Enhancing Security of Modern Computing Systems

Dissertation submitted in partial fulfilment of the requirements
for the award of the degree of

Master of Technology

by

Rishabh Ravi

(Roll No. 200260041)

Under the Supervision of

Prof. Virendra Singh

Specializing in Electronic Systems

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

June, 2025

Dedicated to my beloved parents.

Thesis Approval

This dissertation entitled **Enhancing Security of Modern Computing Systems** by **Rishabh Ravi**, Roll No. 200260041, is approved for the degree of **Master of Technology** from the Indian Institute of Technology Bombay.

.....	.....
Prof. Sachin Patkar	Prof. Sarath Babu
(Examiner 1)	(Examiner 2)
.....	.....
Prof. Virendra Singh	Prof. Sachin Patkar
(Supervisor)	(Chairman)

Date: 28/06/2025

Place: Department of Electrical Engineering, IIT Bombay

Certificate

This is to certify that the dissertation entitled **"Enhancing Security of Modern Computing Systems"**, submitted by **Rishabh Ravi** to the Indian Institute of Technology Bombay, for the award of the degree of **Master of Technology** in Electrical Engineering, is a record of the original, bona fide research work carried out by him under our supervision and guidance. The dissertation has reached the standards fulfilling the requirements of the regulations related to the award of the degree.

The results contained in this dissertation have not been submitted in part or in full to any other University or Institute for the award of any degree or diploma to the best of our knowledge.

.....

Prof. Virendra Singh
Department of Electrical Engineering,
Indian Institute of Technology Bombay.

Declaration

I declare that this written submission represents my ideas in my own words. Where others' ideas and words have been included, I have adequately cited and referenced the original source. I declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated, or falsified any idea/data/-fact/source in my submission. I understand that any violation of the above will cause disciplinary action by the Institute and can also evoke penal action from the source which has thus not been properly cited or from whom proper permission has not been taken when needed.

. . . . . .

Rishabh Ravi
Roll No.: 200260041
Date: 28^{th} June 2025
Place: IIT Bombay

Abstract

Security and performance are two critical considerations in the design of modern chip multiprocessors. Among various data leakage threats, side-channel attacks have emerged as particularly stealthy and effective, exploiting micro-architectural behaviors to extract sensitive information. Designing secure architectures often entails a trade-off with performance, sometimes directly and other times indirectly through the allocation of chip area that could otherwise be used to enhance throughput.

This thesis examines existing side-channel attack techniques with a dual focus: improving their performance to better understand their efficiency and identifying latent vulnerabilities in modern processor designs. In addition, we explore the reverse scenario—dedicating chip area solely for performance improvements—and analyze the potential security implications of such design choices. By studying this interplay between performance optimization and side-channel resilience, we aim to contribute to a deeper understanding of how to co-design secure and high-performing processors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In the current era of information, data security is a critical concern. Modern high-performance computing systems use multiple cores, multi-level cache hierarchies, and various performance-enhancing techniques. These optimized chip multiprocessors (CMPs) often overlook security vulnerabilities in hardware. Modern attacks not only exploit software vulnerabilities but also take advantage of hardware vulnerabilities Yarom and Falkner (2014); Gruss et al. (2016); Liu et al. (2015); Shusterman et al. (2019); Kocher et al. (2019); Lipp et al. (2018). Some attacks, such as flush-based Yarom and Falkner (2014); Gruss et al. (2016), contention-based Liu et al. (2015), and cache-occupancy-based Shusterman et al. (2019) attacks, exploit weaknesses in the shared last-level cache (LLC). There is a significant difference between the access latency of LLC and that of main memory. An attacker can exploit this timing difference by controlling the shared LLC to infer sensitive information from a victim process.

Contention-based and occupancy-based attacks exploit the shared nature of the LLC. In these attacks, the attacker process fills a specific target set or the entire LLC with its own data. This causes the victim process to fetch its data from main memory. The fetched data is then cached in the shared LLC, evicting some of the attacker's data. Later, the attacker accesses its data and measures the access time. By analyzing these timings, the attacker can determine how much of their data was evicted. This allows the attacker to infer the memory access behavior of the victim process and extract sensitive information. Flush-based attacks require a shared LLC and a shared library between the attacker and victim processes. In these attacks, the attacker uses the `clflush` (Cache Line Flush) instruction to evict shared library cache lines from the cache hierarchy. If the victim process accesses the flushed cache line, it is reloaded into the LLC. The attacker then measures the access latency of the same cache line. A shorter access time indicates that the victim accessed it (LLC hit), while a longer access time means it was not accessed (LLC miss) and had to be fetched from main memory. Using this timing difference, the attacker can infer the victim's memory access patterns and extract sensitive information.

Techniques like Flush+Reload Yarom and Falkner (2014) and Prime+Probe Liu et al. (2015) exploit cache access timing analysis to extract sensitive data. Flush+Reload involves flushing and subsequently reloading specific cache lines to detect timing differences. Prime+Probe populates the cache with attacker-controlled data and probes for timing variations resulting from cache hits or misses. These attacks infer victim program memory locations based on cache hit/miss latency differences.

Since these attacks exploit the shared LLC, one possible defense is to partition the LLC among cores. However, static partitioning is inefficient. Different applications and their phases have varying LLC requirements. Static partitioning leads to under-utilization of the LLC. Though secure, this approach is not ideal. Several dynamic

cache partitioning (DCP) techniques have been proposed Qureshi and Patt (2006a); Holtryd et al. (2020); Kaseridis et al. (2009); Manikantan et al. (2012). These methods partition the LLC by assigning a certain number of ways in each cache set to each core. The allocated number of ways changes dynamically based on application behavior. Utility-based cache partitioning (UCP) Qureshi and Patt (2006a) is a well-known DCP technique. UCP assigns more cache ways to applications that can get more cache hits, improving LLC utilization. However, UCP is not secure Boran et al. (2022); Holtryd et al. (2023). It introduces new side-channels that attackers can exploit to extract sensitive data. To address these vulnerabilities, several secure DCP techniques have been developed Boran et al. (2022); Wang et al. (2016); Holtryd et al. (2023). PASS-P Boran et al. (2022) improves security by invalidating cache lines before transferring ownership to another core. This makes it secure against flush-based attacks.

PASSP selects clean cache blocks for transfer. This choice minimizes off-chip write-backs, reducing unnecessary memory traffic. As a result, PASS-P reduces the performance loss without compromising on security. While PASS-P selects clean cache blocks as transfer candidates, it overlooks the presence of dead blocks and blocks that do not trigger back invalidations. Dead blocks are those that are not accessed after being inserted into the cache, remaining idle until eviction. On the other hand, blocks that trigger back invalidations generate coherence requests when evicted, resulting in increased network traffic. By prioritizing dead blocks, along with blocks that do not cause back invalidations and clean blocks, system performance can be enhanced without compromising security. However, PASS-P's suboptimal candidate selection process undermines cache efficiency. It often fails to prioritize transfer candidates that minimize system's performance overhead, such as dead blocks and non-back-invalidation-inducing blocks, leading to inefficient cache utilization and

increased performance penalties. This mismanagement significantly harms overall system performance. Additionally, a clean cache block is not necessarily a dead block, nor does it always avoid triggering back invalidations. However, by selecting any clean block as a transfer candidate without verifying its dead status or back invalidation properties, we risk evicting blocks that could still be useful. This could also lead to the eviction of blocks in private caches, which may cause unnecessary back invalidations and increase network-on-chip (NoC) traffic. Such actions result in inefficient cache utilization and higher cache miss rates, thereby degrading system's performance. These inefficiencies are more evident when partition changes occur frequently, as suboptimal eviction decisions accumulate, further degrading performance. Without an optimal transfer candidate selection mechanism that considers re-reference likelihood and coherence impact, PASS-P fails to strike an optimal balance between security and performance.

However, it is still vulnerable to contention-based attacks. Scale uses randomization to protect against information leakage through cache partition decisions, but it also fails to eliminate all side-channels. These methods do not fully secure UCP. They overlook a new side-channel caused by UCP's "transfer on a miss" policy. UCP makes a global partition decision, which applies to all sets of LLC. But it does not transfer the ownership between cores immediately. Instead, UCP uses a "transfer on a miss" policy. When a miss occurs in a set, UCP checks whether the requesting core has fewer ways than the ways allocated by the partition mechanism. If so, it transfers a way from another core that has more ways than its allocation. This transfer happens on a per-set basis and only when there are misses. As a result, ways are transferred gradually and locally. This behavior creates a side channel that leaks sensitive information. Existing secure partitioning techniques do not address this vulnerability.

1.2 Dissertation Contributions and Organization

Given the aforementioned challenges in defending against cache-based side channel attacks, this thesis proposes ways to boost performance while maintaining security.

This dissertation is organised as follows

Chapter 2 - Background and Related Work This chapter begins by introducing the micro-architectural and cache designs. We then introduce the vulnerabilities of the cache hierarchy and associated side channel attacks. We discuss their mitigations and the new vulnerabilities with performance degradations introduced by them.

Chapter 3 - SCAM This chapter introduces a secure high performance, hierarchy-aware cache replacement policy to overcome the performance penalties of secure policies. We observe that the work of PASSP overlooks the presence of dead blocks and blocks that do not trigger back invalidations. SCAM takes this into account and proposes a secure replacement policy that takes this into account while maintaining security. This work has been accepted in SECRYPT 2025.

Chapter 4 - SPARTON This chapter introduces a previously unseen vulnerability in "transfer on a miss" policy in the works of PASSP and UCP. We introduce the threat model. To address this problem, we propose SPARTON, a secure dynamic cache partitioning technique. SPARTON addresses the vulnerability introduced by the policy. We also introduce a transfer candidate selection policy that reduces performance overhead by minimizing back-invalidations and writebacks. This work has been submitted to ICCD 2025.

Chapter 5 - Conclusion This chapter summarizes the results of the thesis and discusses future research directions.

Chapter 2

Literature Review

This chapter introduces micro-architectural and cache designs. We then introduce the vulnerabilities of the cache hierarchy and associated side channel attacks. We discuss their mitigations and the new vulnerabilities with performance degradations introduced by them.

2.1 MODERN PROCESSOR AND CACHE ORGANIZATION

2.1.1 Chip-Multiprocessors (CMP)

Modern Chip-Multiprocessors (CMPs) consist of multiple cores and a multi-level cache hierarchy. Typically, each core has one or two levels of private cache, while a shared LLC is accessible by all cores. When a cache block is not found in the LLC, it must be fetched from off-chip main memory, which has much higher access latency than the LLC. This timing difference, combined with the shared nature of the LLC,

enables various cache timing side-channel attacks. Contention-based attacks exploit the shared LLC by allowing the attacker process to fill a cache set with its own data, evicting the victim's data. To prevent such cross-evictions, cache partitioning can be used as a strategy to improve LLC security against timing side-channel attacks.

2.1.2 Cache Partitioning

The LLC is typically shared across multiple cores. In an unpartitioned LLC, LLC accesses from different processes running on different cores are treated equally. This can lead to cache lines from one core being evicted by another. In cases where cache lines are shared, one core may benefit from data brought into the LLC by another core. Such interference and sharing can be exploited by attackers to carry out timing side-channel attacks. A potential defense against this is way-partitioning. In a way-partitioned LLC, each core is assigned a fixed number of cache ways in every set. Cache lines of a core are stored only in its allocated ways, ensuring isolation between processes running on different cores. To enforce this isolation, ownership bits are added to each cache line to indicate which core it is assigned to. Each partition maintains an independent recency stack and applies its own replacement policy, without considering the state of other partitions.

2.1.3 Dynamic Cache Partitioning (DCP)

A statically partitioned cache does not utilize the LLC efficiently, as it cannot adapt to the changing behavior of applications. To improve LLC utilization, Dynamic Cache Partitioning (DCP) techniques are employed. DCP policies dynamically adjust the LLC partition based on application behaviour. UCP Qureshi and Patt (2006a) is a well-known DCP technique that partitions the LLC based on utility. In

UCP, utility is defined as the number of additional cache hits an application would achieve if given more cache ways. Based on this metric, UCP allocates more cache ways to the higher-utility applications. When this allocation changes, the ownership of cache ways is not updated immediately. Instead, to maximize LLC utilization, UCP uses a “transfer on a miss" policy. Under this policy, the ownership of cache ways in a specific set changes only when a miss occurs for a core that holds fewer ways than its global allocation. At that point, a way from a core holding more ways than its global allocation is transferred. In each case, the least recently used (LRU) cache line is selected as the transfer candidate. This gradual, per-set transfer allows the core losing ways to continue benefiting from them until it is strictly necessary to reassign ownership. While this method improves overall LLC efficiency, it creates a new side channel that can be exploited by attackers.

