Thesis for The Degree of Doctor of Philosophy

Adaptive Microarchitectural Optimizations to Improve Performance and Security of Multi-Core Architectures

Nadja Ramhöj Holtryd

Division of Computer and Network Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Adaptive Microarchitectural Optimizations to Improve Performance and Security of Multi-Core Architectures

Nadja Ramhöj Holtryd

Advisor:
Professor Per Stenström, Chalmers University of Technology

Co-Advisor:
Madhavan Manivannan, Ph.D., Chalmers University of Technology

Examiner:
Adjunct Professor Fredrik Dahlgren, Chalmers University of Technology

Thesis Opponent:
Professor Moinuddin K. Qureshi, Georgia Institute of Technology

Grading Committee:
Professor Stefanos Kaxiras, Uppsala University
Associate Professor Ramon Canal, Universitat Politècnica de Catalunya
Professor Yan Solihin, North Carolina State University

Deputy Committee:
Professor Alejandro Russo, Chalmers University of Technology

ISBN 978-91-7905-749-7
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5215.
ISSN 0346-718X

Technical Report No 229D
Department of Computer Science & Engineering
Division of Computer and Network Systems
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2023.

Abstract

With the current technological barriers, microarchitectural optimizations are increasingly important to ensure performance scalability of computing systems. The shift to multi-core architectures increases the demands on the memory system, and amplifies the role of microarchitectural optimizations in performance improvement. In a multi-core system, microarchitectural resources are usually shared, such as the cache, to maximize utilization but sharing can also lead to contention and lower performance. This can be mitigated through partitioning of shared caches.

However, microarchitectural optimizations which were assumed to be fundamentally secure for a long time, can be used in side-channel attacks to exploit secrets, as cryptographic keys. Timing-based side-channels exploit predictable timing variations due to the interaction with microarchitectural optimizations during program execution. Going forward, there is a strong need to be able to leverage microarchitectural optimizations for performance without compromising security.

This thesis contributes with three adaptive microarchitectural resource management optimizations to improve security and/or performance of multi-core architectures and a systematization-of-knowledge of timing-based side-channel attacks.

We observe that to achieve high-performance cache partitioning in a multi-core system three requirements need to be met: i) fine-granularity of partitions, ii) locality-aware placement and iii) frequent changes. These requirements lead to high overheads for current centralized partitioning solutions, especially as the number of cores in the system increases. To address this problem, we present an adaptive and scalable cache partitioning solution (*DELTA*) using a distributed and asynchronous allocation algorithm. The allocations occur through core-to-core challenges, where applications with larger performance benefit will gain cache capacity. The solution is implementable in hardware, due to low computational complexity, and can scale to large core counts.

According to our analysis, better performance can be achieved by coordination of multiple optimizations for different resources, e.g., off-chip bandwidth and cache, but is challenging due to the increased number of possible allocations which need to be evaluated. Based on these observations, we present a solution (*CBP*) for coordinated management of the optimizations: cache partitioning, bandwidth partitioning and prefetching. Efficient allocations, considering the inter-resource interactions and trade-offs, are achieved using local resource managers to limit the solution space.

The continuously growing number of side-channel attacks leveraging microarchitectural optimizations prompts us to review attacks and defenses to understand the vulnerabilities of different microarchitectural optimizations. We identify the four root causes of timing-based side-channel attacks: *determinism*, *sharing*, *access violation* and *information flow*. Our key insight is that eliminating any of the exploited root causes, in any of the attack steps, is enough to provide protection. Based on our framework, we present a systematization of the attacks and defenses on a wide range of microarchitectural optimizations, which highlights their key similarities.

Shared caches are an attractive attack surface for side-channel attacks, while defenses need to be efficient since the cache is crucial for performance. To address this issue, we present an adaptive and scalable cache partitioning solution (*SCALE*) for protection against cache side-channel attacks. The solution leverages randomness, and provides quantifiable and information theoretic security guarantees using differential privacy. The solution closes the performance gap to a state-of-the-art non-secure allocation policy for a mix of secure and non-secure applications.

Keywords

Cache Partitioning, Side-channel Attacks, Multi-Core Architectures, Microarchitectural Optimizations, Bandwidth Partitioning, Prefetch Throttling

Acknowledgment

The work underlying this thesis would not have been possible without the kind support I have received. First of all, I would like to thank my advisor Per for his invaluable guidance. With his great knowledge in research he has explained and made complex research methodology understandable and feasible. I would also like to thank my co-advisor Madhavan for his expertise and insightful support during my research studies. I have really enjoyed our collaboration and many discussions. I also thank my former co-advisor Miquel for his enthusiasm and many ideas. I have learned a lot from all of you!

I thank Fredrik Dahlgren who has been my examiner. I am also grateful for my, past and present, colleagues at Chalmers, Alexandra, Sonia, Boel, Georgia, Bhavi, Nikela, Christos, Dmitry, Dan, Sverker, Roc, Behrooz, Fatemeh, Rolf, Monica, Lars, Lena, Pedro, Jan, Johan, Evangelos, Albin, Prajith, Stefano, Petros, Stavros, Pirah, Jing and many others, who created a friendly work environment. A special thank you to Boel for answering my questions about differential privacy with enthusiasm.

Finally, I express my deepest gratitude to my supportive family, especially my parents, my husband Nicklas and our wonderful son Hugo. I dedicate this thesis to them.

This research has been funded by the Swedish Research Council (VR) under the PRIME project (registration no. 2019-04929), European Research Council, under the MECCA project, contract number ERC-2013-AdG 340328, Swedish Research Council (Vetenskapsrådet) under the Approximate Algorithms and Computing Systems Project (Projekt-id: 2014-06221) and from the European Union's Horizon 2020 Programme under the LEGaTO Project (www.legato-project.eu), grant agreement no 780681. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

List of Publications

Appended publications

This thesis is based on the following publications:

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

[II] **N. Ramhöj Holtryd**, M. Manivannan, P. Stenström and M. Pericàs "CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling"
Published in 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2021.

[III] **N. Ramhöj Holtryd**, M. Manivannan and P. Stenström "SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations"
Under review.

[IV] **N. Ramhöj Holtryd**, M. Manivannan and P. Stenström "SCALE: Secure and Scalable Cache Partitioning"
To appear in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2023.

Contents

Chapter 1

Introduction

1.1 Background

Single core performance growth plateaued in the early 2000s, necessitating a shift towards multi-core processors [1]. This shift was mainly prompted by the end of Dennard scaling which meant that it was no longer practical to gain performance by increasing core frequency [2]. In the current era, where performance growth is heavily constrained by technological barriers (end of Dennard scaling [3] and slowdown of transistor scaling according to Moore's law [4]), microarchitectural optimizations are expected to play an increasingly important role in ensuring performance scalability of computing systems.

The emergence of processors with an increasing number of cores increases the off-chip memory bandwidth demand. Memory references are increasingly expensive and frequently limit processor performance. Consequently, microarchitectural optimizations, in the core and in the memory system, play a crucial role in determining overall system performance.

Modern multi-core processors have last level cache (LLC) banks distributed across the chip, as shown in Figure 1.1. The on-chip distances increase with additional cores, resulting in non-uniform access latencies to the cache banks. Microarchitectural resources, e.g., the cache banks and off-chip memory bandwidth, are shared among the cores in order to maximize utilization. Workload consolidation is also used where multiple workloads are executed on the same physical system. However, multiple co-running applications cause shared resource contention which can lead to destructive interference and large performance variations across workload, detrimentally impacting average memory access time.

In order to increase memory system performance, microarchitectural optimizations have been proposed which aim to mitigate contention within a shared resource. To mitigate contention, prior works have proposed partitioning of shared resources,

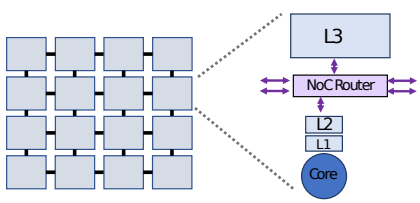


Figure 1.1: Overview of tile-based multi-core architecture.

cache [5–11] and bandwidth [12–14], with the goal of reducing average memory access time and improving performance. Prefetching [15], another memory system optimization, fetches data before it is requested to hide the memory access time. However, inaccurate prefetches have been shown to increase contention [16]. Prefetch throttling [17, 18], adaptively tuning when and what prefetcher settings are used based on application characteristics, has been shown to overcome the drawbacks.

Recent works have shown that coordination of optimizations for multiple microarchitectural resources is advantageous in order to avoid contention bottlenecks and exploit trade-offs between resources [19–25]. It also enables coverage of more and a wider range of applications compared to single resource optimizations. For instance studies have combined cache and bandwidth partitioning [19–21], prefetching and cache partitioning [22, 23], and prefetching and bandwidth partitioning [24, 25] to provide additional performance gains.

The quest for increased performance using microarchitectural optimizations has, however, lead to new and severe security vulnerabilities from side-channel attacks [26, 27]. Recent attacks [28–34] have demonstrated that microarchitectural optimizations, which were assumed to be fundamentally secure for a long time, leak information which can be exploited by an attacker to steal secrets, like encryption keys and user data. These timing-based side-channel attacks exploit the timing variations resulting from the microarchitectural optimizations during program execution. Research has shown that microarchitectural optimizations, widely implemented in commercial processors, like branch predictors [28, 35–38], caches [39–41] and prefetchers [42–47] among others, are prone to attacks. In addition, several not yet commercially implemented optimizations, such as value prediction [48], has also been shown to leak information [49]. Furthermore, efficient attacks continuously emerge targeting defenses, thereby limiting their effectiveness or even rendering the defenses moot altogether [50–61]. Consequently, there is a strong need to be able to leverage microarchitectural optimizations without compromising security.

One of the most well explored categories of timing-based side-channel attacks is through the shared cache LLCs. Efficient utilization of the LLC is essential for processor performance. However, the LLC offers an attractive attack surface because of the channel characteristics, i.e., low noise and high attack bandwidth [62]. Furthermore, cache attacks are challenging to eliminate efficiently because they exploit the timing difference between hits and misses, which is an intrinsic property of caches [63]. There exists three high-level attack categories: i) reuse-based [39,40,64,65] where data is shared between adversary and victim allowing both to access it, ii) conflict-based [41, 66–70] where an adversary creates conflicts to evict lines belonging to the victim, and iii) observation-based [71, 72], where the cache behaviour of the victim leaks information. Current cache partitioning solutions can, in principle, defend against all three categories of attacks by ensuring isolation between adversary and victim processes. But, this comes at the cost of lower cache utilization.

This thesis proposes adaptive microarchitectural optimizations with the goal of improving security and/or performance of multi-core architectures.

1.2 Problem Statements

This thesis is based upon the work presented in Papers **I-IV**. The goal of the work is to provide better performance and/or security for multi-core architectures.

Scalable microarchitectural resource optimization: Prior works [9–11] have shown that locality-aware placement of data in LLCs and fine-grained partitioning are key to designing a well performing cache partitioning solution. Locality-aware data placement reduces cache access times and fine-grained partitioning enables

better adaptation to workloads with varying characteristics. Previous solutions have focused on solving this problem in a centralized manner which have the drawback of introducing an unacceptable overhead when considering frequent reconfigurations for large core counts. Thus in Paper **I**, I aim to address the following question:

Question I: How can we design a scalable cache partitioning solution which supports locality-aware and fine-grained partitioning?

Coordination of optimizations for multiple microarchitectural resources: Previous works [19–25] have proposed coordinated management of only a subset of the optimizations - cache partitioning, bandwidth partitioning and prefetch throttling - with the key insight that managing two instead of one is beneficial since additional trade-offs are enabled. Coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling opens up new trade-offs and interactions, with significant impact on performance, which are not available when considering only a subset of the optimizations. The challenge of managing multiple optimizations is the increased complexity of finding a good allocation, while also considering interactions among resources, which further increases the computational complexity. In the context of coordination of multi-resource optimizations, in Paper **II**, I aim to answer the following question:

Question II: How to enable coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling, avoiding the complexity of evaluation of all possible allocations, while exploiting the new interactions and trade-offs?

Side-channel attacks using microarchitectural optimizations: Prior works have started the important task of analyzing attacks and defenses for microarchitectural optimizations [49, 73–80]. However, most of the works focus only on transient attacks and defenses [74–76, 79, 80] or on quantifying the information leakage [49]. In addition, these analyses fall short of providing a systematic analysis of the similarities across different microarchitectural optimizations, both transient and non-transient, and the underlying root causes which make them vulnerable to attacks. Such an analysis can assist computer architects in understanding the landscape of attacks on a broad range of microarchitectural optimizations and categorize existing defense strategies proposed to thwart such attacks. Thus in Paper **III**, I aim to answer the following question:

Question III: Which are the necessary root causes for timing-based side-channel attacks on microarchitectural optimizations?

Protection against partitioned cache side-channel attacks: Previous works on secure cache partitioning have mostly focused on the enforcement mechanism while assuming static allocation [81–83]. In addition, none of these works satisfy all the requirements of an ideal enforcement mechanism: support for fine-grain partitions, scalability and locality-aware partition placement to reduce average memory access time while being secure. Secure allocation policies have received little attention, despite their significant performance impact. Determining allocations both dynamically and securely is challenging because allocations leak information on applications cache demands. The two existing attempts [84, 85] to secure cache allocations both fall short of providing a performance improvement compared to a shared LLC. To this end in Paper **IV**, I aim to answer the following question:

Question IV: How to provide protection against side-channel attacks through the shared cache, while still enabling high performance through adaptive and scalable partitioning?

1.3 Contributions

This thesis is based on four papers. In the context of single resource optimizations Paper **I** answers the first question, Question I, and the main contribution is:

- A fully distributed and locality-aware cache-partitioning solution consisting of a distributed allocation algorithm which asynchronously negotiates cache allocation decisions together with a flexible enforcement mechanism. The distributed nature of the solution, coupled with low computational overhead, enables a hardware-based implementation. This allows the scheme to scale to large core counts while permitting frequent reconfigurations without invoking the operating system and interrupt program execution.

In Paper **II**, we consider coordination of optimizations for multiple resources and answer the second question, Question II. The main contributions of the paper are:

- An in-depth characterization of the performance impact of cache, bandwidth and prefetching on the entire SPEC CPU2006 suite. Our characterisation results provide several insights: i) a majority of the applications (over 90%) are sensitive to one or multiple optimizations, ii) managing cache, bandwidth and prefetch opens up opportunities for exploiting more interactions and improving performance, and iii) managing cache, bandwidth and prefetch jointly has the potential to outperform combinations of two of the optimizations.
- A microarchitectural resource manager that dynamically coordinates cache partitioning, bandwidth partitioning and prefetch throttling, considering the interactions between them. The solution works by employing individual resource managers to determine appropriate settings for each resource and a coordination mechanism to enable inter-resource trade-offs.

In Paper **III** we consider security of microarchitectural optimizations and answer the third question, Question III. The main contributions in this systematization-of-knowledge paper are:

- Identification of the four root causes for timing-based side-channel attacks using a wide range of microarchitectural optimizations: *determinism*, *sharing*, *access violation* and *information flow*. For a specific attack a subset, or all, of the exploited root causes are necessary for the attack to succeed.
- A framework and systematic analysis of both transient and non-transient execution attacks and defences on a broad range of microarchitectural optimizations, highlighting similarities and differences across the attacks and defenses.

In Paper **IV** we consider protection against side-channel attacks for a single resource, the LLC, and answer the fourth question, Question IV. The main contributions of the paper are:

- A holistic solution for secure, adaptive and scalable cache partitioning which provides protection against cache side-channel attacks. The proposed cache allocation policy leverages randomness to defend against attacks on the cache allocation policy side-channel. The enforcement mechanism supports secure, fine-grained and locality-aware partitioning of cache capacity.
- Demonstration of strong and quantifiable security guarantees within a configurable range, leveraging differential privacy [86]. Outside this range, weak security guarantees are provided while reducing cache allocation policy side-channel bandwidth.

1.4 Organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2 a summary of each paper is presented. Finally, Chapter 3 concludes the thesis, and discusses some possible future research directions.

Chapter 2

Summary of the Papers

This chapter provides summaries of Papers **I** to **IV**. All papers are provided as appendices to the thesis.

2.1 Paper I

This section provides a summary of Paper **I** titled "DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors" and published in the proceedings of *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2020, New Orleans, LA, USA, 2020.*

2.1.1 Summary

A cache partitioning solution consists of two parts: an allocation policy which decides the size of the partitions for each application and an enforcement mechanism to enforce them. Prior work have proposed allocation policies [5, 9, 10, 87, 88], but the drawback is their reliance on a centralized algorithm to determine allocations which limits how frequently reconfigurations can be performed. Prior work on enforcement mechanism has shown that fine-grained partitioning, i.e., support of many and varying partitions, is beneficial [7, 8, 11, 89–92] but the main shortcoming is that they do not take locality into account. A few proposals [9, 10] have tried to address the lack of locality awareness, but their solutions either require costly broadcasts or rely heavily on software support. Furthermore, the allocation policies in these proposals use a centralized algorithm which affects the overhead. A high overhead presents two problems: Firstly, limiting the frequency of reconfiguration which is important in order to adapt to application phase changes, and secondly, introducing unpredictable jitter in application execution.

An ideal cache partitioning, in the context of multi-core systems, needs to have a number of different characteristics. Firstly, it needs to be locality-aware and place partitions in a way which minimizes the on-chip distance. Secondly, it needs to support fine-grained partition sizes and to adapt quickly to application-phase changes, in order to at each instance of time have the most suitable allocation. Finally, it needs to have low enough overhead for performing allocation decisions and cause minimal OS intervention in order to be scalable.

In Paper **I**, we present DELTA, a scalable cache partitioning solution consisting of a fully distributed allocation policy and a locality-aware enforcement policy. In contrast to prior work, DELTA works with standard LRU-replacement policy and is implementable in hardware. DELTA's allocation policy consists of two parts:

one intra-bank and one inter-bank mechanism. The inter-bank algorithm works by asynchronous exchanges of so called challenges among cores. The challenges include the potential performance benefit of increased cache capacity, and the inter-bank algorithm helps applications with larger performance benefit to gain more cache capacity. The intra-bank algorithm redistributes the cache capacity within a cache bank, giving a larger allocation to applications with a larger potential performance gain. DELTA's enforcement mechanism combines bank- and way-level partitioning to enable fine-grained and locality-aware partitions. In order to locate data in the LLC a per-core Cache Bank Table (CBT) is used. The CBT contains mappings between addresses and cache banks, and it is accessed in parallel with the L2 cache in order to find the right LLC bank.

The distributed inter-bank allocation algorithm in DELTA uses two metrics, *pain* and *gain* as part of the asynchronous negotiation. The pain estimates the potential performance decrease with lost cache capacity while gain estimates the potential performance increase with additional cache space. The challenge message contains the potential gain that a given application would experience with additional cache capacity. The challenged cache bank compares its own pain with the gain of the challenging bank. If the gain is greater, a portion of the cache capacity belonging to the challenged bank is remapped to the challenger. The inter-bank allocation algorithm redistributes the cache capacity from applications which have a lower performance decrease when giving up cache capacity to applications which have a higher performance gain from additional cache space.

DELTA is evaluated with detailed simulations on both a 16- and 64-core tiled multi-core architecture using the Sniper simulator [93]. The performance of DELTA is compared against a private cache implementation, a shared NUCA implementation and an idealized centralized solution. The ideal centralized solution is used in order to evaluate the quality of the allocations performed by DELTA and uses the best known cache partitioning algorithm, Lookahead [5]. It does not model the overhead of computing allocation decisions. In the evaluation, an analysis of the overhead of calculating the allocations for DELTA and the best centralized algorithm is shown for different core counts. The analysis shows that the centralized algorithms overhead in time per invocation makes them unusable for large core counts with frequent reconfigurations.

The evaluation on a 16-core multi-core architecture shows that DELTA improves performance by on average 9% compared to an unpartitioned S-NUCA and by 6%, on average, compared to private caches (i.e., equal partitioning). The performance of the allocation using DELTA's is 2% lower than the idealized centralized solution. On a 64-core architecture, DELTA improves performance by on average 16% over an unpartitioned S-NUCA and is within 1% of the idealized centralized solution.

In summary, Paper **I** contributes with a distributed partitioning solution which performs close to an ideal centralized solution. The distributed algorithm has low computational overhead which permits it to be implemented in hardware and allows for frequent reconfigurations while the enforcement scheme enables locality-aware placement.

2.2 Paper II

This section provides a summary of Paper **II** titled "CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling" and published in the proceedings of *30th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2021.*

2.2.1 Summary

Management of optimizations for multiple resources combining either cache and bandwidth partitioning [19–21], or cache partitioning and prefetching [22, 23], or prefetch and bandwidth partitioning [24, 25], has been shown to be beneficial in reducing average memory access time and increasing performance. However, no study so far has considered coordinated management of all three optimizations. Coordinately managing all three optimizations provides several advantages. Firstly, it can potentially enable improved performance in more applications and addressing a broader range of application characteristics. Secondly, new trade-offs and interactions are enabled which have significant impact on performance.

In Paper **II**, we show an in-depth performance characterization of the applications in the SPEC CPU2006 suite. The results show that 90% of the applications have performance sensitivity (over 10% change in IPC) to at least one of the optimizations, and 70% are also sensitive to multiple optimizations. We make several observations regarding the interactions and trade-offs between the different optimizations: i) the allocation of cache and bandwidth affects the performance impact of prefetching, ii) larger bandwidth allocation can compensate for inaccurate prefetches, iii) for an application sensitive to both resources the same performance can be gained, by either increasing cache size or enabling prefetching, or by either increasing bandwidth or cache allocation. In Paper II we also show, using exhaustive search, that for more than 400 random workloads of four applications, coordinately managing three resources is better than any combination of two resources.

In Paper **II**, we present CBP, a coordination optimization for adaptive management of cache partitioning, bandwidth partitioning and prefetch throttling. The design is guided by observations and results from the performance characterization. CBP consists of three local controllers, one for each optimization, and a coordination mechanism, see Figure 2.1. With CBP, the three local controllers are dynamically tuned and guided by heuristics, to give a good allocation of the resources, considering application characteristics and possible trade-offs and interactions. Recalibrations are performed periodically. First, cache space is allocated since altogether avoiding a memory access has higher impact than reducing the latency. As a next step, bandwidth is allocated based on the queuing latency of the memory requests and

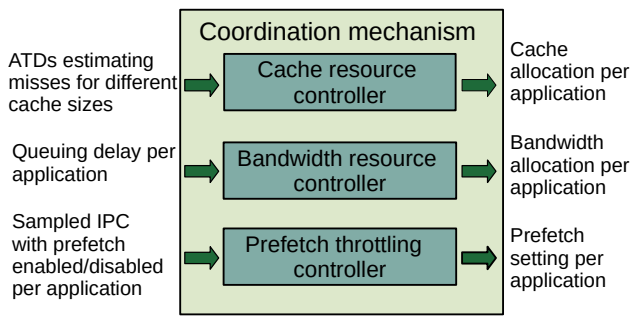


Figure 2.1: Overview of CBP resource manager.

taking into consideration the cache allocation. Finally, the prefetcher setting is determined based on the current allocation of cache and bandwidth. The cache resource controller estimates the number of misses using auxiliary tag directories (ATDs) and allocates the capacity in a way that reduces the aggregate number of cache misses. The bandwidth resource controller uses the queuing delay per application in order to allocate the capacity, where the application experiencing the largest queuing delay will get the largest allocation. The prefetcher setting is determined by sampling, for a short time interval, the IPC with prefetching active/inactive and enabling the prefetcher only if the speedup is large enough.

CBP is evaluated against nine different single- and multi-resource managers. The baseline configuration represents an unpartitioned shared cache with unpartitioned bandwidth and prefetching disabled. The comparison points are CPpf [22] a state-of-the-art scheme combining cache partitioning and prefetching, equal partitioning of cache and bandwidth, as well as resource managers controlling only one resource or two of the three resources. We also investigate the impact of using a different prefetcher and changed coordination and ordering in the resource manager.

CBP is evaluated with multi-programmed workloads on a 16-core multi-core architecture using the Sniper simulator [93]. CBP improves performance by 11% on average compared to CPpf, and by 50% on average compared to the baseline. We use the user-oriented fairness metric Average Normalized Turnaround Time (ANTT) in order to show that CBP does not increase performance at the cost of fairness. CBP increases fairness by 8% compared to CPpf and by 27% compared to baseline. According to the experimental results, the proposed multi-resource manager provides an effective solution for the main research problem. The evaluation shows that coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling is better than any pair-wise optimization and improves upon the state-of-the-art.

In summary, Paper **II** contributes with optimizations which allow for dynamic and adaptive coordination of cache partitioning, bandwidth partitioning and prefetch throttling to achieve better performance, both compared to the state-of-the-art and any subset of optimizations.

2.3 Paper III

This section provides a summary of Paper **III** titled "SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations". The paper is currently under review.

2.3.1 Summary

Microachitectural optimizations are important for providing performance scalability, but have also been shown to lead to security vulnerabilities. Previous works have started the important task of analyzing attacks and defenses for different microarchitectural optimizations [73–80]. However, most of the works focus only on transient attacks and defenses [74–76, 79, 80], SW-based defenses [78] or cover a limited set of non-transient attacks [73]. Pandora [49] considers a broader set of non-transient microarchitectural optimizations and provides microarchitectural leakage descriptors (MLDs) which quantify the information leakage. The MLDs show if a specific optimization can leak and how much information is leaked (1-bit or a few bits). Unfortunately, this information falls short of providing a systematic analysis of the similarities across different microarchitectural optimizations and the underlying root causes which make them vulnerable to attacks.

In Paper **III**, the four root causes for timing-based attacks on microarchitectural optimizations are identified. The four root causes are: *determinism*, *sharing*, *access violation* and *information flow*. Our key insight is that a subset (or all) of the root causes are exploited by attacks and eliminating any of the exploited root causes, in any attack step, is enough to provide protection. Here, determinism causes microarchitectural optimizations to be triggered in the same way under the same pre-conditions, leading to predictable microarchitectural state transitions and timing variations. Sharing of microarchitectural state, which is accessible to both the adversary and the victim, enables the creation of a side-channel. Access violation enables access to a secret outside of the intended protection domain. Finally, information flow refers to exchange of information through microarchitectural state.

We present a framework where the architecture model is represented as a finite state machine (FSM) where the *architectural state*, comprising software-visible registers and memory, is the externally visible interface, that is accessible to a program. A FSM transition is caused when instruction execution leads to a change in the architectural state. The microarchitecture represents an implementation of the FSM specification comprising typically several microarchitectural optimizations, which uses a set of microarchitectural resources to implement the intended functionality. We define *microarchitectural state* (MS) as a snapshot of the state of all the microarchitectural resources in the system at a specific time.

Using this framework, we present an abstract model of an attack in Figure 2.2 which shows the different steps involved to communicate the secret from a victim to an adversary. In our model, we define a step as a tuple of current microarchitectural state and action which leads to a new state, $\{MS_{current},\ action\} \rightarrow MS_{next}$. When the *setup* step is performed the initial state (MS_I) transitions to the primed state (MS_P). The *setup* step ensures that the necessary preconditions are in place to

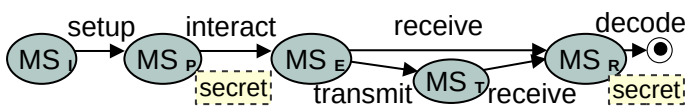


Figure 2.2: MS transitions in different steps of an attack.

encode the secret into MS_P in the next step of the attack. When the *interact* step is performed the secret is accessed and is encoded in the microarchitectural state, which transitions to encoded state (MS_E). The secret is encoded specifically through the state of one or more microarchitectural resources. If the secret is encoded through a microarchitecture resource state, which is accessible to both the victim and the adversary, it can potentially be used as a side-channel to communicate the secret. The *transmit* step is optional, and used in scenarios where the adversary does not use the same microarchitectural state for encoding of the secret and for the side-channel. Finally, when the *receive* step is performed, the adversary accesses the microarchitectural state of the specific resource(s) and observes timing variations based on the encoded secret while the state transitions to received (MS_R).

We present a systematization of timing-based side-channel attacks available in literature on an extensive set of microarchitectural optimizations: cache, prefetching, branch prediction, computational simplification, speculative execution and value prediction. The analysis covers both transient and non-transient attacks. For the attacks using each optimization we answer the following questions: i) which microarchitectural resource(s) is/are used? ii) which root cause(s) are necessary for the attack to succeed in each step of the attack? iii) under which threat model(s) is the attack possible?

We present a systematization of defenses for the timing-based attacks where we show which root cause(s) and attack step(s) the defenses target. The analysis show that the proposed defenses target one or more of the identified root causes. An attack is stopped if any of the root causes are eliminated, in any attack step. We observe that similar defenses can be/are applied across different microarchitectural optimizations, with the same root cause vulnerabilities. Furthermore, there are commonalities in the defense strategies used to protect against attacks on these diverse microarchitectural optimizations. Some of these common defense strategies which we identify include disabling the optimization, to restrict all the root causes; isolating the state related to the optimization, to restrict sharing; applying randomization and/or restriction, to limit information flow and introducing permission checks, to limit resources from, exposing/accessing state outside of the intended domain.

In summary, Paper **III** contributes with the four root causes for timing-based side-channel attacks on a wide range of optimizations, as well as a systematization of published attacks and defenses, which highlights their commonalities.

2.4 Paper IV

This section provides a summary of Paper **IV** titled "SCALE: Secure and Scalable Cache Partitioning". The paper is accepted to IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2023.

2.4.1 Summary

The challenge of defending against cache side-channel attacks is to provide protect against both shared memory-based, conflict-based and occupancy-based attacks while, preferably, providing performance improvement over a shared LLC. Initial works propose to partition cache capacity *statically* among co-running applications [81]. However, static policies do not adapt to changing requirements of applications and provide lower performance than *dynamic* cache partitioning policies [5, 8, 10, 94–98]. Prior works have mostly focused on the enforcement mechanism while assuming static allocation [81–83]. In addition, these works do not satisfy all the requirements of an ideal enforcement mechanism: support for fine-grain partitions, scalability and locality-aware partition placement. Commercially available solutions, such as Intel CAT [99], also do not satisfy these requirements and are furthermore not secure since they allow leakage of information across partitions which can be exploited [82].

Dynamic and adaptive cache allocation, even though desirable from a performance perspective, is challenging because allocations can leak information about individual applications cache demands, which can be used in an attack. There are two existing attempts to provide a secure cache allocation policy: SecDCP [84] and OPTIMUS [85]. SecDCP provides the notion of security tiers (confidential and public applications) and permits one-way leakage from the public tier by only considering public applications' cache demand for determining allocations. OPTIMUS, places a bound on the amount of information leakage by only performing a reconfiguration once at the start of the execution. However, neither solution improves performance over a shared and unpartitioned LLC.

In Paper **IV**, we present SCALE, a dynamic cache partitioning solution that enables secure allocation while considering the cache demand of both secure and non-secure applications continuously during execution. The enforcement mechanism can accommodate a wide range of partition sizes, is scalable and locality-aware. SCALEs approach is based on insights from a detailed characterization of the cache allocation policy side-channel and how it can be exploited to launch conflict-based and occupancy-based attacks to obtain sensitive information. Our analysis shows that information leaks, due to fine-grained changes in cache allocation for the victim, that are caused and/or observed by an adversary, lead to exploits. Protecting against such leaks is important since cryptographic libraries have small working set sizes [100–102]. Consequently, SCALE aims at providing strong guarantees against information leakage due to fine-grained changes in allocation. Leaking high-level occupancy information, e.g., showing that a co-running application is memory intensive, has not been exploited for side-channel attacks. Furthermore, such occupancy information is also leaked by defenses that protect the LLC side-channel through randomizing placement of lines in the cache [103]. Consequently, SCALE therefore provides weaker security guarantees in such cases by permitting controlled leakage about high-level cache occupancy.

SCALEs cache allocation policy leverages *randomness* to achieve the primary design goals, of providing a secure and scalable solution. Specifically, the allocation policy adds noise to the deterministically computed cache allocations. Leveraging differential privacy, we provide quantifiable security guarantees that the amount of noise will provide the requested security level, set by the system administrator. By randomisation we make the allocations non-deterministic, i.e., the same input can lead to different allocations.

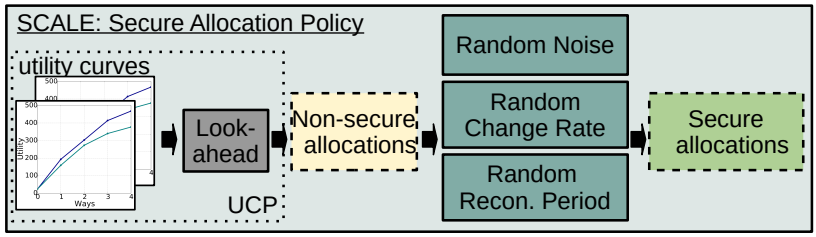


Figure 2.3: Overview of the components in SCALEs secure allocation policy.

An overview of SCALEs cache allocation policy is shown in Figure 2.3. Firstly, utility monitors are used in order to estimate the utility of different cache allocations, for the co-executing applications. This is used by the Lookahead algorithm to compute new insecure allocations for the upcoming reconfiguration. Secondly, the different defense mechanisms introduce randomness into the allocations. The process is repeated at each reconfiguration period. The first defense adds random noise, generated using a Laplace distribution, to update the allocations computed by the non-secure policy. Although noise protects against conflict-based attacks, we notice that patterns can still be observed, which can be leveraged in occupancy-based attacks [71]. To address this, we introduce two additional levels of randomization. The second defense randomizes the allocations change rate, which determines the extent to which allocations can change, in a single reconfiguration period. The third defense randomizes the length of the reconfiguration period, which randomizes the timing of allocation changes while also reducing the channel bandwidth.

The security analysis leveraging differential privacy, demonstrates that SCALE is secure and provides a method for system administrators to configure the randomization according to the security requirements. SCALE allows configurability based on the security requirements on a per application basis. SCALE provides strict security guarantees for allocation changes within the configured range, while providing weak security guarantees and lower channel bandwidth for larger changes. In addition, SCALE also supports running a mix of secure and non-secure applications concurrently while ensuring security and performance gains from cache partitioning.

The secure enforcement mechanism proposed in SCALE, builds on the DELTA enforcement mechanism [98]. DELTA combines bank- and way-level partitioning to support fine-grained partitions and locality-aware mapping of data. The adaptations for SCALE enable secure enforcement, reduce the overheads associated with handling shared data and simplify the design.

SCALE is evaluated on a 16-core tiled multi-core architecture and compared against the state-of-the-art secure cache partitioning solutions, i.e., SecDCP [84], OPTIMUS [85] and ScatterCache [104]. We show that SCALE outperforms prior works and improves performance by 14%, on average, and by up to 39% compared to an unpartitioned shared LLC. In addition, we evaluate SCALEs suitability for commercial designs using an enforcement mechanism similar to Intel CAT [99]. The evaluation shows that the proposed allocation policy leveraging randomization can defend against cache allocation policy side-channel attacks while retaining most of the performance benefits of a state-of-the-art non-secure allocation policy like UCP.

