan C Kleinsorge, Peter Marwedel, and Heiko Falk. Wcet-aware static locking of instruction caches. In *Proceedings of the Tenth Int. Symposium on Code Generation and Optimization*, pages 44–52, 2012.

- [69] Louis-Noël Pouchet and Tomofumi Yuki. Polybench/c. URL: <http://web.cse.ohio-state.edu/~pouchet>, 2, 2016.
- [70] Isabelle Puaut and Alexis Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*. Citeseer, 2006.
- [71] Keni Qiu, Mengying Zhao, Chun Jason Xue, and Alex Orailoglu. Branch prediction-directed dynamic instruction cache locking for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):1–24, 2014.
- [72] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Maurer. Look mum, no vm exits!(almost). *arXiv preprint arXiv:1705.06932*, 2017.
- [73] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2):128–138, 2000.
- [74] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.
- [75] Muhammad Refaat Soliman and Rodolfo Pellizzoni. Wcet-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2017.
- [76] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS’05)*, pages 10–pp. IEEE, 2005.
- [77] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.
- [78] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64. IEEE, 2009.

- [79] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64. IEEE, 2009.
- [80] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, 2011.
- [81] Hui Wu, Jingling Xue, and Sri Parameswaran. Optimal wcet-aware code selection for scratchpad memory. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 59–68, 2010.
- [82] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383. IEEE, 2013.
- [83] Chun Xue, Keni Qiu, Weigong Zhang, Jing Wang, Yuanchao Xu, and Mengying Zhao. Data re-allocation enabled cache locking for embedded systems. *Journal of Systems Architecture*, 77:3–13, 2017.
- [84] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.
- [85] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.
- [86] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2015.
- [87] Wenguang Zheng and Hui Wu. Wcet-aware dynamic d-cache locking for a single task. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, pages 1–10, 2015.

- [88] Wenguang Zheng, Hui Wu, and Qing Yang. Wcet-aware dynamic i-cache locking for a single task. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1–26, 2017.
- [89] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: policing core memory bandwidth from outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248. IEEE, 2023.