2.2 CACHE-BASED SIDE CHANNEL ATTACKS

Although UCP optimizes cache utilization, the flexibility of dynamically changing the partition introduces a critical side-channel vulnerability, enabling data leaks Boran et al. (2022). During partition changes, cache way ownership gets transferred from a core with lower utility to one with higher utility. This transfer creates an opportunity to obtain sensitive data by mounting attacks such as FLUSH+RELOAD and PRIME+PROBE, as detailed later in this section. Preventing these exploits without significantly degrading performance is a major challenge. PASS-P identifies and mitigates this security flaw in UCP Boran et al. (2022). To address this vulnerability, PASS-P invalidates cache lines before transferring ownership. This ensures that sensitive data is not exposed during partition changes. However, such forced invalidation of a cache way can introduce performance degradation for benign

application mixes. To address this, PASSP selects clean cache blocks for transfer. This choice minimizes off-chip writebacks, reducing unnecessary memory traffic. As a result, PASS-P reduces the performance loss without compromising on security

2.2.1 FLUSH+RELOAD

Dynamic cache partitioning schemes are susceptible to Flush+Reload Yarom and Falkner (2014)

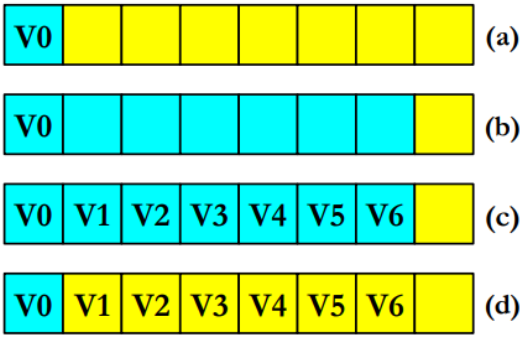

FIGURE 2.1: Flush+Reload Attack

Flush: The attacker increases its utility to displace all But one cache line from each set, subsequently flushing the retained lines as depicted in step (a) of 2.1

Execute: The attacker then decreases its utility to restore the flushed lines, awaiting the victim process's execution, as illustrated in steps (b) and (c).

Reload: The attacker increases its utility again to occupy all but one line in each set and reloads specific addresses of interest, as shown in step (d). The presence of cache hits or misses on these addresses indicates the victim's memory accesses.

In Flush+Reload attacks, both the attacker and victim must share the same code library for the attack to be effective. This sharing enables the attacker to achieve a

cache hit during the Reload phase for addresses accessed by the victim during the Execute phase. Despite dynamic cache partitioning protocols generally preventing any single process from occupying all cache lines, thus avoiding starvation, attackers can still extract sensitive information over multiple iterations. Attackers often employ techniques to significantly slow down the victim's execution, such as launching a denial-of-service attack on the Linux Completely Fair Scheduler (CFS), as detailed by Gullasch et al.(Gullasch et al., 2011). SecDCP Wang et al. (2016) distinguishes processes into confidential and public categories, focusing on safeguarding confidential applications from side-channel attacks. However, SecDCP is still susceptible to Flush+Reload attacks. It only invalidates cache lines transferred from a public to a confidential application if those lines were accessed by the public application. Consequently, lines retrieved by the public attacker during the Reload phase are not invalidated, allowing the attacker to deduce information about the victim's memory accesses. Furthermore, SecDCP does not adjust its partitioning decisions based on the demand of confidential applications, resulting in suboptimal cache partitioning and associated performance degradation

2.2.2 PRIME+PROBE

FIGURE 2.2: PRIME+PROBE Attack

Prime: The attacker increases its utility to evict all but one cache line from each set and populates these lines with its own data, as shown in step (a) of Fig.2.2 **Execute:** The attacker then decreases its utility to restore the evicted lines, allowing the victim to perform its operations, as illustrated in steps (b) and (c) of Fig.2.2 **Probe:** Subsequently, the attacker increases its utility again to evict all but one line from each set and reloads the addresses that were previously primed, as depicted in step (d). The presence of a cache hit or miss on these addresses reveals information about the victim's memory accesses.

2.3 Mitigating Side-Channels

Cache randomization methods aim to obscure the mapping between memory addresses and cache locations, making it challenging for attackers to infer sensitive information through predictable access patterns. One notable randomization technique is **RPCache** Wang and Lee (2006), which introduces a permutation table to randomize cache indexing. This design breaks deterministic cache behavior, reducing an attacker's ability to infer data access patterns. Similarly, **CEASER** Qureshi (2018) leverages a low-latency block cipher to encrypt cache indices, ensuring unpredictable address-to-cache-line mapping. By periodically refreshing encryption keys, CEASER further reduces the predictability of cache access patterns, enhancing security.

While randomization techniques effectively mitigate side-channel risks in static environments, they struggle in dynamic workloads where frequent cache modifications may expose timing differences. Moreover, randomization alone cannot fully prevent contention-based attacks in shared caches. To mitigate side-channel risks in shared

caches, **cache partitioning** isolates cache resources between processes, ensuring that one process cannot influence or infer data from another.

As a defense against cache timing side-channel attacks through shared last-level caches (LLCs), prior work has extensively explored LLC isolation techniques. MI6 Bourgeat et al. (2019) and IRONHIDE Omar and Khan (2020) enforce static partitioning of the LLC into secure and non-secure regions to ensure isolation. However, their static nature leads to underutilization of the cache and an inability to adapt to varying application behaviors. OPTIMUS Omar et al. (2020) introduces dynamic partitioning across applications but continues to use static partitions for each application.

For example, SecDCP (Wang et al., 2016) categorizes processes into confidential and public groups, offering targeted protection for sensitive processes. However, this approach remains vulnerable to specific attacks, such as the Flush+Reload technique, as discussed in Section.3. This compromises the security guarantees of SecDCP, leaving critical gaps in its defense mechanisms.

Other methods, such as COTSknight (Yao et al., 2019) and DAWG (Kiriansky et al., 2018), also enhance cache security but introduce additional complexities. COTSknight uses advanced cache monitoring and allocation mechanisms to detect suspicious behaviors. While effective in some scenarios, it fails to mitigate Flush+Reload attacks entirely and incurs a performance penalty of up to 5% compared to an insecure baseline. DAWG, on the other hand, isolates cache accesses through secure way partitioning. However, this approach leads to performance slowdowns ranging from 0% to 15%, depending on the workload, when compared to an approximate LRU baseline.

PASS-P Boran et al. (2022) and SCALE Holtryd et al. (2023) improve the security of the insecure utility-based cache partitioning (UCP) scheme Qureshi and Patt (2006a). While they provide better isolation than baseline UCP, they fail to mitigate the side-channel introduced by UCP's "transfer on a miss" policy. DAWG Kiriansky et al. (2018) proposes a secure, software-hardware co-designed dynamic partitioning approach. However, since partition changes are managed in software, they incur higher overhead and latency.

For L1 caches, methods like NoMo (Domnitser et al., 2012) offer a tunable tradeoff between performance and security without requiring software or operating system modifications. NoMo makes minor changes to cache replacement policies to mitigate side-channel risks. However, its fully secure configuration aligns with static partitioning, leading to performance degradation of up to 5% and an average slowdown of 1.2% compared to LRU.

2.4 Summary

This chapter surveyed existing techniques for mitigating side-channel attacks in chip-multiprocessors. While these methods enhance security, they often overlook performance considerations. The next chapter introduces a novel approach that aims to maintain strong security guarantees while reducing performance overhead.

Chapter 3

SCAM

This chapter introduces SCAM, an advanced cache management strategy designed to optimize transfer candidate selection. SCAM focuses on prioritizing dead blocks, as well as blocks that neither trigger back invalidations nor cause writebacks. This ensures high performance, without compromising on security. By employing a hierarchical transfer candidate selection policy, SCAM effectively reduces LLC misses, alleviates NoC congestion, and minimizes unnecessary memory overhead.

3.1 Motivation

In this section, we examine the limitations of PASS-P Boran et al. (2022) and identify the research gaps that the proposed **SCAM** framework addresses. While PASS-P successfully provides security akin to static partitioning, it exhibits several shortcomings in its transfer candidate selection process, as detailed in the previous section. These limitations negatively affect the overall performance of the system. We begin

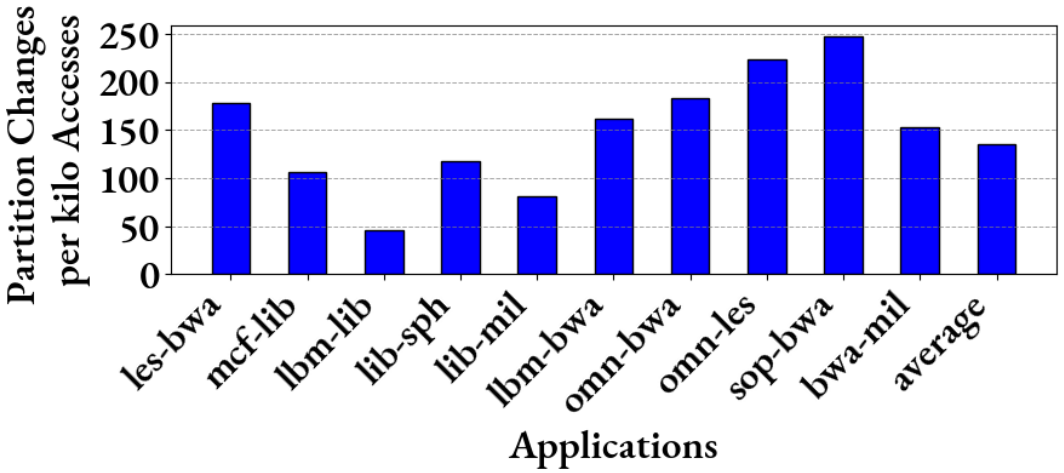

FIGURE 3.1: Number of Instances Partition Changes

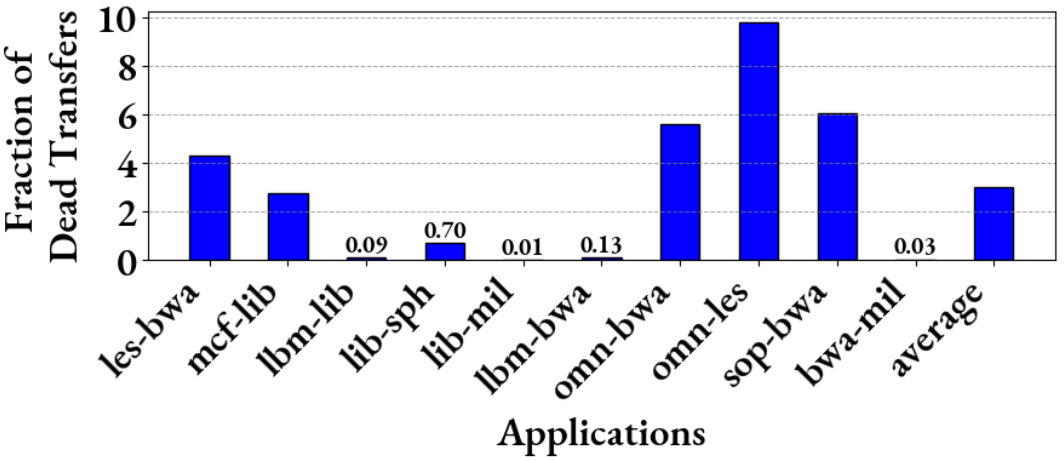

FIGURE 3.2: Fraction of PASS-P transfer candidate that is not dead

by analyzing these shortcomings of PASS-P, followed by a discussion of the techniques incorporated into SCAM to mitigate these weaknesses and enhance system performance.

Before addressing the issue of suboptimal transfer candidate selection, we first analyze the frequency of LLC cache partition changes to emphasize the need for an optimal transfer candidate selection scheme. We conduct an experiment by modeling PASS-P in the Sniper simulator Carlson et al. (2014) and evaluate it using the application mixes listed in Table 3.1. The system configuration used for the

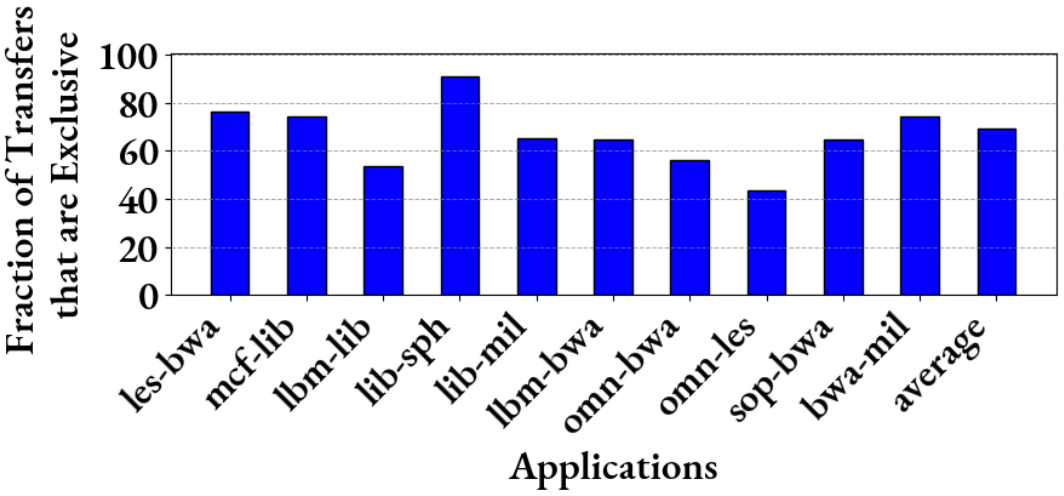

FIGURE 3.3: Fraction of PASS-P transfer candidate that does not generate Back Invalidations

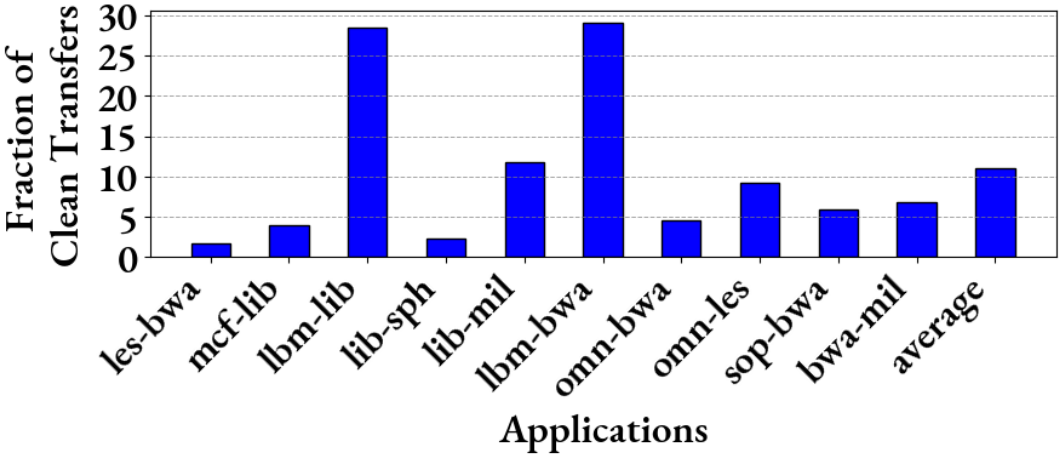

FIGURE 3.4: Fraction of PASS-P transfer candidate that is dirty

evaluation is shown in Table 3.2. Our evaluation methodology follows the approach discussed in Section 3.3. Figure 3.1 presents the results of this experiment, with the x-axis representing the different application mixes and the y-axis showing the number of LLC cache partition changes per kilo LLC accesses. From Figure 3.1, we observe that the frequency of cache partition changes in PASS-P is considerably high. This highlights the critical need for an optimal transfer candidate selection scheme to reduce performance degradation.