In summary, Paper **IV** contributes with a holistic solution for protecting the cache against side-channel attacks which improves performance beyond state-of-the-art and performs close to a non-secure allocation policy. In addition, SCALE provides quantifiable security guarantees using differential privacy.

Chapter 3

Concluding Remarks and Future Work

Microarchitectural optimizations, especially in the memory system, are important to ensure performance scalability of multi-core architectures. However, microarchitectural optimizations can also be exploited in side-channel attacks.

This thesis proposes adaptive microarchitectural resource management optimizations to improve security and/or performance of multi-core architectures and a systematization-of-knowledge of timing-based side-channel attacks and their defenses. Paper **I** concerns an optimization for the cache, where a distributed cache partitioning solution is proposed to avoid high computational overhead when determining allocations. The results show that it is possible to design a distributed solution for cache partitioning, which performs close to an idealized centralized solution. Paper **II** concerns management of multiple microarchitectural resources and proposes a coordinated scheme with cache partitioning, bandwidth partitioning and prefetch throttling. Furthermore, the results for the coordinated multi-resource manager outperform any resource manager for two resources and improves upon state-of-the-art. Paper **III** concerns timing-based side-channel attacks and identifies the four root causes for such attacks. The paper also presents a systematization of attacks using a wide range of optimizations, and their defenses. Paper **IV** concerns protection against timing-based attacks and proposes a secure and scalable cache partitioning solution. The results show that performance, close to a non-secure solution, can be achieved with quantifiable security guarantees.

There are several interesting directions for future work. In the context of Paper **I**, one direction would be to target fairness or Quality of Service (QoS), instead of performance, as is currently done. Another future research direction would be to extend the multi-resource management scheme in Paper **II** to also perform coordinated management of more resources such as core frequency, memory capacity and disk bandwidth, in order to further improve performance. The challenge with adding additional resources would be how to take the additional interactions and trade-offs into consideration without reaching an unacceptable overhead which would eliminate the performance improvement. One interesting direction for future work, in the context of Paper **III**, is to investigate and/or extend the root cause framework to include microarchitectural optimizations for security, such as Intel SGX, and considering power-based side-channels. Another potential direction would be to use a similar approach as proposed in Paper **IV** to secure other microarchitectural optimizations against timing-based side-channel attacks. For example, an optimization such as branch predictor or a prefetcher which can use partitioning for protection, could use

similar defenses to enable secure and high performance partitioning with security guarantees from differential privacy. Likewise, also optimizations with only temporal sharing such as ports, could use randomization backed by differential privacy to provide protection. Another avenue to explore is to combine the distributed cache allocation policy proposed in Paper **I** with the secure cache allocation solution proposed in Paper **IV**. Using a distributed solution for allocation can open up new possible ways to defend the cache, as opposed to a centralized solution. The distributed allocation algorithm could, for example, only allow applications with the same security level to exchange challenges and influence each others cache allocations.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, p. 248–259, May 2000.

[2] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[3] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[4] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[5] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.

[6] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," vol. 2018-Febru, 2018, pp. 104–117.

[7] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-39*, 2012.

[8] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.

[9] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.

[10] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.

[11] X. Wang, S. Chen, J. Setter, and J. F. Martinez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *Proc. HPCA-23*, 2017.

[12] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers." Institute of Electrical and Electronics Engineers Inc., 11 2018, pp. 1–14.

[13] D. R. Hower, H. W. Cain, and C. A. Waldspurger, "Pabst: Proportionally allocated bandwidth at the source and target," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 505–516.

[14] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[15] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 28, pp. 1–69, 5 2014.

[16] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," 2009, p. 316.

[17] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 1, 1993, pp. 56–63.

[18] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. USA: IEEE Computer Society, 2007, p. 63–74.

[19] A. Sahu and S. Ramakrishna, "Creating heterogeneity at run time by dynamic cache and bandwidth partitioning schemes." Association for Computing Machinery, 2014, pp. 872–879.

[20] R. Bitirgen, E. Ipek, and J. F. Martínez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," 2008, pp. 318–329.

[21] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," vol. 19. ACM.

[22] J. Xiao, A. D. Pimentel, and X. Liu, "Cppf: A prefetch aware llc partitioning approach." Association for Computing Machinery, 8 2019, pp. 1–10.

[23] G. Sun, J. Shen, and A. V. Veidenbaum, "Combining prefetch control and cache partitioning to improve multicore performance." Institute of Electrical and Electronics Engineers Inc., 5 2019, pp. 953–962.

[24] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 141, 2011.

[25] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors." Association for Computing Machinery (ACM), 2011, p. 37.

[26] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, jan 2019.

[27] M. D. Hill, "Technical perspective: Why 'correct' computers can leak your information," *Commun. ACM*, vol. 63, no. 7, p. 92, jun 2020.

[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: Reading kernel memory from user space," *Commun. ACM*, vol. 63, no. 6, p. 46–56, may 2020.

[30] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 361–372, jun 2014.

[31] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.

[32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 991–1008.

[33] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3917–3934.

[34] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 753–768.

[35] O. Acıiçmez, c. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 225–242.

[36] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 312–320.

[37] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[38] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *SIGPLAN Not.*, vol. 53, no. 2, p. 693–707, mar 2018.

[39] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.

[40] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732.

[41] C. Percival, "Cache missing for fun and profit," in *http://www.daemonology.net/papers/ htt.pdf*, 2005, pp. 974–987.

[42] "AMD prefetch attacks through power and time," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022.

[43] Y. Chen, L. Pei, and T. E. Carlson, "Leaking control flow information via the hardware prefetcher," 2021.

[44] P. Cronin and C. Yang, "A fetching tale: Covert communication with the hardware prefetcher," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 101–110.

[45] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 131–145.

[46] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 368–379.

[47] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1491–1505.

[48] M. Lipasti and J. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996, pp. 226–237.

[49] J. R. Sanchez Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening pandora's box: A systematic study of new ways microarchitecture can leak private data," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 347–360.

[50] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, *Speculative Probing: Hacking Blind in the Spectre Era*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1871–1885.

[51] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1046–1060.

[52] J. Fustos, M. Bechtel, and H. Yun, "Spectrerewind: Leaking secrets to past instructions," in *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, ser. ASHES'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 117–126.

[53] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 39–54.

[54] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 360–371.

[55] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 9–12, 2020.

[56] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache," *CoRR*, vol. abs/1908.03383, 2019.

[57] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1110–1123.

[58] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 987–1002.

[59] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 955–969.

[60] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988.

[61] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842.

[62] O. Sibert, P. A. Porras, and R. Lindell, "The intel 80x86 processor architecture: Pitfalls for secure systems," 1995.

[63] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 341–353.

[64] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 490–505.

[65] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham: Springer International Publishing, 2014, pp. 299–319.

[66] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418.

[67] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 591–604.

[68] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[69] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[70] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 813–825.

[71] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 639–656.

[72] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912.

[73] J. Szefer, "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses," *J. Hardw. Syst. Secur.*, vol. 3, no. 3, pp. 219–234, 2019.

[74] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 249–266.

[75] C. Canella, S. M. Pudukotai Dinakarrao, D. Gruss, and K. N. Khasawneh, "Evolution of defenses against transient-execution attacks," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 169–174.

[76] G. Hu, Z. He, and R. B. Lee, "Sok: Hardware defenses against speculative execution attacks," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021, pp. 108–120.

[77] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.

[78] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," 2021.

[79] Z. He, G. Hu, and R. Lee, "New models for understanding and reasoning about speculative execution attacks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 40–53.

[80] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Comput. Surv.*, vol. 54, no. 3, may 2021.

[81] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005, page@cs.bris.ac.uk 13017 received 22 Aug 2005.

[82] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[83] S. K. Gururaj Saileshwar and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," 2021.

[84] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[85] H. Omar, B. D'Agostino, and O. Khan, "Optimus: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks," *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1558–1570, 2020.

[86] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 265–284.

[87] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *PACT-15*, 2006.

[88] D. Thiebaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. on Comp.*, 1992.

[89] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ISCA-40*, 2013.

[90] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. HPCA-24*, 2018.

[91] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, 2013.

[92] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.

[93] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.

[94] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *Proc. ICPP*, Washington, DC, USA, 2009.

[95] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.

[96] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-41*, vol. 40, 2012.

[97] W.-C. Kwon, T. Krishna, and L.-S. Peh, "Locality-oblivious cache organization leveraging single-cycle multi-hop NoCs," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS '14*, 2014.

[98] N. Holtryd, M. Manivannan, P. Stenström, and M. Pericàs, "Delta: Distributed locality-aware cache partitioning for tile-based chip multiprocessors," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 578–589.

[99] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 657–668.

[100] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 494–505, jun 2007.

[101] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[102] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage,"

in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 131–145.

[103] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom, "Cachefx: A framework for evaluating cache security," *arXiv preprint arXiv:2201.11377*, 2022.

[104] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692.

Paper I

DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors

N. Holtryd, M. Manivannan, P. Stenström and M. Pericàs

Reprint from

IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2020, New Orleans, LA, USA, 2020.

2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors

Nadja Holtryd, Madhavan Manivannan, Per Stenström, Miquel Pericàs
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email: {holtryd, madhavan, per.stenstrom, miquelp}@chalmers.se

***Abstract*—Cache partitioning in tile-based CMP architectures is a challenging problem because of i) the need to determine capacity allocations with low computational overhead and ii) the need to place allocations close to where they are used, in order to reduce access latency. Although, previous solutions have addressed the problem of reducing the computational overhead and incorporating locality-awareness, they suffer from the overheads of *centrally* determining allocations.**

In this paper, we propose DELTA, a novel *distributed* and locality-aware cache partitioning solution which works by exchanging asynchronous challenges among cores. The distributed nature of the algorithm coupled with the low computational complexity allows for frequent reconfigurations at negligible cost and for the scheme to be implemented directly in hardware. The allocation algorithm is supported by an enforcement mechanism which enables locality-aware placement of data. We evaluate DELTA on 16- and 64-core tiled CMPs with multi-programmed workloads. Our evaluation shows that DELTA improves performance by 9% and 16%, respectively, on average, compared to an unpartitioned shared last-level cache.

***Index Terms*—cache partitioning, multicore architectures, performance isolation**

I. Introduction

Efficient use of cache resources on chip multiprocessors (CMPs) is necessary in order to bridge the speed gap between processor and main memory. The last-level cache (LLC) is usually shared among all cores to maximize utilization. Unconstrained sharing, however, can result in destructive interference between workloads and lead to large performance variation, degrade throughput and violate per-application Quality of Service (QoS) requirements. Cache partitioning can mitigate destructive interference by isolating cache space between cores/applications.

A partitioning solution typically comprises two components: an *allocation policy*, to decide the size of the partitions for each application, and an *enforcement mechanism* to enforce the partitions. With regard to the allocation policy, known approaches target different objectives, e.g. to maximize throughput or improve fairness [1]–[3]. Utility-based cache partitioning (UCP) [1] aims to maximize throughput by assigning cache ways to applications that benefit most from the cache capacity. To do so, UCP leverages the Lookahead algorithm. The algorithm determines dynamic cache allocation based on marginal utility and partitions ways in a monolithic cache between applications but it has a high computational complexity. Approaches to determine cache allocations with lower computational overhead have been proposed [4]. However, as our evaluation of the overheads (described in Section IV) shows, the scalability of these proposals remains limited by their reliance on a centralized allocation algorithm. This reliance presents two problems. First, the execution time of the allocation algorithm limits the frequency of reconfiguration, especially as we scale to large core counts. And second, the invocation of the algorithm introduces unpredictable jitter in application execution. OS noise (jitter) has been identified as a major cause of both execution time unpredictability and untimely synchronization [5], [6]. This is particularly bad for multithreaded applications that rely on bulk synchronous parallelism (BSP) [6], [7]. As a consequence, centralized allocation approaches cannot be utilized when scaling to large core counts and requiring frequent reconfigurations.

Different enforcement mechanisms for cache partitioning have been proposed. Way and set partitioning are proposed in the context of monolithic caches with few cores [1], [2], [8], [9]. These schemes have the drawback of only supporting a limited number of coarse-grained partitions. Solutions have been proposed to enable fine-grained partitioning [10]–[17]. The main shortcoming of these techniques is that they do not take locality into account when partitioning LLCs in tiled CMPs. A few proposals have tried to address this limitation by enabling locality-aware placement [4], [18]. However, the allocation policies used in these proposals rely on a centralized allocation component and inherit its shortcomings.

An ideal cache partitioning solution should be fine-grained to support many and varying partitions, be locality-aware to place data close to where it is used, and adapt quickly to changes in application-phase behavior while still ensuring that allocation operations can be performed in a scalable manner, with low overhead and minimal OS intervention. We propose DELTA, a novel scalable cache partitioning solution for tile-based CMPs, that utilizes a *distributed* allocation policy and a locality-aware enforcement mechanism. In contrast to prior work, our solution uses a completely distributed and asynchronous allocation algorithm, works with a standard LRU-replacement policy, does locality-aware enforcement and is virtually transparent to the full software stack.

DELTA's Allocation Policy: DELTA's allocation algorithm comprises an inter-bank and an intra-bank component. The

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00066

inter-bank algorithm determines capacity allocations by asynchronously exchanging *challenges* among cores. A challenge represents the performance benefit of obtaining increased cache capacity and helps an application with larger performance potential to gain more cache capacity. The algorithm uses coarse-grained shadow-tags to collect reuse-distance information with minimal overhead, which is used as the basis for computing a challenge. The intra-bank algorithm periodically redistributes the cache capacity within a bank by giving more space to the application that has a larger potential performance gain. The distributed nature of the algorithm enables quick and incremental adaptation to program phase changes without requiring a central entity to determine and perform chip-wide reallocation of cache capacity for the different applications.

DELTA's Enforcement Mechanism: DELTA combines way partitioning and bank-level partitioning to achieve a locality-aware and fine-grained partition-enforcement mechanism. The partitioning solution uses per-core *Cache Bank Tables* (CBTs), where mappings between addresses and banks are recorded. When a request needs to access the LLC, the CBT is used to identify the cache bank that the address is mapped to. Inside each bank, way partitioning is used to divide the capacity. The flexibility of the enforcement mechanism makes it possible to keep data close to where it is used.

In summary, we make the following contributions:

(a) We propose DELTA, a fully distributed and locality-aware cache-partitioning solution. The distributed allocation algorithm asynchronously negotiates and makes effective cache allocation decisions. The flexibility of the enforcement mechanism enables locality-aware mappings. The distributed nature of the solution, coupled with low computational overhead, enables a hardware-based implementation. This allows the scheme to scale to large core counts while permitting frequent reconfigurations without invoking the operating system.

(b) We describe a novel allocation policy consisting of a coarse-grained and a fine-grained component which are responsible for carrying out inter- and intra-bank allocations, respectively. The inter-bank algorithm uses challenges to expand into multiple banks, while the intra-bank algorithm allows the allocation to grow within a bank.

(c) We present a reconfigurable NUCA enforcement mechanism that enables locality-aware mapping. The two-level mechanism combines coarse-grained, bank-level partitioning with fine-grained way partitioning. The CBT enforces flexible mapping of addresses to cache banks, which enables placing data close to where it is used.

We evaluate our solution on 16- and 64-core tiled CMPs. With multi-programmed workloads on a 16-core CMP we obtain speed-ups of up to 16% (geom. mean 9%) compared to an unpartitioned S-NUCA and up to 11% (geom. mean 6%) compared to private caches (equal partitioning). On the 64-core CMP DELTA improves performance by up to 28% (geom. mean 16%) over an unpartitioned S-NUCA.

The rest of the paper is organized as follows: Section II describes our proposed solution in detail. Section III discusses the methodology and Section IV presents the evaluation of the proposal. We provide an overview of related work in Section V and conclude in Section VI.

II. DELTA Cache Partitioning

Section II-A provides an overview of DELTA, followed by a detailed presentation of the algorithms and mechanisms in subsequent sections.

A. Overview

Both the allocation policy and the enforcement mechanism have an $inter\text{-}bank$ and an $intra\text{-}bank$ component. The inter-bank component takes care of the allocation and enforcement across cache banks, and the intra-bank part handles the allocation and enforcement within the banks.

DELTA allocation policy: Figure 1 provides an overview of the distributed allocation algorithm. The inter-bank allocation algorithm works by tiles periodically sending out challenge messages, as shown in Figure 1 (Step #1). The mechanism relies on two metrics called *pain* and *gain* (see Table I). A challenge message contains the potential *gain* that a given application would experience if it were to get additional cache capacity. The challenged tile compares its own *pain*, owing to predicted decrease in performance because of lost cache space, with the gain (Step #2), (see Section II-B2 for details about pain and gain). If the gain is greater, the challenged tile gives up space in the cache bank and informs the challenger tile with a response message (Step #3). A portion of the addresses ($[A_k - A_l]$ in Figure 1) belonging to the application running in the challenger tile is remapped to the cache bank in the challenged tile (Step #4). The addresses that have been remapped to a different bank are then invalidated in their previous location. The inter-bank allocation policy helps applications acquire additional cache capacity by mapping data to cache banks in other tiles.

The second part of the allocation algorithm, the intra-bank algorithm, governs changes within a bank. The intra-bank algorithm is invoked periodically in every cache bank. In each interval some ways are transferred to the partition with most gain from the partition with the least. This way, reassignment has little overhead since it does not affect the mapping of addresses to banks, and therefore does not lead to invalidations. While the inter-bank allocation algorithm helps an application to expand its working set into other tiles and get a fixed capacity, the intra-bank algorithm helps in fine-tuning the capacity in banks that an application has already expanded

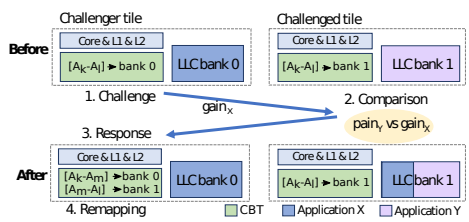


Fig. 1: Overview of steps in DELTA allocation.

$Gain$	Predicted performance increase due to increased cache space
$Pain$	Predicted performance decrease due to lost cache space

TABLE I: Terminology

into. The intra-bank algorithm reports information about ways that an application wins/loses in a cache bank outside the tile and this acts as a feedback to guide inter-bank expansion. The details about the DELTA allocation algorithm are discussed in Section II-B.

DELTA enforcement mechanism: The inter-bank mechanism uses a mapping table as shown in Figure 1, the CBT, to map addresses to cache banks (see Section II-C for details). On a private cache miss the CBT provides the mapping of each address to the LLC bank where it resides. The mappings in the CBT are changed when reconfigurations are triggered by the allocation algorithm. The flexible mapping enabled by the CBT is in contrast to S-NUCA which maps addresses to cache banks statically. The intra-bank mechanism relies on hardware support for way partitioning (discussed in Section II-C) similar to that available in some commodity systems [19].

B. DELTA Allocation

1) Allocation Algorithm: Inter-bank allocation: The pseudo-code for the inter-bank allocation algorithm is shown in Algorithm 1. Each tile (Challenger) starts by computing the pain and gain at the beginning of every inter-bank reconfiguration interval (i_{inter}), which is set to 1ms. An analysis motivating this interval is presented in Section IV-D. A challenge message is only sent if the calculated gain is above a threshold and the size of the allocation is larger than the minimum limit (line 4). The requirement to be above the minimum allocation limit is to avoid placing data far away instead of expanding in the home bank. The choice of which tile to challenge (Challenged) is based on the distance to the tile. Each tile will start by challenging the closest neighbouring tiles, with a hop distance of one, before choosing tiles further away (line 5). A single challenge is issued by every tile in each (i_{inter}) interval if it satisfies the preconditions. The algorithm will only pick a particular tile for a second challenge after it has exhausted other candidates, regardless of whether the previous attempt was successful.

When a challenge is received, the gain from the challenger tile is compared to the pain of the challenged tile. The algorithm uses pain, instead of gain, for comparison in order to accommodate the potentially high impact on performance for the application running on the challenged tile. Furthermore, it also acts as a deterrent to prevent one tile from easily invading and taking over the capacity of neighbouring tiles. In case an application running on tile A is sharing its cache bank with another running on tile B and receives a challenge from a different application running on tile C, the algorithm will compare the Pain_A, Gain_B and Gain_C to determine if the challenge is successful (line 10). In case the challenge is successful, a fixed capacity (number of ways) is allocated in the challenged tile and a response is sent to the challenger tile as a notification (line 12-13). On receiving a successful response the CBT is updated and invalidations are triggered if required (line 18-20). The intra-bank algorithm, described later in this section, discusses how the allocation for the challenger tile can grow to encompass the entire cache bank gradually over time. If the challenged tile does not use its home cache bank (i.e. the core is idle), the algorithm will allocate the whole cache bank to the challenger tile immediately instead of gradually. This is done to make it easier for applications that are running alone to increase their allocation quickly as the cache banks will otherwise remain underutilized.

Algorithm 1: Inter-bank allocation pseudo-code

Input: mlp, allocationForChallanger, interDeltaWays

```
1  In tile Challanger at time period i_inter ;
2  pain = calculatePain(mlp, allocationForChallanger);
3  rawGain = calculateRawGain(mlp,
     allocationForChallanger);
4  if rawGain >gainThreshold AND allocationForChallanger
     >minWays then
5  |  challenged = getClosestNeighbour();
6  |  gain = rawGain / distanceTo(challenged);
7  |  challenge(challenged, challenger, gain);
8  end
9  In tile Challenged on receiving a challenge;
10 partition = partitionWithSmallestGainOrPainInChallenged(
     challengedPain, challengerGain,
     gainChallengedPartitions);
11 if partition then
12 |  updateWayPartition(challenger, partition,
   |    interDeltaWays);
13 |  respondWithNewPartition(challenged, challenger,
   |    true);
14 else
15 |  respondWithNewPartition(challenged, challenger,
   |    false);
16 end
17 On response;
18 if success then
19 |  updateCacheBankTableWithNewPartitionIn(challenged);
20 |  invalidateAddressesWithChangedBankPlacement();
21 end
22 markAsChallenged(challenged);
```

Intra-bank allocation: The pseudo-code for the intra-bank algorithm is shown in Algorithm 2. This is triggered in each tile at every intra-bank reconfiguration interval (i_{intra}) which is set to 0.1ms. The algorithm works by comparing the gain for each partition that shares the cache bank (line 2-3) and reassigns some ways ($intraDeltaWays$) from the partition that has the least gain to the one that has the most (line 5). Here, unlike the inter-bank allocation, the comparison only considers the gain of every application to determine the winner, for two reasons. Firstly, the application running on the tile must have already demonstrated a significant gain, more than the home bank's pain, to have been allowed to expand into a different tile. Secondly, intra-bank changes in allocation are lightweight and do not introduce any invalidation-related overheads (except when leaving a tile). In case an application running on tile A is sharing its own cache bank with two others running on tile B and C, a comparison will happen between Gain_A, Gain_B and Gain_C, to determine which contending

Algorithm 2: Intra-bank allocation pseudo-code

```
   Input: intraDeltaWays
1  In each tile at time period i_intra;
2  partitionSmallest =
     partitionWithSmallestGain(gainsOfPartitionsInBank,
     minWays);
3  partitionLargest =
     partitionWithLargestGain(gainsOfPartitionsInBank);
4  if partitionWithLargestGain !=
     partitionSmallestGain then
5      updateWayPartitioning(partitionLargest,
         partitionSmallest, intraDeltaWays);
6      reportNewAllocation(partitionSmallest,
         partitionLargest);
7  end
```

application will win/lose cache ways.

After reassignment the information about the number of ways are sent back to the respective contending home tiles (line 6). This acts as a feedback mechanism between the intra- and inter-bank algorithm since the current allocation is an important factor in determining the pain and gain, as will be described in the next section. The inter- and intra-bank algorithms are invoked periodically after an initial state where cache capacity is equally partitioned among all tiles.

2) Computing Gain and Pain: The measure of pain and gain is based on simple heuristics that lead to effective allocation decisions. We use gain to predict how an application will react to an increase in cache capacity ($gainWays$) by expanding in/to other tiles. In order to compute gain we take into consideration information about the number of misses provided by the shadow tags, memory-level parallelism (MLP), current capacity allocation outside the home tile and the hop distance. The potential gain for an application running on tile i and expanding into tile j is calculated using the following formula:

$$Gain_{\text{i,j,gainWays}} = \frac{a_{gainWays} * (k+1)^{-1}}{m * (l+1)} \quad (1)$$

where a is the number of misses that potentially can be avoided with $gainWays$ additional ways, k is the number of ways outside of the home tile, m is the MLP of the application running on tile i and l is the hop distance from tile i to j.

The rationale behind factoring in the aforementioned attributes in the gain expression is as follows. Firstly, factoring in the number of avoidable misses provides an estimate of reduction in the number of long-latency memory accesses which influences performance. This value can be read directly from the shadow tags in the monitoring hardware for a given core. MLP is factored in because this coupled with the number of misses helps to get a better estimate of the performance impact of cache allocation decisions. The MLP estimate is obtained through performance counters. Lastly, we factor in the current allocation in remote tiles and the hop distance to introduce fairness and ensure that no single application expands its allocation too aggressively.

We use pain as a heuristic measure to predict how an application will react to losing available cache capacity ($painWays$) on the home tile where it is running. The pain value is never communicated to other cache banks. In order to compute pain we only take information about misses provided by the shadow tags and MLP into account. Unlike the formula for gain, we do not take information about allocations outside home bank and the distance into account because the goal here is to protect the capacity allocation in the home tile where the application is running. Since the pain is not scaled it will grow faster, if there are more misses, which will enable the home bank application to protect its allocation. The pain of losing $painWays$ for the application running on tile j is calculated using the following formula:

$$Pain_{\text{j,painWays}} = \frac{a_{painWays}}{m} \quad (2)$$

where a is the number of misses that will be incurred if the allocation is decreased with $painWays$ and m is the MLP. Our evaluation in Section IV shows that the pain/gain heuristics leads to good cache allocation decisions. We leave further optimization of the pain and gain measures used in this study for future work.

3) Monitoring hardware: We adopt Qureshi's UMON sampled tag array [1] in this work. The original UMON mechanism can predict the number of cache misses under all possible cache allocations (at a single way granularity) based on the access pattern the application has exhibited before. The coarse-grained UMONs, that we use, work by tracking the number of accesses to a shadow tag at coarse-granularity (corresponding to 4 ways). The number of tags required will still be the same but the associated way-hit counter overheads are reduced. The solution also uses dynamic set sampling to decrease the overhead of the monitoring hardware, like the original proposal.

4) Hardware-based implementation: The inter-bank and intra-bank algorithms are implemented in hardware owing to its low computational complexity (see Section IV-E for details). To implement the algorithms each LLC bank controller is provisioned with an ALU capable of computing the pain and the gain and for comparing the values. The inter-bank scheme requires, for each bank, a register array with $N+2$ entries, with $log_2(N)$ bits per entry, to store the pain values of other banks. In addition, each bank also includes a register array with $N+1$ entries, with $log_2(N)$ bits per entry, to store the id of other tiles in increasing order of distance to determine the next tile to send the challenges to. The intra-bank algorithm leverages the state used by the inter-bank algorithm for establishing allocations.

C. DELTA Enforcement

DELTA's enforcement mechanism has two components. The inter-bank enforcement mechanism utilizes a CBT (the detailed design is presented later in this section) which contains the mapping between address ranges and cache banks. The CBT permits the allocations to span multiple banks by mapping portions of the address space to different banks. The CBT is accessed in parallel with the L2 cache to determine in which LLC bank a certain address is mapped to. The

intra-bank enforcement mechanism comprises a standard way-partitioning (WP) unit that keeps track of which ways in the cache bank each core is allowed to insert lines into.

1) Bank partitioning: Each core has a CBT which is organized as a small fully-associative range table that holds the information of address range to bank translation. The CBT works with physical block addresses. We use a simplified version of the range based organization proposed by Gandhi et al. [20]. The total amount of storage required for each CBT is $log_2(N) \times N$ bits, where N is the maximum number of distinct ranges, which is equal to the number of cores/banks. The number of used entries (i.e. ranges) in the CBT is equal to the number of LLC banks allocated by the local core. Only in the rare scenario in which a a single core is active, can a core's CBT grow to map the majority of the chip resources. Therefore, in practice this value is much smaller than the total number of banks and the cost of the associative look-up is negligible.

The CBT is updated when the allocated capacity for a tile expands to/retreats from another tile. When the CBT is updated there is a remapping of address ranges to cache banks. The size of the address range mapped to a bank is proportional to the size of the allocation in that cache bank. Examples illustrating when and how the CBT is updated as allocations expand/retreat are presented in Section II-D.

There are two important design choices for the CBT: (i) how many bits to use, and (ii) which bits to use, i.e. how to map addresses to cache banks. We evaluated different options and found that by using just 8 bits following the set index, as shown in Figure 2, it is possible to effectively distribute the footprint of the application across the different banks. We reverse the bits before we index into the CBT to obtain the bank mapping. The reversing operation turns the least significant bits with the highest entropy to the most significant, which proves to be a reasonable solution for mapping addresses uniformly for the applications we consider.

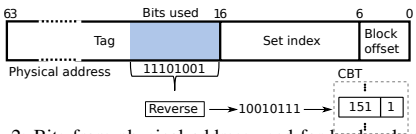


Fig. 2: Bits from physical address used for bank selection.

2) Way partitioning: During insertion, the WP unit uses a bitmask to indicate which cores can insert into a given way in a cache bank. All cores can however access data irrespective of which way it resides. Way-partitioning enforced using bitmasks is practical and has been implemented in commodity systems [19]. The total amount of storage required for each WP unit is $N \times W$ bits, where N is the number of cores and W is the number of LLC ways. DELTA can also work with other fine-grained intra-bank partitioning schemes proposed in literature [14], [15], [21].

3) Invalidation support: A common strategy to handle change in the mapping of an address to a LLC location is to invalidate the line in the cache bank where it currently resides. This invalidation is done by flushing the cache line. Several commodity systems provide ISA level support for this [22]. This is widely used by page coloring mechanisms [23]. However when remapping a large range there is increased overhead due to additional instructions needed for invalidating each address. We therefore rely on hardware support for performing bulk invalidation efficiently. The bulk invalidation unit works by checking the tags to identify addresses that fall in the specified range and invalidates them. This approach does not incur the instruction overhead of cache flushes.

D. Putting it all together

We clarify how the partitioning solution works with the help of two examples that illustrate the different use cases.

Example 1, Inter-bank expansion. The capacity allocation expands to a different tile when a challenge is successful. In Figure 3 we show the process of expansion into a new tile, as well as the state of the CBT and WP before/after the change. We assume that tile 4 has capacity allocated in cache banks in tile 4 and 0, as indicated in the CBT for tile 4. Since, the core in tile 4 sees a considerable gain from expanding its allocation it issues a challenge to tile 5 (#1). Tile 5 compares the gain coming from tile 4 with its own pain of losing cache capacity (#2). Since the gain for tile 4 is considerably larger, tile 5 decides to assign $interDeltaWays$ of ways from its allocation to tile 4 and updates its WP unit (#3). In this case, ways 12-15 are assigned to tile 4 and a response message is sent to tile 4. On receiving the message the tile updates its CBT to also include tile 5 (#4). This is followed by remapping addresses in range 192-255 from tile 4 to tile 5, and invalidating them where they were previously located (#5). Note that expansion process does not require invalidations in tile 5.

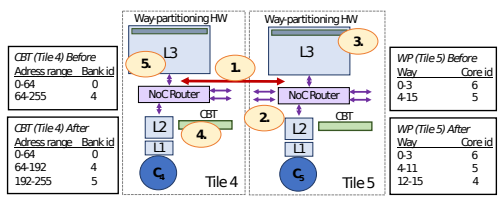


Fig. 3: Example of expansion.

Example 2, Intra-bank algorithm and retreat. The intra-bank algorithm determines whether allocations within a bank expand or shrink. The decision on which partition expands or shrinks is based on the gain of the different applications that share the cache bank. Whenever a partition expands or shrinks the WP unit is updated to reflect the new allocation for the partitions. A shrink will result in a retreat if a partition loses all the ways it was assigned in the cache bank. This scenario is shown in Figure 4. After the intra-bank algorithm is triggered in tile 5 the algorithm decides that tile 4 must give up the entire capacity in the cache bank since it has the least gain among

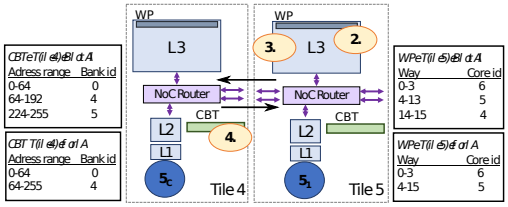


Fig. 4: Example of retreat.

those sharing the cache bank (#1). As a consequence, the WP unit is changed to show the new configuration where ways 14-15 are reassigned to tile 5. The change triggers a message back to tile 4 informing about the retreat. The CBT in tile 4 will now be updated (#2). The addresses previously mapped to tile 5 are remapped, in this example addresses in range 224 to 255 are now remapped back to tile 4. All addresses in the range will consequently be invalidated in tile 5 (#3).

The detailed evaluation of DELTA is presented in Section IV. We show in Section IV-E that the overheads introduced by DELTA are marginal.

E. Support for multithreading

When running multi-threaded workloads containing shared data, accesses to the same line from two different tiles will end up inserting the blocks in two different LLC banks, breaking coherence. To address this, we propose to distinguish between private and shared data at a page granularity and handle them differently. Detection of shared pages including cross-process sharing is performed by the TLB. We adopt a classification scheme, proposed in R-NUCA [24] and used in other NUCA schemes [4], to dynamically classify pages as private or as shared incrementally and lazily. Lines belonging to shared pages are mapped to the cache banks using a fixed S-NUCA strategy whereas lines belonging to private pages are mapped to banks based on the mappings available in the CBT. When a page is first classified as shared all the lines belonging to the page are invalidated.

The allocation algorithm also needs a minor change. On receiving a challenge, the processIDs of the different threads are compared, and the challenge will only be successful if they are different. The rationale is to not let threads from the same application (homogeneous multi-threaded) compete for capacity since it can adversely impact application progress. We expect the performance of this extension with multi-threaded application to be similar to R-NUCA since private data and most of shared data are dealt with in a similar way. Multi-threaded workloads are analyzed in Section IV-C.

III. Experimental Methodology

A. Simulated Architecture

We evaluate our proposal on a 16/64 core tiled CMP architecture modeled using the Sniper Simulator [25]. Details about the baseline architecture are shown in Table II. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and a LLC bank of 512KB. The cache latencies assumed have been modelled using CACTI 6.5 [26].

Cores	16 / 64 cores, x86-64 ISA, 4GHz, OOO, Nehalem-like, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, inclusive, 9-cycle data and 2-cycle tag latency, LRU
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 / 8x8 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 / 8 MCUs, 1 channel/MCU, latency 80 ns, 12.6GB/s per channel
DELTA parameters	reconfiguration interval i_{inter}=1ms i_{intra}=0.1ms, $gainThreshold$=0.5, $minWays$=4, $interDeltaWays$=4, $intraDeltaWays$=1, $gainWays$=4, $painWays$=4

TABLE II: Configuration of the simulated 16- and 64-core tiled CMP.