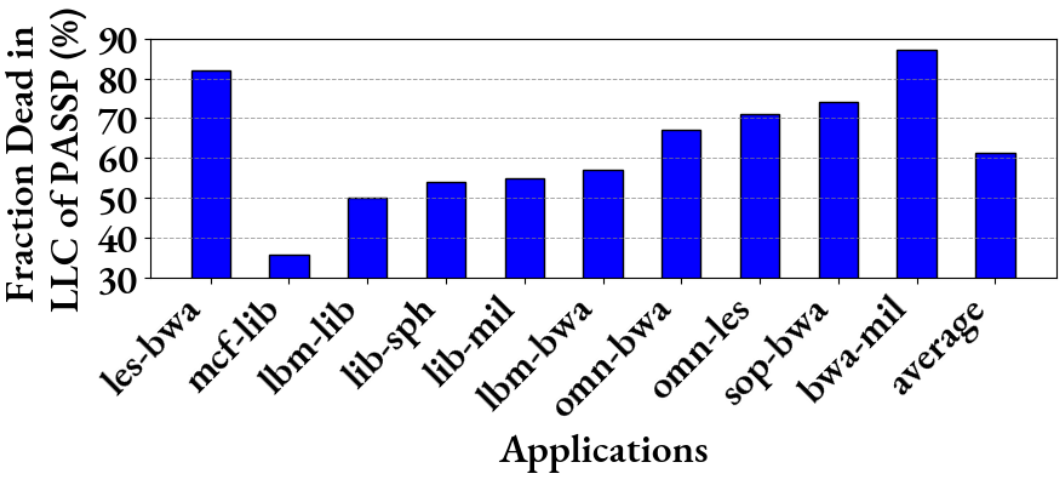

FIGURE 3.5: Percentage of Dead Cacheblocks Residing in LLC

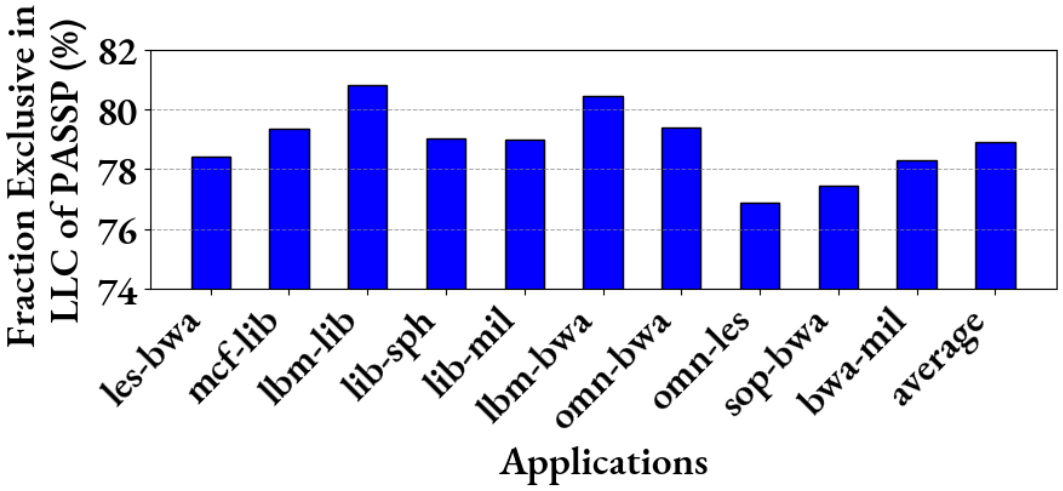

FIGURE 3.6: Percentage LLC Cacheblocks residing in Private Caches

Now to further examine the severity of issues with PASS-P, we conducted an experiment to analyze the types of cache blocks PASS-P transferred during all these partition changes at LLC. We classify the transferred cache blocks into three classes:

1. **Dead blocks**: Dead blocks are those that are not accessed again after being inserted into the cache until eviction.

2. **Blocks causing back invalidations**: Blocks causing back invalidations are

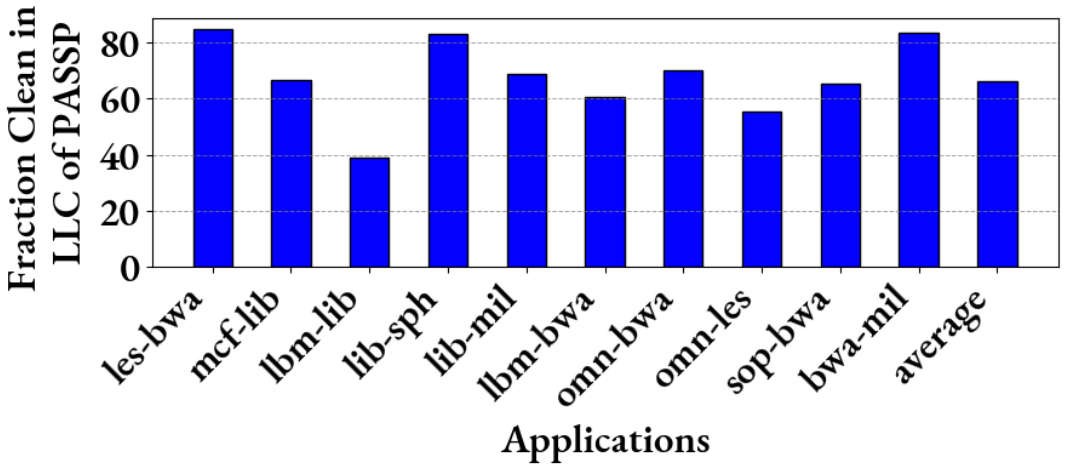

FIGURE 3.7: Percentage of Clean Cacheblocks residing in LLC

those that, when evicted, generate coherence requests to private caches, increasing network traffic. These cache blocks reside in both private and shared caches.

3. **Clean blocks**: Clean blocks are unmodified and do not require writebacks when evicted.

A comprehensive empirical analysis reveals significant limitations in PASS-P's algorithm for selecting transfer candidates. Figures 3.2, 3.3, and 3.4 illustrate the frequency at which PASS-P evicts suboptimal cache blocks, including non-dead blocks, blocks that trigger back invalidations, and dirty blocks. The simulation environment and evaluation methodology are consistent with those used in the previous experiment. To generate the plots shown in Figures 3.2, 3.3, and 3.4, we incorporated counters into the simulator to track the number of non-dead blocks, dirty blocks, and blocks that trigger back invalidations that got transferred due to PASS-P. Additionally, the necessary infrastructure for detecting the properties of cache blocks was introduced as described in Section 4.2.

Figure 3.2 shows that PASS-P evicts non-dead blocks in approximately 3% of cases. This suboptimal decision increases LLC miss rates, leading to performance degradation as frequent cache block insertions occur at the LLC level. Furthermore, as depicted in Figure 3.3, PASS-P selects blocks that trigger back invalidations in nearly 30% of instances. This behavior generates unnecessary coherence traffic in the network-on-chip (NoC), further stressing system resources. In addition, Figure 3.4 reveals that PASS-P evicts dirty blocks in 8% of cases when no clean blocks are available within the allocated cache ways. This results in unnecessary writebacks to main memory, adding additional overhead and further diminishing performance.

The previous analysis indicates that PASS-P tends to select suboptimal transfer candidates. However, it is also necessary to investigate whether optimal transfer candidates exist in a set during a cache miss and partition change. To address this, we performed an experiment using the same setup described earlier to analyze the distribution of different cache block properties within the LLC when PASS-P is in operation. Figures 3.5, 3.6, and 3.7 display the distribution of various cache block types—dead blocks, blocks that cause back invalidations, and clean blocks—within the LLC under the PASS-P scheme. Figure 3.5 reveals a substantial number of dead blocks in the LLC. These dead blocks, which have already completed their useful lifetime, are ideal candidates for eviction. Evicting dead blocks would significantly reduce performance degradation by minimizing unnecessary cache insertions and replacements. Furthermore, Figure 3.6 illustrates that a considerable fraction of cache blocks trigger back invalidations when evicted. These blocks generate coherence traffic, which increases network congestion and puts additional strain on the system's resources. Such traffic can contribute to system bottlenecks and reduce overall performance.

In addition, Figure 3.7 shows that a large portion of the LLC is occupied by clean

blocks. These clean blocks can be evicted without incurring additional writebacks to main memory, which would otherwise add unnecessary system overhead. Given these observations, it becomes clear that a more effective eviction strategy is required—one that prioritizes dead blocks for eviction while minimizing the impact of blocks that cause back invalidations and unnecessary writebacks. A refined selection approach can not only enhance cache utilization but also improve overall system performance, all while maintaining the security guarantees in dynamically partitioned cache systems. Thus, optimizing the transfer candidate selection process becomes essential to achieving an efficient balance between performance and security.

To address the limitations identified in PASS-P, we introduce SCAM, an advanced cache management strategy designed to optimize transfer candidate selection. SCAM focuses on prioritizing dead blocks, as well as blocks that neither trigger back invalidations nor cause writebacks. This ensures high performance, without compromising on security. By employing a hierarchical transfer candidate selection policy, SCAM effectively reduces LLC misses, alleviates NoC congestion, and minimizes unnecessary memory overhead. As a result, SCAM significantly boosts overall system performance, outperforming PASS-P in both efficiency and effectiveness.

In summary, SCAM marks a substantial improvement in secure cache management for shared LLCs in multi-core systems. It overcomes the key limitations of PASS-P by refining the transfer candidate selection process and optimizing cache block eviction strategies. SCAM not only enhances system performance but does so without compromising security. This makes it a more efficient and resilient solution for cache management in dynamic partitioning systems. In the following section, we will delve into the details of SCAM's design and implementation.

3.2 Proposal

UCP is a dynamic cache management technique that optimizes shared cache resources in multi-core systems by adjusting last-level cache (LLC) partitioning based on workload demands. This flexible partitioning prevents cache contention, enhances data locality, and reduces conflict misses. Unlike static schemes, UCP reallocates cache space in real-time, balancing performance and energy efficiency while improving system throughput, especially in multi-application environments. However, UCP's dynamic reallocation introduces a vulnerability. Transferring cache ownership between cores can expose memory access patterns, allowing attackers to exploit techniques like FLUSH+RELOAD or PRIME+PROBE. This side-channel risk arises from cache line transfers, which inadvertently reveal access patterns.

PASS-P mitigates these vulnerabilities by implementing a cache line invalidation strategy. It ensures that cache lines are invalidated before transferring ownership, preventing timing-based attacks during cache access. While PASS-P effectively secures cache partitioning, it still faces performance challenges related to transfer candidate selection, coherence traffic, and cache utilization.

PASS-P predominantly selects clean cache blocks for eviction, but not all clean blocks are optimal. Some might be useful in the future or could trigger unnecessary back invalidations, leading to increased memory traffic. These inefficient evictions negatively impact performance by increasing cache misses and adding load to the system. SCAM improves upon this by prioritizing dead blocks and blocks that do not cause back invalidations or writebacks, enhancing cache efficiency and reducing system overhead. SCAM achieves better performance without sacrificing security. We will now examine the specifics of SCAM, its role in LLC management, and how it facilitates efficient cache management while maintaining security.

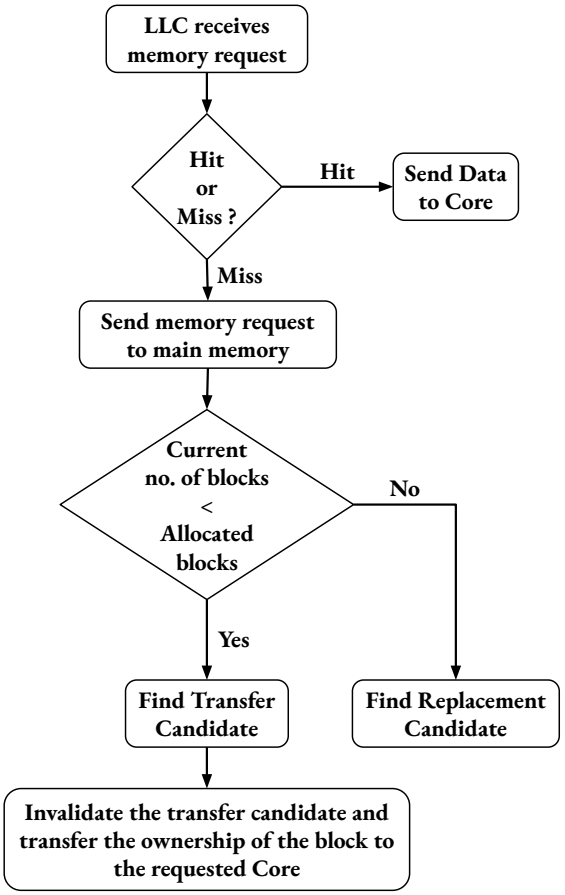

FIGURE 3.8: LLC Access Flowchart

3.2.1 SCAM: Secure Shared Cache Partitioning Scheme

Figure 3.8 illustrates the process of accessing the last-level cache (LLC) and the role of SCAM in optimizing LLC cache management scheme. When an LLC receives a memory access request, it first determines whether the requested data is present in the cache. In the case of a cache hit, the data is sent to the requesting core, while also being forwarded to higher-level caches to maintain inclusivity. Conversely, if an LLC miss occurs, the request is forwarded to the main memory, and the retrieved data must be inserted into the LLC while also being accessed by the core.

Before inserting new data into the LLC, several conditions must be evaluated to ensure efficient cache utilization. First, it is necessary to verify whether the number

of cache ways currently owned by the requesting core is less than its allocated quota, as determined by the dynamic cache partitioning (DCP) scheme's decision engine. If the core has already reached its allocation limit, a cache block from the corresponding set must be evicted based on the replacement strategy, creating space for the new block. However, if the number of cache ways owned by the requesting core is below the allocated threshold, ownership of a cache way must be transferred from another core to the requesting core.

This transfer mechanism is precisely SCAM (Secure Shared Cache Partitioning Scheme). SCAM employs a hierarchical selection process, as outlined in *Algorithm* 1, to determine the optimal cache block for eviction. By following a structured decision-making approach, SCAM enhances system's performance without compromising on cache security. Its step-by-step selection strategy effectively optimizes cache utilization, and enhances overall system efficiency. SCAM follows a step-by-step selection strategy as outlined below:

1. **First Priority:** SCAM gives the highest priority to *dead blocks* that are *clean* and do not cause *back invalidations*. A dead block is a cache block that will not be reused in the future. A clean block has not been modified since it was brought into the cache. Evicting such blocks minimizes performance impact and avoids unnecessary back invalidation requests to higher cache levels.

2. **Second Priority:** If no primary candidate is available, SCAM selects any dead block that avoids back invalidations. This ensures efficient eviction without disrupting other levels of the cache hierarchy.

3. **Third Priority:** If no secondary candidate exists, SCAM searches for any *clean block* that avoids back invalidations. Such blocks are evicted to minimize writebacks and performance penalties.

4. **Fourth Priority:** If the above options are unavailable, SCAM searches in the following order of priority: *any dead block, any clean block, or any block that avoids back invalidations.* The first matching block is selected for eviction.

5. **Fallback Candidate:** If none of the above candidates are found, SCAM evicts the *Least Recently Used (LRU)* cache block as a last resort.

To implement this hierarchical process, SCAM scans half of the cache set, starting from the LRU position. This ensures that the most appropriate block is identified for eviction while limiting unnecessary cache disruptions. If no preferred candidate is found during the scan, SCAM defaults to evicting the LRU block. This fallback mechanism ensures robust and consistent cache management under all scenarios. By following this systematic approach, SCAM reduces performance overhead and improves the efficiency of cache operations while maintaining secure and optimal cache partitioning. SCAM uses additional metadata to track the status of cache blocks:

1. **Deadblock Bit**: This bit indicates whether a cache block has been accessed since it was loaded. A value of 1 means the block is "dead" (unreferenced).

2. **Back-invalidation Bit**: This bit shows whether evicting the block will cause back invalidations in upper-level caches. A value of 1 means back invalidations will occur.

When a block is inserted into the cache, the deadblock bit is set to 1, and the back-invalidation bit is also set to 1. If the block is accessed, the deadblock bit is reset to 0, indicating the block is active. When the block is evicted from all private caches, the back-invalidation bit is reset to 0. This metadata allows SCAM to make informed decisions about which blocks to evict or transfer.

Algorithm 1: Transfer Candidate Selection Algorithm

```
1  Function FindTransferCandidate():
       // First Priority
2      for B@LRU to B@(LRU − f × curr_alloc) do
3          if B.NotInPrC and B.Dead and B.Clean then
4              return B
5          end
6      end
       // Second Priority
7      for B@LRU to B@(LRU − f × curr_alloc) do
8          if B.NotInPrC and B.Dead then
9              return B
10         end
11     end
       // Third Priority
12     for B@LRU to B@(LRU − f × curr_alloc) do
13         if B.NotInPrC and B.Clean then
14             return B
15         end
16     end
       // Fourth Priority
17     for B@LRU to B@(LRU − f × curr_alloc) do
18         if B.NotInPrC or B.Dead or B.Clean then
19             return B
20         end
21     end
       // Fallback Candidate
22     return B@LRU
```

3.2.2 Lower Partition Bound

Dynamic cache partitioning schemes, such as SCAM, aim to allocate cache blocks efficiently among all cores in a system. SCAM uses a heuristic that assigns at least one cache block per set in the LLC to each application. However, this allocation method can cause underutilization of private L2 caches. The problem occurs because the LLC's capacity may not be sufficient to fully support the private L2 cache of each core. For example, if a core is allocated only one block per set in the LLC, the total number of usable blocks in the LLC will be smaller than the capacity of the

core's private L2 cache. This mismatch results in wasted L2 cache resources, leading to reduced overall system performance.

To overcome this limitation, SCAM enforces a minimum allocation of two cache blocks per set for every application. By increasing the lower allocation limit, SCAM ensures that the private L2 caches are better utilized. This adjustment minimizes resource wastage and enhances system performance. Through this improved allocation strategy, SCAM achieves a more balanced use of the cache hierarchy, reducing inefficiencies and supporting higher application performance across all cores.

3.2.3 Decision-Making Engine

SCAM employs a decision-making engine to optimize the partitioning of the shared LLC. This engine draws inspiration from the Utility-based Cache Partitioning (UCP) method Qureshi and Patt (2006b). The detailed operation of this engine is described below:

3.2.3.1 Initialization:

In a dual-core system with an m-way shared LLC, the cache is evenly divided between the two cores at the start. Each core is allocated $\frac{m}{2}$ ways. Additionally, two Auxiliary Tag Directories (ATDs) are configured to evaluate alternative partitioning schemes:

- **ATD1:** Simulates a configuration where the first core is allocated $(\frac{m}{2} - 1)$ ways and the second core receives $(\frac{m}{2} + 1)$ ways.
- **ATD2:** Simulates a configuration where the first core is allocated $(\frac{m}{2} + 1)$ ways and the second core receives $(\frac{m}{2} - 1)$ ways.

3.2.3.2 Epoch-Based Evaluation:

The cache controller tracks the number of cache misses during each epoch for the actual LLC and the two ATDs. At the end of each epoch, the controller selects the partitioning configuration (from the actual LLC, ATD1, and ATD2) that results in the fewest misses for the next epoch. To ensure adaptability while retaining historical workload behavior, the miss counters for all configurations are reset to half their current value at the beginning of each epoch.

3.2.3.3 Dynamic Adjustments:

During runtime, the controller validates whether the current set partition aligns with the selected configuration whenever a cache miss occurs. If the set partition does not match, ownership of the cache line is transferred to the other core. These adjustments are only triggered by cache misses, while cache hits leave the existing partitioning unchanged. This approach minimizes unnecessary performance disruptions and preserves the stability of the cache system.

By leveraging this adaptive partitioning mechanism, SCAM ensures that the shared LLC dynamically responds to workload demands while maintaining low performance overhead. In the next section, we evaluate SCAM's performance against PASS-P. The results demonstrate SCAM's ability to deliver enhanced system performance and robust security in dynamic cache partitioning scenarios.

3.3 Evaluation Framework

In this section, we provide a detailed explanation of the simulation environment used for evaluating **SCAM**. We also present a comparative analysis of **PASS-P** and SCAM.

TABLE 3.1: Application list

Application Name	Abbreviation	LLC Miss Rate (in %)	Application Name	Abbreviation	LLC Miss Rate (in %)
leslie 3d	les	80.2	sphinx3	sph	91.56
GemsFDTD	gem	66.99	wrf	wrf	77.95
mcf	mcf	63.49	bwaves	bwa	96.42
soplex	sop	56.51	milc	mil	83.64
lbm	lbm	98.96	omnetpp	omn	58.54

TABLE 3.2: Experimental Framework

Simulator	**Sniper Multi-core Cycle Accurate Simulator**
CPU core	Dual core, 2.67 GHz, 4-wide fetch, 128-entry ROB
L1 I/D Cache	32 KB, 4-way, LRU, private, 4 cycles access time
L2 Cache	512 KB, 8-way, LRU, private, 8 cycles access time
Last Level Cache (LLC)	2 MB per core, 16-way, LRU, shared, 30 cycles access time

To evaluate performance, we used the cycle-accurate **Sniper Simulator** Carlson et al. (2014). This tool allowed us to analyze two benchmarks running concurrently on separate cores. The simulation configurations are summarized in Table 3.2. Our study focuses on benchmark pairs selected from the **SPEC CPU2006** suite Henning (2006). These benchmarks frequently update the UCP partitioning strategy, making them suitable for testing SCAM's dynamic cache management.

For each application, we used traces representing one billion instructions. These traces were identified using **Simpoints** to reflect typical application behavior. Before starting the analysis, the system was warmed up with 200 million instructions to ensure steady-state performance.

The results of our evaluation are shown in Fig. 3.9 and Fig. 3.11. The x-axes in these figures are labeled with the application names and their respective types. For better clarity, Table 3.1 lists all evaluated applications. These applications were chosen based on their cache behavior, particularly their **Last-Level Cache (LLC)** miss

TABLE 3.3: Transfer Candidate Property

Property	Quantity for PASS-P (%)	Quantity for SCAM (%)	Property	Quantity for PASS-P (%)	Quantity for SCAM (%)
only dead	2.18	0.81	only not in private cache	0.53	2.65
only clean	0.11	0	dead and clean	11.13	2.06
dead and not in private cache	6.28	20.98	dead, clean and not in private cache	76.39	70.98
clean and not in private cache	3.36	2.51	only LRU and not dead, not clean, and in private cache	0.02	0

rates. Applications with moderate (50% to 75%) to very high (above 75%) LLC miss rates were selected.

High LLC miss rates are significant because they increase the likelihood of destructive interference in shared caches. Such interference is critical to evaluate the effectiveness of SCAM in reducing performance bottlenecks. Additionally, the selected applications frequently triggered partition decision changes when run concurrently, providing valuable insights into SCAM's dynamic partitioning capabilities.

The results presented in this section represent the **average performance metrics** across all possible application combinations from the SPEC CPU2006 benchmark suite. This approach ensures a comprehensive analysis of SCAM's behavior across a wide range of workloads.

In the next subsection, we compare SCAM with PASS-P in detail. We highlight the key improvements in performance achieved by SCAM, demonstrating that these enhancements do not compromise the strong security guarantees provided by dynamic cache partitioning.

3.4 Results & Analysis

In this section, we conduct a detailed performance evaluation of **SCAM** and compare its efficiency with the **PASS-P** scheme across diverse application combinations.

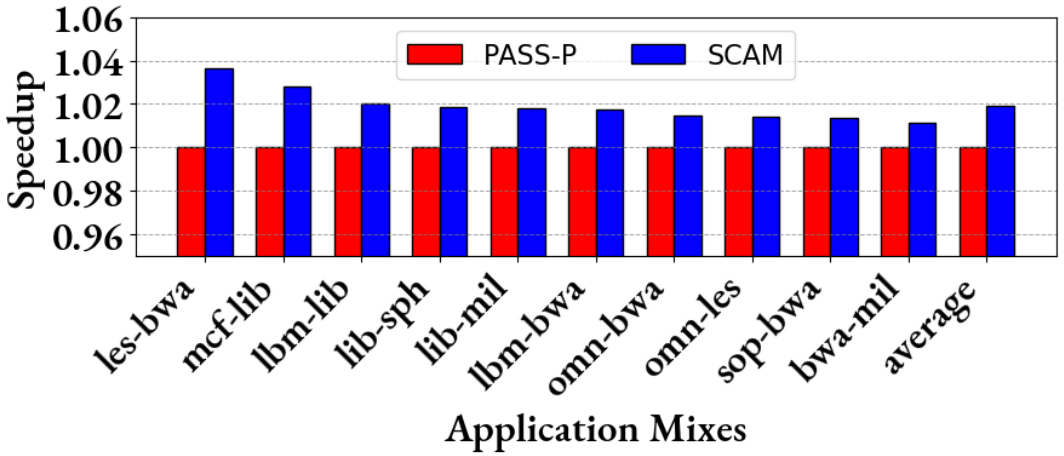

FIGURE 3.9: System's Performance Analysis

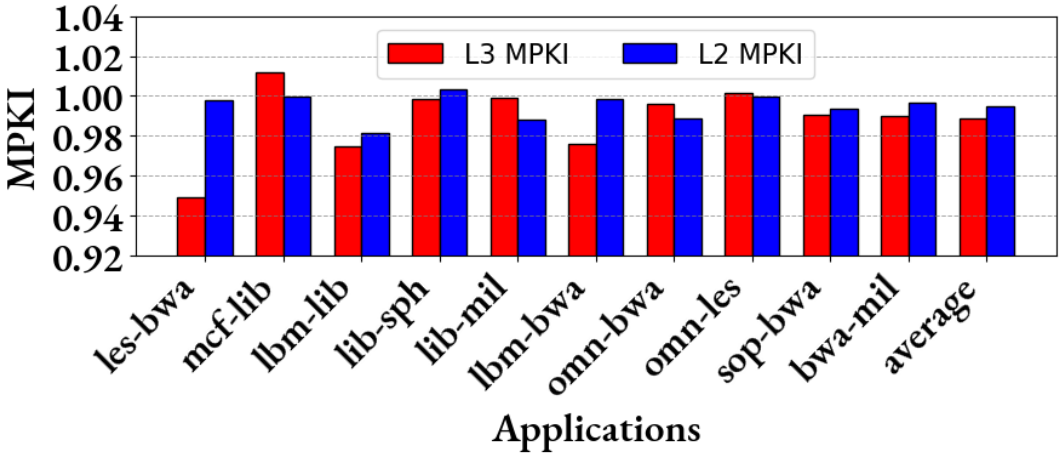

FIGURE 3.10: Cache Performance Analysis

The results demonstrate SCAM's ability to outperform PASS-P in both performance and cache management. As shown in Fig.3.9, SCAM achieves a maximum performance improvement of 4% and an average enhancement of 2% compared to PASS-P. These results indicate that SCAM's optimized transfer candidate selection has a direct and measurable impact on system performance. Further validation is provided by Fig.3.11, which highlights the differences in cache block transfers between SCAM and PASS-P. On average, SCAM selects different cache blocks for transfer 18% of the time, with a maximum difference of **34%** observed for a specific application mix (*omn-les*). This significant variation underscores the importance of SCAM's careful