In Section IV-E we demonstrate that state-of-the-art allocation algorithms, Lookahead or Peekahead, cannot compute locality-aware allocations in a scalable manner. This is because the time needed to compute allocations and locality-aware placement far exceeds the 1 ms reconfiguration interval that we target in this study especially as we scale to larger core counts. In order to fairly compare our distributed solution against the centralized solutions, we model an *ideal centralized* solution that calculates both allocations and locality-aware placement in zero time (no overhead). The ideal solution represents an upper bound on dynamic allocation decisions using the best known centralized algorithm, Lookahead. We use Lookahead as a reference since Peekahead too computes the same allocations as Lookahead albeit with lower overhead. The ideal centralized scheme uses the DELTA enforcement mechanism to support locality-aware mapping in banked LLCs. UMONs are used in each core to measure misses for all possible cache capacity allocations for each application. The cost of invalidations that occur due to remapping of addresses to banks (invalidation+re-fetch) are modelled in detail for both DELTA and the ideal centralized scheme. In addition, we also evaluate an unpartitioned, static NUCA implementation with line-interleaved LLC addresses (unpartitioned S-NUCA), and private LLC, with equal static partitioning of capacity per core (private) for comparison.

DELTA dynamically considers allocations in increments of 32KB from a cache size of 128KB up to 6MB (per application) for the 16-core configuration and 128KB to 24MB for the 64-core case. Each core reserves a minimum of 128KB (see Table II) in the LLC to avoid potential back-invalidations due to the inclusive cache hierarchy.

B. Workloads

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [27]. Workload mixes are constructed by classifying applications in one of the four categories - *cache-insensitive*,

cache-sensitive low, *cache-sensitive low medium* and *thrashing*, depending on sensitivity to different cache sizes. The classification is performed by running each application for 1B instructions (after fast-forwarding for 1B instructions) with cache sizes of 128KB, 512KB and 8MB. The applications that show improvement in IPC of over 10%, as the cache size increases, are classified as sensitive for a particular cache-size region. Sensitive applications that show improvement in the 128KB to 512KB region are classified as cache-sensitive low and those that also show improvement in the 512KB to 8MB are classified as cache-sensitive low medium. The detailed classification is presented in Table III. Lastly, cache insensitive and thrashing applications experience less than 10% improvement in the 128KB to 8MB range. Among these, we classify applications with a number of Misses-Per-Kilo-Instruction (MPKI) above five as thrashing and the rest as cache insensitive.

Insensitive (I)	povray(po),sjeng(sj),namd(na), zeusmp(ze),GemsFDTD(Ge)
Thrashing (T)	bwaves(bw),libquantum(li),milc(mi)
Cache-sensitive low (L)	h264ref(h2),gromacs(gr),astar(as), gamess(ga),lbm(lb),tonto(to), wrf(wr),leslie3d(le),hmmer(hm)
Cache-sensitive low medium (LM)	dealII(de),omnetpp(om),xalancbmk(xa), gobmk(go),bzip2(bz),gcc(gc),mcf(mc), soplex(so),perlbench(pe),sphinx3(sp), calculix(ca),cactusADM(cac)

TABLE III: Classification of SPEC CPU2006 benchmarks.

Name	Composition	Benchmarks
w1	LM	de,om(2),pe,ca,bz,go(2),ca,hm,le,go,bz,gc,so,mc
w2	L+LM	bw,sj,na,ze,li,mi,ca,sp,de,om,go,go,bz,gc,mc,pe
w3	T+L	to(2),bw(3),lb(2),li(3),h2,mi,gr,as,ga,mi
w4	T+LM	delII,bw(3),so,li(2),hm,pe,mi(3),go,om,bz,go
w5	I+L+LM	gc,po,Ge,as,pe,wr,ga,cac,to,hm,sj,h2,bz,ze,gr,so
w6	I+T+L+LM	na,de,li,gr,wr,so,mi,as,mi,to,ze,om,bw,h2,Ge,hm
w7	I+T+LM	sj,bw(2),bz,wr,li(2),gc,mi,de,na,om,ze,mi,go,Ge
w8	I+T+L	po,bw(2),h2,sj,li(2),gr,na,mi,as,Ge,ga,wr,lb,mi
w9	I+LM	po,om,sj(2),go,na(2),le,ze,go,Ge,bz,wr,ca,sp,gc
w10	I+L	po,to,sj,h2(2),na,lb(2),ze(2),gr,Ge,as,wr,ga,po
w11	T+L+LM	sp,bw,h2,om,li,gr,go,mi(2),as,hm,bw,ga,le,lb,calulix
w12	random	go,lb,ca,sp,bw,go,li(2),ga,h2,ze,to,so,gr,mi,pe
w13	random	lb,to,pe,go,gc,mi,li(2),na,h2,cac,ze(2),ca,so,as
w14	random	de,bw,mc,li,pe,mi,ca,wr,go,po,hm,na,go,ze,so,Ge
w15	random	to(2),po,lb,li,mi,lb,wr,h2,sj,gr,na,as,ze,ga,Ge

TABLE IV: Workload mixes.

We construct a total of 15 workload mixes by combining the applications from the categories described above. Applications from each category are picked randomly while not allowing duplicates unless all applications in a category have already been picked. Details about workload mixes are presented in Table IV. We construct workload mixes for 64 cores by replicating the 16-core workload four times. The applications in a workload mix are mapped to cores randomly.

C. Methodology

We fast-forward for 8B/2B instructions for the 16/64 core simulations. Detailed simulations are carried out until all benchmarks have completed at least 500M/125M instructions and statistics are reported based on the first 500M/125M instructions for each application. We simulate fewer instruction in fast-forward and detailed mode for 64-core CMP to reduce simulation time. The methodology is in line with earlier works [4], [14], [15].

D. Metrics

We use IPC as a measure of performance. We report the geometric mean of IPCs of the applications in a workload, as a performance metric for the workload. We also report the following fairness and throughput metrics: average normalized turnaround time (ANTT) and system throughput (STP) [28]. ANTT is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i}{CPI_{i,private}}$ and STP is given by $\sum_{i=1}^{N}\frac{CPI_{i,private}}{CPI_i}$ ANTT and STP are commonly used for performance evaluation of multi-programmed workloads.

IV. Evaluation

We first compare DELTA against alternative cache organizations and allocation algorithms (described in Section III-A). Next, we show results for multithreaded applications followed by the impact of reconfiguration frequency. Finally, we provide an analysis of DELTA's overheads.

A. Multi-programmed mixes on 16-core CMP

Figure 5 shows the performance for multi-programmed mixes normalized to the unpartitioned S-NUCA. On average, DELTA improves performance by 9% (up to 16%) over S-NUCA whereas the ideal centralized solution shows an average improvement of 12% (up to 22%). In comparison, the private scheme shows an average improvement of 3% over S-NUCA. The results for comparing the fairness and throughput of DELTA and the ideal centralized scheme are shown in Figure 6. On average, DELTA is 2% behind in terms of ANTT and 5% behind in terms of STP, than the ideal centralized scheme. Note that a lower value signifies greater fairness with ANTT while a higher value is equivalent to larger throughput with STP.

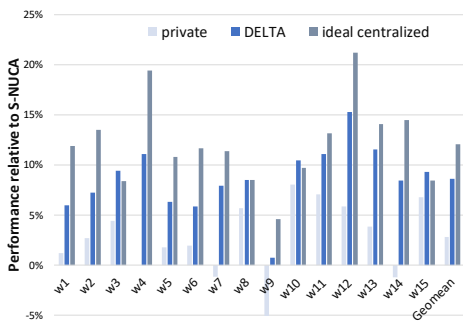


Fig. 5: Performance of workload mixes normalized to unpartitioned S-NUCA on a 16-core CMP.

As can be seen in Figure 5, the ideal centralized scheme is better than DELTA in 11 out of 15 mixes. In four cases DELTA performs on par or better. In order to understand the performance gap between the ideal centralized scheme and DELTA we investigate a single workload in detail. Figure 7 shows the performance of different applications in a single

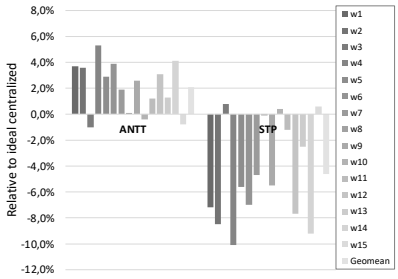


Fig. 6: Fairness comparison between ideal centralized and DELTA.

workload using the ideal centralized scheme, normalized to DELTA (in gray). In addition, it also shows the performance of the private scheme normalized to DELTA (in blue). As can be seen in the figure, most of the applications perform almost identically with the exception of xalancbmk and soplex. For these two applications, the ideal centralized solution performs considerably better than DELTA, by 45% and 35%, respectively. Note that DELTA performs better than the private scheme for these two applications by 12% and 36%.

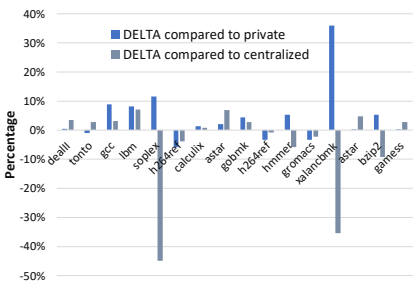


Fig. 7: Normalized performance for applications in *w2* on a 16-core CMP.

To understand the trend for xalancbmk and soplex, we compare the allocations of cache capacity, in terms of number of ways, made by the ideal centralized scheme and DELTA, for the different applications in the workload. The ideal centralized algorithm gives a larger allocation of 42 respectively 50 ways on average to xalancbmk and soplex in comparison to DELTA which gives 26 and 20 ways. This behaviour can be attributed to the *farsighted* nature of the ideal centralized scheme i.e. it uses information about the entire miss curves for all applications to determine allocations. DELTA, in contrast, is *nearsighted*, i.e. uses a limited window of the miss-rate curves to determine the pain/gain which influences allocations. As xalancbmk and soplex do not see a considerable improvement in the limited window, DELTA does not allocate as much cache capacity as the ideal centralized scheme. The difference in the size of allocations impacts the performance because these applications are sensitive to additional cache capacity.

In Figure 8 we show the performance of individual applications in one of the workloads where DELTA is on par with the ideal centralized scheme. We see that individual applications mostly perform as well as or better than the centralized scheme even though DELTA is nearsighted. The same trend holds also for the other workloads (w3,w8,w10,w15).

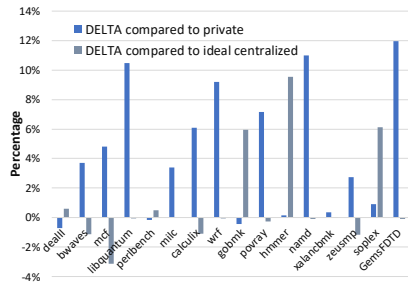


Fig. 8: Normalized performance for applications in *w3* on a 16-core CMP.

B. Multi-programmed mixes on 64-core CMP

To investigate how our proposal scales to larger core counts we evaluate a 64-core CMP. Figure 9 shows the performance for the individual multi-programmed workload mixes. DELTA, on average, improves performance by 16% (up to 28%) over S-NUCA, while the ideal centralized scheme improves performance by 17% (up to 35%). The private scheme performs better for 64-cores than for 16-cores, but is generally regarded as an inefficient solution since it cannot handle underutilized scenarios. The results for comparing fairness and throughput between ideal centralized and DELTA indicate that the difference between the two schemes is 1% for STP and less than 1% for ANTT (not shown). The results also indicate that DELTA makes good allocation decisions, on par with an ideal scheme, in spite of the distributed nature of the algorithm that increases the number of re-configurations (steps) required to span across all the banks in a CMP.

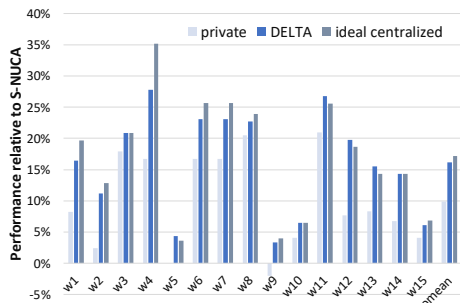


Fig. 9: Performance of workload mixes normalized to unpartitioned S-NUCA on a 64-core CMP.

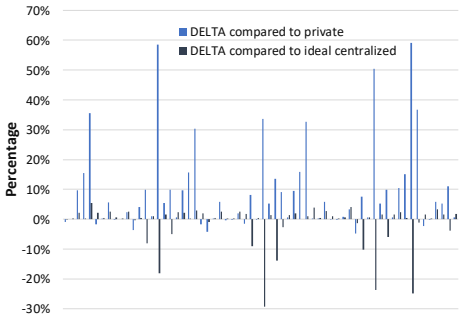


Fig. 10: Normalized performance for applications in *w2* on a 64-core CMP.

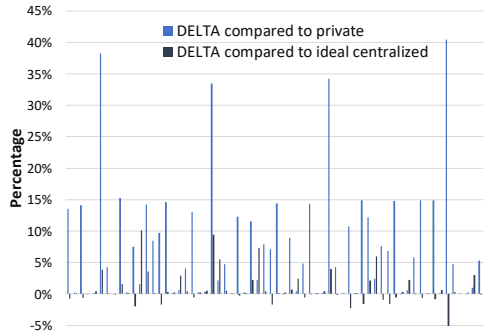


Fig. 11: Normalized performance for applications in *w13* on a 64-core CMP.

The performance gap between DELTA and the ideal centralized scheme has diminished on average, compared to the results for a 16-core CMP. For seven workloads (w3, w5, w10, w11, w12, w13, w14) DELTA is on par or better than ideal centralized. For workload w2, as shown in Figure 10, we observe that DELTA falls behind the ideal centralized scheme in the 64-core CMP experiments, similar to the trend seen with the 16-core CMP. For many applications in this workload DELTA is still better than the ideal centralized scheme but in the few cases where the reverse is true, the difference in performance is comparatively larger. For applications such as xalancbmk and soplex the ideal centralized scheme surpasses DELTA by giving a larger allocation, because it is farsighted.

We investigate one of the workloads (w13) where DELTA performs better than the ideal centralized scheme, as shown in Figure 11. The farsightedness of the ideal centralized scheme results in it allocating over 250 ways to applications such as lbm and libquantum. Moreover, the centralized scheme does not consistently detect the benefit of giving these applications a large allocation, and switches the cache capacity allocation between a large and small allocation. In general, giving larger allocation to a few applications puts severe constrains on the allocations for the other applications and degrades the overall performance. DELTA does not suffer from making these unadvantageous allocations and performs better for the mixes containing applications like lbm and libquantum.

In summary, the evaluation shows that DELTA performs almost as well as the ideal centralized scheme as we scale to 64 cores. Furthermore, this demonstrates that a dynamic distributed scheme can give good allocations, without incurring the overheads associated with computing allocations centrally.

C. Multi-threaded applications

We estimate the performance of DELTA using SPLASH2 suite in order to understand how the scheme performs with multithreaded applications. Figure 12 shows the speed-up obtained by DELTA over the S-NUCA implementation and compares it to the private cache configuration. We execute each application on the 16-core CMP and using large input sets (from Sniper) to obtain performance data for the baselines. We use number of cycles for the longest running thread within the parallel region, which we identify as the region of interest (ROI), as a measure of performance.

We follow a two step process to estimate the performance of DELTA. Firstly, we measure the ratio of private/shared pages and cache blocks. These results are shown in Table V. For this we develop a pintool [29] that instruments all loads and stores in the region of interest to measure inter-thread sharing at page and cache block granularity. Next, we estimate the performance of DELTA by performing a piece-wise reconstruction of the execution in which private accesses are modeled according to private LLC baseline's performance, and shared accesses are modeled according to the S-NUCA baseline performance. To simplify, we assume that the LLC accesses are uniformly distributed across pages. Private pages are reclassified at most once, and the S-NUCA mapping is never reverted. Hence, for long running applications this overhead is negligible. We expect the estimation to be accurate since DELTA maps lines from private-pages to the private bank and utilizes S-NUCA mapping for lines from the shared pages.

By design (see Section II-E), the performance of DELTA is usually between the performance of the S-NUCA and private baselines, depending on the amount of private/shared pages. Over the entire SPLASH2 suite, the average performance of DELTA compared to both the private LLC configuration and S-NUCA configuration is within 1% for both cases. The actual

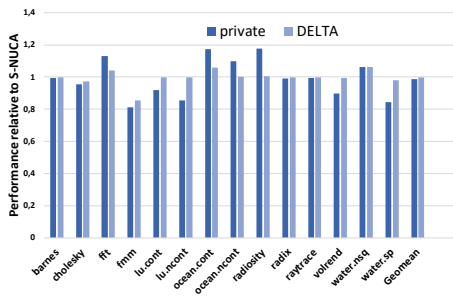


Fig. 12: Normalized performance for splash2 on a 16-core CMP.

App.	barnes	cholesky	fft	fmm	lu.cont	lu.ncont	ocean.cont
Page	8.2	62	33	73	0.5	0.5	38
Block	9.3	66	34	65	0.3	0.3	98.6
App.	water.sp	radiosity	radix	raytrace	volrend	water.nsq	ocean.ncont
Page	10	3	5.2	17	5.7	99.8	1.1
Block	70	4.6	10	24	21	91	99

TABLE V: Percentage of private pages and blocks.

performance for each benchmark depends considerably on the amount of sharing, with variations of up to 20%. For example, in `lu.ncont`, which has high ratio of sharing (>99%), DELTA's performance is almost equal to the S-NUCA performance, while the private LLC configuration suffers performance loss of approx. 10%. On the other hand, in `water.nsq`, where almost all pages are private, DELTA's performance is equivalent to that of the private LLC configuration, achieving a speed-up of 6% over S-NUCA.

The results in Table V indicate that several benchmarks have a low amount of private pages as opposed to private blocks. On reason for this is the existence of shared data, such as boundary elements in structured grid simulations, in pages that are mostly private. In general, this will lead to less locality-optimized cache access for these benchmarks. Due to the additional costs associated with distant memory accesses (both in DRAM and in caches), modern HPC software development encourages programming styles that result in higher number of private pages. This is designed to ensure threads operate on local memory thereby reducing the amount of shared pages [30]. Moreover, an important trend in algorithm design are Communication-avoiding Algorithms (CAA) [31] which, attempt to reduce sharing. Hence, we expect the architecture of DELTA to achieve even better performance on modern multi-threaded workloads with a considerable amount of private data in comparison to S-NUCA. Detailed modeling and optimization of DELTA for emerging multithreaded workloads is left for future research.

D. Frequency of reconfiguration

In order to understand the impact of the frequency by which cache allocations are computed, we simulate a cache partitioning solution that uses an ideal implementation of Lookahead for computing allocations with zero overheads (see Section III-A for details about the ideal centralized implementation) at two cache allocation frequencies (1 ms and 100 ms), on the baseline system. Figure 13 shows the impact of allocation frequency on performance of five different workload mixes each comprising 16 SPEC CPU2006 benchmarks (see Table IV). The results demonstrate that while frequent allocations do not benefit all workloads, they do provide the opportunity to improve performance for several of the workloads considered, because of better adaptation to phase changes.

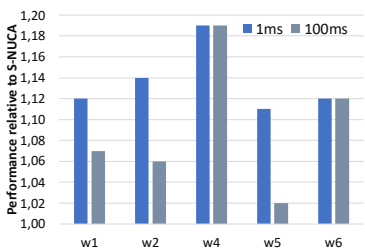


Fig. 13: Impact of frequency of reconfigurations on a 16-core CMP.

E. Overheads

We analyze the different sources of overheads in the centralized scheme and DELTA.

1) Computational overheads: The worst-case time complexity of the Lookahead algorithm is $O(N \times W^2)$ where N is number of cores and W is number of ways. We can consider the algorithm to have cubic complexity, since the number of ways needs to be at least as many as the number of cores (for way-partitioning). The best case complexity is $O(N \times W)$, i.e. quadratic. Peekahead, which considers only the points of the miss rate on a convex hull, has a complexity of $O(N \times W)$, in the best/average case and $O(N \times W^2)$ in the worst-case. The time to compute cache allocations for different core counts using Lookahead and Peekahead, with 16 ways per core, is presented in Table VI.

The Lookahead algorithm takes 5.32 ms on average to compute allocations for a CMP with 16 cores (16-tile CMP with each bank containing 16 ways). Peekahead takes 0.89 ms on average for the same scenario. For larger core counts the overhead is even larger. Note that the data presented in the table do not take into account the additional computations needed to perform locality-aware data placement, which for large core counts has been shown to exceed capacity allocation overheads [32].

For DELTA the complexity can be attributed to the inter- and intra-bank allocation algorithm. The pain and gain computation step takes constant time, i.e. $O(1)$ complexity. The inter/intra bank allocation algorithm requires finding the core with the MIN and MAX gain/pain values. This operation is similar to finding the min. and max. in an unsorted array. The simplicity of the DELTA reconfiguration algorithms enables a hardware implementation with low overheads. Even if the algorithms were to be implemented in software, the overhead of DELTA's inter- and intra-bank allocation algorithms assuming a 64-core CMP would be 0.015 ms and 0.007 ms, three orders of magnitude lower than state-of-the-art.

2) Message overheads: We calculate the number of additional messages sent in the worst-case at each reconfiguration interval (assuming $i_{inter} = 1ms$ and $i_{intra} = 0.1ms$) in a 16-core CMP. For the centralized scheme the total number of messages is $2 \times N$, where N is the number of cores in the system, and this results in $16 \times 2 = 32$ additional

Cores	2	4	8	16	32	64
Lookahead	0.02	0.05	0.46	5.32	73.07	1230
Peekahead	0.03	0.07	0.23	0.89	3.34	13.12

TABLE VI: Overhead for Lookahead and Peekahead in ms per invocation.

messages. DELTA however needs $2 \times N$ messages for intra-bank allocation and $N \times 10 \times 2$ for inter-bank allocation which results in a total of 352 messages. However, the number of messages on the NoC pertaining to L2 misses alone, during the same interval, is 320K on average. This indicates that the overhead in terms of additional messages for DELTA is marginal ($\sim$0.1%), even in the worst-case.

3) Invalidation overheads: Invalidations are needed when remapping addresses to LLC locations regardless of whether the allocation algorithm is centralized or distributed. Invalidation overheads can be primarily attributed to two causes: the overheads of performing the invalidations and how often/much data need to be invalidated. We address the first by performing bulk invalidations (see Section II-C). We mitigate the second by only requiring invalidations for inter-bank reconfigurations. Intra-bank reconfigurations do not lead to invalidations except when a retreat is triggered in rare cases (see Section II-D).

V. Related work

Cache Partitioning: Several solutions have been proposed in literature for cache resource partitioning. Way partitioning is the most popular and it is implemented in commodity systems [2], [19], [33], [34]. It is a simple technique that works by limiting which ways a core can insert into. The major limitation is that it requires cache associativity to scale with the number of cores, which is not easily done [35]. Set partitioning is another approach, which can be implemented with hardware or software support [8], [9], [23], [36]. Hardware based schemes require flexible indexing of the cache. Software schemes use page coloring and rely on OS support for partitioning. Page coloring, however, cannot support superpages and incurs high overhead for reconfiguration. The aforementioned solutions also have the drawback of only supporting a limited number of coarse-grained partitions.

Fine-grained partitioning solutions can be broadly classified in three categories, i) hybrid techniques [10], ii) clustering techniques [11], [12] and iii) replacement-based techniques [13]–[16]. Hybrid techniques like SWAP, combine set and way partitioning in order to get more fine-grained partitions. Clustering techniques like KPart, group applications into clusters and then assigns clusters to way partitions, to emulate fine-grained partitioning. The replacement-based techniques adapt the cache replacement policy to enable fine-grained partitions with different sizes. However, in the context of tile-based CMPs, these approaches leave room for further improvement since they do not take locality into account.

A few proposals performs locality-aware placement for tiled CMPs [4], [18]. CloudCache [18] uses virtual private cache partitions that span across banks and performs locality-aware placement of the partitions. The drawback of this proposal is that it uses N-chance spilling [37] on evictions and requires costly broadcasts. Jigsaw [4] lets software define *shares* and then maps data to them by assigning a share *id* to every page in the application. Allocation and enforcement is done at share granularity instead of application/core granularity as in prior proposals. The proposal relies heavily on software support. Furthermore, in the aforementioned solutions the allocation decision is made by a central hardware or software component which limits scalability.

To lower the overhead of a central component, XChange [38] uses a market-based approach where some of the computations for multi-resource management are done in each core/bank. However when used for partitioning a single resource, XChange will result in an equal partitioning. Moreover the scheme is based on a shared L2 cache structure and thus does not enable locality-aware placement, unlike DELTA.

Non-Uniform Cache Access (NUCA): Efficient usage of NUCA caches has been an extensively researched topic [24], [39]–[45]. The simplest approach, Static NUCA (S-NUCA), spreads the data over all cache banks with a fixed mapping and exposes variable access latencies.

D-NUCA schemes try to combine the best from private and shared cache designs, where private designs have isolation but suffers from under-utilization and shared designs have dynamic utilization but suffers from long on-chip latencies and interference.

A few proposals, like DELTA, try to minimize long on-chip distance. They mostly focus on multi-threaded applications and achieve this by replication and placement, where frequently used lines are copied and placed in the nearest cache bank [42]–[44]. Spilling has been proposed as a way to overcome the problem of underutilized private caches. It works by inserting a copy of a line in another cache bank before it is evicting from the cache hierarchy [37], [46].

Both spilling and replication have two problems. Firstly, both operations decrease the capacity of the cache, where a tradeoff is made between latency and capacity. Secondly, costly directory lookups are needed in order to find data. In our proposal, we avoid both these drawbacks since we place data in a locality-aware way without replicating and do not need a directory to find the data.

Some D-NUCA proposals implement a different partitioning strategy, where separation is done between shared and private regions instead of applications [47]–[50]. Elastic Cooperative Caching (ECC) [50] uses a distributed approach to divide the cache bank between shared/private using way partitioning. The scheme also uses spilling to extend capacity to other banks and therefore inherits the drawbacks of spilling. In contrast to this work we enforce strict per application partitions that can span multiple banks.

R-NUCA [24] classifies accesses into three categories (instructions, private data, shared data) and uses static rules for placement and replication for each type. The drawbacks of the scheme is that it uses a static placement scheme for the different classes not taking dynamic nor application specific behaviour into account, which DELTA does. Note that, compared to DELTA, none of the aforementioned techniques give strict interference protection for data.

Coherence framework: CDR [51] reduces the scope of cache coherence from global to VM-, application-, or page-level to enable shared memory between servers or to minimize on-chip distances in a manycore. Unlike DELTA it does not

provide strict cache partitioning. Furthermore, creation of sharer domains does not take the cache requirements of each application into account but instead allows the different threads of the same application to form a domain.

To the best of our knowledge, DELTA is the first distributed solution for fine-grained and locality-aware cache partitioning which permits permits hardware implementation and scales to many-core architectures.

VI. Conclusions

We present DELTA, a fully distributed and locality-aware partitioning solution for tile-based CMPs. The solution is scalable through its novel challenge-based allocation algorithm, which allocates cache capacity in a distributed way based on the performance gain of each application.

We show that the distributed algorithm has low computational overhead which permits hardware implementation and enables frequent reconfiguration. The allocation algorithm is supported by a flexible enforcement mechanism that enables locality-aware placement. Our evaluation demonstrates that the distributed partitioning solution performs close to an ideal centralized solution and scales to large core counts.

Acknowledgment

This research has been funded by the European Research Council, under the MECCA project, contract number ERC-2013-AdG 340328 and the Swedish Research Council (VR), under the ACE project. The simulations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

References

[1] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.
[2] L. R. Hsu *et al.*, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *PACT-15*, 2006.
[3] D. Thiebaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. on Comp.*, 1992.
[4] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.
[5] R. Riesen *et al.*, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, 2009.
[6] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proc. SC'10*, 2010.
[7] T. Jones, "Linux kernel co-scheduling for bulk synchronous parallel applications," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 57–64.
[8] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. HPCA-14*, 2008.
[9] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. MICRO-39*, 2006.
[10] X. Wang *et al.*, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *Proc. HPCA-23*, 2017.
[11] H. Cook *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ISCA-40*, 2013.
[12] N. El-Sayed *et al.*, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. HPCA-24*, 2018.
[13] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, 2013.
[14] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management," in *Proc. ISCA-39*, 2012.
[15] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.
[16] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.
[17] R. Iyer, "Cqos: A framework for enabling qos in shared caches of cmp platforms," in *Proc. ICS-18*, 2004.
[18] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.
[19] A. Herdrich *et al.*, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proc. HPCA-22*, 2016.
[20] J. Gandhi *et al.*, "Range translations for fast virtual memory," *IEEE Micro*, 2016.
[21] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. MICRO-47*, 2014.
[22] "Intel® 64 and IA-32 Architectures Developer's Manual."
[23] M. Awasthi *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. HPCA-15*, 2009.
[24] N. Hardavellas *et al.*, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proc. ISCA-36 2009*, 2009.
[25] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.
[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO-40*, 2007.
[27] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002.
[28] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[29] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[30] D. Ott, "Optimizing Applications for NUMA," 2011.
[31] J. Demmel, "Communication avoiding algorithms," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 1942–2000.
[32] N. Beckmann, "Design and analysis of spatially-partitioned shared caches," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.
[33] S.-H. Yang *et al.*, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *Proc. HPCA-8*, 2002, pp. 151–161.
[34] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. HPCA-8*, 2002, pp. 117–128.
[35] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proc. MICRO-43*, 2010, pp. 187–198.
[36] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proc. MICRO-34*, 2001.
[37] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proc. HPCA-15*, 2009.
[38] X. Wang and J. F. Martinez, "XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proc. HPCA-21*, 2015.
[39] S. Srikantaiah *et al.*, "MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy," in *Proc. HPCA-17*, 2011.
[40] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The Direct-to-Data (D2D) Cache: Navigating the cache hierarchy with a single lookup," in *Proc. ISCA-42*, 2014.
[41] ——, "A Split Cache Hierarchy for Enabling Data-Oriented Optimizations," in *Proc. HPCA-23*, 2017.
[42] M. Zhang *et al.*, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. ISCA-32*, 2005.
[43] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.
[44] P. A. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A New Approach to Replication in Distributed Shared Caches," in *Proc. PACT-26*, 2017.
[45] Q. Shi *et al.*, "LDAC: Locality-Aware Data Access Control for Large-Scale Multicore Cache Hierarchies," *ACM TACO*, 2016.
[46] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. ISCA-33*, 2006.
[47] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proc. HPCA-13*, 2007.
[48] L. Zhao *et al.*, "Towards hybrid last level caches for chip-multiprocessors," *ACM SIGARCH Comput. Archit. News*, 2008.
[49] J. Merino, V. Puente, and J. A. Gregorio, "ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture," in *Proc. HPCA-6*, 2010.
[50] E. Herrero *et al.*, "Elastic cooperative caching," in *Proc. ISCA-37*, 2010.
[51] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proc. MICRO-48*, 2015.

Paper II

CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling

N. Ramhöj Holtryd, M. Manivannan, P. Stenström and M. Pericàs

Reprint from

30th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2021.

CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling

Nadja Ramhöj Holtryd, Madhavan Manivannan, Per Stenström, Miquel Pericàs
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email: {holtryd, madhavan, per.stenstrom, miquelp}@chalmers.se

2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT) | 978-1-6654-4278-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/PACT52795.2021.00023

***Abstract*—Reducing the average memory access time is crucial for improving the performance of applications running on multi-core architectures. With workload consolidation this becomes increasingly challenging due to shared resource contention. Techniques for partitioning of shared resources - cache and bandwidth - and prefetch throttling have been proposed to mitigate contention and reduce the average memory access time. However, existing proposals only employ a single or a subset of these techniques and are therefore not able to exploit the full potential of coordinated management of cache, bandwidth and prefetching. Our characterization results show that application performance, in several cases, is sensitive to prefetching, cache and bandwidth allocation altogether. Furthermore, the results show that managing these together provides higher performance potential during workload consolidation as it enables more resource trade-offs. In this paper, we propose CBP a coordination mechanism for dynamically managing prefetching throttling, cache and bandwidth partitioning, in order to reduce average memory access time and improve performance. CBP works by employing individual resource managers to determine the appropriate setting for each resource and a coordinating mechanism in order to enable inter-resource trade-offs. Our evaluation on a 16-core CMP shows that CBP, on average, improves performance by 11% compared to the state-of-the-art technique that manages cache partitioning and prefetching and by 50% compared to the baseline without cache partitioning, bandwidth partitioning and prefetch throttling.**

I. Introduction

Memory access time has a significant impact on application performance. Effective utilization of the memory system is therefore necessary. Typically, resources in the memory system (e.g., last-level cache (LLC) and off-chip memory bandwidth) are shared among multiple cores as they help in improving resource utilization during workload consolidation. However, sharing can detrimentally impact average memory access time and performance due to resource contention. Prior work has proposed partitioning of shared resources – cache [1]–[5] and bandwidth [6]–[8] – and prefetch throttling [9] to mitigate contention, reduce or hide memory access time and improve performance.

Recent work has proposed combining cache and bandwidth partitioning [10]–[12], prefetching and cache partitioning [13], [14] and bandwidth partitioning and prefetching [15], [16] to provide additional performance gains. The key insight from this work is that coordinated management of two techniques is more advantageous than considering each in isolation because of the trade-offs that are made possible. However, prior works have not considered combining all three techniques. This misses out on several opportunities. The goal of this paper is to explore the potential of combining all three techniques and thereby facilitate inter-resource interactions and tradeoffs.

In this paper we show that coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling provides the following advantages. Firstly, it makes it possible to address more applications and cover a broader range of workloads, as shown in our in-depth performance characterization (see Section II). The results show that 90% of the applications in the SPEC CPU2006 suite have performance sensitivity (over 10% change in IPC) to at least one of the techniques, and 70% are also sensitive to multiple techniques. Secondly, managing these techniques jointly opens up the opportunity to new and improved trade-offs. There are synergistic interactions between the techniques and they cannot be realized if cache partitioning, bandwidth partitioning and prefetch throttling are not jointly managed.

As an example, consider the case of a workload comprising of two applications. The first application, *lbm*, is sensitive to bandwidth and prefetching while the second, *xalancbmk*, is sensitive to cache size and has lower performance when prefetching is enabled. The best solution when managing all three techniques is to give xalancbmk a large cache allocation, small bandwidth allocation and disable the prefetcher, while giving lbm a large bandwidth allocation, small cache allocation while keeping the prefetcher active. Figure 1 shows the performance from coordinated management of all three

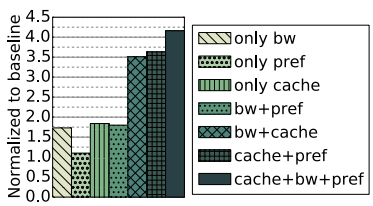


Fig. 1: Workload with lbm and xalancbmk. Total bandwidth 16 GB/s, total cache size 2MB. Executing applications for 1B instructions, more details in Section IV. Settings: lbm-prefethingactive, 12 GB/s, xalancbmk- prefetcher inactive, 4 GB/s, determined from characterization (Section II). Cache partition sizes are decided dynamically.