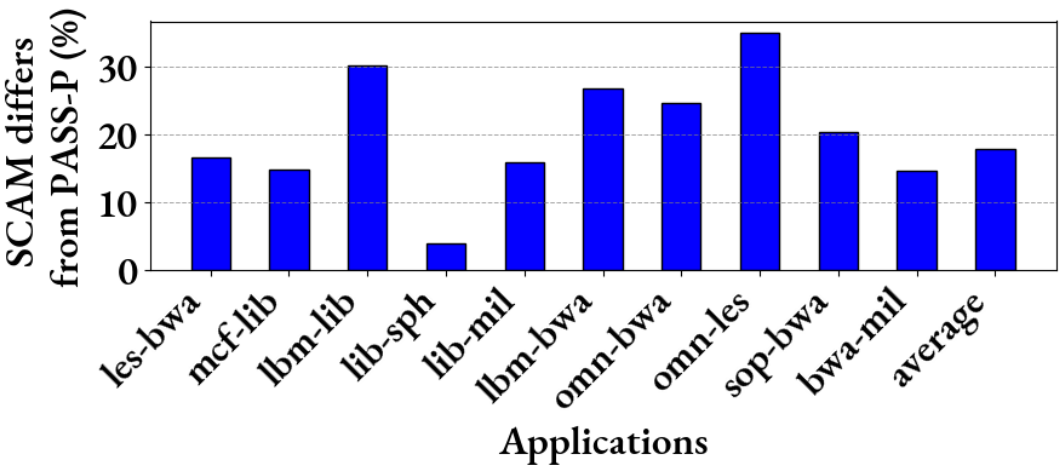

FIGURE 3.11: Fraction of times SCAM transfers a cache block different from that of PASS-P

approach to transfer candidate selection during cache partition adjustments.

Fig.3.10 illustrates the impact of SCAM on L2 and LLC misses per kilo instructions (MPKIs). On average, SCAM reduces L2 MPKI by 1%, with the most significant reduction reaching **2%** for certain application mix (*lbm-lib*). Also, SCAM reduces LLC MPKI by an average of 1.2% with the reduction being as high as 5% for *les-bwa* application mix. These improvements reflect SCAM's ability to enhance cache efficiency by reducing cache misses and optimizing resource utilization. However, it is important to note that some application mixes exhibit an increase in both L2 and LLC MPKIs. This seemingly counterintuitive outcome is attributed to shifts in cache access patterns caused by SCAM's optimization strategies. These variations highlight the complex interplay between SCAM's dynamic partitioning approach and the workloads' cache access behaviors.

Table.3.3 provides a comparative analysis of the properties of cache blocks transferred by PASS-P and SCAM during partition changes. The table presents the average values observed across all application mixes used in our evaluation. To evaluate the effectiveness of PASS-P and SCAM, we profiled the characteristics of all

cache blocks that underwent transfer during the execution phase in which performance statistics were recorded. This profiling enables a direct comparison of the decision-making processes of PASS-P and SCAM. From the data in Table.3.3, it is evident that SCAM successfully avoids transferring cache blocks that merely satisfy the clean property without meeting additional criteria (refer to the *only clean* property in Table.3.3). This ensures that SCAM does not prioritize cache blocks solely based on clean property, preventing unnecessary evictions that could impact performance.

Additionally, SCAM refrains from transferring LRU blocks that do not satisfy any of the three key properties required for optimal eviction decisions (refer to *only LRU and not dead, not clean, and in private cache* property in Table.3.3). By doing so, SCAM prevents inefficient cache block transfers that would otherwise degrade system performance. Furthermore, an analysis of the *only not in private cache* and *dead and not in private cache* categories in Table.3.3 highlights SCAM's ability to mitigate NoC traffic pressure. By prioritizing blocks that are unlikely to be re-referenced after insertion, SCAM ensures efficient cache utilization while reducing unnecessary communication overhead. These optimizations collectively enhance cache management efficiency, improve system throughput, and reinforce security against side-channel threats.

Overall, the evaluation results emphasize the importance of SCAM's strategic selection of transfer candidates. By prioritizing dead blocks and considering back invalidations, SCAM improves performance without compromising the security of the shared **Last-Level Cache (LLC)**. This dual focus on performance and security establishes SCAM as a robust and effective dynamic cache partitioning strategy.

3.4.1 Hardware Overhead Compared to PASS-P

SCAM enhances transfer candidate selection with minimal hardware overhead, ensuring efficient cache management without added complexity. Each cache block is augmented with an additional bit per core to track its presence in private caches, enabling more informed eviction decisions and reducing unnecessary data movement.

To further optimize cache utilization, SCAM introduces a re-reference bit per block to detect dead blocks. This bit indicates whether a block has been accessed since insertion, allowing SCAM to prioritize evicting blocks unlikely to be reused. By distinguishing between frequently accessed and obsolete data, SCAM minimizes performance overhead while improving cache efficiency.

Moreover, SCAM leverages the existing dirty bit in cache metadata to identify clean blocks, eliminating the need for additional storage. This ensures that clean blocks can be selected as transfer candidates without incurring unnecessary writebacks, maintaining both performance and security. With these lightweight yet effective modifications, SCAM refines cache management while preserving low hardware complexity. The next section explores related research on shared cache management, highlighting existing solutions and their limitations.

3.5 Summary

SCAM marks a substantial improvement in secure cache management for shared LLCs in multi-core systems. It overcomes the key limitations of PASS-P by refining the transfer candidate selection process and optimizing cache block eviction

strategies. SCAM not only enhances system performance but does so without compromising security. This makes it a more efficient and resilient solution for cache management in dynamic partitioning systems.

This chapter sets the foundation for understanding existing secure cache management techniques. Building on this, the next chapter investigates a previously overlooked side-channel vulnerability in PASS-P. By identifying its root cause, we develop a targeted mitigation strategy that preserves performance while enhancing security.

Chapter 4

SPARTON

This chapter introduces SPARTON, a secure dynamic cache partitioning technique. SPARTON addresses the vulnerability introduced by the "transfer on a miss" policy. In SPARTON, we restrict each core's allocation change to at most one way per partition decision. In addition, SPARTON transfers ownership across all sets at once. This removes the side-channel created by the "transfer on a miss" policy. Like PASS-P, SPARTON invalidates cache lines before any transfer. This ensures protection against flush-based, contention-based, and occupancy-based attacks. SPARTON selects transfer candidates to reduce performance overhead. It minimizes the number of back-invalidations and writebacks caused by invalidating transferred ways.

4.1 Novel Side-Channel Attack on UCP

Our threat model considers a Chip-Multiprocessor (CMP) system, where an attacker process runs on one core and a victim process runs on another. The LLC employs an insecure dynamic cache partitioning scheme, like UCP. We assume that the attacker

can influence the partitioning decision by carefully controlling its own memory accesses. Additionally, the attacker is allowed to access LLC sets other than the target sets, which enables it to influence the global cache partitioning decision independently from accessing the target cache sets. We also assume that the epoch length of the cache partitioning scheme is short enough to allow the attacker to launch an attack across multiple epochs. The attacker is assumed to have knowledge of both the epoch length and the cache partitioning algorithm. Furthermore, we assume that the dynamic cache partitioning scheme uses a “transfer on a miss" policy when applying new partition decisions. This policy is used in UCP to delay cache line transfers and thus maximize LLC utilization and performance. UCP Qureshi and Patt (2006a) satisfies all of the assumptions outlined in this threat model. The attack described in this section can be launched against any cache partitioning scheme that adheres to these conditions. To the best of our knowledge, this is the first work to identify the side channel created by the “transfer on a miss" policy.

Consider a system with two cores, where the attacker process runs on one core and the victim process on the other. Both cores share an 8-way last-level cache (LLC) managed by a utility-based cache partitioning (UCP) scheme. As illustrated in Fig. 4.1, assume that in phase P0, the cache partition is set to 7:1—seven ways allocated to the attacker and one to the victim. During this phase, the attacker primes the target cache set by inserting its own cache lines (A0 to A6), following a Prime+Probe Liu et al. (2015) methodology. Next, the attacker deliberately reduces its utility, prompting UCP to shift the cache partition from 7:1 to 1:7. Now, the victim is allocated seven ways, and the attacker only one. However, due to the “transfer on a miss" policy, the cache state for the target set does not update immediately. This state corresponds to phase P2. Next, the attacker remains idle while

the victim accesses the target cache set, guided by its secret-dependent memory behavior. Assume that the victim accesses four new cache lines (V1 to V4), resulting in four cache misses. Each of these misses triggers the “transfer on a miss" policy, whereby one of the attacker's cache lines is invalidated and the corresponding way is transferred to the victim process. Thus, the victim gradually takes ownership of the cache set. Subsequently, the attacker increases its utility again, prompting UCP to revert the partition back to 7:1. As before, the cache state of the target set remains unchanged until misses occur. In phase P4, the attacker probes the target cache set by accessing its original cache lines (A0 to A6) and measures the access latencies. As shown in Fig. 4.1, the attacker observes four cache misses, corresponding to the four cache lines evicted during the victim's accesses. This allows the attacker to infer how many new cache lines the victim loaded into the target cache set, effectively revealing information about the victim's memory access pattern. The side channel arises specifically due to the “transfer on a miss" policy. Existing defense mechanisms, including PASS-P Boran et al. (2022) and SCALE Holtryd et al. (2023), do not mitigate this vulnerability. To counter this novel side-channel attack, we propose SPARTON—a secure dynamic cache partitioning scheme for LLCs designed to eliminate this side-channel and enhance security.

4.2 Proposal

As explained in Section 4.1, a side-channel attack can be mounted on a dynamically partitioned LLC that employs the “transfer on a miss" policy. The root cause of this side channel is the access-dependent, per-set transfer of cache line ownership. To eliminate this vulnerability, our proposed scheme, SPARTON, replaces the “transfer on a miss" policy with a proactive, simultaneous transfer of ownership across all

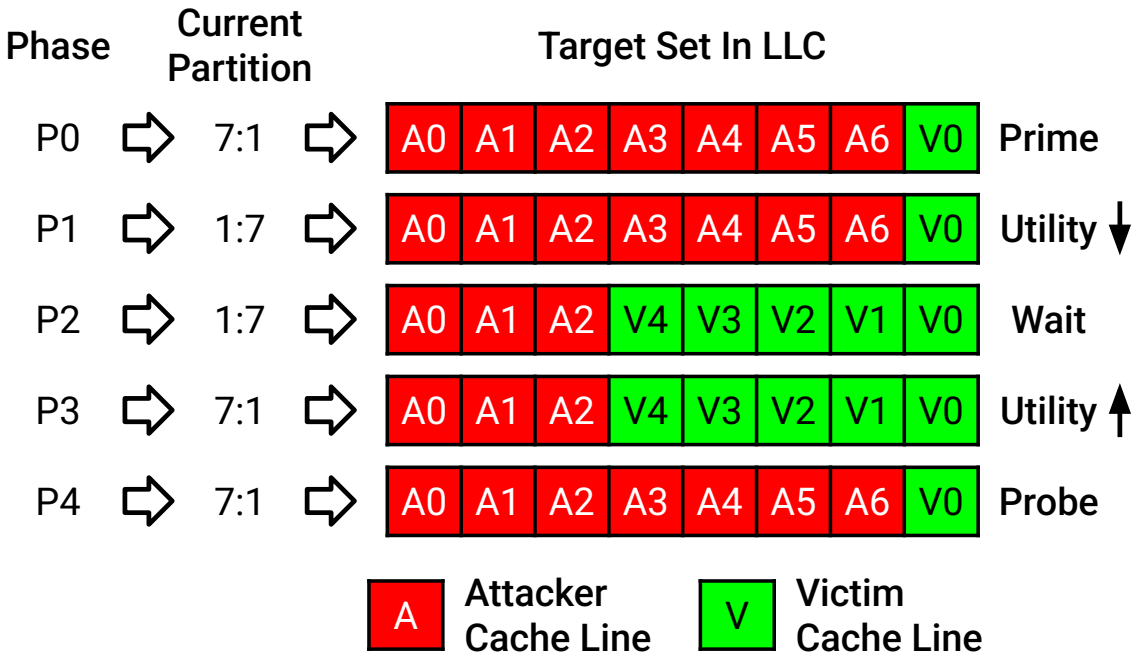

FIGURE 4.1: Novel Side-channel Attack on UCP

LLC sets whenever the partition decision changes. However, transferring a large number of cache lines introduces performance overhead. To reduce this overhead, SPARTON restricts the partition update policy such that the number of cache ways assigned to any core can change by at most one per epoch. This significantly limits the number of cache lines that must be transferred. Before transferring a cache line, SPARTON invalidates it to prevent data leakage. This step, however, presents two main challenges. First, if the cache line is modified (dirty), it must be written back to main memory. Second, if the cache line is present in any private cache, it must be back-invalidated to maintain the inclusivity of the LLC. Both actions incur additional overhead. To minimize this cost, SPARTON employs a priority-based transfer policy. The highest priority is assigned to cache lines that are neither modified nor cached privately, ensuring minimal overhead while preserving security.