978-1-6654-4278-7/21/$31.00 ©2021 IEEE
DOI 10.1109/PACT52795.2021.00023

techniques (*cache+bw+pref*) compared to managing a subset of the techniques. The results show that the solution managing all three techniques is better than others managing two of the techniques, and leads to an additional performance gain of 15%. The main challenge of coordinately managing all three techniques is the complexity of evaluating all possible allocations dynamically and determining the best possible allocation while exploiting the large number of possible trade-offs.

Guided by our characterization results, we propose CBP, a technique for dynamic and coordinated management of Cache partitioning, Bandwidth partitioning and Prefetch throttling for multi-programmed workloads. CBP consists of three local controllers, one for each resource, that together with a coordination mechanism manages and allocates the resources. CBP dynamically tunes the three local resource controllers in an iterative fashion in order to address the complexity of navigating the multi-resource search space. While we explore different coordination policies, the one that works best is the following: First, cache and bandwidth are allocated. The cache and the bandwidth allocation controllers provide a minimum allocation to all cores to ensure that requests from non memory-intensive applications do not experience significant delay because of requests from other co-running applications. The remaining cache capacity is allocated with the goal of minimizing the aggregate number of LLC misses while the remaining bandwidth is allocated based on the queuing delay. This allocation is carried out mainly based on the statistics collected from the previous interval. As a next step, the prefetch setting is determined for the current interval by sampling the impact of different prefetch settings on performance for the current allocation of bandwidth and cache. The prefetcher performance influences the next reallocation of cache space and bandwidth. The feedback mechanism between the different techniques dynamically adapts the allocations in order to reach a good configuration depending on the characteristics of the individual applications in the workload. Our approach of combining local resource controllers with a feedback mechanism reduces the computational complexity.

In summary, we make the following contributions:

(a) We present an in-depth characterization of the performance impact of cache, bandwidth and prefetching on the entire SPEC CPU2006 suite. Our characterisation results provide several insights: i) a majority of the applications (over 90%) are sensitive to one or multiple techniques, ii) managing cache, bandwidth and prefetch opens up opportunities for exploiting more trade-offs and improving performance for consolidated workloads, and iii) managing cache, bandwidth and prefetch jointly has the potential to outperform combinations of two of the techniques.

(b) We propose CBP, a technique to dynamically manage the three resources in coordination. The solution is based on simple heuristics in order to sidestep the complexity associated with evaluating all possible configurations and choosing the best performing configuration. CBP works by employing individual resource managers to determine the appropriate setting for each resource and a coordinating mechanism to enable inter-resource trade-offs.

(c) We evaluate our solution with multi-programmed workloads on a 16-core tiled CMP. CBP improves performance by up to 36% (geom. mean 11%) compared to the state-of-the-art technique that manages cache partitioning and prefetching in a coordinated manner and by up to 86% (geom. mean 50%) compared to an S-NUCA without cache partitioning, bandwidth partitioning and prefetching.

The rest of the paper is organized as follows. Section II motivates the need for a coordinated approach using cache partitioning, bandwidth partitioning and prefetch throttling. Section III describes our proposed solution in detail. We then discuss the methodology in Section IV and Section V presents the evaluation of the proposal. We provide an overview of related work in Section VI and conclude in Section VII.

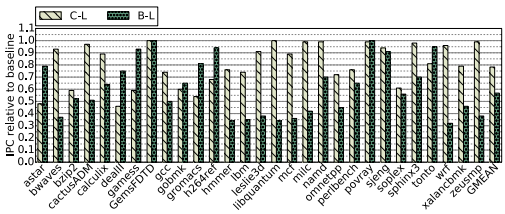
II. Characterization

In order to motivate the need for coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling, we perform a detailed characterization study of applications in the SPEC2006 CPU suite. The aim of this study is to: i) characterize applications to determine the extent to which they are performance sensitive to cache, bandwidth and prefetch settings, ii) understand the different resource interactions, their impact on performance and the inter-resource trade-offs that are possible, and iii) demonstrate the performance potential of coordinated management of all three resources over a subset of resources.

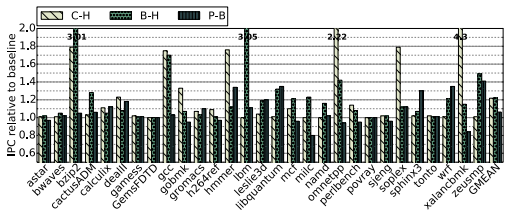
A. Sensitivity to cache, bandwidth and prefetch settings

To understand the sensitivity of applications to cache, bandwidth and prefetch settings, we model a system consisting of one out-of-order core with a 3-level cache hierarchy using the Sniper simulator [17]. Details about the methodology are provided in Section IV. For this experiment, the baseline LLC and bandwidth allocation is 512 KB and 4 GB/s, respectively. We run the application in steady-state for 1B instructions and use IPC as a measure of performance.

Figure 2 shows the performance impact of changing the cache allocation, bandwidth allocation and enabling of prefetching normalized to the baseline allocation without prefetching. Note that we only change the setting for one resource at a time. In Figure 2a, C-L and B-L represent low allocation settings where the cache allocation is decreased to 128 KB and the bandwidth allocation is decreased to 1 GB/s, respectively, while prefetching is disabled. Similarly, in Figure 2b, C-H and B-H represent high allocation settings, where the cache allocation is increased to 2MB and the bandwidth allocation is increased to 16GB/s, while prefetching is disabled. Finally, P-B represents the setting where prefetching is enabled with the baseline cache and bandwidth allocation. We classify applications as *performance sensitive* to a specific resource if the modified allocation results in a 10% deviation from the baseline IPC. We refer to applications that are performance sensitive to a change in cache allocation as cache sensitive (CS), sensitive to change in bandwidth allocation as



(a) Slowdown when decreasing the cache size to 128kB (C-L), and slowdown when decreasing the bandwidth allocation to 1GB/s (B-L) in comparison to the baseline allocation, with prefetching disabled.



(b) Performance improvement from increasing cache allocation to 2MB (C-H), and increasing bandwidth allocation to 16GB/s (B-H) in comparison to the baseline allocation, with prefetching disabled. Performance improvement from enabling prefetching for the baseline allocation (P-B).

Fig. 2: Performance impact of changing cache size, bandwidth allocation and prefetcher setting. There are 6 CS-BS-PS applications, 8 CS-BS, 6 BS-PS, 3 CS, 3 BS and 3 applications are insensitive (I) to all three techniques.

Application	CS	BS	PS	Application	CS	BS	PS
astar	✓	✓		bwaves		✓	
bzip2	✓	✓		cactusADM		✓	
calculix	✓	✓	✓	dealII	✓	✓	✓
gamess	✓			GemsFDTD			
gcc	✓	✓		gobmk	✓	✓	
gromacs	✓	✓		h264ref	✓		
hmmer	✓	✓	✓	lbm	✓	✓	✓
leslie3d		✓	✓	libquantum		✓	✓
mcf	✓	✓		milc		✓	✓
namd		✓		omnetpp	✓	✓	
perlbench	✓	✓		povray			
sjeng				soplex	✓	✓	✓
sphinx3		✓	✓	tonto	✓		
wrf		✓	✓	xalancbmk	✓	✓	✓
zeusmp		✓	✓	**Count**	17	23	12

TABLE I: Summary of performance sensitivity to cache, bandwidth and prefetch setting. Note that the classification of applications is based on the specific settings for cache, bandwidth and prefetch that we have evaluated.

bandwidth sensitive (BS) and sensitive to prefetch throttling as prefetch sensitive (PS). Table I summarizes the sensitivity of the applications to the three resources.

The sensitivity results for cache size show that nearly 60% of the applications (17 out of 29) are sensitive to changes in cache allocation. The extent to which applications are performance sensitive varies greatly with a performance increase of up to 4x in some cases. Furthermore, a larger number of applications are sensitive in the low allocation setting in comparison to high allocation setting (17 compared to 11). The sensitivity results for bandwidth allocation also shows a similar trend, as more applications are sensitive in the low allocation setting (23 compared to 15). Also, the extent of performance sensitivity varies greatly with an increase of up to 3x. The sensitivity results for prefetch throttling indicates that nearly 38% of the applications (10 out of 29) are sensitive to prefetching and experience a speedup. However, some applications (2 out of 29) suffer from a slowdown due to prefetching. In summary, we make the following observation:

OBSERVATION 1. *In SPEC CPU2006 suite 90% of the applications are sensitive to one resource, while 70% are sensitive to multiple resources and the extent of sensitivity varies greatly.*

B. Inter-resource interactions and trade-offs

Next, we investigate inter-resource interactions and trade-offs that are enabled when jointly managing cache, bandwidth and prefetching. We focus on intra-application resource interaction initially. There are four possible types of interactions within an application: cache-bandwidth-prefetch, bandwidth - prefetch, cache - prefetch and cache - bandwidth.

Regarding the cache-bandwidth-prefetch trade-off, we want to find out how the performance impact from prefetching varies with the allocation of cache and bandwidth. Figure 3 shows the performance impact of prefetching for three different cache/bandwidth settings normalized to the respective baseline setting without prefetching. The cache and bandwidth setting for an application in a low allocation scenario (P-L) is 128KB and 1GB/s, the baseline allocation scenario (P-B) setting is 512KB and 4GB/s while the high allocation scenario (P-H) setting is 2MB and 16GB/s.

For some applications, lower bandwidth and cache allocation leads to higher sensitivity for prefetching as seen in *hmmer*. This is because avoiding a miss altogether, as a consequence of accurate prefetching, can have a larger impact in low allocation settings where the bandwidth is scarce and the memory queuing delays tend to be longer. Also, there are applications like *gcc* which experience higher prefetch sensitivity with a larger cache and bandwidth allocation. The

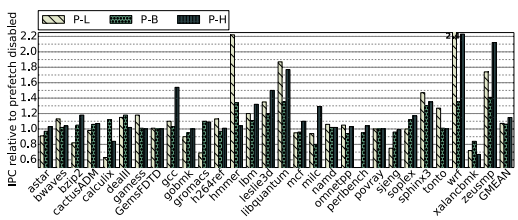
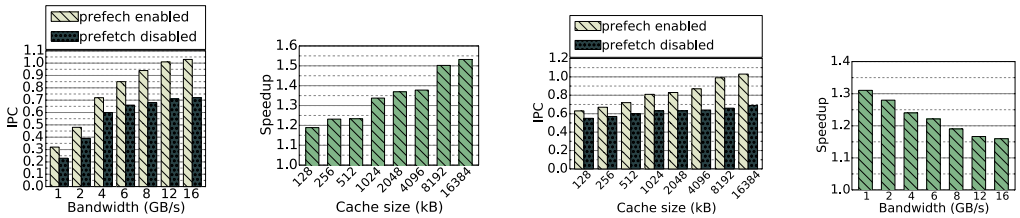


Fig. 3: Performance impact of enabling prefetching relative to allocation of cache and bandwidth, for allocation settings; L:128kB,1GB/s B:512kB,4GB/s H:2MB,16GB/s



(a) IPC with and without prefetching for different bandwidth allocation.

(b) Performance improvement from prefetching depending on cache allocation.

(c) IPC with and without prefetching for different cache allocation.

(d) Performance improvement from increasing cache allocation to 2MB depending on bandwidth allocation.

Fig. 4: leslie3d - example of interactions and its impact on performance within a single application.

results indicate that applications tend to be prefetch sensitive in some settings and prefetch insensitive in others. We make the following observation:

OBSERVATION 2. *Allocation of cache and bandwidth influences prefetch sensitivity. Furthermore, applications tend to be prefetch sensitive in some settings and prefetch insensitive in others.*

In the interest of space we use *leslie3d* as a representative example to illustrate the other pairwise resource interactions and trade-offs, since it is sensitive to all three techniques. Note that the baseline setting will be used for the resources unless specified otherwise. The bandwidth - prefetch interaction manifests in two different ways. Firstly, prefetching typically increases the number of memory accesses and this in turn increases the bandwidth pressure. In the case of *leslie3d* prefetching results in a 15% increase in the number of memory requests in comparison to the baseline. Note that prefetch misses, i.e. prefetched blocks that are evicted before use, can result in a further increase in pressure on the memory bandwidth. Secondly, the performance improvement from prefetching can be influenced by the bandwidth allocation. The results in Figure 4a show the performance for different bandwidth allocations with and without prefetching. We make the following observation:

OBSERVATION 3. *A larger bandwidth allocation can compensate for increased bandwidth demands, due to inaccurate prefetches, leading to increased performance with prefetching.*

The cache - prefetch interaction also manifests in two main ways. Firstly, the performance loss from reduced cache allocation can be offset if prefetching is effective. Figure 4c shows the IPC for different cache allocations with and without prefetching. The results show that the performance of a 128KB allocation with prefetching is better than a 512KB allocation without prefetching. Secondly, larger cache sizes (if it is used efficiently) can lead to higher speedup from prefetching. Figure 4b shows the performance improvement from prefetching with different cache allocations normalised to the respective cache allocation without prefetching. The results show that prefetching is effective with lower cache allocation and that its effectiveness can increase with additional allocation. The reason for this behaviour is that a larger cache reduces the number of memory accesses which has the same effect as increasing the available bandwidth, i.e. a lower queuing delay, which is more forgiving when there is an increase in memory accesses caused by inaccurate prefetches (in leslie3d there is a 15% increase in dram accesses caused by prefetching). These results lead to the following observation:

OBSERVATION 4. *A trade-off can be made between either increasing cache size or enabling prefetching, leading to the same performance, for applications which are performance sensitive to both cache and prefetching.*

As for the cache - bandwidth interaction, a lower bandwidth allocation can result in a larger sensitivity to cache size. Figure 4d shows the performance improvement from increasing the cache allocation from 512kB to 2MB with different bandwidth allocation settings. The results show that performance improvement from additional cache allocation is much higher in low bandwidth allocation settings (see the result for 1GB/s bandwidth allocation). This is because the average cost of a miss is much higher in the case of lower bandwidth allocation. The results also show that a large cache allocation can reduce the performance sensitivity to bandwidth allocation (see the result for 16GB bandwidth allocation). These results lead to the following observation:

OBSERVATION 5. *A trade-off can be made between either increased cache space or increased bandwidth allocation, for applications which are performance sensitive to both cache and bandwidth.*

We now describe how the observed intra-application interactions and trade-offs can be leveraged in the inter-application setting for multi-programmed workloads. Let us revisit the example of running a simple workload comprising two application (*lbm* and *xalancbmk*) on a dual-core system with 2MB LLC capacity and 16GB/s bandwidth, discussed in Figure 1. For achieving the best aggregate performance we expect xalancbmk to get the majority of the cache (nearly 1.75MB), while lbm is given a smaller cache allocation of 256KB. For bandwidth, we would expect lbm to have a large allocation (12GB/s) of the available bandwidth and xalancbmk to get a

smaller allocation (4GB/s). This is reflected in Observation 5 about the trade-off between cache and bandwidth where we would prioritize the application that shows the highest sensitivity for the resource. Furthermore, as reflected in Observation 2, we expect prefetching to be more effective for lbm since it has a large allocation of bandwidth. In the case of xalancbmk, prefetching leads to lower performance regardless of the allocation of cache and bandwidth.

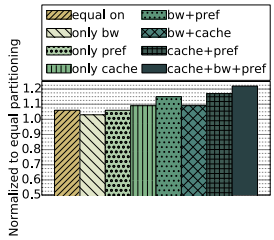
C. Generalizability of performance characterization

The results in the performance characterization are based on a range of configurations for cache (128KB up to 16MB) and for bandwidth (1GB/s up to 16GB/s). This range of explored configurations represents common design points in existing multicore chips. We do not expect that a wider range of configurations to fundamentally change the observations. We investigate the impact of using two other prefetchers, GHB [18] and AMPM [19]. The average performance impact from prefetching increases from 6% using the stride prefetcher to 10% with GHB and to 19% with the AMPM prefetcher. With the AMPM prefetcher 16 applications are positively performance sensitive compared to 10 applications with the stride prefetcher. We show the impact of using these prefetchers with CBP in Section V-A. The results show that using an aggressive prefetcher increases the performance sensitivity and the potential for coordinated management as well.

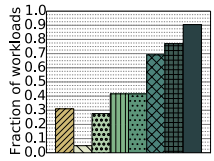
D. Potential for coordinated management

In order to show the potential for coordinated management of cache, bandwidth and prefetch throttling we run 640 randomly generated workloads each comprising 4 SPEC 2006 CPU applications. We compare the performance of jointly managing all the three resources to other resource managers that only manage a subset of these resources. For this experiment the baseline allocation of cache and bandwidth for each application is 512kB and 4GB/s. We use an exhaustive search algorithm to find the best static configuration (over 1B instructions) for the different resources when running each workload. Figure 5a shows average (geometric mean) performance with different resource managers normalized to the baseline settings without prefetching. *equal on*, depicts the performance when prefetching is enabled for all applications and improves performance by 6% while *only pref*, depicts the performance when prefetching is selectively activated and improves performance by 9%. *cache+bw+pref* results show that coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling improves performance by 5% compared to the best combination of two techniques (22% compared to 17%).

Figure 5b shows the number of workloads (among the 640 workloads considered) that experience a performance gain of at least 10% using the different resource managers discussed previously. The results show that 90% (597) of the workloads are sensitive to the resource manager that jointly manages all three techniques. A smaller fraction of the workloads are sensitive to resource managers that manage a subset of these techniques (77% are sensitive to *cache+pref* resource manager and 69% are sensitive to the *cache+bw* resource manager).



(a) Performance potential of using different resource managers.



(b) Fraction, of the workloads, with at least 10% performance improvement.

Fig. 5: Potential for coordinated management measured using 640 random workloads of 4 SPEC CPU2006 applications. Performance is obtained using an exhaustive search algorithm that evaluates bandwidth settings (2GB/s, 4GB/s, 6GB/s), cache settings (256kB, 512kB, 1024kB) and prefetching settings (active/inactive), in conjunction, to determine the resource allocation for each application in the workload that maximizes aggregate performance.

In summary, the results from the characterization study demonstrate that around 90% of applications in the SPEC 2006 CPU suite are sensitive to different resources and that coordinately managing them opens up new possibilities for improving performance by trading resource allocations. Furthermore, jointly managing these resources has the potential to cover a broader range of workloads and outperform resource managers that manage a subset of resources. In the next section we will discuss how the proposed resource manager, CBP, determines cache, bandwidth and prefetch settings for different applications in a workload.

III. CBP Resource Manager

Section III-A provides an overview of the CBP resource manager. Section III-B discusses the individual resource controllers while Section III-C describes how the coordination mechanism ties the local resource controllers together. Finally, the implementation and overhead of the proposed mechanism is discussed in Section III-D.

A. Overview

CBP is a coordinated mechanism for dynamically managing cache partitioning, bandwidth partitioning and prefetch throttling. The design consists of one local controller for each of the three techniques, and a coordination mechanism, as shown in Figure 6.

The cache allocation controller estimates the number of misses for different cache sizes using auxiliary tag directories (ATDs) [1] and uses this as input for determining cache allocation. The cache allocation per application is determined such that it reduces the aggregate number of cache misses for the entire workload. The bandwidth allocation controller uses memory request queuing delay experienced by the applications as input and allocates the available bandwidth in proportion

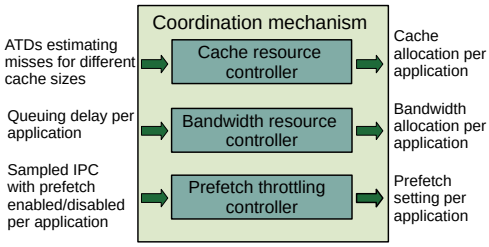


Fig. 6: Overview of CBP resource manager.

to the delay. The bandwidth allocation controller, assigns a larger allocation of the available bandwidth to applications that experience longer queuing delay, and a comparatively lower allocation to those that experience shorter queuing delays. Note that the cache and the bandwidth allocation controllers provide a minimum allocation to all applications to ensure that requests from applications that are not memory-intensive do not experience significant delay because of requests from other co-running applications. Lastly, the prefetch controller samples IPC with and without prefetching to determine whether the prefetcher should be enabled/disabled for each application.

The three techniques are dynamically tuned using the coordination mechanism in an iterative manner such that the local controller takes into consideration the decisions taken by the other controllers.

B. Local resource allocation controllers

A partitioning solution for shared resources like cache and bandwidth typically comprises two components: allocation policy, that determines how a resource is divided among multiple co-running applications, and an enforcement mechanism, that enforces the partitioning decision. Similarly, prefetch throttling involves a policy to determine the best prefetch setting to use and a mechanism implemented in hardware to enforce the setting. In the context of CBP, the policy component is of particular interest because it enables inter-resource trade-offs. We discuss the allocation policy in this section and defer the details of the enforcement mechanism to Section III-D.

1) Cache partitioning: The cache allocation controller uses the Lookahead algorithm [1], to determine cache allocation. In a nutshell, the algorithm computes the utility for each application where utility is the measure of how many additional misses can be reduced with allocation of cache ways. It then computes the number of ways that maximizes the utility for each application (while ensuring this is less than the total number of ways available for allocation). Finally, it compares the utility values for the different applications, determines the application that has the highest utility and assigns the precomputed number of ways that maximizes the utility for that application. The process repeats, with recomputation of utility for each application and reassignment of available cache ways to the application that has the largest utility, until the rest of

Algorithm 1: Bandwidth allocation controller pseudo-code

```
   Input  : A list queuingDelayPerApplication
   Output: A list bandwidthAllocationPerApplication
1  At time period reconfiguration_interval;
2  remainingBandwidth = (totalBandwidth-
     min_bandwidth_allocation*totalNumberOfCores
     totalDelay = 0;
3  for i ← 0 to totalNumberOfCores−1 do
4  |   totalDelay += queuingDelayPerApplication[i];
5  |   bandwidthAllocationPerApplication[i] =
   |     min_bandwidth_allocation;
6  end
7  for i ← 0 to totalNumberOfCores−1 do
8  |   bandwidthAllocationPerApplication[i] +=
   |     (queuingDelayPerApplication[i]/totalDelay) *
   |     remainingBandwidth;
9  end
```

the available capacity is distributed. The allocation controller relies on sampled ATDs to estimate, based on past behaviour, the number of misses that can be avoided with additional allocation of cache ways for each application. In order to adapt to an inclusive cache hierarchy, we assign a minimum allocation of cache space (*min_ways*) to all the applications before distributing the remaining capacity.

2) Bandwidth partitioning: We propose a bandwidth allocation algorithm, that partitions bandwidth proportional to the memory queuing delay experienced by each application. The pseudo-code for the proposed bandwidth allocation controller is outlined in Algorithm 1. The controller assigns a minimum bandwidth allocation (*min_bandwidth_allocation*) for each application. The minimum allocation is important because it ensures that the less memory intensive applications have a low memory latency. A sensitivity study for the minimum bandwidth parameter is presented in Section V-C3. The remaining bandwidth is set for distribution among the applications (see line 2). The algorithm computes the total queuing delay by summing up the individual queuing delays experienced by each application (line 4) while assigning the minimum allocation to each application (line 5). As the next step, the remaining bandwidth is allocated proportionally to the queuing delay experienced by the application (line 7-9). For precision we need performance counters for monitoring queuing delays similar to the performance counters for L2 stall cycles (PMU event “STALLS L2 PENDING”) which are available in Intel processors [20].

3) Prefetch throttling: The prefetch throttling policy determines the best prefetcher settings for each application. The pseudo-code for the prefetch throttling controller is outlined in Algorithm 2. The algorithm considers two possible settings – the prefetcher enabled vs. disabled – but can easily be extended to support other aggressiveness settings as well. The algorithm uses the sampled IPC values, for each application obtained, with different prefetcher settings over a sample period (*prefetch_sampling_period*) as input. The algorithm first computes the speedup from prefetching for each application using the sampled IPC values. If the speedup is below a threshold (*speedup_threshold*) the prefetcher is deactivated for the next prefetch interval (*prefetch_interval*) (line 3-4). If

Algorithm 2: Prefetch throttling controller pseudo-code

```
   Input  : Two lists ipcWithPrefetchingActive,
            ipcWithPrefetchingInactive
   Output: A list prefetchSettingPerCore
1  At time period prefetch_interval;
2  for i ← 0 to totalNumberOfCores−1 do
3      if (ipcWithPrefetchingActive[i]/
         ipcWithPrefetchingInactive[i] >
         speedup_threshold then
4          prefetchSettingPerCore[i] = 0;
5      else
6          prefetchSettingPerCore[i] = 1;
7      end
8  end
```

the speedup is above the threshold prefetching is activated for the next prefetch interval (line 6). The prefetch-throttling controller is generic enough to support any type of prefetcher.

C. Coordination Mechanism

The goal of the coordination mechanism is to ensure that each local controller takes into account the decisions taken by other controllers. This is necessary to exploit the trade-offs outlined earlier in Section II. There are two essential tasks carried out in order to establish this: i) inter-controller interaction ii) controller ordering.

Inter-controller interaction: Figure 7 provides an overview of the interactions between the different resource allocation controllers. Firstly, we describe how the bandwidth allocation controller decisions takes into account the decisions made by the cache and the prefetch controller. The bandwidth allocation controller, makes decisions based on the queuing delay of each application which is affected by the number of memory accesses. The cache allocation controller through a larger cache allocation can reduce the number of memory accesses (Interaction #1). This leads to a lower bandwidth allocation for applications that can efficiently use the cache. The prefetch throttling controller influences the bandwidth allocation decision mainly through prefetch misses (prefetched data that is not used) (Interaction #2). This is because prefetch misses lead to more memory requests and potentially a higher queuing delay.

Next, we describe how the prefetch throttling controller takes into account the decisions made by the other controllers. The prefetch-throttling controller makes decisions by sampling the IPC with different prefetcher settings over a specific interval. The sampled IPC values, used to determine the prefetcher setting, reflects the effect of cache and bandwidth allocation decisions made by the respective resource

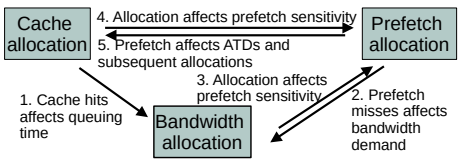


Fig. 7: Interactions among the different resource allocation controllers in CBP .

controllers (Interaction #3-4). Finally, we describe how the cache allocation controller is affected by the prefetch throttling controller (Interaction #5). If an application benefits from prefetching, this reflects on the miss count values monitored in the ATDs. Since the cache allocation is computed based on the counter values observed in the ATD, this ends up affecting the subsequent cache allocation decision, resulting in a smaller cache allocation for prefetch sensitive applications.

Section V-A evaluates alternate designs that only exploit a subset of the interactions discussed above to understand the performance impact of failing to exploit the different inter-resource interactions and trade-offs.

Controller ordering: Since the decision taken by one controller has the potential to influence those taken by others, the order in which local controllers make allocation decisions is important. We discuss the timeline showing when the different resource allocation controllers are invoked and how they interact with each other, in an iterative manner, using the example illustrated in Figure 8. The three resource controllers, are invoked after a specific interval that we refer to as the *reconfiguration_interval* in the sequence shown in the figure.

First cache and bandwidth are equally partitioned among all applications at time 0, since information about misses and queuing delays is initially unavailable, as shown in Step 0. This is followed by sampling the IPC of applications with different prefetch settings for a specific interval (twice the *prefetch_sampling_period*) as shown in Step 1. Note that the prefetch controller is ordered (invoked) after cache and bandwidth allocation to exploit the observation that the prefetch effectiveness is determined by the current allocation of cache and bandwidth. Based on the sampled IPCs the prefetch throttling controller determines the appropriate prefetcher setting for each prefetcher for the current *reconfiguration_interval*.

Cache and bandwidth allocation controllers are again invoked after the *reconfiguration_interval*, as shown in Step 2. Since the allocation of cache and bandwidth are based on statistics collected during the previous interval, the relative ordering among them has little impact. A cache allocation decision for the next interval (shown in step 2.1) is influenced by the number of hits and misses observed in the previous interval. The ATD values will be halved after each reconfiguration, in order to be sensitive to changes in the last time interval while the per application queuing delays are accumulated with those from the previous interval. The bandwidth

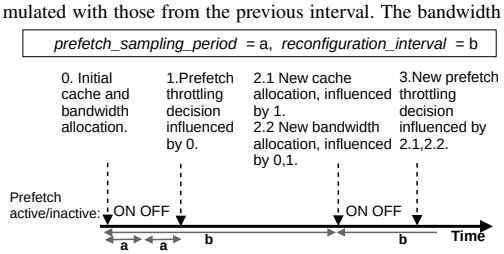


Fig. 8: Timing and interactions of CBP resource manager.

allocation decision shown in Step 2.2, is influenced both by the previous bandwidth and cache allocation and prefetcher setting in the previous interval as discussed previously. Finally, the prefetcher throttling controller shown in Step 3 is influenced by the new cache and bandwidth allocation. The impact of using a different order between prefetch, cache and bandwidth is evaluated in Section V-A. The interactions among the different resource allocation controllers take place over multiple iterations, and are key to finding an effective solution.

D. Implementation

The computational overhead of CBP resource management is low since the design uses heuristics to guide the allocation decisions instead of exhaustively evaluating the different possible allocations.

The cache allocation controller needs hardware support in order to estimate the number of misses with different cache sizes. We use sampled ATDs [1] as discussed previously to compute the effect of different cache allocations on the misses. When enforcing cache partitioning there is an overhead associated with invalidations due to reconfiguration decisions. This is modelled faithfully by invalidating the addresses and re-fetching them when accessed, this includes the latency and impact on bandwidth from accessing memory. We have used the enforcement mechanism proposed by Holtryd et al. [21] since it is suitable for a modern tile-based CMP and is both fine-grained and locality-aware. The enforcement mechanism uses per-core Cache Bank Tables (CBTs), where mappings between addresses and banks are recorded. When a request needs to access the LLC, the CBT is used to identify the cache bank that the address is mapped to. Inside each bank, way partitioning hardware divides the capacity. The enforcement results in a partition granularity of 32kB on our system, see Section IV. We incur hardware cost for implementing ATDs and cache partition enforcement [1], [21].

The bandwidth partition enforcement is done in likeness with Intel Memory Bandwidth Allocation (MBA) technology [22], [23] which is commercially available. The solution uses delays as a way to allocate the bandwidth. An application with a high delay has a low allocation, and experiences a longer queuing delay for each memory access. In our solution the additional delay is added after the LLC, instead of after the L2, as in the original proposal.

The overhead of prefetch-throttling comes from sampling an application with different prefetcher settings. This is because deactivating prefetching for an application can be detrimental for its performance, especially when prefetching is effective. Likewise, it is detrimental to turn it on prefetching (for a sample period) for an application whose performance is hurt by prefetching.

IV. Experimental Methodology

A. Simulated Architecture

We evaluate our proposal on a 16-core tiled CMP architecture modeled using the Sniper Simulator [17]. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and an LLC bank of 512KB. The cache latencies assumed have been modelled using CACTI 6.5 [24]. Details about the baseline architecture are shown in Table II. A sensitivity study is provided for the CBP parameters in Section V-C.

B. Methodology

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [25]. We create 14 workload mixes (each comprising 16 applications) by randomly selecting applications from the entire SPEC CPU2006 suite. Details about workload mixes are presented in Table III.

We fast-forward for 16B instructions (in total) and then carry out detailed simulation until all benchmarks have completed at least 500M instructions. Statistics are reported based on the detailed simulation of 500M instructions. After this period the applications continue to run and compete for resources to avoid having a lighter load on long running applications. The methodology is in line with earlier works [4], [5], [26].

C. Metrics

We report normalized weighted speedup over baseline for each workload.This is computed by $\frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{i,RM}}{IPC_{i,baseline}}$, in order to evaluate system performance for multi-programmed workloads. RM refers to the system with a resource manager that manages cache, bandwidth and prefetcher settings and the baseline refers to a system with unpartitioned cache and bandwidth and without prefetching.

We also report average normalized turnaround time (ANTT) for each workload since this is a user-oriented performance metric which shows fairness. ANTT is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{CPI_{i,RM}}{CPI_{i,baseline}}$

Table IV shows the resource managers we evaluate, in addition to CBP, and the corresponding settings they use for cache, bandwidth and prefetching. The baseline configuration represents a system with unpartitioned cache, unpartitioned

Cores	16 cores, x86-64 ISA, 4GHz, OOO, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, inclusive, 9-cycle data and 2-cycle tag latency, LRU
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 MCUs, 1 channel/MCU, latency 80 ns, 16GB/s per channel
Prefetcher	stride-based, located in L2, 4 prefetches stop at page boundary, 8 flows/core
Alternative prefetchers	GHB Distance prefetching(G/DC), width=4, depth=1, GHB =512, Index table = 512 AMPM, 256 pages, degree 4
CBP parameters	*reconfiguration_interval*=10ms *prefetch_sampling_period*=0.5ms, *speedup_threshold* = 1.05, *prefetch_interval*=10ms, *min_bandwidth_allocation*=1, *min_ways*=4

TABLE II: Configuration of the simulated 16-core tiled CMP.

w#	Types	Benchmarks
w1	4CS-BS-PS,5CS-BS,3BS-PS,3CS,1BS	xalancbmk(xa),gromacs(gr),libquantum(li)(2) h264ref(h2),zeusmp(ze),tonto(to),soplex(so), lbm(lb),perlbench(pe),calculix(ca),milc(mi) sphinx3(sp),bwaves(bw),gobmk(go),gamess(ga)
w2	3CS-BS-PS,5CS-BS, 5BS-PS,2CS,1BS	lb,to,pe,go,gcc(gc),mi,li(2),namd(na), h2,cactusADM(cac),ze(2),ca,so,astar(as)
w3	6BS-PS,CS,5BS,4I	bw(2),povray(po)(2),sjeng(2),sp(2),na(2),ze, GemsFDTD(Ge),cac,li,mi,wrf(wr)
w4	CS-BS-PS,2CS-BS,5BS-PS,3CS,2BS,3I	po,bw(2),h2,sjeng(sj),li(2),gr,na,mi(2),as,Ge, ga,wr,lb
w5	5CS-BS-PS,10CS-BS,BS-PS	dealII(de),omnetpp(om)(2),go(2),hmmer(hm),xa leslie3d(le),bzip2(bz)(2),gc,so,mcf(mc),pe,ca(2)
w6	3CS-BS-PS,5CS-BS,4BS-PS,2CS,2BS	sp,bw(2),h2,om,li,gr,go,mi(2),as,hm,ga,lc,lb,ca
w7	2CS-BS-PS,2CS-BS,3BS-PS,5CS,4I	po(2),to,sj,h2(2),na,lb(2),ze(2),gr,Ge,as,wr,ga
w8	4CS-BS-PS,4CS-BS,2CS-PS,3BS-PS 3BS	de,bw(3),xa,mi(3),om,li(2),bz,go,so,hm,pe
w9	2CS-BS-PS,5CS-BS,2BS-PS,3CS,BS,2I	gc,po,to,hm,sj,h2,bz,ze,gr,so,Ge,as,pe,wr,ga,cac
w10	2CS-BS-PS,3CS-BS,6BS-PS,CS,2BS,2I	sj,bw(2),de,na,li(2),om,ze,mi(2),xa,Ge,bz,wr,gc
w11	2CS-BS-PS,4CS-BS,4BS-PS,CS,2BS,3I	po,om,sj,go,na(2),le,ze,xa,Ge,bz,wr,ca,sj,sp,gc
w12	6CS-BS-PS,8CS-BS,2CS	de,to,go,h2(2),hm,gr,xa,as(2),bz,ga,gc,lb,so,ca
w13	3CS-BS-PS,2CS-BS,4BS-PS,4CS,3I	to,po,h2,sj,gr,na,as,ze,ga,Ge,lb(2),li,to,mi,wr
w14	5CS-BS-PS,2CS-BS,5BS-PS,CS,BS,2I	de,bw,go,po,hm,na,xa,ze,so,Ge,mc,li, pe,mi,ca,wr

TABLE III: 16-core workload.