4.2.1 Working of SPARTON

SPARTON builds upon the UCP framework, leveraging its utility-based dynamic cache partitioning approach. Similar to UCP, SPARTON divides execution time into fixed-length epochs of CPU cycles. During each epoch, Utility Monitors (UMONs) Qureshi and Patt (2006a) track LLC accesses to estimate the utility of each core's application. At the end of the epoch, SPARTON uses UCP's look-ahead algorithm to compute a new cache way allocation. Unlike UCP, SPARTON restricts the allocation change for any core to a maximum of one way per epoch. To apply this constraint, SPARTON takes the new and old allocations as input and calculates the net change in allocation for each core (see Algorithm 2). Cores with non-zero changes are then grouped into gainers and losers—cores that are gaining and losing cache ways, respectively. Since the total number of cache ways is fixed, the net sum of changes must be zero. With the one-way change restriction, SPARTON ensures balance by forming an equal number of gaining and losing cores. To do this, both groups are sorted in descending order of the magnitude of change in allocation. Then, transfer pairs are created by selecting one core from each group until one of the groups is empty. SPARTON only updates allocations for these transfer pairs, transferring one cache way per pair. To eliminate side-channels introduced by UCP's "transfer on a miss" policy, SPARTON proactively transfers ownership across all sets simultaneously. To minimize performance overhead, SPARTON triggers the transfer for each core pair on an LLC miss by the gaining core. Since an LLC miss already incurs high-latency access to main memory, SPARTON overlaps the transfer operation with this latency, effectively hiding its cost. For each core pair, one cache line per set is invalidated and reassigned according to SPARTON's priority-based transfer policy (described in Section 4.2.2). This process is repeated for all core pairs, ensuring secure and low-overhead ownership transitions during

partition changes.

Algorithm 2: SPARTON: Transfer Algorithm

Input: Old allocation A_{old}, New allocation A_{new}

```
// 1) Compute change in allocation
foreach i do
    core[i].Δ ← A_new[i] − A_old[i]
end
// 2) Identify gainers and losers
Gains ← {core[i] | core[i].Δ > 0}
Losses ← {core[j] | core[j].Δ < 0}
// 3) Sort in descending order of |Δ|
SortDescendingMagnitute(Gains)
SortDescendingMagnitute(Losses)
// 4) Form transfer pairs
Pairs ← ∅
while Gains ≠ ∅ and Losses ≠ ∅ do
    core_gain ← Gains.pop()
    core_lose ← Losses.pop()
    Pairs ← Pairs ∪ {(core_gain, core_lose)}
end
// 5) Perform transfer for each pair
foreach (core_gain, core_lose) ∈ Pairs do
    // Triggered on an LLC miss by core_gain
    foreach LLC Set S do
        T = SelectTransferCandidate(core_lose, S)
        if (T.modified) then
            MainMemory ← T
        end
        if (T.InPrC) then
            BackInvalidate(T)
        end
        T.valid ← false
        T.ownership ← core_gain
    end
end
```

4.2.2 Selection of Transfer Candidate

SPARTON transfers cache lines across all sets at the same time. To minimize overheads, it must prevent back-invalidations and writebacks. Each LLC cache line has two properties: *Modified (M)* and *In Private Cache (InPrC)*. The *Modified* flag shows if the contents of the cache line have been modified. Invalidating such a line triggers a writeback to main memory. The *InPrC* flag shows if the line is present in any private caches. Invalidating this line sends back-invalidation messages to those caches.

The selection algorithm works on the cache lines assigned to the core losing one way. The algorithm chooses a transfer candidate based on the *Modified* and *InPrC* flags. It follows this priority order:

- **Priority P0**: Invalid lines. These cause no overhead.
- **Priority P1**: Not modified and not in any private cache.
- **Priority P2**: Not modified but present in private caches. This only causes back-invalidation.
- **Priority P3**: Modified but not in private caches. This only causes a writeback.
- **Priority P4**: Least recently used (LRU) cache line as no better option is found.

SPARTON assigns higher priority (P2) to cache lines that are not modified than to those not present in private caches (P3). This is because writing back modified cache lines requires sending data to off-chip main memory, which is expensive. The selection algorithm can be implemented using a small finite state machine (FSM)

for each set. Since SPARTON transfers ownership for only one pair of cores at a time, a single FSM per set is sufficient.

4.2.3 Security Analysis

The attack described in Section 4.1 operates in five phases. Out of these, two phases involve a change in the cache partition. For the attack to achieve maximum bandwidth, the attacker must switch the cache partition from one extreme to the other within a single epoch. SPARTON prevents this by limiting partition changes to a maximum of one way per epoch. As a result, the attacker needs several epochs to reach the desired partition configuration. Furthermore, SPARTON does not use the "transfer on a miss" policy. Instead, it transfers cache lines in all sets simultaneously. In UCP, cache line transfer depends on a miss in the core gaining a way, and only in the corresponding set. In contrast, SPARTON triggers the transfer across all sets when the gaining core experiences a miss in any set. Therefore, the attacker cannot determine whether a miss occurred in the target set or in a different set.

An attacker may attempt to improve the attack by using multiple attacker processes on multiple cores. In SPARTON, partition changes create two possible cases: **Case (1):** The victim's core does not get paired with any other core. **Case (2):** The victim's core is paired with another core. In Case (1), there is no change in the number of cache ways assigned to the victim. Thus, no data leakage can occur. In Case (2), the victim's core is paired with another core (which may be benign or malicious), and one cache way is transferred. This transfer affects all sets simultaneously and is triggered by the first miss on the core gaining a way. As a result, even in this case, the attacker cannot determine whether the miss occurred in the target set. Finally, other core pairs not involving the victim do not influence the number of cache ways

assigned to the victim. Therefore, cache line transfers involving unrelated pairs do not affect the security of the victim process.

4.2.4 Hardware Overhead

SPARTON is implemented on top of the baseline UCP. Therefore, it requires the same hardware components as UCP, such as UMONs and ownership bits. In addition, SPARTON must determine whether a cache line is modified and whether it is present in any of the private caches. This information is already tracked by the coherence protocol Papamarcos and Patel (1984). As a result, SPARTON does not require any additional per-line storage in the LLC for transfer candidate selection, provided the coherence metadata is readily accessible. If coherence information is not directly accessible, SPARTON requires 2 additional bits per LLC cache line to track the “Modified" and “In Private Cache" states. Moreover, SPARTON requires a small finite state machine (FSM) per cache set to manage cache way transfers. Finally, SPARTON maintains a small structure to store the pairs of core IDs for which transfers are pending. In an n-core system, the maximum number of entries in this structure is $(n \times (n-1))/2$. The size of each entry depends on the width of the core ID field.

4.3 Evaluation

4.3.1 Simulation Environment

To evaluate the performance overhead of the proposed secure cache partitioning scheme, we use the ChampSim https://github.com/ChampSim/ChampSim simulator. ChampSim has been widely used in recent studies on LLC optimizations Jain and Lin (2016); Shah et al. (2022). Table 4.1 presents the detailed configuration of the baseline system used in our analysis. The simulated system features a 3-level cache hierarchy. Each core has private L1 and L2 caches. The last-level cache (LLC) is shared among all cores and is inclusive, which makes it vulnerable to cache timing side-channel attacks. By default, we configure the LLC as a 2 MB/core, 16-way set-associative cache. The baseline uses a utility-based dynamic cache partitioning (UCP) Qureshi and Patt (2006a) scheme without any security enhancements. All analyses in this work use an epoch length of 5 million cycles.

TABLE 4.1: Baseline System Configuration

Core	2 OoO cores, Freq. = 4 GHz, ROB = 256 entry, LQ = 128 entry, SQ = 72 entry, Dispatch width = Issue width = Commit width = 4.
L1-I/D Cache	Private, 32 KB, 8-Way, 6 cycles, LRU
L2 Cache	Private, 256 KB, 8-Way, 12 cycles, LRU
L3 Cache (LLC)	Shared, 2 MB/core, 16-way, 30 cycles, UCP (Utility-based Cache Partitioning)
Main memory	800MHz, 4 KB Page, tRP = tRCD = tCAS = 12.5ns

4.3.2 Workloads

Our evaluation utilizes the publicly released traces of SPEC 2006 Henning (2006) and SPEC 2017 Bucek et al. (2018) benchmark suites from DPC3

https://dpc3.compas.cs.stonybrook.ed (2019). For each application, we use the trace corresponding to the highest weighted SimPoint Perelman et al. (2003). Based on single-core simulations, we classify the applications into two categories based on their LLC accesses per kilo-instructions (APKI). We select 8 memory-intensive applications (M-type) with the highest LLC APKI and 8 compute-intensive applications (C-type) with the lowest LLC APKI (see Table 4.2).

To evaluate the performance overhead of our proposed secure cache partitioning scheme, SPARTON, we simulate all 120 unique pairs formed from the 16 selected applications on a dual-core system. Each simulation starts with a warm-up of 50 million instructions per application, followed by a detailed simulation of 200 million instructions within the region of interest. If one application completes its 200 million instructions earlier than the other, it continues to execute to maintain contention in the shared LLC and to actively participate in dynamic partitioning decisions.

TABLE 4.2: Selected Workloads

Application	**LLC APKI**	**Type**	**Application**	**LLC APKI**	**Type**
619.lbm s	112.586	M	631.deepsjeng s	0.584	C
654.roms s	112.586	M	465.tonto	0.584	C
444.namd	59.566	M	600.perlbench s	0.065	C
450.soplex	59.566	M	628.pop2 s	0.065	C
401.bzip2	50.524	M	416.gamess	0.031	C
470.lbm	50.524	M	434.zeusmp	0.031	C
607.cactuBSSN s	42.289	M	603.bwaves s	0.011	C
435.gromacs	42.289	M	454.calculix	0.011	C

For performance analysis, we divide all 120 application pairs into three categories: (1) MM, where both applications are memory-intensive (M-type); (2) MC, where one application is memory-intensive (M-type) and the other is compute-intensive (C-type); and (3) CC, where both applications are compute-intensive (C-type). In

the following analysis, we choose five representative application pairs from each category (see Table 4.3) to show in the plots. These representatives include the two best-performing pairs, the two worst-performing pairs, and one pair with average performance. We also report the mean performance for each category, as well as the overall mean across all 120 pairs. Note that mean values are calculated using the complete set of application pairs in each category, not just the selected representatives.

TABLE 4.3: Representative Workload Pairs

SN	Application 0	Application 1	Notation
1	450.soplex	401.bzip2	MM-1
2	450.soplex	435.gromacs	MM-2
3	619.lbm_s	607.cactuBSSN_s	MM-3
4	401.bzip2	470.lbm	MM-4
5	444.namd	470.lbm	MM-5
6	444.namd	600.perlbench_s	MC-1
7	607.cactuBSSN_s	600.perlbench_s	MC-2
8	444.namd	416.gamess	MC-3
9	401.bzip2	631.deepsjeng_s	MC-4
10	450.soplex	631.deepsjeng_s	MC-5
11	465.tonto	600.perlbench_s	CC-1
12	628.pop2_s	416.gamess	CC-2
13	631.deepsjeng_s	454.calculix	CC-3
14	600.perlbench_s	416.gamess	CC-4
15	434.zeusmp	603.bwaves_s	CC-5

4.3.3 Weighted Speedup

We use the weighted speedup metric to evaluate the performance impact of the proposed SPARTON scheme, similar to the prior works on LLC optimizations Qureshi and Patt (2006a); Boran et al. (2022); Holtryd et al. (2023); Jain and Lin (2016); Shah et al. (2022). This metric avoids bias caused by fast-running applications in multi-core simulations. Weighted speedup is calculated as sum of $(IPC^i_{ROI}/IPC^i_{Single})$ across all cores. Here, IPC^i_{ROI} is the IPC measured in the region of interest (ROI)

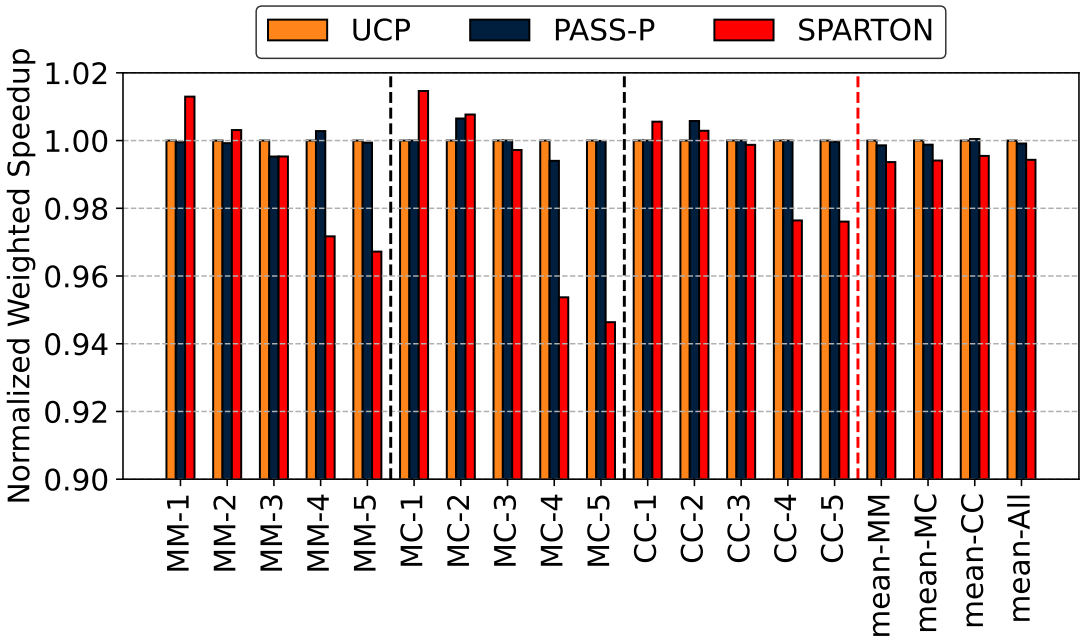

FIGURE 4.2: Normalized Weighted Speedup

for the application running on the i^{th} core in a multi-core simulation, and IPC^{i}_{Single} is the IPC of the same application when running alone on a single core.