RM	cache	bandwidth	prefetch
baseline	unpartitioned	unpartitioned	disabled
equal off	equal	equal	disabled
only cache	dynamic (see 3.2.1)	unpartitioned	disabled
only bw	unpartitioned	dynamic (see 3.2.2)	disabled
only pref	unpartitioned	unpartitioned	dynamic (see 3.2.3)
bw+pref	unpartitioned	dynamic	dynamic
bw+cache	dynamic	dynamic	disabled
cache+pref	dynamic	unpartitioned	dynamic
CPpf	dynamic	unpartitioned	enabled
CBP	dynamic	dynamic	dynamic

TABLE IV: The settings for cache, bandwidth and prefetch in the evaluated configurations.

bandwidth and with prefetching disabled. *equal off* configuration represents a system where cache and bandwidth are equally partitioned and prefetching is disabled. *only cache* represents the configuration where cache is partitioned as described in Section III-B1, while bandwidth is unpartitioned and with prefetching disabled. Likewise, *only bw* represents the configuration where bandwidth is partitioned as described in Section III-B2 while the cache is unpartitioned and with prefetching disabled. As for *only pref*, prefetch throttling is performed as described in Section III-B3 while cache and bandwidth remains unpartitioned. The resource managers that jointly manage two out of the three resources (*bw+perf, bw+cache, cache+perf*) in a coordinated manner can leverage a subset of the interactions described in Section III-C. We also compare against *CPpf* [13], a recently proposed technique for jointly managing prefetching and cache partitioning. In *CPpf*, prefetch friendly applications are allocated small partition sizes (because the benefit from prefetching can offset the performance drop from small allocation) while the rest of the cache is allocated to the non prefetch friendly applications. In our implementation, we give minimum allocation to the prefetch friendly applications and use UCP (see Section III-B1), to partition the remaining capacity among the non prefetch friendly applications. We use UCP and per application partitioning, in order to not put *CPpf* at a disadvantage in comparison to other schemes. Finally, CBP jointly manages all the three techniques dynamically.

V. Evaluation

We first compare CBP to other resource managers that manage a subset of resources. Next, we investigate the performance impact of changing coordination, ordering and prefetcher. Finally, we carry out sensitivity analysis for the different design parameters, to understand its impact on the performance of CBP.

A. CBP Performance Analysis

Figure 9 shows normalized weighted speedup for each of the 14 workloads with the bars representing the different resource managers. We use normalized weighted speedup as a measure of performance for the entire workload. *equal off* improves performance in 12 of the mixes and improves performance by 10% on average over the baseline. *only bw* improves performance in 7 of the mixes and on average by 4% over the baseline. *only pref* improves performance for 12 workloads and provides an average improvement of 9%. *only cache* improves performance for all the workloads and provides an average improvement of around 28%.

Coordinated management of bandwidth partitioning and prefetch throttling (*bw+pref*) leads to higher performance in comparison to the baseline in 12 workloads and an average overall improvement of 10%. Coordinated management of bandwidth and cache partitioning (*cache+bw*) improves performance across all workloads and provides an average performance improvement of 37% (up to 64%). Coordinated cache partitioning and prefetch throttling (*cache+pref*) improves performance across all workloads on average by 39% (up to 57%). *CPpf*, cache partitioning influenced by prefetching, improves performance by 39% (up to 63%).

Among the resource managers that perform coordinated management of two resources, *cache+pref* and *CPpf* achieve the best performance. The results also show that the im-

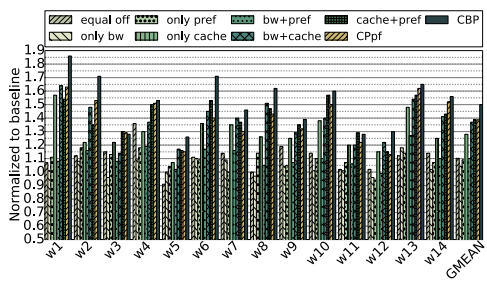


Fig. 9: Performance results, shows normalized weighted speedup over baseline.

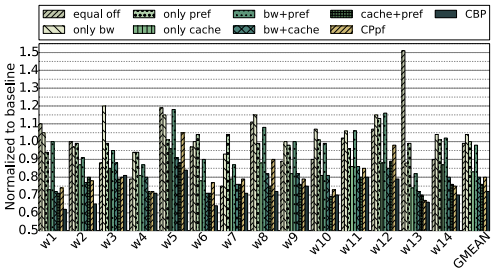


Fig. 10: Fairness results, shows average normalized turnaround time (ANTT) over baseline, where lower is better.

provement achieved with coordinated management of two techniques is larger than summing up the improvements from individual techniques. This shows that coordinated management helps exploit synergistic interactions among the different techniques, which cannot otherwise be leveraged.

Finally, CBP, outperforms previous schemes. CBP turns out to be the best performing coordinated resource manager in 14 of the 15 workloads and provides an average improvement of 50% (up to 86%). CBP improves performance by an additional 11% in comparison to the best performing resource manager that does coordinated management of two techniques, as well as state-of-the-art. In one workload, w3, CBP achieves slightly lower performance (2%) in comparison to *cache+pref*. This is because bandwidth partitioning is not very effective for this specific workload.

Figure 10 shows the average normalized turnaround time which shows the fairness of the different resource managers. Note that a lower value signifies greater fairness. On average, CBP shows 27% better fairness than the baseline and 4% better fairness than the best combination of two techniques, *cache+pref*. *cache+pref* has 4% better fairness than *CPpf*.

Case study: We investigate the performance of a single workload in detail to understand how CBP improves performance in comparison to resource managers that manage a subset of the techniques. Figure 11 shows the IPC for individual applications in a specific workload (w2) normalized to the baseline IPC. We have classified the applications in this workload into two groups. Group 1 comprises applications, from lbm to gcc (in the figure), for which the *cache+pref* resource manager performs better than the *bw+cache* resource manager. Group 2 comprises the rest of the applications in the workload, from soplex to namd, where the *bw+cache* resource manager performs better than the *cache+pref* resource manager. *cache+pref* resource manager provides the best performance for applications in group 1 because the applications are comparatively more memory intensive and get a larger share of the available bandwidth using this resource manager. Applications in group 2 benefit from bandwidth partitioning since they then get a fair bandwidth share, and in addition are not sensitive to prefetching. When we perform coordinated management of all the three resources, we would ideally prefer to have allocation decisions made by *cache+pref* resource manager for applications in group 1 and *bw+cache* resource manager for applications in group 2. With CBP, some applications in group 1 end up with a lower allocation of bandwidth (compared to *cache+pref*) which hurts their performance (see lbm, perlbench, cactusADM) while the rest of the applications in the group see a performance improvement from getting the right amount of the allocation. For the applications in group 2, CBP manages to match the performance of *bw+cache* resource manager. In summary, CBP enables better trade-offs, resulting in a solution that improves overall performance for the workload and outperforms other resource managers that only manage a subset of the techniques.

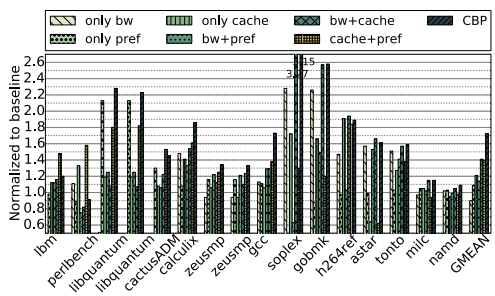


Fig. 11: Results for workload 2.

B. Performance impact of changing coordination, ordering and prefetcher

Impact of coordination: Figure 12 shows the performance impact of coordination and ordering by comparing CBP to resource managers that exploit only a subset of the interactions of CBP, and to a variation of CBP that uses a different ordering. First, we compare the performance of CBP with another resource manager, *RM1*, that uses an accuracy-based prefetch-throttling policy. Note that *RM1* coordinately manages cache and bandwidth and exploits the rest of the inter-resource interactions as CBP except those with prefetching. The lack of prefetch interaction is due to the design of the prefetch throttling mechanism that determines the prefetch setting based on the number of accurate prefetches without being influenced by the cache and bandwidth allocation. Using

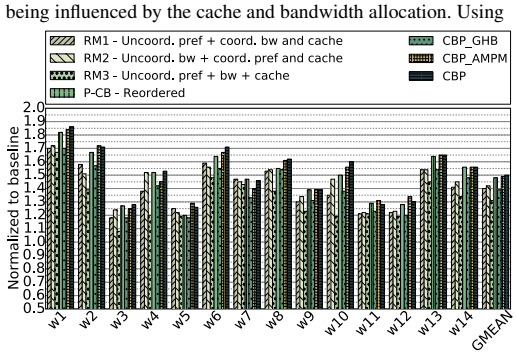


Fig. 12: Performance impact of changing coordination, ordering and prefetcher.

RM1 results in a 7.5% slowdown compared to CBP since it does not exploit bandwidth-prefetch and cache-prefetch interaction. Next, we compare against a different resource manager, *RM2*, that uses a new bandwidth allocation policy. This bandwidth allocation policy only considers the number of demand requests when determining the bandwidth allocations without considering prefetch requests. Cache partitioning and prefetch throttling is performed in a coordinated manner as in CBP. *RM2* leads to a 5.5% slowdown since it does not exploit the interaction between prefetch and bandwidth. We then combine these two policies for bandwidth and prefetch with cache partitioning, which represents an uncoordinated resource manager, *RM3*. *RM3*, leads to a performance decrease of 15% compared to CBP, since it does not exploit a majority of the interactions between cache, bandwidth and prefetch. These results show the importance of exploiting the different inter-resource interactions and the benefit of coordination in the design of CBP.

Impact of ordering: Secondly, we investigate the impact of ordering. In CBP, cache and bandwidth are first allocated and then the prefetch setting is determined based on sampling for the current allocation. In *P-CB* prefetch sampling is first performed followed by the new allocation decision for cache and bandwidth. This leads to 2% slowdown since the decision about the prefetch setting is based on the previous allocation. The impact of changing the order is limited since the reconfigurations are performed frequently (every 10ms).

Impact of prefetcher: Lastly, we investigate the impact of changing the type of prefetcher. Using the GHB prefetcher, *CBP_GHB*, leads to a speedup of 1.47× over baseline, while using the AMPM prefetcher (*CBP_AMPM*) leads to a speedup of 1.49×. Although the single application performance is better with the more aggressive prefetchers, the bandwidth consumption is higher, which has a negative impact on the overall workload performance. This results in nearly the same performance as with the simpler stride prefetcher.

C. Sensitivity analysis

We investigate the sensitivity of CBP to different design parameters in this section.

1) Impact of reconfiguration interval: The reconfiguration interval, determines how frequently the different resource allocation controllers are invoked when running a workload. We investigate the sensitivity of CBP to different reconfiguration interval values, in order to determine an appropriate interval. Figure 13a shows the average (geo. mean) performance when using three different reconfiguration intervals - 1ms, 10ms, 100ms and 1000ms. A shorter reconfiguration period has the potential to adapt faster to phase change behaviour. However, it also incurs a higher overhead of invoking the local controllers. The results show that using a reconfiguration period of 1s leads to a 20% slowdown compared to using 10ms, which demonstrates the importance of frequent adaptation. Overall, the results show that using a 10ms period provides a good trade-off between quick adaptation and the overhead incurred for IPC sampling.

2) Impact of cache size: The results thus far assume that each tile has a baseline cache allocation of 512KB which leads to a total LLC capacity of 8MB for a 16-core CMP. We next study the impact of changing the cache capacity available for a single tile to 1MB and 2MB. Figure 13b shows the average performance achieved using CBP with different per-tile capacity normalized to the baseline configuration with the same capacity. The results show that increasing the total LLC capacity to 16MB leads to a performance improvement of 43% over baseline while increasing the cache size to 32MB leads to a performance improvement of 41%.

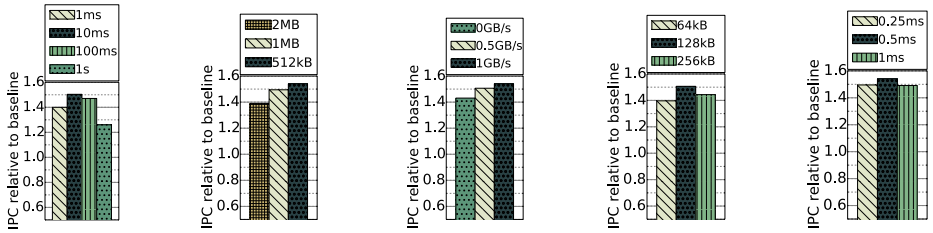
3) Impact of changing bandwidth partitioning parameters: We investigate the sensitivity to the minimum bandwidth allocation, used in the bandwidth allocation algorithm presented in Section III-B2. Figure 13c shows the difference in performance with a minimum bandwidth allocation of 0GB/s, 0.5GB/s and 1GB/s, normalized to the baseline. Not using a minimum allocation leads to a 5% performance decrease compared to using a minimum allocation of 1GB/s. This is because a minimum allocation ensures a shorter queuing delay for applications that are non-memory intensive. Note that the majority of the bandwidth is still allocated to the bandwidth intensive applications. Changing the minimum allocation from 1GB/s to 0.5GB/s did not have a considerable impact on the performance of CBP.

4) Impact of changing cache partitioning parameters: We investigate the sensitivity to the minimum cache allocation, used in the algorithm presented in Section III-B1. Figure 13d shows the difference in performance with a minimum cache allocation of 64kB, 128kB and 256kB, normalized to the baseline. Using a minimum allocation of 64kB leads to 7.5% performance decrease compared to using a 128kB minimum allocation. One reason for the importance of the minimum allocation is that the cache hierarchy is inclusive and the L2 cache has a size of 128kB. Using a a minimum allocation of 256kB leads to a 4.2% slowdown compared to using 128kB. This is because larger minimum allocation reduces the effective capacity available for partitioning.

5) Impact of changing prefetch sampling interval: We finally investigate the impact of changing the *prefetch_sampling_period* used in the prefetch throttling controller III-B3. Figure 13e shows the impact of changing the sampling period on performance normalised to the baseline. The intervals we use for evaluation are 0.25ms, 0.5ms and 1ms. The advantage of using a shorter sampling period is that it carries a lower overhead, while the drawback is the risk of over/under estimating the performance benefit from prefetching. The results indicate the sampling interval of 0.5ms achieves the best performance.

VI. Related work

Isolated Management: Several techniques have been proposed in the literature that focus specifically on cache partitioning, bandwidth partitioning and prefetch throttling. Cache partitioning techniques [1]–[5] help improve performance and



(a) Sensitivity for reconfiguration period. (b) Sensitivity for larger cache size. (c) Sensitivity for minimum bandwidth allocation. (d) Sensitivity for minimum cache allocation. (e) Sensitivity for sample period in prefetch throttling.

Fig. 13: Sensitivity analysis.

achieve better utilization of available cache resources, by avoiding interference among co-running applications and reducing the number of accesses to memory. Bandwidth partitioning techniques [6]–[8], [27], reduce average memory access penalty, by dynamically determining how bandwidth must be shared among the co-running applications. Prefetching can hide memory access latency by fetching the data before it is requested [9], [28]–[32]. However, inaccurate prefetches can impact application performance since it can increase the number and cost of demand misses [33]. Prefetch throttling [9], [34], involves adaptively tuning when and what prefetcher settings are used dynamically based on application characteristics and has been shown to provide better performance and address drawbacks of prefetching. The aforementioned works, consider each of the techniques in isolation and leaves room for improvement, as shown in this work, since they do not take the interaction between cache partitioning, bandwidth partitioning and prefetch throttling into account.

Coordinated Management: Several works have proposed combining two of the techniques in order to exploit the benefits from coordination. These works can be broadly classified into the following groups: i) coordinated cache and bandwidth partitioning [10], [12], ii) coordinated prefetching and cache partitioning [13], [14], and iii) coordinated bandwidth partitioning and prefetching [16]. Sahu et al. propose [10] a method for cache and bandwidth partitioning, using a CPI model for bandwidth and set partitioning for the cache. CoPart [12] combines bandwidth and cache partitioning using a user-level run-time. Unlike CBP, their goal is to improve fairness. Recently, CPpf [13] and Sun et al. [14] propose a coordinated approach for cache partitioning and prefetch where prefetch friendly applications where given a smaller cache allocation. Unlike CBP which maintains per-application partitions, cache partitioning in these two proposals is performed for groups of applications. Ebrahimi et al. [15] propose general mechanisms to make memory scheduling techniques prefetch aware. However, these works cannot use the additional interactions and trade-offs which are available when coordinately managing all three resources, which we have shown is important for performance.

Some works have also proposed coordinated management of multiple resources. For instance, CLITE [35] uses bayesian optimization to provide theoretically-grounded resource partitioning to meet QoS targets of multiple resources (e.g., cores, caches, memory bandwidth, memory capacity, disk bandwidth etc.) among multiple co-located jobs. Bitirgen et al. [11] use machine learning to manage power, cache and bandwidth in a coordinated way to anticipate system-level performance impact of allocation decisions. However, in neither of these works is prefetch throttling considered, which we have shown is important in order to realise the full potential of coordinated resource management. To the best of our knowledge, CBP is the first coordinated resource manager for cache partitioning, bandwidth partitioning and prefetch throttling.

VII. Conclusions

We have presented CBP, a mechanism for coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling. The design is motivated by our in-depth characterisation of the performance impact of cache, bandwidth and prefetch allocation and their interactions. CBP combines local resource allocation controllers with a coordination mechanism that dynamically manages and allocates the resources, in a way which considers both inter- and intra-application interactions. Our evaluation on a tiled 16-core CMP demonstrates that CBP improves performance by up to 86% (geo. mean 50%) compared to a system without partitioning and prefetching and by up to 36% (geo. mean 11%) over the state-of-the-art technique that manages cache partitioning and prefetching in a coordinated manner.

VIII. Acknowledgements

This research has been funded by the Swedish Research Council (VR) under the projects ACE (registration number 621-2014-6221) and PRIME (registration number 2019-04929). In addition, the work has received funding from the European Union Horizon 2020 research and innovation programme under the LEGaTO project with grant agreement No. 780681, and the MECCA project with contract number ERC2013-AdG 340328. The simulations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at the Chalmers Centre for Computational Science and Engineering (C3SE) partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

REFERENCES

[1] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.

[2] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.

[3] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," vol. 2018-Febru, 2018, pp. 104–117.

[4] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-39*, 2012.

[5] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.

[6] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers." Institute of Electrical and Electronics Engineers Inc., 11 2018, pp. 1–14.

[7] D. R. Hower, H. W. Cain, and C. A. Waldspurger, "Pabst: Proportionally allocated bandwidth at the source and target," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 505–516.

[8] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[9] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 1, 1993, pp. 56–63.

[10] A. Sahu and S. Ramakrishna, "Creating heterogeneity at run time by dynamic cache and bandwidth partitioning schemes." Association for Computing Machinery, 2014, pp. 872–879.

[11] R. Bitirgen, E. Ipek, and J. F. Martínez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," 2008, pp. 318–329.

[12] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," vol. 19. ACM.

[13] J. Xiao, A. D. Pimentel, and X. Liu, "Cppf: A prefetch aware llc partitioning approach." Association for Computing Machinery, 8 2019, pp. 1–10.

[14] G. Sun, J. Shen, and A. V. Veidenbaum, "Combining prefetch control and cache partitioning to improve multicore performance," *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, pp. 953–962, 5 2019.

[15] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 141, 2011.

[16] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors." Association for Computing Machinery (ACM), 2011, p. 37.

[17] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.

[18] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, 2005.

[19] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 499–500.

[20] "Intel® 64 and ia-32 architectures developer's manual: Vol. 3b." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[21] N. Holtryd, M. Manivannan, P. Stenström, and M. Pericàs, "Delta: Distributed locality-aware cache partitioning for tile-based chip multiprocessors," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 578–589.

[22] "Intel® 64 and IA-32 Architectures Software Developer Manuals." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#nine-volume

[23] "Introduction to Memory Bandwidth Allocation." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html

[24] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO-40*, 2007.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002.

[26] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.

[27] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, "Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019.

[28] F. Dahlgren and P. Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," vol. 7. IEEE Comput. Soc. Press, 1996, pp. 385–398.

[29] K. Nesbit, A. Dhodapkar, and J. Smith, "Ac/dc: an adaptive data cache prefetcher." IEEE, 2004, pp. 135–145.

[30] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, pp. 90–97, 1 2005.

[31] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization*, vol. 9, pp. 1–29, 3 2012.

[32] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching." IEEE, 6 2018, pp. 83–95.

[33] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," 2009, p. 316.

[34] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. USA: IEEE Computer Society, 2007, p. 63–74.

[35] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 193–206.

Paper III

SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations

N. Ramhöj Holtryd, M. Manivannan and P. Stenström

Under review.

SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations

Nadja Ramhöj Holtryd, Madhavan Manivannan and Per Stenström
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email: {holtryd, madhavan, per.stenstrom}@chalmers.se

***Abstract*—Microarchitectural optimizations are expected to play a crucial role in ensuring performance scalability. However, recent attacks have demonstrated that microarchitectural optimizations, which were assumed to be secure, can be exploited. Moreover, new attacks surface at a rapid pace limiting the scope of existing defenses. These developments prompt the need to review microarchitectural optimizations with an emphasis on security, understand the attack landscape and the potential defense strategies.**

We analyze timing-based side-channel attacks targeting a diverse set of microarchitectural optimizations. We provide a framework for analysing non-transient and transient attacks, which highlights the similarities. We identify the four root causes of timing-based side-channel attacks: *determinism*, *sharing*, *access violation* and *information flow*, through our systematic analysis. Our key insight is that a subset (or all) of the root causes are exploited by attacks and eliminating any of the exploited root causes, in any attack step, is enough to provide protection. Leveraging our framework, we systematize existing defenses and show that they target these root causes in the different attack steps.

1. Introduction

Computer architecture is facing a security crisis [79], [80]. Recent attacks [28], [97], [100], [114], [149], [170], [173] have demonstrated that microarchitectural optimizations, which were assumed to be fundamentally secure for a long time, leak information which can be exploited to steal secrets. Furthermore, efficient attacks continuously emerge targeting defenses, thereby limiting their effectiveness or even rendering the defenses moot altogether [20], [22], [27], [29], [52], [58], [137], [138], [140], [161], [176], [184]. Simultaneously, with the slowing down of Moore's Law, microarchitectural optimizations are expected to play an increasingly important role in ensuring performance scalability. Consequently, there is a strong need to be able to leverage microarchitectural optimizations without compromising security.

Microarchitectural optimizations, widely implemented in commercial processors, like branch predictors [4], [5], [49], [51], [100], caches [68], [136], [188] and prefetchers [1], [40], [44], [66], [158], [175] among others, are prone to attacks. A recent paper [146] demonstrates that several optimizations proposed in literature, but not known to be commercially implemented as yet, such as value prediction [111], also are vulnerable. This underscores the importance of conducting a thorough review of microarchitectural optimizations with an emphasis on security.

Prior works have started the important task of analyzing attacks and defenses for different microarchitectural optimizations [33], [35], [37], [54], [78], [82], [118], [163], [185]. However, most of the works focus only on transient attacks and defenses [33], [35], [78], [82], [185], SW-based defenses [37] or cover a limited set of non-transient attacks [54], [163]. Pandora [146] considers a broader set of non-transient microarchitectural optimizations and provides *microarchitectural leakage descriptors* (MLDs) which quantify the information leakage. The MLDs show if a specific optimization can leak and how much information is leaked (1-bit or a few bits). Unfortunately, this information falls short on providing a systematic analysis of the similarities across different microarchitectural optimizations and the underlying root causes which make them vulnerable to attacks. Such an analysis can also help with the categorization of existing defense strategies and with the potential identification of attacks and defenses.

Our goal, in this paper, is to perform a systematic analysis to highlight the common root causes which make microarchitectural optimizations vulnerable to exploits that reveal secrets. In order to enable analysis of a diverse set of microarchitectural optimizations, we present an abstract model of the architecture and the microarchitectural state transitions involved in an attack. Using this model as a framework, we analyze several timing-based side-channel attacks available in the literature on an extensive set of microarchitectural optimizations: cache, prefetching, branch prediction, computational simplification, speculative execution and value prediction. We also analyse additional microarchitectural optimizations like cache compression, pipeline compression, register-file compression, silent stores and computation reuse but omit them from the discussion due to space constraints.

Our analysis reveals four root causes which are exploited in order to succeed with attacks targeting the diverse set of microarchitectural optimizations covered. The root causes are **determinism**, **sharing**, **access violation** and **information flow**. Here, determinism causes microarchitectural optimizations to be triggered in the same way under the same pre-conditions, leading to predictable microarchitectural state transitions and timing variations. Sharing of microarchitectural state, which is accessible to both the adversary and the victim, enables the creation of a side-channel. Access violation enables access to a secret

outside of the intended protection domain. Finally, information flow refers to exchange of information through microarchitectural state. We note that a subset of these root causes have been identified individually in the context of specific attacks [28], [43], [82], [116], [163]. However, in our analysis we show that a subset of, or all, the root causes are common across attacks on a broad set of microarchitectural optimizations.

We show that the proposed defenses that focus on the vulnerabilities in different microarchitectural optimizations can be classified as targeting one or more of the identified root causes. We observe that similar defenses can be / are applied across different microarchitectural optimizations, with the same root cause vulnerability. For instance, partitioning can thwart attacks using the cache [98], [115], SMT [2] and branch prediction [177], [193], by affecting sharing, information flow and determinism. In addition, the defenses can also be applied to address the applicable root causes in the different steps of the attack. Eliminating any of the root causes, exploited by an attack, in any of the attack steps can protect against the attack. We also discuss potential attacks and defenses for vulnerable microarchitectural optimizations.

Overall, our analysis demonstrates the versatility of the framework to capture a diverse set of attacks and defense strategies for different microarchitectural optimizations. We expect that our framework can be easily extended to study microarchitectural optimizations we do not explicitly cover in this paper. We also believe that it can assist computer architects in understanding the landscape of attacks on a broad range of microarchitectural optimizations, categorizing existing defense strategies proposed to thwart such attacks, and in designing secure microarchitectural optimizations.

In summary, we make the following contributions:

- We identify four root causes: **determinism**, **sharing**, **access violation** and **information flow**, that enable timing-based side-channel attacks on a wide range of microarchitectural optimizations.
- We provide a framework and analyze both transient and non-transient execution attacks on a broad range of microarchitectural optimizations, highlighting similarities and differences.
- We analyze available defenses using our framework and make a classification based on the root causes they address. Based on the analysis, we discuss potential attack and defense possibilities for microarchitectural optimizations.

The paper is structured as follows. Section 2 presents our framework and defines the root causes. Section 3 and 4 use the framework to systematize the attacks and defenses, respectively. Section 5 discuss general observations before we conclude in Section 6.

2. Systematization Framework

We first present an abstract architecture model and outline the steps for carrying out attacks based on the model. We then use this model as a framework to identify the root causes that enable attacks. Finally, we present an actual attack in the context of this framework.

2.1. Abstract model and side-channel attack

The architecture model is represented as a finite state machine (FSM) where the *architectural state (AS)*, comprising SW-visible registers and memory, is the externally visible interface, that is accessible to a program. An FSM transition is caused when instruction execution leads to a change in the AS.

The microarchitecture represents an implementation of the FSM specification and typically comprises several microarchitectural optimizations, denoted $\{O_1, O_2, \ldots, O_n\}$, to enable an efficient implementation. A microarchitectural optimization uses a set of microarchitectural resources, denoted $R = \{R_1, R_2, \ldots, R_m\}$, to implement the intended functionality. This model permits resources to be shared across different optimizations. We define *microarchitectural state (MS)* as a snapshot of the state of all the m microarchitectural resources in the system at time instance t, denoted $MS = \{state(R_1), state(R_2), \ldots, state(R_m)\}_t$.

It is important to note that while the change in AS caused by an FSM transition remains the same across different implementations of a given FSM specification, the change in *MS* varies depending on the optimizations triggered, resources used and the implementation. Even when considering a specific implementation, there is a one-to-many mapping relationship between AS and *MS*; i.e., a single AS can have several equivalent *MS*. Furthermore, the time it takes for an implementation to make a transition between different *MS* (caused by an action) may vary and this property is typically exploited by attacks. As an example, executing a load instruction will cause a change in the AS while the insertion of a corresponding line in the cache hierarchy as a consequence of executing the load instruction will cause a change in the state of the cache(s) which is a microarchitectural resource.

We next consider an abstract model of an attack that shows the different steps involved while leveraging *MS* as the side-channel to communicate the secret from a victim to an adversary. In our model, we define a step as a tuple of current state and action which leads to a new state, $\{MS_{current}, action\} \rightarrow MS_{next}$. Figure 1 shows the different steps listed in this model and is based on the attacks proposed in literature [33], [35], [37], [78], [82], [118], [163], [185]. We assume *MS* is in the initial state (MS_I) before any of the steps in the attack are carried out. When the *setup* step is performed MS_I makes a transition to the primed state (MS_P). The *setup* step ensures that the necessary preconditions are in place to encode the secret into MS_P in the next step of the attack. When the *interact* step is performed the secret is accessed and is encoded in the microarchitectural state (MS_E). The secret is encoded specifically through the state of one or more microarchitectural resources. If the secret is encoded through a microarchitecture resource state, that is accessible to both the victim and the adversary, it can potentially be used as a side-channel to communicate the secret.

Attacks optionally utilize the *transmit* step in case the encoded microarchitectural resource state is not accessible

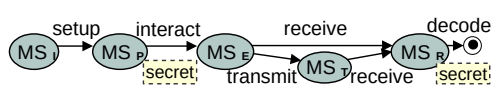


Figure 1. *MS* transitions in different steps of an attack.

by the adversary or the specific *MS* based side-channel is noisy (i.e. the channel is prone to high error rate and has low channel bandwidth). When the transmit action is performed the secret is usually re-encoded through the state of a different shared microarchitectural resources (MS_T) which can address the aforementioned transmission limitations. When the *receive* step is performed, the adversary accesses the microarchitectural state of the specific resource(s) and observes timing variations based on the encoded secret while the state transitions to MS_R. Finally, in the decode step, the timing variations observed are used as the basis to infer the secret.

The steps outlined above that cause *MS* transitions and secret information to be leaked can be performed by the adversary, the victim or both, depending on the type of attack. In the abstract model it is required that the state of at least one microarchitectural resource is shared between the victim and an adversary to enable information flow and consequently communicate the secret. The microarchitectural resources that are shared between the victim and the adversary are specific to the implementation and the threat model (see Section 2.3 for details).

Prior works define attack steps differently which leads to fewer/more steps. For example, Xiong et al. [118] defines three attack steps while Hu et al. [82] use six attack steps. In contrast, our attack model include five steps where each step consists of action(s) performed by adversary and/or victim on microarchitectural resource(s) *MS* which leads to a new *MS*. Note that the difference between the *interact* and *transmit* step is that the former accesses and encodes the secret on *MS* while the latter re-encodes the secret on shared *MS*.

We exemplify the abstract model by describing the steps in the well-known flush+reload [188] attack. This attack uses a single shared microarchitectural resource, a shared cache (*SC*). The goal of the attack is to infer the secret which is revealed through the victim's cache accesses because of data-dependent control flow. A prerequisite for the attack is that the cache lines of interest are mapped to a shared page that is accessible by the adversary as well as the victim. During the *setup* step, the adversary uses the *clflush* instruction to evict the lines belonging to the shared pages from the cache. The fact that the pages are shared allows the adversary to evict data that is accessed by a victim. The state of the cache after this step is $MS_P\{R_{SC}[[target]{=}null]\}$. During the *interact* step the victim executes and interacts with the secret which is encoded in the *SC* state by the presence/absence of the specific target cache line(s). The cache state changes to $MS_E\{R_{SC}[[target]{=}A]\}$. Since the cache is shared the adversary can detect the state change that has occurred as a result of the interaction. The adversary, during the *receive* step, accesses the cache line(s) (target) and measures the time. Through timing the adversary deduces which line(s) the victim has inserted and thereby infers the secret.

2.2. Root causes

We define the root causes of an attack in the context of the abstract model and exemplify with an actual attack.

2.2.1. Determinism. We define determinism as the characteristic of a microarchitectural optimization whereby microarchitectural resource(s) used by an optimization, under the same pre-conditions, is/are triggered in the same manner and cause a predictable microarchitectural state transition and timing variation. In other words, determinism causes an expected *MS* transition and timing variation upon an action by the adversary and/or the victim. In the abstract model of the attack, determinism enables the adversary to control *MS* transitions from MS_I through to MS_R across multiple steps.

2.2.2. Sharing. We define sharing as the characteristic of a microarchitectural optimization whereby the state of the microarchitectural resource(s) used by an optimization is/are shared between a victim and an adversary. In the abstract model sharing allows for the creation of a side-channel between the victim and the adversary's protection domain through *MS*.

2.2.3. Access violation. We define access violation as the characteristic of a microarchitectural optimization whereby one/many microarchitectural resource(s) permit(s) access of secret data which is outside the protection domain of the program on the microarchitectural level. This consequently enables information to flow outside the intended protection domain and occurs either in the *interact* or the *receive* step of the attack which causes secret information to be encoded into *MS*.

2.2.4. Information flow. We define information flow as the characteristic of a microarchitectural optimization to exchange information through the state of one or many microarchitectural resource(s). Information flow enables the adversary to infer the secret by observing the state change of microarchitectural resource(s).

The flush+reload attack, discussed earlier, exploits determinism, sharing and information flow in each of the steps of the attack. Determinism guarantees that the three state transitions occur in the attack. Firstly, the eviction of the target line(s) from the cache in *setup*, followed by insertion of a cache line in *interact*. Finally, timing differences are observed based on the presence and/or absence of specific cache lines in *receive*. Likewise, information flow and sharing guarantee that the secret is encoded and communicated, through *MS* of the shared *SC*, from the victim to the adversary, across the different steps. Access violation is not exploited in this attack since the *interact* step, executed by the victim, does not lead to an access outside its own protection domain, i.e., there is not an access violation on the microarchitectural level. In general, attacks can exploit a subset or all the root causes as we will show in Section 2.4 and 3.

2.3. Threat Model

We consider four types of threat models in our classification. Across the different threat models, the secret, that the adversary attempts to steal, resides in a different protection domain from that of the adversary.

An adversary can execute on a separate core from the victim, referred to as *CrossCore*; be time-multiplexed on the same core as the victim process, referred to as *SameThread*; run on distinct SMT threads executing on the same core, referred to as *SMT* or run in isolation,

referred to as *Solo*. In the *Solo* threat model the adversary only needs to have a pointer to the location of the victims data (kernel memory). The threat model determines which set of microarchitectural resources are shared or private in the attack setting on a given machine. A *CrossCore* threat model leads to a scenario where fewer microarchitectural resources are shared. In contrast, assuming the *SameThread* or the *SMT* threat model leads to potentially more microarchitectural resources being shared between victim and adversary, leading to a broader attack surface.

Another dimension of the threat model is based on whether the adversary or the victim performs the different actions in an attack. In a typical attack the adversary performs one or more steps. However, it has been shown that an adversary can manipulate the victim to perform some of the required actions through the use of specific gadgets. This is especially useful in scenarios where the adversary does not have access to a shared microarchitectural resource state to facilitate the *MS* transitions. This strategy increase the scope of possible attacks even in cases where the threat models limit the attack surface.

One example is the Spectre v2 attack [100] which requires training the Branch Target Buffer (*BTB*) as part of the *setup* step. Without gadgets such attacks would only be possible with the *SMT*/*SameThread* threat models since the *BTB* is not shared between cores. However, when the victim can be manipulated to perform the training, a *CrossCore* threat model can be used. The manipulation from the adversary can be performed by calling a function in the victims code with a controlled input, i.e., the action of triggering a gadget. The gadget can be constructed using Return Oriented Programming (ROP) [155] where code snippets ending with a return instruction are used by changing the return address and thereby chaining the different snippets together. However, these attack scenarios depend on the availability of gadgets and/or vulnerabilities, such as buffer overflows, and most have not been demonstrated outside of specific environments [35].

For some optimizations and attack scenarios the side-channel can be noisy and/or obscured by other optimizations, thereby making it difficult to decode the secrets based on *MS* transition and the consequent timing variations. Amplification gadget(s) can be used by the adversary to enhance the timing differences and ease the decoding of the secret. A simple example is on the cache side-channel, where prefetching can obscure the secret-related accesses. This can be circumvented by using a linked list [168] or by spreading accesses across pages [100] since most prefetchers only target linear or strided access patterns and do not prefetch across page boundaries.

2.4. Case Study

Next, we will describe an actual attack, Spectre v1, using the abstract model, the root causes we have identified and the threat model, as a framework.

2.4.1. Spectre v1. Spectre v1 [100] leverages three different optimizations: branch prediction, speculative execution and shared cache. The example code for the attack is

```
if(x < array1_size){y = array2[array1[x]*4096;}
```

Figure 2. Example code for Spectre v1 attack [100].

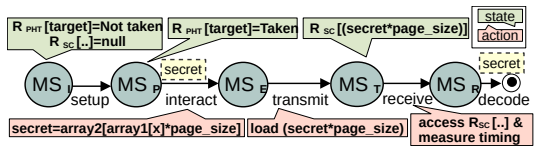


Figure 3. *MS* transitions and actions in Spectre v1.

shown in Figure 2, where x is controlled by the adversary and is used to represent the address delta between the base address of *array1* and the secret's location. The attack involves speculatively executing the if-clause code block, by training the branch predictor to predict taken. When the taken code block is speculatively executed the adversary can cause speculative access to *array2* indexed using the adversary controlled x. Even when the speculatively executed instructions are eventually rolled back this still leaves a trace in the cache, as a consequence of the access to *array2*, which is then used to infer the secret.

Figure 3 show an overview of the attack steps and MS transitions. The *setup* step consists of (mis)training the branch predictor Pattern History Table (*PHT*) by adversary/victim to change the prediction for the targeted conditional branch from $MS_I\{R_{PHT}[target]=Not\ taken\}$ to $MS_P\{R_{PHT}[target]=Taken\}$. The necessary (mis)training can either be performed by the adversary, restricting the threat model to SMT/SameThread, or be performed by the victim. To make the victim perform the (mis)training of the specific conditional branch a gadget must be located in the victim binary containing instructions which execute and train the target *PHT* entry to mispredict on the conditional branch. Setup of the *SC*, as in flush+reload, is also needed since it will be used later in the *transmit* step.

In the *interact* step the victim executes speculatively and accesses the secret because of the (mis)prediction. Speculative execution allows the CPU to temporarily violate program semantics by transiently execute code that accesses the secret and leave a trace in the *MS*. In the *transmit* step the secret is re-encoded in the state of another microarchitectural resource through a secret dependent access to the *SC* (*load (secret*page_size)*). This access causes the *SC* to transition to $MS_T\{R_{SC}[(secret*page\_size)]\}$. The *MS* transition occurs before the processor detects that the speculative execution was erroneous and rolls back the register state, leaving a trace in the state of the *SC*. In the *receive* step the adversary accesses the cache, measures the time and infers the secret based on timing variations for cache lines.

The root causes which enable this attack are determinism, sharing, access violation and information flow. Determinism guarantees that, the adversary can cause a *BTB* state update in an intended entry in the *setup* step, which is then used in the *interact* step. In addition, determinism ensures that the *SC* becomes primed, as a result of the flush, in *setup* and that the secret-dependent loads will result in timing variations corresponding to the lines presence/absence observed in the *receive* step. Access violation enables access to the secret which resides in a different protection domain, through speculative execution. Sharing is exploited in both the *BTB* and the *SC* during the *setup* step and in the cache during the *receive* step. Information flow is allowed through each of the state of the shared resources, across the different attack steps.

3. Systematization of Attacks

In this section we use the abstract model, the root causes and the threat model as a framework to systematize attacks on a broad set of microarchitectural optimizations. We describe a typical attack on each optimization and analyze the necessary root causes exploited in the different steps of the attack. We group attacks wherever possible and also discuss dissimilarities between the attacks on the same microarchitecture optimization. Note that the goal of this analysis is not to exhaustively discuss all the attacks proposed in literature. Rather, through the discussion, our aim is to highlight commonalities and differences across different attacks that target a microarchitectural optimization by discussing the following questions:

1) Which microarchitectural resource(s) are exploited in an attack?
2) Which root cause(s) are necessary to enable the attack and in which attack step(s)?
3) Under which threat model(s) is/are the attacks possible?

To simplify the discussion we categorize the microarchitectural optimizations into two broad groups – non-transient and transient optimizations – and analyze attacks on each of them. The systematization of attacks is presented in Table 1. Finally, we discuss microarchitectural optimizations without any published attacks that are prone to leaks.

3.1. Non-transient attacks

3.1.1. Cache. The last-level cache is typically shared among cores to improve cache utilization and to reduce costly off-chip accesses. The *SC* is vulnerable to side-channel attacks and is an attractive attack surface because of the channel characteristics (i.e., low noise and high attack bandwidth [160]). There exist three high-level attack categories: i) reuse-based [68], [72], [89], [188] where data is shared between adversary and victim allowing both to access it, ii) conflict-based [88], [93], [113], [132], [133], [136], where an adversary creates conflicts to evict target lines belonging to the victim, and iii) observation-based [69], [159], i.e., brute-force conflicts. In observation-based attacks the conflicts and observed behaviour relates to any cache line and is not restricted to a selected target, as in conflict-based attacks. We note that the same categories are common across attacks on other shared resources and have implications for the defense strategies (see Section 4).

Prime+probe [136], a typical conflict-based attack [88], [93], [113], [132], [133], [136], is used in the absence of data sharing between an adversary and a victim. The threat model is *CrossCore*, since the *SC* is shared between cores. In the *setup* step, the adversary first finds the set of cache lines (eviction-set) which will create conflicts in the targeted index shared with cache lines belonging to the victim and evict them from the cache. The state changes to $MS_P\{R_{SC}[[index]{=}A_{adversary}]\}$. Next, in the *interact* step, the victim accesses the secret during execution which is in turn encoded in the *SC* state through the presence/absence of the specific target cache line(s). The state changes to $MS_E\{R_{SC}[[index]{=}B_{victim}]\}$. In the *receive* step the adversary accesses the target cache line(s) and based on the timing variations infers the secret.

The root causes determinism, sharing and information flow enable the attack, as shown in Table 1. Determinism provides different guarantees in the three steps of the attack. First, in the *setup* step it ensures that conflicts can be created at a specific index in the cache which causes an eviction of the targeted cache line. Second, in the *interact* step it ensures the secret is encoded through the *MS* of the *SC*. Third, it also causes timing variation corresponding to the presence/absence of the cache line. Sharing and information flow ensures that the secret is encoded and communicated, through *MS* of the *SC*, from the victim to the adversary, across the different steps.

Observation-based attacks [69], [159] work by observing set conflicts in all the sets in the cache instead of actively causing them in a few sets like prime+probe. These attacks leverage the observation that the same conflict patterns will re-occur because the cache behavior of a program is deterministic. An alternative *setup* step is used, without relying on using *clflush* or a specific eviction-set, which involves accessing a large buffer [123], [151], [159] to flush all cache lines in the *SC* and creating conflicts across all the sets in the cache. This approach, however, has the drawback of a lower bandwidth since filling the *SC* is time consuming. These observation-based attacks are challenging to defend against since they rely on determinism and the shared *MS* of the *SC* in the *interact* and *receive* steps. This limits the defenses which can be used to avoid the attack (for details see Section 4.1.1).

A number of attacks show that other state can also be used for attacks, such as the state of replacement metadata [98], way prediction [113], interconnect [134], [178], cache banks [189], translation lookaside buffer (*TLB*) [3], [61] or memory management unit (*MMU*) [172]. Using other microarchitecture resource state (related to *SC*) enables to circumvent defenses on the *SC*. Van Schaik et al. [172] propose a prime+probe-like attack where an eviction-set is found and used in the *MMU* instead of the *SC* to overcome defenses targeting conflict-based attacks in the *SC*. Wan et al. [178] show an attack using temporal contention on the mesh interconnect, while Paccagnella et al. [134] show an attack on the ring interconnect. These attacks exploit determinism, sharing and information flow as previously discussed in the *SC* attacks. However, the sharing is of other microarchitectural resources as the *MMU* or the mesh interconnect, instead of the *SC*. We do not exhaustively cover all *SC* related attacks since our goal is to discuss a representative set to demonstrate the applicability of our framework.

3.1.2. Prefetching. Prefetchers predict addresses that will be used by a program and proactively fetches them to help hide memory access latency. The attacks exploiting prefetching can be broadly grouped into two fundamentally different categories, 1) SW-based and 2) HW-based. Attacks that belong to the latter category exploit the availability of a HW prefetcher (microarchitectural resource) and specifically utilize the *MS* of the prefetch tables and/or the *SC* as the side channel while attacks that belong to the former category exploits different prefetch instructions directly which exhibits different timing depending on the

Microarchitectural optimization	Attack(s)	Resource(s)	Attack steps				Threat model
			A_S	A_I	A_T	A_R	
Shared cache (SC)	flush+flush [68], flush+reload [72], [89], [188]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	prime+probe [88], [93], [132], [133], [136]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	observation [159], C5 [123]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	collide+probe [113]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2
	load+reload [113]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	3
	xlate+probe/abort [172]	R_{SC}, R_{MMU}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	TLBleed [3], [61]	R_{SC}, R_{TLB}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2,3
	CacheBleed [189]	$R_{SC^T_{bank}}$	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	3
	MemJam [126]	R_{SC}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	LoR [134]	R_{ring^T}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	MeshUp [178] MeshAround [45]	R_{mesh^T}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	CacheTiming [23], [169]	R_{SC}	- [A]	D/I [V*]	-	D/I [A]	1/-[2]
Prefetching (P)	Prefetch SCAs [1], [67]	R_P,R_{SC},R_{TLB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	prefetch+reload, prefetch+prefetch [74]	R_P,R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	LeakingControlFlow [40]	R_P,R_{SC},R_{TLB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	DMP [146], [175]	R_P,R_{SC}	D/S/I [V*/A]	D/S/A/I [V*/A]	-	D/S/I [A]	2
	Unveiling [158]	R_P,R_{SC}	D/S/I [A]	D/S/I [V]	-	D/I [A]	1,2,3
	FetchingTale [44]	R_P,R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2,3
Branch prediction (BP)	JumpOverASLR [49]	R_{BTB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	PredictingKeys [4]–[6], [49]	R_{BTB}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2,3
	BranchScope [51]	R_{PHT}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	BranchShadowing [109]	R_{PHT},R_{LRB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
Computational simplification (CS)	Subnormal FP [16], [102]	R_{FPU}	(D/S/I)[1] [A]	D (S/I)[1] [V]	-	(D/S/I)[1] [A]	-
	Early termination [63]	R_{MUL}	- [A]	D [V*]	-	- [A]	2,3
Transient attacks: Speculation-based	Spectre v1 [100], [167], v1.1 [99]	R_{SC},R_{PHT},R_{BHB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	Spectre v2 [20], [39], [100], [184]	R_{SC},R_{BTB},R_{BHB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	Spectre v4 [124], LVI [171]	R_{SC},R_{STL}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2
	Spectre v5 (ret2spec) [104], [121]	R_{SC},R_{RSB},R_{BTB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	BranchSpec [90]	R_{PHT}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3
	NetSpectre [151]	R_{PHT},R_{AVX2}/R_{SC}	D/S/I [V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	-[2]
	CROSSTALK [141]	R_{SC},R_{LFB},$R_{staging\_buf.}$	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1
	SMoTherSpectre [26]	R_{SC},R_{PHT},R_{ports^T}	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	3
	SpectreRewind [52]	R_{SC},R_{BTB},R_{ports^T}	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3
	Speculative interference [22]	R_{SC},R_{BTB},R_{MSHR},R_{RS},R_{EU^T}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2,3
	ROB cont. [7]	R_{PHT},R_{ROB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2,3
Transient attacks: Exception-based	Meltdown [114], [167]	R_{SC},R_{BTB} [PF-US]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	Foreshadow [170], [182]	R_{SC},R_{TLB} [PF-P]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	Spectre1.2 [99]	R_{SC},R_{TLB} [PF-RW]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	LazyFP [162]	R_{SC},R_{FPU},R_{SIMD} [#NM]	D/S/I [V]	D/S/A/I [A]	D/I [A]	D/I [A]	2,3
	Fallout [125]	R_{SC},R_{SB}	D/S/I [A]	D/S/A/I [A&V]	D/I [A]	D/I [A]	2,3
	RIDL [173] ZombieLoad [149]	R_{SC},R_{LFB}	D/S/I [A]	D/S/A/I [A&V]	D/I [A]	D/I [A]	2,3
	LVI [171]	R_{SC},R_{FPU},R_{SB},R_{LFB} [PF]	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3,4

TABLE 1. ATTACK SYSTEMATIZATION. ROOT CAUSES: DETERMINISM (D), SHARING (S), ACCESS VIOLATION (A), INFORMATION FLOW (I). ATTACK STEPS: SETUP (A_S), INTERACT (A_I), TRANSMIT (A_T), RECEIVE (A_R), PERFORMED BY ADVERSARY [A] OR VICTIM [V]. GADGET*. THREAT MODELS: 1) CROSSCORE, 2) SAMETHREAD, 3) SMT, 4) SOLO.[1]IN SW.[2]REMOTE.^{T}TEMPORAL RESOURCE.

state of the *TLB* and/or the *SC*. The attacks exploiting SW-based and HW-based prefetching mechanisms can be further subdivided into two categories, 1A and 2A) Attacks which exploit the lack of permissions check and 1B and 2B) Attacks which exploit a secret-dependent prefetching pattern.

One typical attack from category 1A, a SW-based attack that exploits the lack of permissions check, is the address-translation attack by Gruss et al. [67] where the goal of the full attack is to translate between virtual and physical addresses from unprivileged user-space and overcome the protection provided by user-space and kernel-space Address Space Layout Randomization (ASLR). We focus on the first phase of the attack where the adversary searches through possible addresses and tests if two virtual addresses, a and a', map to the same physical address by performing the attack. Here, address a' can be a kernel address or a non-mapped address and not be directly accessible to the adversary. In the *setup* step the adversary flushes the candidate collision address, a. The state changes to $MS_P\{R_{SC}[index_a]=empty\}$. In the *interact* step, the adversary prefetches the address a', and performs an access violation. The access violation is due to speculative dereferencing of kernel-space registers from user-space [152], and not because of the prefetch instruction as suggested by Gruss et al. [67]. The state changes to $MS_E\{R_{SC}[index_a]=a'\}$. In the *receive* step the adversary accesses address a and based on the timing variations infers if there is a match between a and a'.

The root causes exploited by the attack are determinism, sharing, access violation and information flow, as shown in Table 1. Determinism enables all the three steps of the attack. First, in *setup* step, it causes the cache line corresponding to a to be evicted. Second, in the *interact* step, it causes the prefetch of a' to be encoded through the *MS* of the *SC*. Third, it makes timing variation to be observed corresponding to the presence/absence of the cache line. Access violation enables the attack by permitting the adversary to prefetch inaccessible address(es). Sharing and information flow guarantees that the secret is encoded and communicated, through *MS* of the *SC*.

In the second category in SW-based attacks, 1B, the attacks [74] use a SW-controlled prefetch instruction (*PREFETCHW*), in the *setup* and the *receive*, to reveal cryptographic keys through the data-dependent access pattern of an application.

The attacks in category 2A, HW-based without permissions check, use HW prefetchers to prefetch addresses outside of a sand-box [175] or in kernel-space [40]. The prefetcher is trained in the *setup* step, in order to issue a prefetch to the target address in the *interact* step. Chen et al. [40] show that an adversary trained prefetcher can prefetch kernel addresses. In [175] a data memory-dependent prefetcher (DMP) is trained to perform out-of-bounds reads on pointers, since the prefetcher is allowed to use memory content to prefetch irregular address patterns. The root causes exploited by the attacks are determinism, sharing, access violation and information flow. In both the attacks the state of the *SC* is used to encode the accesses in the *interact* step and to measure the timing difference in the *receive* step. There is no need of an additional *transmit* step to encode the secret in the *SC*, since the prefetcher directly interacts with the *SC* in the *interact* step. Access violation is exploited in the *interact* step since the prefetch is issued without permission checks.

Lastly, the attacks [40], [158] in category 2B, leak secrets through secret data-dependent prefetching pattern(s). Shin et al. [158] show how data-dependent prefetch activity can be used to leak secret keys through *MS* in the *SC*. Similar to the previously discussed attacks using SW-based prefetching, which also leak secrets through data-dependent prefetch access patterns, the root causes are determinism, sharing and information flow.

3.1.3. Branch prediction. Branch predictors record history of branch outcomes in order to predict the direction of control flow after a branch instruction, to improve instruction flow. There are two high-level strategies for attacks using the branch predictor, reuse-based [51], [100], [109] where entries set by one process may influence the other and conflict-based [4]–[6], [49] where contention is used to evict the entry inserted by the other process.

A typical conflict-based attack is JumpOverASLR [49] where the goal of the adversary is to determine the position of a code block in the address space of a victim. Knowing the position of a code block can help break the protection provided by ASLR since the randomization is based on an offset. The attack is launched multiple times using different index values, searching for a collision in the *BTB*. A collision in the *BTB* can be used to infer the address used by the victim and the offset used by ASLR. In the *setup* step, the adversary inserts an entry in the *BTB* which might create a collision with the entry later inserted by the victim code. This changes the *MS* to primed $MS_P\{R_{BTB}[index_I]{=}addr.\ A\}$. Next, in the *interact* step the victim executes and inserts a different target address at the same *BTB* position, creating a collision. The state transitions to $MS_E\{R_{BTB}[index_I]{=}addr.\ B\}$. In the *receive* step the adversary executes code which will trigger the *BTB* entry at $index_I$ and measures the execution time. If the *BTB* entry was changed by the victim it would result in a longer execution time since the target address is incorrect (B instead of A).

The root causes which enable this attack are determinism, sharing and information flow, as shown in Table 1. Determinism guarantees that, the adversary can cause a *BTB* state update in an intended entry, induce a conflict on the same entry when the victim executes and measure timing variations due to the conflict. Likewise, information flow and sharing guarantees that the secret is encoded and communicated, through *MS* of the *BTB*, from the victim to the adversary, across the different steps. Note that this attack is restricted to the *SameThread* threat model (although it can be applied in *SMT*) and does not extend to *CrossCore* because *BTB* state is not shared across cores.

There also exist conflict-based attacks which exploit that the branch predictions can reveal data-dependent control flow [4]–[6]. For example in [5] secrets are inferred based on the predictions made in the *BTB*.

The reuse-based attack use the *PHT* instead of the *BTB* [51]. In BranchScope [51], the branch predictor is manipulated into using only the directional branch predictor, *PHT*, where the directional prediction inserted by the adversary is changed by the victim which reveal the direction of conditional branches. The attack can also be used against SGX enclaves, since the *PHT* is shared between processes executing in SGX and outside. The same root causes as in JumpOverASLR enable the attack.

3.1.4. Computational simplification. Computational simplification comprises techniques which eliminate or simplify instruction execution. One example is the zero-skip multiplier and the same principle can be applied on different instruction types as square root, AND/OR and to different pipeline stages. Attacks on this type of optimizations have been studied [16], [43], [63]. In [16] an attack is described using subnormals in a floating-point division unit, to create visible timing differences. Großschädl et al. [63] describe an attack using early-termination of multiplication where the multiplication is terminated when all remaining digits are zero, creating observable timing differences. The goal of the attack is to leak secret keys from cryptographic SW such as RSA. There are two prerequisites of the attack, firstly, that the adversary is able to control the plaintext which will be encrypted and secondly, that the timing can be observed on a side-channel. In the *setup* the adversary calls the cryptographic function on the victim with a plaintext. In the *interact* step the victim encrypts the plaintext and will experience different timings depending on the values of the key. Großschädl et al. does not describe which side-channel could be used in order to allow the adversary to observe the timing difference. We note that either the *MS* of the *SC* or execution unit contention could be used. A gadget is likely needed at the victim for re-encoding of the secret to the side-channel.

The root cause exploited is determinism which enables the data-dependent behaviour of the early-termination optimization and the timing variability. In addition, the side-channel, which enables the adversary to observe the timing differences, exploits sharing and information flow.

3.2. Transient attacks

We finally discuss transient execution attacks which exploit speculative out-of-order (OoO) execution to execute code transiently (i.e. executed but never committed). We use the categorization provided in related works [35], which divide the transient attacks broadly into two groups, speculation-based [7], [22], [26], [39], [52], [90], [100], [121], [124], [141], [151] and exception-based [99], [114], [125], [162], [167], [170], [171], [173] attacks.

3.2.1. Speculation-based Attacks. The attacks that fall in this category exploit transient execution, due to branch prediction and/or address/value speculation, to access the secret. An overview of the attacks is shown in Table 1. Spectre v2 [100] is a typical speculation-based attack. The attack exploits an indirect branch to execute a gadget which interacts with the secret in the victims protection domain, leaving a trace in the *MS*. The prerequisites are an indirect branch that can be (mis)trained and a known gadget in the victim's binary that can be manipulated to interact with the secret. The threat models are *SameThread* and *SMT*, since *BTB* is a resource private to a core. However, *CrossCore* can be used if a gadget is used to make the victim perform the (mis)training. In the *setup* step the adversary/victim (mis)trains the *BTB* to insert a new entry containing the address of the gadget for the indirect branch. The state changes to $MS_P\{R_{BTB}[[index_{target}]=addr.\ A_{gadget}]\}$. The root causes in the *setup* step are determinism, sharing and information flow. Determinism guarantees that the adversary can cause a *BTB* state update in an intended entry, while sharing and information flow enables the state change caused in the *BTB* to be observed by the victim. Note that setup of the *SC* is also performed, i.e. *clflush*, since it will be used later in the *transmit* step.

Next, in the *interact* step the victim executes the gadget speculatively, accesses the secret and changes the *MS*. The root causes are determinism, access violation and information flow. Determinism guarantees that the (mis)trained *BTB* entry is used. Access violation enables access to the secret through execution of the gadget which results in temporary violation of program semantics, i.e., instructions that access the secret are executed and are later squashed. In the *transmit* step the secret is re-encoded in the state of the *SC*, by issuing load(s) to the target address(es) by the victim. The root causes exploited are determinism, sharing and information flow since the *SC* contains the cache line(s) and the *MS* of *SC* is shared between the adversary and the victim. Finally, in the *receive* step the adversary accesses the target cache line(s) and based on the timing variations infers the secret. The root causes are the same as in the previous step, with the difference that determinism ensures observable timing variations based on the state of the *SC*, i.e., the presence/absence of the target cache line(s).

In contrast to Spectre v2, which uses the *BTB*, other microarchitectural resources have been used to manipulate the control flow, e.g., *PHT* [100] or the Return Stack Buffer (*RSB*) [104], [121]. In addition, address speculation can be targeted for manipulating Store-To-Load (*STL*) forwarding that happens in the Load Store Queue (*LSQ*) [124]. Many of these different attack variants still use *SC* as the side-channel for transmission.

Next, we discuss attacks which are more restrictive since other microarchitectural resource(s) (not *SC*) is/are used for the transmission of the secret. One example is BranchSpec [90] where the *PHT* is used in the *transmit* and *receive* step. The root causes exploited by the attack are the same as in Spectre v2, while the threat model is more restrictive since the *PHT*, used as the side-channel, is not shared between cores. Another attack, SMoTherSpectre [26], uses port contention to encode the secret and transmit to a co-running SMT thread. Likewise, temporal contention in the floating-point division unit is exploited in SpectreRewind [52]. Like SpectreRewind, the attack proposed by Behnia et al. [22] shows that the secret can be encoded by affecting the timing and order of older instructions, which are issued before the secret dependent instruction(s) in program order. This is in contrast to prior works [95], [110], [186] that focused on studying the secret-dependent effect on younger instructions, issued after the secret dependent instruction(s). In the attack, the non-speculative instructions timing is affected either through the miss status handling register (*MSHR*) or execution unit contention. As an example, let's consider the attack using the *MSHR* described by Behnia et al. [22]. In the *setup* step the adversary evicts a number of cache lines Y. In the *interact* step the a gadget, depending on the secret value, either issues independent loads to the cache lines Y (filling up the *MSHR* entries) or issues loads to the same cache line (using one entry in the *MSHR*). When the target victim load occurs it will experience different timing depending on the *MS* of the *MSHR*.

3.2.2. Exception-based Attacks. The attacks that fall in this category exploit transient execution, due to delayed exception handling to access the secret. Meltdown [114] is a typical attack from this category where the adversary exploits transient execution due to delayed exception handling, to read arbitrary kernel memory. In the *setup* step the adversary causes an exception by accessing a kernel address that resides in a kernel memory page without suitable permissions causing a page fault, e.g., PF-US. Because of the deferred exception handling the execution continues transiently. In the *interact* step the adversary executes code that uses the loaded value from the faulting kernel address. By suppressing the exception can the transient execution continue [114]. In the *transmit* step the secret is encoded in the *MS* of the *SC*, through a load to the data buffer, in order for the adversary to retain the information after the transient execution is rolled back after exception handling. In the *receive* step the secret is inferred from the state of the *SC*, through the presence/absence of cache line(s). Note that all the steps of the attack are performed by the adversary.

The root causes enabling the attack are determinism, sharing, access violation and information flow (Table 1). Determinism ensures transient execution due to delayed exception handling in the *setup* step and that the secret is encoded in the state of the *SC* in the *interact* step. Sharing enables the access from the adversary to the victim kernel address in the *interact* step. Information flow enables the transiently accessed secrets to be communicated to the non-transient execution, through the *MS* of the *SC*. Access violation enables the adversary, in the *interact* step, to access kernel data which it does not have the right privileges to access.

Attacks have shown that different types of exceptions, i.e., page fault (PF), can be used in the *setup* step. For instance, Foreshadow uses PF-P [170], [182], Spectre v1.2 [99] uses PF-RW while LazyFP [162] uses #NM (device not available). Other types of page fault exceptions can also be used as shown by Canella et al. [35]. Furthermore, in addition to reading from kernel memory, attacks have also exploited delayed exception handling to leak data across addresses spaces, virtual machines and even from secure enclaves [170], [182].

Another group of attacks, referred to as the microarchitectural data sampling (MDS) attacks, exploit the state of internal buffers in the CPU, as the Line Fill Buffers (*LFBs*) [149], [173] or the Store Buffer (*SB*) [125], in conjunction with delayed exception handling. Specifically, the attacks leverage the observation that values from these buffers can be leaked as a consequence of accesses that trigger an exception. In the ZombieLoad v1 attack [149] the adversary uses the kernel virtual address (k) corresponding to the user-space address of the victim (u) where the secret resides. Both virtual addresses k and u map to the same physical address s. In the *setup* step, the adversary monitors the victim by performing repeated flush+reload attacks on the address corresponding to the instruction just before the loading of the secret. This enables the attacker to synchronize with the victim and determine when it can start accessing the state of the buffers to retrieve the secret. In addition, the contents of the data buffer are also flushed from the *SC*. Next, in the *interact* step, the victim performs the load of the secret key, *load u*. This load operation will cause the secret to be inserted in the *LFB* (on a cache miss). The adversary performs a faulting load i.e. the ZombieLoad, to the kernel address of the secret (*load k*), which causes the adversary to retrieve the secret from the *LFB*. In the *transmit* step the adversary uses the secret as an index to a data buffer to encode the secret into the *MS* of the *SC*, before the transient execution is rolled back. In the *receive* step the adversary accesses the data buffer entries to infer the secret based on the timing variations arising due to *SC* state. In contrast to Meltdown, victim's accesses cause the secret to be inserted in the internal buffers which are then leaked by loads that trigger exceptions. All four root causes enable this attack. Determinism enables all the steps of the attack while sharing the *MS* of the *LFB* allows for information flow between the victim and the adversary. Access violation occurs during transient execution when the adversary is allowed to read stale data from the *LFB*. Unlike the MDS attacks discussed previously, Load Value Injection (LVI) [171] uses the different types of exception-based vulnerabilities to inject data/code and control victim's execution by controlling the values in the internal CPU buffers. This attack exploits the same root causes as the MDS attacks discussed previously.

3.3. Vulnerable optimizations

Several microarchitectural optimizations available in literature have not been reviewed in detail with an emphasis on security. We discuss possible attacks using our framework for one such representative optimization, value prediction. We also analysed vulnerabilities in a few other optimizations, including the ones explored by Vicarte et al. [146], but omit them due to page constraints.

Value prediction: This is a speculative optimization that aims to increase instruction-level parallelism (ILP) and hide memory access latency by predicting values for load misses and consequently breaking instruction dependencies [111], [112], [135], [154], [156]. Accurate predictions can improve ILP by increasing the overlap between memory access(es) and useful computation(s) while mispredictions lead to pipeline squashes and re-execution of instruction(s). In a nutshell, value prediction is implemented using table-based structures and samples history to enable prediction.

Both reuse-based and conflict-based attacks are possible, similar to the attacks described for the branch predictor. One possible attack that exploits the reuse behavior is to let the victim train the predictor leading to the secret being encoded in the predictor state. The adversary can then trigger a prediction and use this to infer the secret. Another possible attack strategy is to let the adversary (mis)train the predictor in order to induce the victim to access a secret (cause an access violation) using a gadget, akin to injection attacks. This can then be leaked to the adversary through a side-channel, such as the *SC*. Finally, conflict-based attacks could also be mounted by using secret dependent predictor use behavior and monitoring the state of the prediction tables to infer secrets. The root causes exploited by the aforementioned potential attacks are determinism, sharing and information flow. Determinism permits the *MS* of the prediction table to be accessed and manipulated depending on the attack requirements. Sharing permits the *MS* of the prediction tables to be accessible to both the adversary and the victim and enables information flow between adversary and victim. Access violation could also be exploited if the predictions cause the adversary and/or the victim to access data from outside the intended protection domain.

In summary, we observe that the necessary conditions for most attacks are determinism, sharing and information flow and that there are few variations in the combinations of root causes. We also note that the goal of the attacks is either to i) exploit data-dependent implementations to leak encryption keys, or ii) circumvent privilege checks to typically read kernel data. All attacks in the later category exploit access violation.

4. Systematization of Defenses

We present a systematization of defenses against attacks targeting different microarchitectural optimizations. We specifically discuss optimizations for which several attacks and defenses exist in literature: cache, prefetching, branch prediction, computational simplification and transient execution attacks. For each of the defenses, we discuss which root cause(s) and the attack step(s) the defenses target. Eliminating any of the root causes, exploited in a specific attack, in any of the attack steps can provide protection. Table 1 show the different root causes for each attack, which can be targeted by a defence.

We categorize defenses into groups, wherever possible, in case there are similarities. In addition, we also describe the protection level offered against the discussed attacks using the optimization and the threat model targeted by the defense. Our goal is not to exhaustively cover defenses against all possible attacks targeting a microarchitectural optimization. Rather, it is to explore broad defense strategies, in which attack step and root cause they can be applied and their limitations.

4.1. Defenses against Non-transient attacks

4.1.1. Cache. Several defenses have been proposed for securing the shared cache against side-channel attacks. We have classified the different defenses for the *SC* based on the root cause(s) and attack step(s) they target, into five

broad categories, see Table 2. Each row in the table shows which root causes are restricted by the defence, in which attacks step using which microarchitectural resource.

Disabling *clflush* [187], [188] only addresses reuse-based attacks which typically uses the instruction in the *setup* step. This affects the three root causes determinism, sharing and information flow primarily in the *setup* step.

The defenses in the next category, partitioning (part.), target sharing and information flow, in all the attack steps, by providing isolation between processes/threads in the *SC* state using partitioning [30], [73], [75], [96], [98], [115], [130], [131], [147], [153], [179], [180]. Partitioning can, in principle, provide full protection against reuse-based, conflict-based and observation-based attacks where victim and adversary share the *SC* state. However, the defenses provide different levels of protection depending on the level of isolation, i.e., whether all or only some of the data is partitioned. For example, MI6 and IRONHIDE [30], [131] statically partition both *SC* and DRAM and can thereby enable full protection of the *SC*, albeit at a comparatively higher performance cost. In contrast, STEALTHMEM [96] only provides partial protection for a limited number of cache lines per core, which are not allowed to be evicted. This will lead to higher performance but also a lower protection level, since the state of the unprotected cache lines are shared.

The next category, randomization (rand.), targets conflict-based attacks and is typically achieved by modifying the mapping of addresses to sets in the *SC* [117], [139], [140], [164], [166], [183]. Randomization target determinism and information flow. The strategy affects determinism by complicating the process of creating an eviction-set needed to evict a target cache line at a specific address. The change in mapping leads to limited information flow. This makes randomization effective especially against conflict-based attacks. However, in spite of defense, the *SC* is still shared and insertions by the adversary can result in evictions for the victim process and vice versa. This limits the effectiveness against observation-based attacks since the working-set size of an application can be observed and can be leveraged by attacks [55].

The category replacement-based defenses (repl.) – insertion and/or eviction – leverage randomization to provide protection [47], [92], [94], [116], [143], [180]. These proposals mainly target determinism and information flow through the *SC* state. Specifically, through randomising insertion and/or eviction, the *SC* do not react in the same way under the same preconditions. Information flow is limited by reducing/avoiding set conflicts, i.e., by preventing an adversary from evicting data inserted by the victim. These defenses offer protection but eventually leak information since determinism, sharing and information flow in the *SC* are not completely eliminated [163].