Figure 4.2 presents the weighted speedup of UCP, Pass-P, and SPARTON, normalized to a baseline system using UCP at the LLC. On average, across all application pairs, SPARTON incurs only a 0.6% performance loss compared to the baseline UCP. The maximum observed performance degradation is 5.4%, seen in the MC-5 application pair. As shown by the category-wise averages, SPARTON maintains low performance overhead across all three workload types (MM, MC, and CC), while providing strong security guarantees. In some cases, SPARTON even slightly improves performance. This happens when the transfer candidate selected does not trigger any writeback or back-invalidation, reducing transfer overhead.

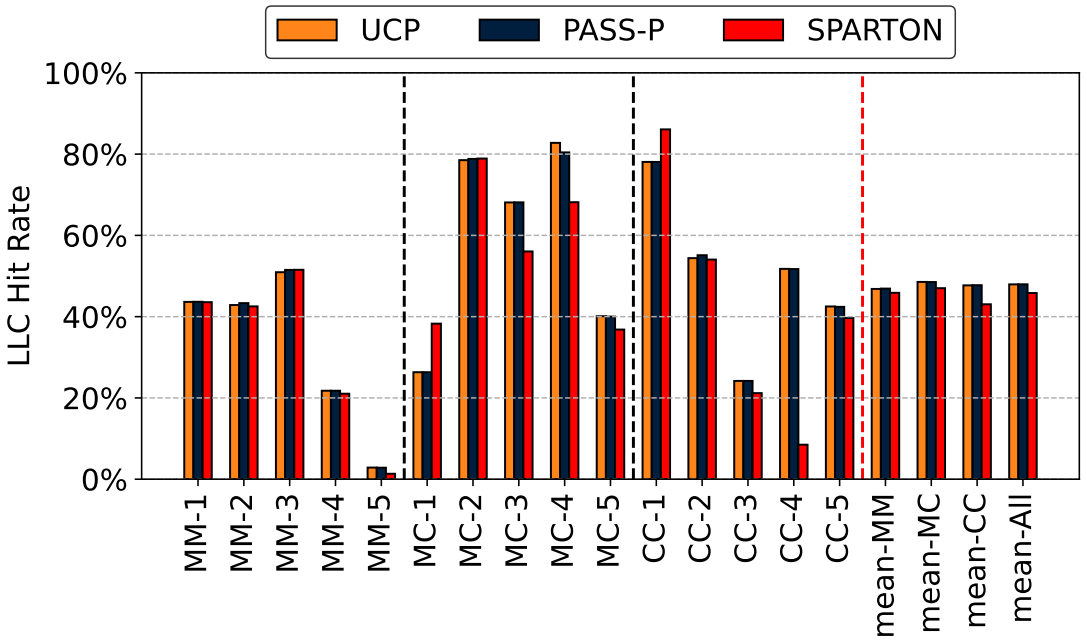

FIGURE 4.3: LLC Hit Rate

4.3.4 LLC Hit Rate

To better understand the impact of the proposed policy, we also analyze its effect on the LLC hit rate. Across all application pairs, the LLC hit rate decreases from 47.95% with UCP to 45.81% with SPARTON. For most pairs, this degradation is marginal. In some cases, the LLC hit rate even improves, leading to a corresponding gain in weighted speedup. For the CC category, the average hit rate shows the largest drop, from 47.70% with UCP to 43.04% with SPARTON.

4.3.5 Transfer Candidates

As described in Section 4.2, the proposed scheme, SPARTON, selects transfer candidates based on a predefined priority order. Before ownership is updated, all transferred cache lines are invalidated to ensure security against timing side-channel attacks. As a result, many sets often select invalid lines as transfer candidates, since

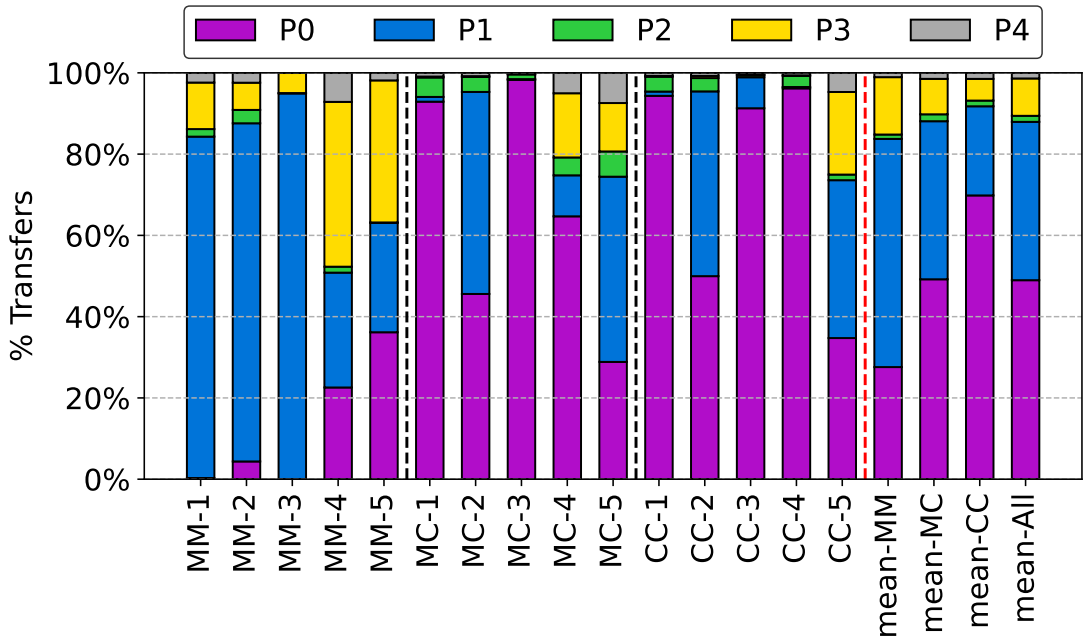

FIGURE 4.4: Percentage of Transfer Candidates per Priority

not all invalid lines are filled before the next partition change. As shown in Fig. 4.4, on average, an invalid cache line is available for transfer in 48.97% of cases across all application pairs. This percentage increases to 69.83% for compute-intensive (CC) application pairs. These applications typically have smaller working sets, leading to lower LLC utilization and a higher availability of invalid lines. Furthermore, in 38.97% of cases, SPARTON successfully selects a cache line that does not trigger any back-invalidation or writeback. Overall, 87.94% of transfers avoid costly back-invalidation and writeback, contributing to the low performance overhead observed with SPARTON.

4.3.6 LLC Occupancy

During each partition change, SPARTON transfers ways across all sets and invalidates them before updating ownership. To evaluate the impact of this mechanism

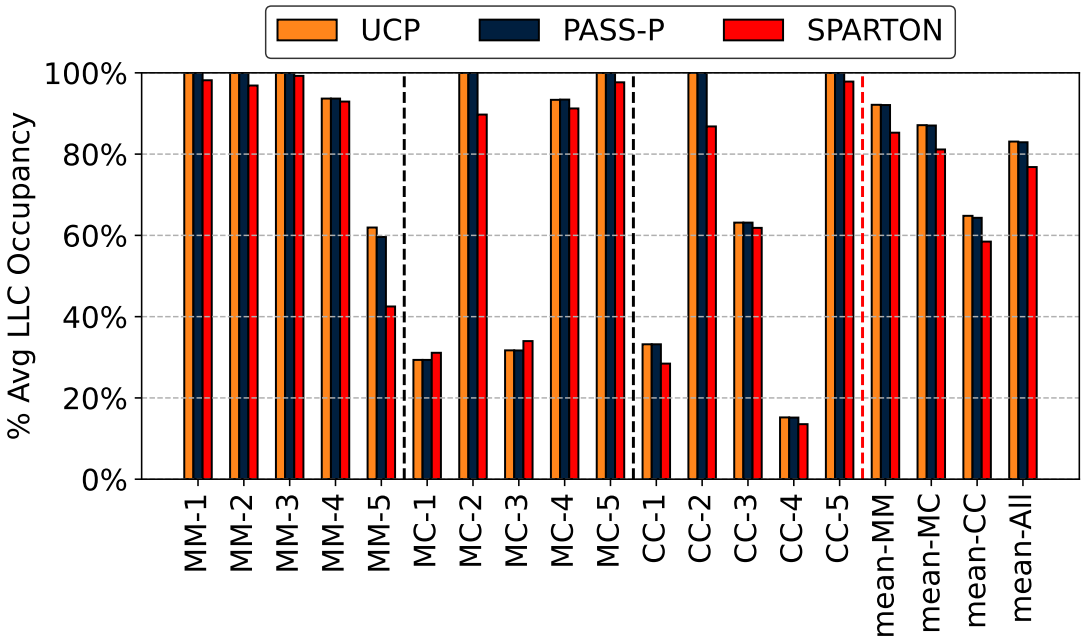

FIGURE 4.5: Average LLC Occupancy

on LLC utilization, we analyze and compare the average LLC occupancy of SPARTON with that of UCP. Average occupancy is measured by counting the number of valid cache lines at each LLC insertion and dividing it by the capacity of the LLC. As shown in Fig. 4.5, SPARTON reduces the average LLC occupancy from 83.09% (UCP) to 76.82% across all application pairs—a reduction of only 6.27%. For compute-intensive (CC) application pairs, which typically have smaller working sets, the occupancy drops from 64.81% with UCP to 58.47% with SPARTON. This reduction is expected, as compute-intensive workloads leave more cache lines invalid, and SPARTON's transfer mechanism further contributes to lower occupancy.

4.3.7 Partition Changes

In each epoch, UCP can change the cache partition from any configuration to any other. To reduce the number of ways transferred between cores, SPARTON limits partition changes to one way at a time. This restriction ensures that only one way

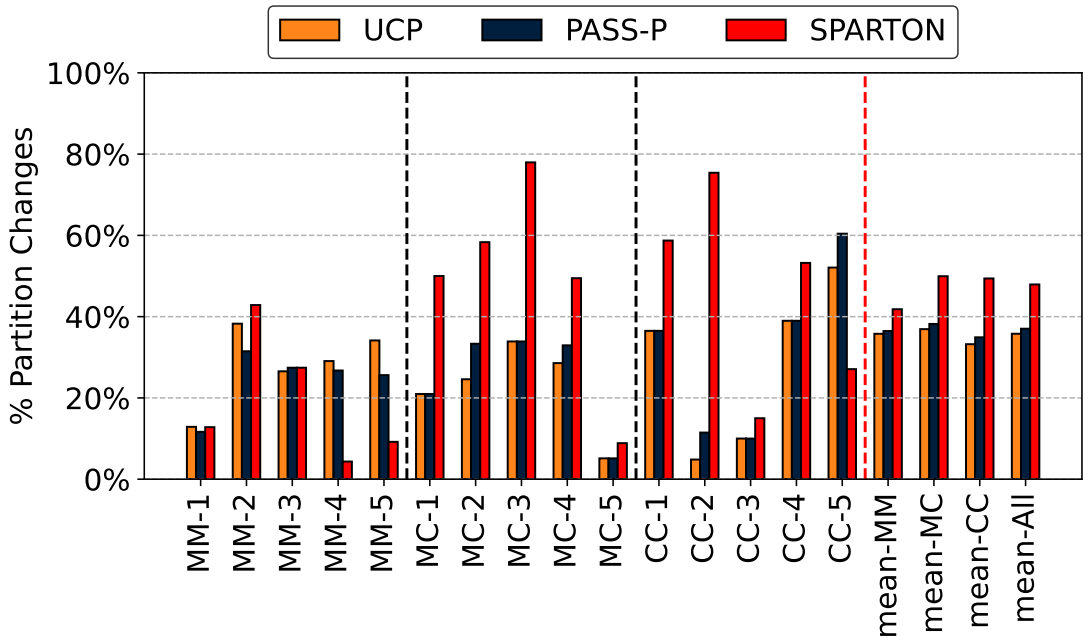

FIGURE 4.6: Percentage of Partition Changes

per set per pair of cores is transferred during each partition change. However, it also requires more epochs to reach the optimal partition. To evaluate this trade-off, we analyze the percentage of epochs in which partition changes occur for UCP, Pass-P, and SPARTON. As shown in Fig. 4.6, the average percentage of partition changes increases from 35.79% with UCP to 47.93% with SPARTON. This increase reflects SPARTON's gradual adjustment approach while simultaneously restricting the attacker's ability to exert control over partition decisions.

4.3.8 Back Invalidation Analysis

To assess the effectiveness of SPARTON in minimizing back-invalidations due to transfer of ways, we evaluated the percentage of back-invalidations. As shown in Fig. 4.7, the average back-invalidation rate increases from 10.84% for UCP to 18.88% for SPARTON. We further analyzed the source of back-invalidations by separating those caused by way transfers from those due to the cache replacement

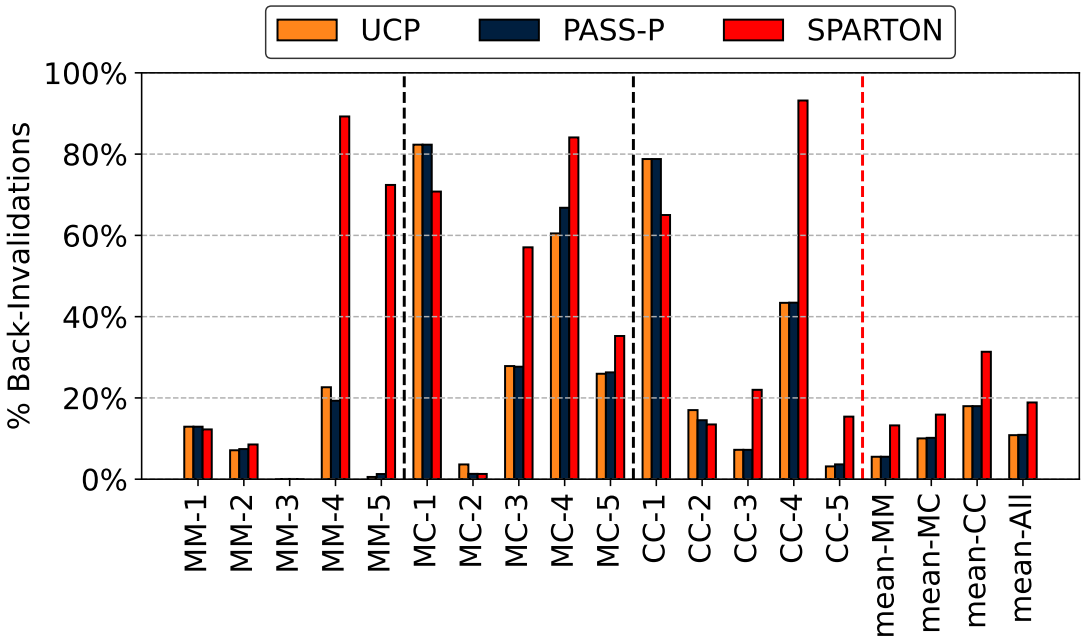

FIGURE 4.7: Percentage of Back-invalidations

policy. In SPARTON, only 10.87% of all back-invalidations result from ownership transfers. The remaining majority are attributed to the replacement policy. These results demonstrate that SPARTON's transfer candidate selection effectively minimizes back-invalidations caused by partition changes by prioritizing cache lines that are not present in private caches.

4.3.9 Writebacks Analysis

Writing back modified cache lines from the LLC is costly, as it involves off-chip memory operations and increases network traffic. To minimize this overhead, Pass-P prioritizes unmodified cache lines as transfer candidates. Similarly, SPARTON gives higher priority (P0, P1 and P2) to choosing an unmodified cache line as transfer candidate. To evaluate the impact of SPARTON on writebacks, we analyzed the overall percentage of writebacks (Fig. 4.8) and the percentage specifically caused by transfer candidates. On average, the total writeback percentage increases from

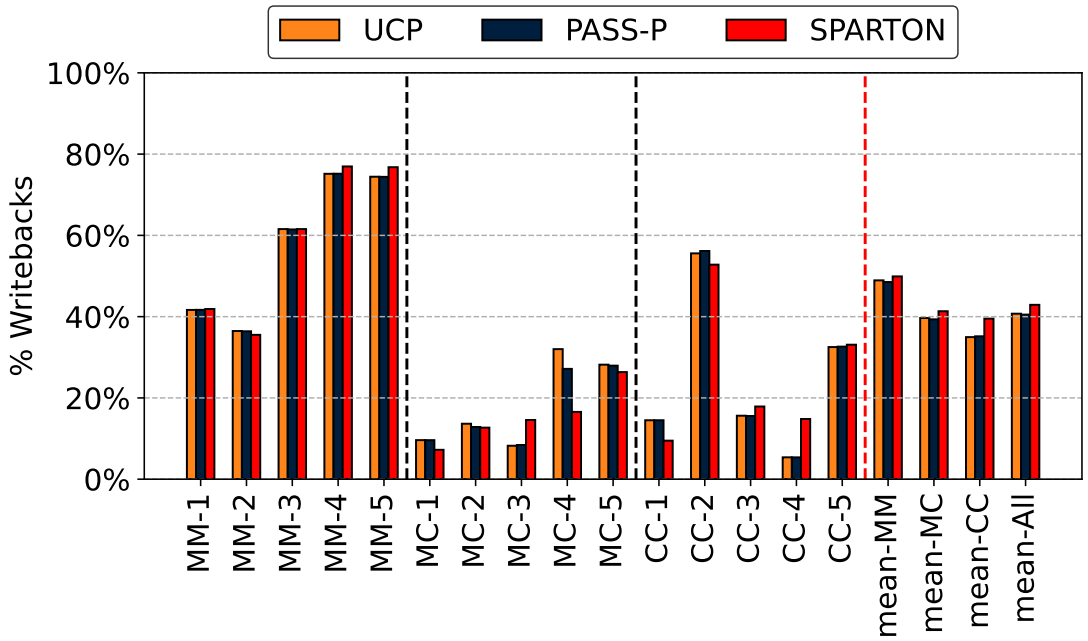

FIGURE 4.8: Percentage of Writebacks

40.73% in UCP to 42.91% in SPARTON. In SPARTON, only 9.24% of all writebacks result from transfer candidates. These results show that SPARTON effectively limits writeback overhead.

4.4 Summary

This chapter proposes SPARTON, a secure dynamic cache partitioning technique. SPARTON addresses the vulnerability introduced by the "transfer on a miss" policy. In SPARTON, we restrict each core's allocation change to at most one way per partition decision. In addition, SPARTON transfers ownership across all sets at once. This removes the side-channel created by the "transfer on a miss" policy. Like PASS-P, SPARTON invalidates cache lines before any transfer. This ensures protection against flush-based, contention-based, and occupancy-based attacks. SPARTON

selects transfer candidates to reduce performance overhead. It minimizes the number of back-invalidations and writebacks caused by invalidating transferred ways.

The following chapter concludes the thesis by summarizing key findings and contributions. It also outlines potential directions for future research, highlighting opportunities to further enhance security and performance in processor architectures.

Chapter 5

Conclusions and Future Work

The first part of this research demonstrates that secure cache partitioning can be optimized for performance without compromising security. To achieve this, we introduce SCAM, a novel secure cache management framework that addresses the performance limitations inherent in the transfer candidate selection strategy employed by PASS-P, a state-of-the-art secure dynamic cache partitioning (DCP) protocol. Experimental results indicate that SCAM yields a performance improvement of up to 4% compared to PASS-P. By minimizing back invalidations and prioritizing the removal of dead blocks, SCAM not only enhances cache efficiency but also strengthens security by mitigating potential attack vectors. Designed with versatility in mind, SCAM seamlessly integrates with shared cache levels in systems utilizing various DCP protocols. This adaptability ensures robust and scalable cache management across a diverse range of application scenarios. Furthermore, SCAM provides a foundation for addressing emerging security threats, such as Meltdown Lipp et al. (2018) and Spectre Kocher et al. (2019). Future extensions to mitigate these advanced vulnerabilities will further solidify SCAM's position as a high-performance and secure cache management solution.

Moving forward, we identified a previously unexplored side-channel vulnerability in dynamic cache partitioning (DCP) schemes like UCP, which rely on a “transfer on a miss" policy during partition changes. To mitigate this, we proposed SPARTON, a secure and efficient enhancement to UCP. SPARTON defends against this side-channel by simultaneously transferring cache ways across all sets and invalidating them before changing ownership, preventing information leakage through partition changes. Additionally, it restricts partition changes to a single way per epoch, limiting the attacker's ability to influence partition decisions. To minimize performance impact, SPARTON employs a transfer candidate selection strategy that prioritizes reducing costly back-invalidations and writebacks. Experimental evaluation across 120 application pairs from SPEC2006 and SPEC2017 benchmark suites shows that SPARTON incurs only a 0.6% average performance overhead compared to insecure UCP. Furthermore, in 87.94% of transfers, SPARTON successfully selects candidates that do not cause back-invalidations or writebacks, demonstrating the effectiveness of its policy. Overall, SPARTON provides a practical and efficient solution to secure dynamic cache partitioning, mitigating side-channel threats while preserving cache performance.

References

Boran, N.K., Joshi, P., Singh, V., 2022. PASS-P: Performance and Security Sensitive Dynamic Cache Partitioning, in: Proceedings of the 19th International Conference on Security and Cryptography, SECRYPT 2022.

Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Arvind, Devadas, S., 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.

Bucek, J., Lange, K.D., v. Kistowski, J., 2018. SPEC CPU2017: Next-Generation Compute Benchmark, in: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.

Carlson, T.E., Heirman, W., Eyerman, S., Hur, I., Eeckhout, L., 2014. An evaluation of high-level mechanistic core models. ACM Transactions on Architecture and Code Optimization (TACO) doi:10.1145/2629677.

Gruss, D., Maurice, C., Wagner, K., Mangard, S., 2016. Flush+Flush: A Fast and Stealthy Cache Attack, in: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.

Henning, J.L., 2006. SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News .

Holtryd, N., Manivannan, M., Stenström, P., Pericàs, M., 2020. DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).

Holtryd, N.R., Manivannan, M., Stenström, P., 2023. SCALE: Secure and Scalable Cache Partitioning, in: 2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST).

https://dpc3.compas.cs.stonybrook.ed, 2019. The 3rd data prefetching championship.

https://github.com/ChampSim/ChampSim, . ChampSim.

Jain, A., Lin, C., 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).

Kaseridis, D., Stuecheli, J., John, L.K., 2009. Bank-aware Dynamic Cache Partitioning for Multicore Architectures, in: 2009 International Conference on Parallel Processing.

Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J., 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2019. Spectre Attacks: Exploiting Speculative Execution, in: 40th IEEE Symposium on Security and Privacy (S&P'19).

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., 2018. Meltdown: Reading Kernel Memory from User Space, in: 27th USENIX Security Symposium (USENIX Security 18).

Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B., 2015. Last-Level Cache Side-Channel Attacks are Practical, in: 2015 IEEE Symposium on Security and Privacy.

Manikantan, R., Rajan, K., Govindarajan, R., 2012. Probabilistic Shared Cache Management (PriSM), in: 2012 39th Annual International Symposium on Computer Architecture (ISCA).

Omar, H., D'Agostino, B., Khan, O., 2020. OPTIMUS: A Security-Centric Dynamic Hardware Partitioning Scheme for Processors that Prevent Microarchitecture State Attacks . IEEE Transactions on Computers .

Omar, H., Khan, O., 2020. IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications, in: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).

Papamarcos, M.S., Patel, J.H., 1984. A low-overhead coherence solution for multiprocessors with private cache memories, in: Proceedings of the 11th Annual International Symposium on Computer Architecture.

Perelman, E., Hamerly, G., Van Biesbrouck, M., Sherwood, T., Calder, B., 2003. Using SimPoint for accurate and efficient simulation, in: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.

Qureshi, M.K., 2018. Ceaser: mitigating conflict-based cache attacks via encrypted-address and remapping, in: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Press. p. 775–787. URL: https://doi.org/10.1109/MICRO.2018.00068, doi:10.1109/MICRO.2018.00068.

Qureshi, M.K., Patt, Y.N., 2006a. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, in: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06).

Qureshi, M.K., Patt, Y.N., 2006b. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches, in: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), IEEE. pp. 423–432.

Shah, I., Jain, A., Lin, C., 2022. Effective Mimicry of Belady's MIN Policy, in: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA).

Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y., 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel, in: 28th USENIX Security Symposium (USENIX Security 19).

Wang, Y., Ferraiuolo, A., Zhang, D., Myers, A.C., Suh, G.E., 2016. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection, in: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC).

Wang, Z., Lee, R.B., 2006. Covert and side channels due to processor architecture, in: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06), pp. 473–482. doi:10.1109/ACSAC.2006.20.

Yarom, Y., Falkner, K., 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, in: Proceedings of the 23rd USENIX Conference on Security Symposium.

List of Publications

Articles Published/Submitted in International Conferences

1. "SCAM: Secure Shared Cache Partitioning Scheme," Varun Venkitaraman, **Rishabh Ravi**, Tejeshwar Thorawade, Nirmal Boran, Virendra Singh
   Accepted in International Conference on Security and Cryptography (SECRYPT) 2025

2. "SPARTON: Secure Dynamic Partition Scheme for Last-level Cache" Tejeshwar Thorawade, **Rishabh Ravi**, Varun Venkitaraman, Nirmal Boran, Virendra Singh
   Submitted to International Conference on Computer Design (ICCD) 2025

Acknowledgements

I take this opportunity to acknowledge and express my gratitude to all those who supported and guided me during the dissertation work. I am grateful to Varun and Tejeshwar for the help they provided that enabled me to complete this dissertation successfully.

Rishabh Ravi