Lastly, constant-time programming paradigm [24], [25], [43] can be used to avoid data-dependent implementations which affects determinism and information flow. However, this is challenging to utilize in practice since it cannot be generically applied.

4.1.2. Prefetching. Existing defenses to protect against prefetch-based attacks can be categorized broadly into five groups, as shown in Table 3. Disabling the prefetcher [1], [40], [44], [158] impacts determinism, sharing, access violation and information flow, in all the attack steps. This strategy is equally applicable to attacks using HW- or SW-based prefetching. However, the performance cost can be high since prefetching can provide significant speedups.

The defense in the second group, by Gruss et al. [67], propose introducing privilege checks on prefetch instructions. This would cause a segmentation fault when there is an attempt to prefetch kernel data. This prevents access violation in the *interact* step and affect all SW- and HW-based attacks that exploit the lack of permission checks [1], [40], [67], [146], [175].

Another strategy is to provide stronger isolation between kernel and user-space to protect against attacks that leverage the lack of permission checks [65]. This approach has been adopted in both Linux [57] and Windows [91]. This would provide protection against SW-based attacks using the prefetch instruction [1], [67] and against HW-based attacks [40], [146], [175]. This approach restricts sharing and information flow through the page tables (and TLB), in the *interact* step of the attack.

The next group of defenses target attacks that exploit the state of HW-based prefetchers (prefetch tables). Specifically, these defenses replicate and flush prefetcher state at context switches [40], [44]. This ensures that state is no longer shared across context switches which restricts sharing and information flow. This only affects the HW-based attacks which rely on *MS* in the prefetcher. Furthermore, the threat model targeted is limited to *SameThread* since the technique does not affect concurrently executing SMT threads sharing prefetcher state.

Another strategy is to change the SW implementation to ensure that any prefetch activity is not dependent on any secret. This would affect the attacks relying on data-dependent execution paths and observing prefetch patterns [40], [74], [158]. This can be achieved using constant-time programming practices [24], [158], for example rewriting table based look-ups to be immune to prefetches [60], [158]. This strategy affects determinism and information flow and can theoretically protect against attacks that exploit prefetch patterns. However, it is challenging to implement this broadly in practice.

Defense	Res.	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable clflush [187], [188]	R_{SC}	D/S/I	-	-	-	1,2,3	◐
part.: [46], [73], [115], [180]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	◐
[96], [147], [153], [179]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	◐
part.: static [30], [98], [131]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	●
rand. [117], [139], [140], [166]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
[164], [183]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
repl. [47], [92], [94], [116], [143], [180]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
const. time [24], [25], [43]	-	-	D/I	-	-	1,2,3	●

TABLE 2. DEFENSES FOR ATTACKS USING THE *SC*. PROTECTION (P): FULL●/PARTIAL◐.

Defense	Res.	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable [1], [40], [44], [158]	$R_{Pref.}$	D/S/I	D/S/A/I	-	D/S/A/I	1,2,3	●
privilege checks [67]	$R_{Pref.}$	-	A	-	A	1,2,3	◐
kernel/user isol. [65]	R_{TLB}	-	S/A/I	-	-	2	◐
flush [40], [44]	$R_{Pref.}$	I	I	-	I	2	◐
replicate [40], [44]	$R_{Pref.}$	S/I	S/I	-	S/I	2	◐
const. time [25], [43], [60], [158]	-	-	D/I	-	-	1,2,3	◐
SC defenses	R_{SC}	D/S/I	-	-	D/S/I	1,2,3	◐

TABLE 3. DEFENSES FOR ATTACKS USING THE PREFETCHER.

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
if-conversion [41], [51]	-	D/S/I	D/S/I	-	D/S/I	2,3	●
enc.: [108], [192]	R_{BTB}/R_{PHT}	D/I	D/I	-	D/I	2,3	◐
[50], [62], [193]	R_{BTB}/R_{PHT}	D/I	D/I	-	D/I	2,3	◐
flush [177], [193]	$R_{BTB}/R_{PHT}/R_{BHB}$	I	I	-	I	2	●
part. [177], [193]	R_{BTB}/R_{PHT}	S/I	S/I	-	S/I	2	◐
rand. [81], [191]	R_{BTB}/R_{PHT}	D	D	-	D	2,3	◐

TABLE 4. DEFENSES FOR ATTACKS USING THE BRANCH PREDICTOR.

Lastly, it should be noted that some of the attacks rely on the shared state of the *SC* [40], [67], [74], [158], [175]. The defenses proposed for the *SC* could be used to defend against these attacks as well.

4.1.3. Branch prediction. The defenses against branch prediction based attacks can be grouped into five categories, see Table 4. The first group relies on SW-based techniques, as if-conversion, where the compiler restructures code to avoid conditional branches and use predication instead [41]. This restricts determinism, sharing and information flow since branching is avoided and is akin to disabling the direction prediction. However, the applicability to real-world code with complex control flow is limited [51]. Furthermore, highly predictable branches have been shown to perform poorly when if-converted [41].

The next category of defenses use randomization to thwart attacks. Specifically, encryption of the *BTB*/*BTB* have been proposed [50], [62], [108], [192] to prevent the adversary from easily manipulating the branch prediction logic. Encryption restricts determinism and information flow, in all the steps of the attack. Determinism is affected in the *setup* step because the target is encrypted which makes manipulating collisions difficult. Information flow is also hindered since a process can only access correct entries in the presence of a valid key. The limitation of these encryption-based solutions is that they cannot guarantee protection against brute-force approaches.

Defenses in the next category flush the state of the branch predictor i.e. *BTB*/*BTB*/*BHB*, on context switches [177], [193]. This would affect information flow in the context of the *SameThread* threat model. However, the performance overhead is usually high [177] and the effectiveness is restricted to the *SameThread* model.

Partitioning has been shown to thwart attacks on the branch predictor [51], [193]. Partitioning affects sharing and information flow, in all the steps of the attack, since the state of the *BTB*/*BTB* is isolated. HyBP [193] combines isolation and encryption, and selectively replicates parts of the predictor state, while using encryption for the larger tables. The focus of these proposals is to use partitioning/replication to avoid the high performance cost of flushing the entire branch prediction state upon a context switch. Replicating the entire prediction state among SMT threads, although a possibility, is prohibitively expensive.

The last category makes the state transition of the predictor probabilistic, by affecting the saturating counters, as proposed by Zhao et al. [191]. This defense restricts determinism in all the steps of the attack. However, the protection offered by the technique is limited since an adversary, through repeated measurements, can eventually infer the secret from the state of the branch predictor.

4.1.4. Computational simplification. Two broad strategies have been proposed for protecting against attacks using computational simplification, see Table 5. One strategy is to selectively disable the optimization for parts of the program which accesses sensitive information [43], [63]. Disabling restricts determinism in the *interact* step.

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable [43], [63]	R_{MUL}/R_{FPU}	-	D	-	-	2,3	●
const. time: [17], [43], [63]	R_{MUL}/R_{FPU}	-	D	-	-	2,3	◐
[16], [24], [25], [142]	R_{MUL}/R_{FPU}	-	D	-	-	2,3	◐

TABLE 5. DEFENSES FOR ATTACKS USING COMP. SIMPLIFICATION.

The other strategy is to change the implementation to avoid any data-dependent timing variations, even for computational simplification. This strategy targets determinism in the *interact* step. One way to achieve data-independent implementation is to use constant-time programming practices [16], [24], [25], [43]. In [16] a FP library (LibFTFP) is shown, providing a fixed-point data type with all library operations executing in constant time.

Lastly, in the case of the browser-based attack [16] the SW-construct which enables the sharing and information flow, can be disabled i.e. cross-origin SVG-filters [102].

4.2. Defenses against Transient Attacks

We describe defenses for transient execution attacks, where we first discuss defenses for speculation-based attacks, and then defenses for exception-based attacks.

4.2.1. Defenses against Speculation-based Attacks. The defenses against speculation-based attacks can be broadly grouped into four high-level categories based on the defense strategy: localized defenses, disabling defenses, restriction defenses and isolation defenses, see Table 6.

Localized defenses leverage the defenses available for individual optimizations/resources that interact with speculative execution, such as the BP and/or *SC*. Branch prediction can be targeted in the *setup* step to stop the adversary from being able to (mis)train the predictor, see Section 4.1.3. The defense is only applicable for the attacks which uses the specific predictor resource. Another approach is to target the side-channel that permits information flow across protection domain through transient execution. In most attacks the *SC* is used since it can offer the highest bandwidth, which makes the defenses in Section 4.1.1 applicable. However, studies have demonstrated that a large variety of microarchitectural resources can be

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
local: BPU	R_{BTB}/R_{PHT}	D/S/I	-	-	-	2,3	◐
SC	R_{SC}	D/S/I	-	-	D/S/I	1,2	◐
disable: IBPB [12], sb [18]	R_{BTB}/R_{PHT}	D/S/I	-	-	-	1,2,3	◐
SSBD [12], [13]	R_{STL}	D/S/I	-	-	-	1,2,3	◐
SMT [59], [76], [107]	-	D/S/I	D/S/I	-	D/S/I	3	●
restrict: IBRS,STIBP [12], [14]	R_{BTB}/R_{PHT}	D/S/I	-	-	-	1,2,3	◐
retpoline [15], [87]	R_{RSB}	I	-	-	-	1,2,3	◐
randpoline [31]	R_{RSB}	D	-	-	-	1,2,3	◐
lfence, serial. [12]	-	-	D/I	-	-	1,2,3	◐
fence: [105], [165], [174]	$R_{\mu OP\_Q}$	-	D/I	-	-	1,2,3	◐
[36], [129], [157]	-	-	D/I	-	-	1,2,3	◐
delay: [19], [53], [148], [181]	R_{ROB}, $R_{reg.}$	-	D/I/A	-	-	1,2,3	◐
[32], [190]	R_{ROB}, $R_{reg.}$	-	D/I/A	-	-	1,2,3	◐
[110], [119], [145]	R_{ROB}, $R_{reg.}$	-	-	D/I	-	1,2,3	◐
rollback [144]	R_{SC}	-	-	D/I	D/I	1,2	◐
isolate: [8], [9], [95], [186]	R_{buf}/R_{L0}	-	-	D/S/I	-	2	◐

TABLE 6. CLASSIFICATION OF DEFENSES FOR TRANSIENT ATTACKS.

used, such as execution units, ports, *PHT*, *MSHR* etc. This is a challenge because multiple resources may need to be protected, since they can all act as potential side-channels. By defending the resource which enables the easily accessible (high bandwidth and low noise) side-channels, the attack bandwidth can be reduced. In addition, limiting the threat model (for instance to *CrossCore*) can restrict the number of resources which can be used as side-channels.

The second category involves disabling the optimization. This approach has been proposed in specific scenarios where other defenses are not applicable. Support for disabling indirect branch predictions using a barrier, Indirect Branch Predictor Barrier (IBPB) [12], [14], has been adopted in commodity HW. Likewise, to thwart Spectre v4 attack, the STL mechanism can also be disabled using Speculative Store Bypass Disable (SSBD) microcode updates from Intel and AMD [12], [13]. These techniques affect the root causes determinism, sharing and information flow in the predictor, in all the steps where it is used. However, the performance cost is potentially high since the predictions are restricted. Another defensive measure is to limit the threat model by disabling SMT [59], [76], [107]. However, disabling SMT comes at a potentially high performance cost.

The third category is restriction-based defences. Here, the high-level idea is to restrict the speculative execution, selectively, to avoid the attacks. Speculation restriction can be performed in the different steps of the attack and using different mechanisms in HW and/or SW. One HW mechanism, adopted in commodity HW to avoid (mis)training, is Indirect Branch Restricted Speculation (IBRS) [12], [14], which affects the *setup* step. Using IBRS restricts the training of indirect targets inside an enclave. Likewise, using Thread Indirect Branch Prediction (STIBP) restricts the use of prediction entries trained in another SMT thread. Speculative execution have also been restricted using micro-code updates [86]. SW-based defenses [15], [31], [87] also aim to restrict speculation by avoiding the state transition $MS_I \rightarrow MS_P$, and preventing the speculation triggered by the the branch prediction. Here, the (exploitable) indirect branch is replaced by a different retpoline sequence. This will cause the misspeculated code to execute a controlled loop sequence until speculation has been resolved. To lower the performance cost of the technique both a probabilistic variant, randpoline [31], and a HW variant [15] have been proposed. The root cause exploited is information flow. A recent paper, RETBLEED [184], have shown that the retpoline strategy only provides partial protection, since it can be circumvented using manipulation of return instructions.

Another mechanism to restrict speculation is to introduce fences to limit transient execution. Many existing attack mitigations use the serializing *lfence* instruction before sensitive parts of the code. In order to improve the usability and performance cost, defenses have been proposed which selectively and automatically choose when to use fences, either in SW [105], [157], [165], [174] or in HW [165]. For example Shen et al. [157] split code into small blocks and insert fences between the entry point and a potentially leaking memory access to defend against Spectre attacks. The root causes affected are determinism and information flow, depending on the solution. Another example is Context-Sensitive Fencing (CSF) [165] which uses customized decoding from instructions to micro-ops to insert fences after a conditional branch instruction and before a subsequent load instruction. The root causes affected by fences are determinism and information flow, in the *interact* step. Another way similar to fences is Speculative Load Hardening (SLH) [36] a compiler-level technique where the idea is to introduce a data dependency on the condition, in order to guarantee that the control flow is valid. The technique is supported in LLVM and GCC [48]. Oleksenko et al. [129] restrict speculation by introducing a data dependency in order to guarantee that a load will only start if the comparison is in registers or L1 cache. However, the technique is only effective if the load is performed after the comparison.

Another method to restrict speculations is to wait for authorization or until data is no longer transient [32], [53], [110], [119], [145], [148], [181] which affect the execution in the *interact* and/or *transmit* steps. Here, the defenses delay to avoid the access violation which leads to leakage (in *interact*) or stall the load which would update *MS* of *SC* (in *transmit*). For example NDA (Non-speculative Data Access) [181] provide different policies for controlling control flow and data propagation in *interact*. The root causes affected are determinism, sharing and information flow. SpectreGuard [53] also affects the *interact* step and proposes to mark secret data and selectively restrict speculation only for data from sensitive pages. These defences affect the root cause determinism, access violation and/or information flow in the *interact* step. CondSpec [110] affects *transmit* by handling loads differently, a load that hits in the *SC* can read the data and complete its execution while a load that experience a cache miss will be stalled and re-issued later. This affects the root cause access violation and/or information flow. CleanUpSpec [144], on the contrary, restricts how speculative updates are encoded in the *MS* of the *SC*. Speculative accesses are allowed to progress and make changes to the *SC* but these are removed in case of miss-speculation. This has been shown to be insufficient in certain conditions [22]. This defense affects the root causes determinism and information flow, in the *transmit* and *receive* step of the attack.

The performance cost of the restriction-based defenses depends on how restrictive the rule set is, i.e., how much execution differs from the unconstrained speculation scenario. Introducing protections at a later attack step generally leads to more flexibility, since more speculation can be allowed, and comes at a lower performance cost [82]. The trade-off is that more possible side-channels can be used for the state transitions $MS_I \rightarrow MS_T$ and $MS_T \rightarrow MS_R$, which makes ensuring full protection challenging.

The last category targets isolation by introducing a shadow structure to hold the speculative *MS* until it is deemed safe [8], [9], [95], [186]. For example, in MounTrap [9] an L0 filter cache is used for speculative data, which is only allowed to propagate to the rest of the cache hierarchy after commit. This allows the speculation and potential access violation to occur but not affect *MS* of the *SC*, for example. This restricts information flow from the transient execution to the non-transient execution.

4.2.2. Defenses against Exception-based Attacks. Exception-based attacks typically exploit implementation oversights in HW. The Meltdown attack exploits a race

condition between authorization and access [77] which enables transient execution to continue with an unauthorized value, leading to access violation. Newer CPUs contain patches whereas existing ones are mostly protected through microcode updates or other workarounds [33]. For instance, the issue has been addressed on newer Intel microarchitectures [12], Whiskey Lake and onward, by returning zero when accessing privileged memory [34]. In the case of the MDS-attacks the leaks is attributed to a use-after-free vulnerability where stale data is read in the internal registers [149], allowing unintended information flow through shared CPU buffers. The issue is solved by flushing the internal buffers to restrict the information flow. Similarly, to defend against LazyFP, FPU registers are flushed on context switches when changing protection domains with SGX, for hypervisors and for logical cores. In addition, since Linux 4.6 eager FPU switching is used by default [120]. This disables the fault since the FPU is always available. Foreshadow has been mitigated on Intel CPUs through setting a physical page number field of unmapped page-tables to refer to non-existing physical memory [85], [182], thereby restricting access violation. The LVI attacks [171] are no longer possible when the corresponding fault or microcode assist is mitigated.

Another defense strategy is to provide stronger address space isolation [56], [65], [84], [103], which mitigates or limits the possible access violation in the attacks. For example MemoryRanger [103] isolates drivers, kernel and user space into separate address spaces using Extended Page Table (EPT). This defense restricts access violation in the *interact* step, by providing isolation.

4.3. System-level defenses

Broader system-level strategies for providing protection have also been proposed. Leveraging constant-time programming paradigm is one such strategy where the key idea is to rewrite the SW-implementation to avoid timing variability. The challenge with constant-time programming is to provide protection for all types of attacks. Another strategy is to decrease the accuracy of timing measurement [42], [83], [101], [122]. This strategy affects determinism in the *receive* step, and can be used as a defense against several attacks on different optimizations. The strategy leads to lower attack bandwidth but has been shown to be ineffective in providing complete protection since many attacks can amplify the timing difference [124] or use other timer mechanisms [150]. Another strategy is to use formal models [38], [70], [71], [77], [127] in order to automatically synthesize vulnerabilities. This strategy could potentially find all exploitable scenarios in the design phase itself. However, identifying all vulnerabilities complicate the model building process and current proposals therefore only target a limited set. Finally, leveraging programming language based security [106] is another option wherein defenses are co-designed leveraging programming language annotations to mark sensitive parts of the programs, along with suitable HW primitives. However, a recent analysis performed by Naseredini et al. [128] found that most programming languages and their execution environments does not have support for Spectre mitigations. This shows the challenge of relying on programming languages and execution environments to provide complete protection.

5. Discussion

Commonalities: We have shown that the root causes that enable the attacks are common, across a wide range of microarchitectural optimizations. Furthermore, through our analysis of the proposed defenses, we have shown that these target one or more root cause, across the different steps of an attack. There are commonalities even in the defense strategies used to protect against attacks on these diverse microarchitectural optimizations. Some of those include disabling the optimization, to restrict all the root causes; isolating the state related to the optimization, to restrict sharing; applying randomization and/or restriction, to limit information flow and introducing permission checks, to limit resources from, exposing/accessing state outside of the intended domain.

Using these common strategies together with the root causes, enable us to envision new defenses for vulnerable microarchitectural optimizations. We apply these common strategies to defend against potential attacks exploiting value prediction (Section 3.3). Possible defenses against those attacks could involve flushing the table at context switches, isolating and/or partitioning the table to avoid conflict and reuse-based attacks, introducing randomization to limit information flow. Another option could involve introducing non-determinism in the value prediction mechanisms. Lastly, the simplest mitigation would be to provide mechanisms to selectively disable the optimization when running sensitive parts of the program.

Observations: Firstly, the ease of exploiting a microarchitectural optimization and the severity of the leak vary widely. There exist limiting factors which are not easy to quantify, such as the availability and capabilities of gadgets in the victim code [35]. The link between attack bandwidth and effectiveness in practical scenarios is also difficult to quantify.

Secondly, we observe that, in several cases, vulnerabilities are not due to the fundamental behaviour of the microarchitectural optimization but are rather a result of design and/or implementation. This points in the direction of promoting a deeper understanding of the root causes of vulnerabilities and the potential defense strategies in the design and implementation phases, which we hope can be assisted by our framework, rather than as an afterthought. We believe that microarchitectural optimizations can still be a promising avenue to provide performance scalability in future technology nodes without having to compromise on security.

Future work: Our focus, in this paper, has been on vulnerabilities in microarchitectural optimizations that target performance. We have not exhaustively covered all microarchitectural optimizations and/or resources, for example the NoC and DRAM. However, we expect our framework to be easily extended to cover attacks and defenses on other optimizations and resources. We have not focused on SW vulnerabilities, such as buffer overflows, which poses considerable security risks. Furthermore, we have not investigated microarchitectural optimizations for security, such as Intel SGX, nor power-based side channel attacks nor performance degradation attacks [10], [11],

[21], [64]. Investigating and/or extending the root cause framework to include optimizations for security and considering power-side channels is left for future work.

6. Conclusions

We identify four root causes that enable timing-based side-channel attacks on an extensive set of microarchitectural optimizations. We provide a framework and systematize both transient and non-transient attacks and defenses, highlighting the similarities and differences. Based on our analysis we discuss potential attacks and defenses for vulnerable optimizations. We believe our framework can assist computer architects in understanding the landscape of attacks and defenses, and to provide guidance in designing secure microarchitectural optimizations.

References

[1] AMD prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[2] SecSMT: Securing SMT processors against Contention-Based covert channels. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[3] TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, page 312–320, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, page 225–242, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In Steven D. Galbraith, editor, *Cryptography and Coding*, pages 185–203, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions. *IEEE Computer Architecture Letters*, 20(2):162–165, 2021.

[8] Sam Ainsworth. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 592–606, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 132–144. IEEE Press, 2020.

[10] Alejandro Cabrera Aldaya and Billy Bob Brumley. Hyperdegrade: From ghz to mhz effective cpu frequencies, 2021.

[11] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435, New York, NY, USA, 2016. Association for Computing Machinery.

[12] AMD. Speculative execution side channel mitigations, 2018, 2018.

[13] AMD. Amd speculative bypass store disable, technical report, 2022.

[14] AMD. Software techniques for managing speculation on amd processors, 2022, 2022.

[15] Nadav Amit, Fred Jacobs, and Michael Wei. {JumpSwitches}: Restoring the performance of indirect branches in the era of spectre. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 285–300, 2019.

[16] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

[17] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1369–1382, New York, NY, USA, 2018. Association for Computing Machinery.

[18] ARM. Arm architecture reference manual armv8, 2018, 2018.

[19] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–164, 2019.

[20] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, Boston, MA, August 2022. USENIX Association.

[21] Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention, 2019.

[22] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1046–1060, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Daniel J. Bernstein. Cache-timing attacks on aes. 2005.

[24] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, pages 32–49, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[25] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[26] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[28] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, Boston, MA, August 2022. USENIX Association.

[29] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123, 2020.

[30] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. Efficient mitigation of side-channel based attacks against speculative execution processing architectures, September 14 2021. US Patent 11,119,784.

[32] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. Model checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification. In Claude Carlet, Sylvain Guilley, Abderrahmane Nitaj, and El Mamoun Souidi, editors, *Codes, Cryptology and Information Security*, pages 462–479, Cham, 2019. Springer International Publishing.

[33] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of defenses against transient-execution attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, GLSVLSI '20, page 169–174, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 481–493, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 249–266, USA, 2019. USENIX Association.

[36] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant1 mitigation), 2018, 2018.

[37] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses, 2021.

[38] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, 2019.

[39] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 142–157, 2019.

[40] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Leaking control flow information via the hardware prefetcher, 2021.

[41] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel® itanium™ processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, page 182–191, USA, 2001. IEEE Computer Society.

[42] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 204–217, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[44] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 101–110, 2019.

[45] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2857–2874, Boston, MA, August 2022. USENIX Association.

[46] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. *HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments*. USENIX Association, USA, 2020.

[47] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[48] R Earnshaw. Mitigation against unsafe data speculation (cve-2017-5753, 2018.

[49] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[50] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Trans. Archit. Code Optim.*, 13(1), mar 2016.

[51] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *SIGPLAN Not.*, 53(2):693–707, mar 2018.

[52] Jacob Fustos, Michael Bechtel, and Heechul Yun. Spectrerewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, ASHES'20, page 117–126, New York, NY, USA, 2020. Association for Computing Machinery.

[53] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[54] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018.

[55] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. Cachefx: A framework for evaluating cache security, 2022.

[56] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. Lazarus: Practical side-channel resilient kernel-space randomization. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 238–258, Cham, 2017. Springer International Publishing.

[57] Thomas Gleixner. x86/kpti: Kernel page table isolation (was kaiser), 2017, 2017.

[58] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. *Speculative Probing: Hacking Blind in the Spectre Era*, page 1871–1885. Association for Computing Machinery, New York, NY, USA, 2020.

[59] Google. Product status: Microarchitectural data sampling (mds), 2022.

[60] Vinodh Gopal, James Guilford, Erdinç Öztürk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and constant-time implementation of modular exponentiation. 01 2009.

[61] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association.

[62] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51, 2020.

[63] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology – ICISC 2009*, pages 176–192, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[64] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 409–418, 2002.

[65] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[66] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[67] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[68] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing.

[69] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[70] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018.

[71] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware/software contracts for secure speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, S&P 2021. IEEE, 2021.

[72] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[73] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. *IVcache: Defending Cache Side Channel Attacks via Invisible Accesses*, page 403–408. Association for Computing Machinery, New York, NY, USA, 2021.

[74] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1458–1473, 2022.

[75] S. Kariyappa Gururaj Saileshwar and Moinuddin Qureshi. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. 2021.

[76] Red Hat. Simultaneous multithreading in red hat enterprise linux, 2022.

[77] Zecheng He, Guangyuan Hu, and Ruby Lee. New models for understanding and reasoning about speculative execution attacks, 2020.

[78] Zecheng He, Guangyuan Hu, and Ruby Lee. New models for understanding and reasoning about speculative execution attacks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 40–53, 2021.

[79] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[80] Mark D. Hill. Technical perspective: Why 'correct' computers can leak your information. *Commun. ACM*, 63(7):92, jun 2020.

[81] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, page 42–47, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Guangyuan Hu, Zecheng He, and Ruby B. Lee. Sok: Hardware defenses against speculative execution attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 108–120, 2021.

[83] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[84] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient defence against meltdown attack for unpatched VMs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 255–266, Boston, MA, July 2018. USENIX Association.

[85] INTEL. L1 terminal fault, 2018, 2018.

[86] INTEL. Affected processors: Transient execution attacks & related security issues by cpu, 2022.

[87] Intel. Retpoline: A branch target injection mitigation, technical report, 2022.

[88] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[89] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319, Cham, 2014. Springer International Publishing.

[90] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. Branchspec: Information leakage attacks exploiting speculative branch instruction executions. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 529–536, 2020.

[91] Ken Johnson. Kva shadow: Mitigating meltdown on windows, 2018, 2018.

[92] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[93] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[94] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. Non deterministic caches: a simple and effective defense against side channel attacks. *Des. Autom. Embed. Syst.*, 12(3):221–230, 2008.

[95] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[96] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, August 2012. USENIX Association.

[97] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, jun 2014.

[98] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[99] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

[100] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[101] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, August 2016. USENIX Association.

[102] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 69–81, Vancouver, BC, August 2017. USENIX Association.

[103] Igor Korkin. Divide et impera: Memoryranger runs drivers in isolated kernel spaces, 2018.

[104] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, page 3, USA, 2018. USENIX Association.

[105] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[106] Dexter Kozen. Language-based security. In *International Symposium on Mathematical Foundations of Computer Science*, pages 284–298. Springer, 1999.

[107] M. Larabel. Openbsd disabling smt/hyper threading due to security concerns, 2022.

[108] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. *ACM Trans. Archit. Code Optim.*, 17(3), aug 2020.

[109] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 557–574, USA, 2017. USENIX Association.

[110] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 264–276, 2019.

[111] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237, 1996.

[112] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 138–147, New York, NY, USA, 1996. Association for Computing Machinery.

[113] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 813–825, New York, NY, USA, 2020. Association for Computing Machinery.

[114] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020.

[115] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[116] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[117] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[118] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6), jul 2021.

[119] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. Dolma: Securing speculation with the principle of transient non-observability. In *USENIX Security Symposium*, 2021.

[120] A. Lutomirski. x86/fpu: Hard-disable lazy fpu mode, june 2018., 2018.

[121] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery.

[122] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012.

[123] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64, Cham, 2015. Springer International Publishing.

[124] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution, 2019.

[125] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space, 2019.

[126] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538–570, aug 2019.

[127] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 72–86, New York, NY, USA, 2022. Association for Computing Machinery.

[128] Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, and Daniel Gruss. Systematic analysis of programming languages and their execution environments for spectre attacks, 2021.

[129] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[130] Hamza Omar, Brandon D'Agostino, and Omer Khan. Optimus: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *IEEE Transactions on Computers*, 69(11):1558–1570, 2020.

[131] Hamza Omar and Omer Khan. Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 111–122, 2020.

[132] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery.

[133] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[134] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical, 2021.

[135] Arthur Perais and André Seznec. Eole: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance. *ACM Trans. Comput. Syst.*, 34(2), apr 2016.

[136] C. Percival. Cache missing for fun and profit. In *http://www.daemonology.net/papers/ htt.pdf*, pages 974–987, 2005.

[137] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 987–1002, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[138] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache. *CoRR*, abs/1908.03383, 2019.

[139] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[140] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 360–371, New York, NY, USA, 2019. Association for Computing Machinery.

[141] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867, 2021.

[142] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast Floating-Point operations on x86 processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 71–86, Austin, TX, August 2016. USENIX Association.

[143] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[144] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[145] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 723–735, New York, NY, USA, 2019. Association for Computing Machinery.

[146] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2021.

[147] Sercan Sari, Onur Demir, and Gurhan Kucuk. Fairsdp: Fair and secure dynamic cache partitioning. In *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pages 469–474, 2019.

[148] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[149] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[150] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing.

[151] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I*, page 279–299, Berlin, Heidelberg, 2019. Springer-Verlag.

[152] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing: Reviving foreshadow. 2021. 25th International Conference on Financial Cryptography and Data Security : FC 2021 ; Conference date: 01-03-2021 Through 05-03-2021.

[153] Brian C. Schwedock and Nathan Beckmann. Jumanji: The case for dynamic nuca in the datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 665–680, 2020.

[154] André Seznec. Exploring value prediction with the eves predictor. 2018.

[155] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[156] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 423–435, New York, NY, USA, 2017. Association for Computing Machinery.

[157] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2297–2299, 2018.

[158] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery.

[159] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, Santa Clara, CA, August 2019. USENIX Association.

[160] Olin Sibert, Phillip A. Porras, and Robert Lindell. The intel 80x86 processor architecture: Pitfalls for secure systems, 1995.

[161] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 955–969, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[162] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018.

[163] Jakub Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardw. Syst. Secur.*, 3(3):219–234, 2019.

[164] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 2020.

[165] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 395–410, New York, NY, USA, 2019. Association for Computing Machinery.

[166] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[167] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols, 2018.

[168] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, jan 2010.

[169] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[170] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 991–1008, USA, 2018. USENIX Association.

[171] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, 2020.

[172] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 937–954, Baltimore, MD, August 2018. USENIX Association.

[173] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[174] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *arXiv preprint arXiv:2005.00294*, 2020.

[175] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1491–1505, 2022.

[176] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54, 2019.

[177] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. Brb: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477, 2019.

[178] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1506–1524, 2022.

[179] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[180] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, jun 2007.

[181] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[182] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [**?**].

[183] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.

[184] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[185] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3), may 2021.

[186] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441, 2018.

[187] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 347–360, New York, NY, USA, 2017. Association for Computing Machinery.

[188] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[189] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017.

[190] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[191] Lu-Tan Zhao, Rui Hou, Kai Wang, Yu-Lan Su, Pei-Nan Li, and Dan Meng. A Novel Probabilistic Saturating Counter Design for Secure Branch Predictor. *J. Comput. Sci. Technol.*, 36(5):1022–1036, 2021.

[192] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1267–1272, 2021.

[193] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. Hybp: Hybrid isolation-randomization secure branch predictor. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 346–359, 2022.

Paper IV

SCALE: Secure and Scalable Cache Partitioning

N. Ramhöj Holtryd, M. Manivannan and P. Stenström

To appear in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2023.

SCALE: Secure and Scalable Cache Partitioning

Nadja Ramhöj Holtryd, Madhavan Manivannan and Per Stenström
Chalmers University of Technology, Sweden
Email: {holtryd, madhavan, per.stenstrom}@chalmers.se

***Abstract*—Dynamically partitioned last-level caches enhance performance while also introducing security vulnerabilities. We show how cache allocation policies can act as a side-channel and be exploited to launch attacks and obtain sensitive information. Our analysis reveals that information leaks due to predictable changes in cache allocation for the victim, that is caused and/or observed by the adversary, leads to exploits.**

We propose SCALE, a secure cache allocation policy and enforcement mechanism, to protect the cache against timing-based side-channel attacks. SCALE uses randomness, in a novel way, to enable dynamic and scalable partitioning while protecting against cache allocation policy side-channel attacks. Non-determinism is introduced into the allocation policy decisions by adding noise, which prevents the adversary from observing predictable changes in allocation and thereby infer secrets. We leverage differential privacy (DP), and show that SCALE can provide quantifiable and information theoretic security guarantees. SCALE outperforms state-of-the-art secure cache solutions, on a 16-core tiled chip multi-processor (CMP) with multi-programmed workloads, and improves performance up to 39% and by 14%, on average.

I. Introduction

Shared last-level caches (LLCs) offer a broad attack surface for an adversary to exploit timing side-channels and infer secrets. There are three LLC attack categories: conflict-based [39], shared-memory-based [68] and occupancy-based attacks [52]. Current secure cache partitioning solutions can, in principle, defend against all three categories of attacks by ensuring isolation between adversary and victim processes. But, this comes at the cost of lower cache utilization.

Initial works propose to partition cache capacity *statically* among co-running applications [38]. However, static policies do not adapt to changing requirements of applications and provide lower performance than *dynamic* cache partitioning policies [3], [20], [21], [27], [28], [33], [44], [47]. Dynamic cache partitioning solutions comprise two parts: i) an allocation policy which determines the partition allocations in order to maximize an aggregate metric (e.g. cache hit rate), and ii) an enforcement mechanism to maintain the partitions. Prior works on secure cache partitioning have mostly focused on the enforcement mechanism, while assuming static allocations [8], [9], [25], [38], [45], [57]. In addition, none of these works satisfy all the requirements of an ideal enforcement mechanism – support for fine-grain partitions, scalability and locality-aware partition placement to reduce on-chip access latency while being secure. Furthermore, commercially available solutions, as Intel CAT [19], do not satisfy these requirements and allow information leakage across partitions [25].

Secure allocation policies have received little attention, despite their significant performance impact. Determining allocations both dynamically and securely is challenging because allocations leak information on applications' cache demands. There are two existing attempts to secure cache allocation: SecDCP [63] and OPTIMUS [35]. SecDCP provides the notion of security tiers (confidential and public applications) and permits one-way leakage from the public tier by only considering public applications' cache demand for determining allocations. OPTIMUS, places a bound on the information leakage by only performing allocation once at the start of the execution. Our evaluation (in Section V) shows that both proposals are ineffective and do not provide improvement over the baseline. In summary, existing allocation policies and enforcement mechanisms do not provide support for secure and scalable cache partitioning.

This paper proposes SCALE – a novel dynamic cache partitioning solution which protects the LLC against timing-based side-channel attacks while considering the cache demand of both secure and non-secure applications. The enforcement mechanism can accommodate a wide range of partition sizes and is scalable and locality-aware. SCALEs allocation policy is based on insights from a detailed characterization of the cache allocation policy side-channel and how it can be exploited to launch conflict- and occupancy-based attacks.

Our analysis shows that information leaks, related to small working set changes, which lead to predictable cache allocation changes, enable exploits. Consequently, SCALE aims at providing strong protection against such exploits. This protection is especially important for cryptographic libraries since they have small working set sizes [22], [51], [64]. Leaking high-level occupancy information (e.g, showing that a co-running application is memory intensive) has not been exploited for side-channel attacks. SCALE therefore permits leakage about imprecise high-level cache demand, and provides quantifiable and weak security guarantees in such cases. In contrast, exact LLC occupancy is leaked by defenses that protect the LLC through randomizing placement of lines in the cache [16].

SCALEs cache allocation policy leverages *randomness*, in a novel way, by adding noise to cache allocation decisions computed in a deterministic manner, to achieve its primary design goals. This paper is also the first, to the best of our knowledge, to apply differential privacy (DP) [13], [14] to protect against cache side-channel attacks. Although noise protects against conflict-based attacks, we notice that patterns can still be observed, which can be leveraged in occupancy-based attacks [52]. To address this, we introduce two additional levels of randomization. The first randomizes the extent of permitted change in allocation between two consecutive reconfiguration periods while the second randomizes the period.

SCALE allows configurability based on the security requirements for each application. Our security analysis demonstrates

that SCALE provides strong and quantifiable information theoretic security guarantees for allocation changes within the configured range, while providing weaker guarantees and lower channel bandwidth for larger changes. In addition, SCALE supports a mix of secure and non-secure applications. Our evaluation shows that SCALE defends against allocation policy side-channel attacks while retaining most of the performance benefits of a non-secure allocation policy like UCP.

The secure enforcement mechanism proposed in SCALE, builds on the DELTA enforcement mechanism [20]. DELTA combines bank and way partitioning to support fine-grained partitions and locality-aware mapping of data. The adaptations for SCALE enable secure enforcement, reduces the overheads associated with handling shared data and simplifies the design. SCALEs enforcement mechanism supports secure, fine-grain and locality-aware partitioning of cache capacity. To show the applicability of SCALEs allocation policy to commercially available designs, we evaluate it together with an enforcement mechanism comparable to a secure version of Intel CAT.

In summary, we make the following contributions:

- We demonstrate conflict- and occupancy-based attacks on the cache allocation policy side-channel and propose a framework, inspired by prior work [32], [66], for characterizing the channel bandwidth and error rate.
- We propose SCALE, a holistic solution for secure and scalable dynamic cache partitioning. SCALEs allocation policy leverages randomness, in a novel way, to defend against attacks on the cache allocation policy side-channel.
- Our DP-based security analysis shows that SCALE provides quantifiable and information theoretic security guarantees. SCALE outperforms state-of-the-art secure cache partitioning solutions, on a 16-core CMP, and improves performance by up to 39% and by 14%, on average, compared to an unpartitioned shared LLC.

II. Background and Motivation

A. Cache Allocation Policy

We use UCP [44], a dynamic cache partitioning solution to illustrate the operations of a cache allocation policy. The allocation policy in UCP consists of two steps. The first step, estimates the utility for each application which helps determine how much each application will benefit from additional capacity. UCP uses utility monitors (UMONs) to estimate the utility of different cache sizes, for each application. The second step, compares the estimated utility for each application under different cache sizes and calculates allocations using the Lookahead algorithm [44], [56]. In a nutshell, Lookahead compares the utility values for applications, identifies the application with the highest utility and assigns the capacity that maximizes the utility. These two steps are repeated after a preset time (reconfiguration period), and the allocations are determined based on the statistics collected in the previous period(s). In commodity systems, utility is estimated using performance counters or by using Intel's Cache Monitoring Technology [15], [19]. Note that utility estimation and capacity allocation are predictable and deterministic, i.e., the utility and the assignment always remain the same for a given input.

B. Cache Allocation Policy Side-Channel

A typical side-channel attack consists of three steps: interfere, wait and analyse [25]. The cache allocation policy can act as a side-channel and leak information from the victim to the adversary.

Conflict-based attack: We illustrate a conflict-based attack on the cache allocation policy side-channel, with the data dependent square-and-multiply exponentiation algorithm used in RSA and ElGamal decryption in GnuPG v1.4.13. In the algorithm different values for the exponent bit lead to different code execution paths. Specifically, a '1' in the exponent results in a square-reduce-multiply-reduce step while a '0' results in a square-reduce step, see Figure 1a.

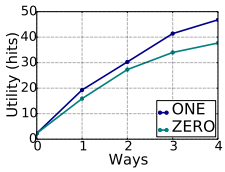
For the purposes of illustration we assume that the victim and the adversary share a 4-way cache, as shown in Figure 1c. In the interfere step, the adversary primes the allocation policy, to equal allocation of two ways, to ensure that the utility changes from executing the square-and-multiply exponentiation algorithm with bit '1' in the exponent will lead to an allocation increase, while executing the same with bit '0' will result in no allocation change. Next, the victim executes the algorithm and changes the cache allocation only in case it encounters a '1' in the exponent. The difference in the code execution path reflects on the utility estimate (see Figure 1b) and leads to predictable allocation changes (see Figure 1c) due to the deterministic nature of the allocation policy. Finally, by accessing data in its own partition and measuring the timing, the adversary can determine its own allocation, infer the allocation for the victim and conclude if the bit accessed by the victim is a '1' or a '0'. Leaking the exponent bits in this way can lead to the recovery of the private key in both RSA and Elgamal decryption [32]. We have omitted amplification/synchronization details, which can be handled as in other side-channel attacks [1], [53], [59]. One simple amplification step is to ensure that the utility change, from the exponent calculation, will result in an allocation change. This can be ensured by manipulating the working set sizes of either the victim or adversary, by for example accessing a data set of appropriate size.

Note that a non-secure and deterministic allocation policy can be attacked and manipulated, in order to extract information about a co-running victim process, even in the presence of a secure enforcement mechanism. A coarse-grained enforcement mechanism, as Intel CAT [19], does not provide protection for the cache allocation side-channel. The example above has shown how small and predictable utility changes can be observed through the cache allocations and reveal secrets.

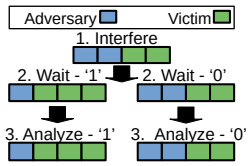
Occupancy-based attack: In addition to conflict-based attack, observing changes to cache allocation can launch an occupancy-based attack and reveal sensitive information, similar to the attack in Shusterman et al. [52]. The previous example illustrates this. Figure 1d shows how the cache allocation of the adversary changes across two different runs of the algorithm using a single exponent bit sequence. Here, the adversary interferes and resets the allocation to equal partitioning before waiting for the victim to transmit each successive bit. The results show that different executions, using the same exponent

```
Square-and-multiply for
exponentiation e[0..n-1]:
r = r² mod m
if(e[i] ==1){
   r = r * b mod m
}
```

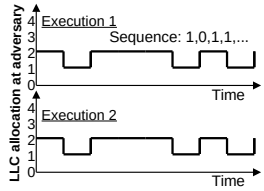
(a) Pseudo-code for the square-and-multiply exponentiation algorithm.



(b) Utility for '1' and '0' in the square-and-multiply alg.



(c) Conflict-based attack.



(d) Occupancy-based attack.

Fig. 1: Cache allocation policy side-channel attacks on the data dependent square-and-multiply exponentiation algorithm.

bit sequence, provide the same cache allocation sequence due to the deterministic nature of the cache allocation policy. Monitoring this information could allow an adversary to learn the fingerprint of allocations made by the victim.

C. Threat model

Our threat model assumes that the victim and the adversary code runs on different processes. We focus on cache timing attacks, (i.e. conflict-based [39], shared memory-based [17], [18], [68] and occupancy-based [30], [52]), on the LLC since L1 and L2 caches are typically private. Furthermore, we also consider cache allocation policy side-channel attacks described in Section II-B. We assume that the victim is always classified as a process with security requirements, while the adversary can be classified as either secure or non-secure. In addition, we consider that threads belonging to the same process (multi-threaded application) cannot act as both adversary and victim. Furthermore, we assume that the adversary can time its own cache accesses.

Similar to prior work, our threat model excludes attacks relying on contention in LLC ports or in the NoC, bandwidth, and attacks on branch data structures, TLBs or shared functional units etc. in the core. These attacks have other defences which can be combined with ours [37], [49], [61]. In case the L1 and L2 are shared among distrusting hyper-threads, we assume equal way-partitioning of private caches and use flushing on context-switches. SCALE does not consider physical side-channel attacks as power [26] or sound [2].

III. SCALE: SECURE PARTITIONING

A. Secure Allocation Policy

The allocation policy in SCALE builds upon the non-secure allocation policy in UCP, outlined in Section II-A. We re-design the allocation policy to provide security guarantees while giving the performance benefit of dynamic cache partitioning. The key idea is to secure the allocation policy by adding noise to the allocations. This protects against conflict- and occupancy-based attacks on the cache allocation policy side-channel.

Figure 2 provides an overview of SCALEs allocation policy. We assume that all applications running on the system are secure, and discuss changes to the allocation policy in the presence of non-secure applications in Section III-A6. The allocation policy in SCALE employs utility monitors to estimate

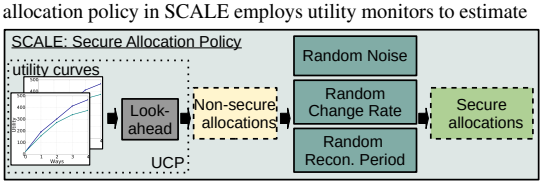


Fig. 2: Overview of SCALE secure allocation policy.

the utility and the Lookahead algorithm to compute non-secure allocations that are deterministic. SCALE utilizes the current allocations and the new allocations computed for the upcoming reconfiguration period, using the Lookahead algorithm, as input. It then applies different defense mechanisms that introduce randomness into the allocations. The process is repeated at each cache reconfiguration period.

The first defense adds random noise, generated using a Laplace distribution, to update the allocations computed by the non-secure policy. Noise provides security guarantees against information leakage, where strong guarantees are only provided for a protected range. The protected range and the amount of noise added is configurable based on the security requirements, as discussed in Section III-A5. The second defense randomizes the allocations change rate, which determines the extent to which allocations can change, in a single reconfiguration period. Randomizing the rate of change causes the change in allocation to take place gradually, over several reconfiguration periods. Furthermore, the allocation changes will be different, at each reconfiguration period. The third defense randomizes the length of the reconfiguration period, which randomizes the timing of allocation changes while also reducing the channel bandwidth.

Security and performance evaluation of SCALE are provided in Section IV and V, respectively. Note that while we propose and evaluate SCALEs defenses by building on top of UCP, these defenses could also be applied to other non-secure allocation policies as [3], [15], [28], [33], [62]. We discuss the individual defense mechanisms in detail below.

1) Noise using Laplace: We add noise generated using the Laplace distribution [14] to the cache allocations computed using the Lookahead algorithm. Laplace is used because it has desirable security properties, see Section IV-B1. The expression for computing allocation for interval t is given by: $A(t) = UCP_{allocation} + noise$, where $UCP_{allocation}$ is the allocation computed using Lookahead for the current period and *noise* represents the noise generated using the Laplace distribution. The Laplace probability density function (PDF) with mean equal to zero is defined as:

$$Lap(x|b) = \frac{1}{2b} exp(-\frac{|x|}{b}) \quad (1)$$

where b is the scale-factor which decides the shape of the distribution. We use a separate parameterized Laplace model for each core to generate noise. Details about noise configuration to suit security requirements, are discussed in Section III-A5.

The pseudo-code for *Noise* is shown in Algorithm 1, where noise is calculated in the function *generateRandomNoise()* (line 4). The output of this function is the amount of noise to be added to or subtracted from the current allocation. The function *randRound()* chooses to round up or down the noise value with

Algorithm 1: Random Noise

```
1  Function Noise(allocations) is
2      sumOfAllocations=0;
3      for core in totalCores do
4          randAllocations[core] = allocations[core] +
             randRound(generateRandomNoise(core));
5          sumOfAllocations += randAllocations[core];
6      end
7      adjustCapacity(randAllocations, sumOfAllocations);
8      return randAllocations;
9  end
10 Function adjustCapacity(randAllocations, sumOfAllocations) is
11     while (sumOfAllocations != totalNumberOfWays) do
12         core = generateRandomId;
13         if (sumOfAllocations > totalNumberOfWays) and
             (randAllocations[core] > minWays) then
14             randAllocations[core]−−;
15             sumOfAllocations−−;
16         end
17         if (sumOfAllocations < totalNumberOfWays) then
18             randAllocations[core]++;
19             sumOfAllocations++;
20         end
21     end
22 end
```

equal probability which leads to better security guarantees because of quantization, as shown in [10]. Since noise is added to the allocations computed for each application independently, the algorithm may reach a state where the sum of the adjusted allocations exceeds the total available cache capacity. In order to address this case, we adjust the capacity as shown in line 7. The function *adjustCapacity()* will subtract/add one way from a randomly chosen application until the allocations match the full cache capacity (lines 10-22) while ensuring that each application gets a minimum allocation of *minWays*.

To illustrate how noise is added we use an example with two applications, A and B, sharing a cache of 16 ways. First, the Lookahead algorithm will determine the best allocation (4 and 12 ways for A and B, resp.). Next, noise is generated for each application using Laplace. The noise generated for A and B is -1 and +3, resp., resulting in a final allocation of 3 ways for A and 15 ways for B. The sum of allocated capacity is 18 ways and this exceeds the total available capacity of the cache. *adjustCapacity()* then decreases the allocations by (randomly) choosing B followed by A, resulting in the allocations of 2 ways for A and 14 ways for B, which will fit in the cache.

2) Random Change Rate: The random change rate (*ChangeRate*) limits the extent to which allocations can change for each application in a single reconfiguration period, and makes the new allocation dependent on the previous allocation: $A(t) = A(t-1) + randRound((A(t-1) - UCP_{allocation}) * changeRate)$, where $A(t)$ is the allocation at time period t. The pseudo-code is shown in Algorithm 2. This algorithm takes the current allocation and the newly computed allocation for the upcoming reconfiguration period as input, creating a feedback-loop. The algorithm limits the extent of change by computing the difference between the current and the new allocations (line 4) and using this as input to *generateRandomChangeRate()* (line 5). The output of this function is the allowed amount of change, drawn from a uniform random distribution. The change determined through this process is added to the current allocation to obtain the randomized allocation for the upcoming period (line 6). To ensure that the entire capacity is allocated we call *adjustCapacity()* (line 9) shown in Algorithm 1.

Algorithm 2: Random Change Rate

```
1  Function RateChange(allocations, currentAllocations) is
2      sumOfAllocations,change=0;
3      for core in totalCores do
4          change = allocations[core] - currentAllocations[core];
5          randChange =
             randRound(generateRandomChangeRate(change,core));
6          randAllocations[core] = currentAllocations[core] +
             randChange;
7          sumOfAllocations += randAllocations[core];
8      end
9      adjustCapacity(randAllocations, sumOfAllocations);
10     return randAllocations;
11 end
```

3) Random Reconfiguration Period: The defense *RandRecon* randomly chooses a reconfiguration period for the next reconfiguration event after each event. This will randomize the timing of when an allocation change is observable. Furthermore, using a longer reconfiguration period decreases the channel bandwidth.

4) Combined defences: The allocation at a randomized reconfiguration interval t, with *Noise* and *ChangeRate*, is: $A(t) = randRound(A(t-1) + randRound((A(t-1) - UCP_{allocation}) * changeRate) + noise)$. *ChangeRate* is not used if $UCP_{allocation}$ at current and last period is the same, $A(t) = randRound(A(t-1) + noise)$.

5) Configurability: SCALE is configurable based on the security requirements determined by the system administrator. For each secure application, the granularity of allocation change that need to be protected, i.e., the protected range, as well as the level of security, can be individually configured. The *Noise* parameters *capacitySetting* and ε are used to specify the protected range and the needed security guarantees, respectively, (Section IV-B1). Using these parameters the amplitude, i.e., the scale-factor b, of the noise will be determined. *capacitySetting* can be set to any value between 32KB (1 way) and 512KB (16 way), in steps of 32KB. SCALE provides strong security guarantees for allocation changes within the protected range and weak security guarantees and reduced channel bandwidth for larger changes. Applications that only require smaller protection granularity or weaker security guarantees, indicated using *capacitySetting* and ε, can expect better performance. For *ChangeRate* we limit the *change_rate_interval* between 10% to 25%. *RandRecon* is configured using the setting *reconfig_period_interval*. The performance impact of the choice of settings are shown in V-C while the security impact is shown in IV-B. For an application with strict security requirements, where any information about working set sizes cannot be leaked, equal static partitioning can be used ($\varepsilon = 0$).

6) Secure and Non-Secure Applications: SCALE can concurrently run applications with and without any security requirement. Here, applications are grouped into secure and non-secure applications. SCALE divides the total cache capacity proportionally among them. For instance, if half of the applications are non-secure then half the LLC capacity is allotted to that group. SCALE invokes Lookahead separately for the non-secure group and partitions the assigned capacity. This provides fairness for non-secure applications in addition to

ensuring that an allocation change in a non-secure application will not impact the allocation of secure applications. Only the secure applications will be affected by the defences in SCALE.

7) Utility Information and SW Support: The cache allocation policy needs HW support in order to estimate the number of misses with different cache sizes. We use sampled UMONs [44] to estimate utility at a way granularity. The utility counters are reset after each reconfiguration period. SCALE allocation policy is executed as a low-level SW runtime and requires OS intervention, at every reconfiguration period.

B. Secure Enforcement Mechanism

1) The DELTA Scheme: We assume the non-secure enforcement mechanism in DELTA [20] since it is suitable for tile-based CMPs, enables locality-aware placement of data and uses fine-grain partitions (e.g. 256 partitions on a 16-core CMP with 16 ways per bank).We first outline DELTAs enforcement mechanism. Next, we explain how SCALE is adapted to ensure security, reduce overheads and simplify the design.

DELTA overview: DELTA provides two major functions by ensuring: (i) that capacity assigned to an application, by the allocation algorithm, is placed in the bank(s) with the least number of network hops and (ii) that a request for a physical address is forwarded to the bank where data is mapped.

DELTA utilizes a per-core[1] cache bank table (CBT) to store the mapping of portions of physical address space to LLC banks, see Figure 3. CBT allows the allocations to span multiple banks. The design determines which bank an address maps to by using a simple 1-cycle linear and inverted hash function that takes 8-bits of the address, after the set index, reverses it and uses the result as an index for the CBT. Note that the hash function is used for distributing addresses across occupied banks and that CBT contents are private to each application. The CBT lookup, using this index, yields the bank id to which the address is mapped to. Once the request reaches the corresponding bank a tag lookup is performed. On a miss, the way partitioning (WP) table in the bank, that keeps track of mapping between cache ways and *core_id*, identifies the candidate ways for inserting the line. On a hit, similar to Intel's CAT [19], data is permitted to be accessed across partition boundaries since DELTA enforces the invariant that only a single copy of a cache line will reside in the LLC. CBT and WP enable fine-grained partition sizes by combining bank and way partitioning. The enforcement supports a minimum partition granularity of 32KB on our simulated system which is the size of a single way in a cache bank. Details regarding the implementation and the storage cost for CBT and WP are provided in DELTA [20].

The CBT is updated when the allocated cache capacity expands/retreats across banks. As an example, consider the allocation policy increases the allocation of an application running on *core0* from 512KB to 768KB. The mapping for CBT on *core0* should be updated from one bank (*bank0*) to span across two banks (*bank0* and *bank1*) resulting in a CBT update. In this case, the original CBT entry for *core0* changes from *0-255→0* to *0-191→0*; *192-255→1* to ensure a fair distribution

[1]per-thread in case of simultaneous multithreading

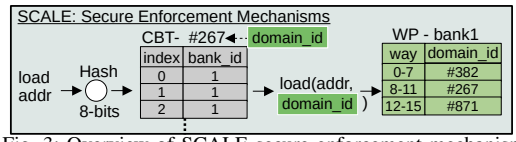


Fig. 3: Overview of SCALE secure enforcement mechanism. New additions/modifications shown in green.

of addresses across both the banks. The address range mapped to a bank is proportional to the size of the allocation.

The CBT update step is preceded by invalidation of all addresses belonging to the remapped region from the cache bank(s) where the data is initially mapped. In this case, cache lines in the remapped address range (*191-255*) is invalidated from *bank0*. The contents in WP table in *bank0* and *bank1* are updated to reflect this change. However, contents in the newly assigned ways in *bank1* are not invalidated. All subsequent requests that map to this index in the CBT on *core0*, will be sent to *bank1* instead of *bank0*. In addition, data associated with a page are invalidated from the caches when a private page becomes shared. This is because DELTA uses the CBT only to determine the mapping of data in private pages. Information about whether an access is part of a private or a shared page is obtained from the TLB, during translation. To access lines in shared pages DELTA resorts to the default SNUCA mapping.

Issues: Although DELTA enforcement provides support for fine-grained and locality-aware placement of data, there are a number of security issues in the design: (1) Accesses outside partition boundaries are allowed, (2) Caches cannot differentiate between accesses coming from different processes on the same core, (3) The replacement policy leaks information through metadata, (4) A single copy of shared data is retained, (5) Data ways in the newly assigned banks are not invalidated.

Adaptations: Figure 3 provides an overview of SCALEs enforcement mechanism. SCALE utilizes *domain_id*, like other secure cache partitioning proposals [25] for identifying the user processes and the kernel (id:0). All cache requests will be tagged with the *domain_id* field. We modify the WP table in each cache bank to store the mapping between cache ways and *domain_id* instead of *core_id*. On a cache lookup and insertion, information about *domain_id* is used to restrict access to specific ways, addressing issue 1 and 2.

Replacement metadata: We evaluate two variants to avoid replacement policy metadata leaks. The first isolates the replacement metadata by partitioning the LRU state [25] while the second simply uses random replacement policy. Both variants protect against replacement metadata leaks and address issue 3, while offering different trade-offs with regard to overhead and performance (Section V-C).

Cache coherence: In order to support multi-threaded applications, SCALE assigns the same *domain_id* to different threads of the same application. This will ensure that data shared among the threads of the application will be cache coherent. In other cases, when data is shared by different processes, it is duplicated, even if a copy already exists in the same bank since the *domain_id* fields for the two requests are different. Additionally, cache flush instructions (CLFLUSH,CLWB) are

appended with the *domain_id*, and only affect the partition with matching id. These changes address issue 4.

Finally, in the original design CBT remappings only trigger invalidation in the bank where address ranges are unmapped (*bank0* in the example above). In SCALE, we invalidate cache lines in the ways which are reallocated to another process, to address issue 5. In the example discussed before, ways in *bank1* which will be allocated to *core0* are also invalidated.

Context switches: It is important to retain CBT mappings across context switches. We therefore store the contents of CBT along with the *domain_id* for all processes in a specific region in the kernel address space. The contents from each process specific CBT are saved and restored at context switch boundaries, like registers. Similarly, the contents of CBT are also saved and restored when handling system calls that involve switching to kernel mode from user mode. Note that these switches do not require invalidating cache contents.

Inter-Process Communication: Data transfer between processes needs to be handled in a special manner to ensure security [25]. Data transfer between processes in different partitions needs to be handled through system calls and OS kernel involvement to ensure isolation. Data transfer between kernel and user-space requires HW support to enable functions, as *copy_from_user* and *copy_to_user*, to read data from addresses mapped to a user's *domain_id* and write data to addresses in the kernel's *domain_id*.

IV. Security Evaluation

This section characterizes the cache allocation policy side-channel and analyses the security provided by SCALE.

A. Channel Characterization

1) Methodology: We use two willing collaborating processes, the *Sender* and the *Receiver*, that exchange information using a covert channel. Such an approach has been used in prior works [32], [66], for characterizing other side-channels. In this experiment, Sender transmits a sequence of message bits to Receiver, using the cache allocation policy side-channel with both a fine-grained and coarse-grained enforcement mechanism (32 resp. 16 ways). We use a return-to-zero (RZ) self-clocking encoding scheme, where the signal returns to a base value before every transmission, since Sender and Receiver execute concurrently without synchronization. We characterize the channel using two metrics - channel bandwidth and error rate. Channel bandwidth provides a measure of how much information can be exchanged over a period of time, measured in $\frac{N_{bits}}{time}$ (Kb/s). The error rate provides a measure of the integrity of information communicated through the channel, measured in $\frac{N_{errors}}{N_{bits}}$. We also characterize the impact of SCALE defences on the channel, with *capacitySetting*=128KB (4 and 2 ways for fine- and coarse-grained, resp.) and $\varepsilon = 1$. The goal is to demonstrate that SCALE can defend against a covert channel which is implemented by inducing allocation changes that are within the protected range.

Sender and Receiver follow a specific protocol to transmit and receive information. Sender can transmit one of the three possible messages: 1, 0 or *new_bit* (indicating return to base before new bit transmission). Each of the messages leads to a specific allocation in the LLC for Sender, which affects Receiver's allocation and the timing measured by Receiver. In this experiment the total cache size is 1MB, transmitting 1 corresponds to accessing a 832KB data buffer, transmitting 0 corresponds to accessing a 576KB data buffer and transmitting a new bit corresponds to accessing a 704KB data buffer.

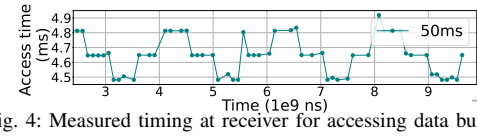
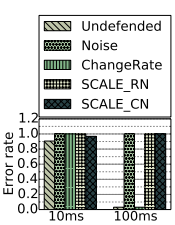


Fig. 4: Measured timing at receiver for accessing data buffer.

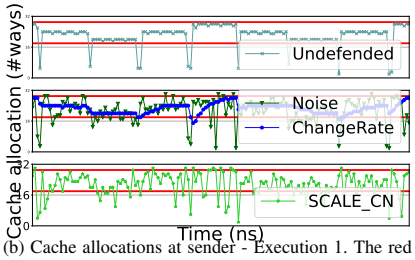
Receiver accesses a 960KB data buffer slowly twice, both in the forward and backward direction. It should be executed at a slower rate to ensure that Sender's access pattern determines the cache allocations. Depending on the allocation for Sender, the allocation for Receiver will change and impact the measured Receiver's access time for the buffer. Figure 4 shows the measured access times for Receiver's buffer. Here, we can clearly see the timing values corresponding to the message bit sequence $1,0,1,0,1,0,1,0$. We classify the timing value between 4.9ms to 4.75ms as 1, the value between 4.7ms and 4.55ms as *new_bit*, and 4.5ms to 4.35ms as 0.

Receiver receives a bit by measuring the time to access the data buffer. It compares the timing for accessing the buffer with the thresholds for 1 and 0. If the measured time does not correspond to a message or if the flag *waitForNewBit* is set, it will continue in the loop and periodically access the buffer. If the access time observed by Receiver matches the thresholds set for 1 or 0 (and *waitForNewBit* flag is not set), it exits the loop and registers the message. It also sets the flag *waitForNewBit* to mark that the next message bit can only be received after receiving a *new_bit* signal. Once Receiver observes a timing value for a *new_bit* signal the flag is reset which allows new message bits to be processed.

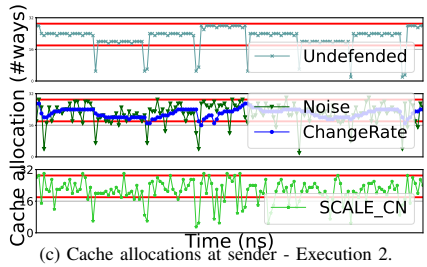
2) Results: Figure 5a shows the error rate for exchanging a 32-bit message using the covert channel and fine-grained enforcement. The results are the same for the coarse-grained enforcement mechanism, which are omitted due to space constraints. We plot the error rate for the undefended case and when using SCALE (individual defenses and combined). The two sets of bars denote the error rate while varying the length of $T_{transmit}$. The results show that a short transmit time leads to a high error rate even for the undefended case. This is because short transmit times do not leave room for Receiver to access the data buffer fully before Sender starts to transmit the next bit. However, when using a long transmit time the undefended scenario has a very low error rate. In contrast, *Noise* introduces a high error rate and has security guarantees. Therefore, it acts as an effective defense against conflict-based attacks on this side-channel and provides partial protection for occupancy-based attacks. *ChangeRate* alone has a low error rate because the long transmit time allows the allocations to eventually reach the expected value over multiple reconfiguration periods. Note that the main target for *ChangeRate*, unlike *Noise*, is occupancy-based attacks. *SCALE_RN* which combines *Noise* with *RandRecon* introduces a high error rate, mostly due to *Noise*. The effect of *RandRecon* will be to randomize the timing



(a) Impact of SCALEs defences on error rate.



(b) Cache allocations at sender - Execution 1. The red boxes shows the protected range.



(c) Cache allocations at sender - Execution 2.

Fig. 5: Error rate and cache allocations.

of allocation changes as well as decrease the attack bandwidth while mostly preserving the performance of a shorter period. Finally, the results for *SCALE_CN* show that combining *Noise* and *ChangeRate* is also effective.

The bandwidth for an ideal covert channel, where the transmission and synchronization for a single message bit is done in one reconfiguration period, is 0.05Kb/s assuming a period of 10ms. Our implementation of an undefended channel achieves an attack bandwidth of 0.005Kb/s when $T_{transmit}$ is 100ms. This is several order of magnitudes smaller than the 1.2 Mb/s reported for LLC side-channel in prior work [32].

Figure 5b and 5c show the changes in cache allocations over time for Sender for two different runs using the same message bit sequence with the fine-grained enforcement mechanism. Receiver can infer Sender's allocation by monitoring its own allocation and by subtracting it from the total available capacity. The allocation trace for *Undefended* shows a clear pattern in allocations for Sender (i.e., 26, 22 and 18 ways), corresponding to sending 1, 0 or *new_bit*) in both runs, as expected. The intermittent changes to 2 ways is due to bookkeeping code in Sender, not strictly part of the attack.

The red boxes in the figure show the range of allocation change, the *capacitySetting*, where SCALE provides strong security guarantees. For larger allocation changes the attack bandwidth is decreased and weak security guarantees are provided. Allocation changes with *ChangeRate* are smaller compared to the undefended case, leading to a stepwise increase to the larger allocations. *SCALE_CN* shows the allocations when combining *ChangeRate* and *Noise*. The intermittent changes, however, allow us to show the impact of the defences on large allocation changes. Over time these larger allocation changes are observable, while exact information about the actual working set size and allocation remains protected.

B. Security Analysis

We analyse and quantify the security provided by SCALE.

1) Security of SCALE allocation policy: In order to quantify the security of SCALE's allocation policy we use differential privacy (DP), which can provide information theoretic privacy guarantees. DP was proposed in the setting of public sharing of information about a population dataset where noise is added to ensure privacy for an individual in the dataset while providing reasonably accurate statistics for the population [12]. DP has also been applied in the context of continuous infinite streams of data [14]. We use DP because SCALE also introduces noise to ensure privacy of allocations assigned to an individual application in a workload. In the analysis, we consider the worst case scenario from a security perspective, of one adversary and one victim process. A randomization algorithm Alg gives ε-differential privacy for all neighbouring data sets D and D', (which differ by a single entry) for all $S \in Range(Alg)$:

$$P[Alg(D) \in S] \leq e^{\varepsilon} P[Alg(D') \in S] \quad (2)$$

Using DP, we can provide an information theoretic security guarantee regarding leakage across D and D'. The sensitivity captures the maximum difference between the two data sets, defined as: $\Delta f = max||f(D) - f(D')||$ [14]. In SCALE, the sensitivity is configured using *capacitySetting*, which captures the magnitude of the protected range. The amount of noise added is determined based on the sensitivity requirement specified using *capacitySetting* and ε. Specifically, the Laplace scale-factor b, used in *Noise*, is determined using these two attributes as follows: $b = \frac{\Delta f}{\varepsilon}$. By using random variables drawn from *Lap(b)*, *Noise* will preserve (ε,0)-differential privacy [14].

The relationship between *capacitySetting*, ε, and the Laplace scale-factor b is shown in Figure 6. If the sensitivity, i.e., the difference between D and D' that needs to be protected is large, more noise need to be added (i.e. higher b) to maintain the same ε value. Likewise, if a stronger security guarantee is needed (lower ε), a higher b is needed. As an example, if $\varepsilon = 0.5$ is required and $\Delta f = 64KB$ (2 ways) then $b = 4$. This framework allows us to quantify the security provided for allocation changes. For example, if a 1-MB change occurs and $b = 4$, then the corresponding value for $\varepsilon = 32/4 = 8$. An ε of 8 provides a quantifiable measure of the given (weak) security guarantees. Overall, the framework allows the system administrator to configure SCALE based on the security requirements.

For a conflict-based attack to be successful, controlled allocation changes are necessary in the two steps of the attack. First, the adversary need to interfere with the victim's allocation in an intended manner. Second, when the victim changes the allocations, these changed allocations need to be observable. Even if we, as in the analysis above, conservatively assume that the adversary can interfere as intended (noise=0), the framework still guarantees that enough noise is added to safeguard the privacy of the victims allocation, i.e., that an adversary cannot determine the expected deterministic allocation of the victim.

Repeated queries: We have so far considered the security guarantees for one reconfiguration period. In a replay-based attack as shown in [53], the adversary can fix its allocation

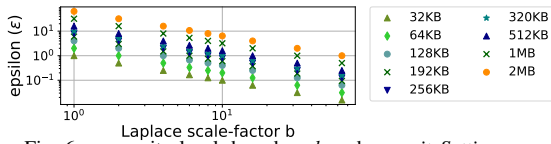


Fig. 6: ε-security level, based on b and *capacitySetting*.

and make the victim re-execute using the same allocation repeatedly, i.e., D and D' can be kept constant over several reconfiguration intervals. In such a scenario, the privacy budget for the victim decreases after each reconfiguration until it is eventually exhausted (noise added over long periods averages to zero, thereby exposing the original allocation). The total privacy budget under composition becomes the number of repeated queries times ε. A memoized noise value (one for each allocation) is used, instead of generating a new noise value using the Laplace distribution at each reconfiguration interval. The memoized noise value is not allowed to be 0. We utilize this memoization mechanism to guarantee that we never leak information as a consequence of overspending the total privacy budget. This technique, involving caching responses, is a proven way for providing event-level DP under continuous observation without impacting the privacy budget [14]. The performance impact of using memoized noise is shown in V-C.

Occupancy-based attack: An occupancy-based attack can succeed if it is possible to create an observable and reproducible fingerprint. In order to observe the same pattern of allocations, the output of *Noise*, *ChangeRate* and *randRound* functions at each time step need to be the same. The probability of observing the same pattern over n reconfiguration intervals, with $UCP_{allocation}$ changes, is $Pr[\frac{1}{(\#randRound*\#changeRate)^n}]$, even in the conservative case when the output of the *Noise* function remains the same due to memoization and the reconfigurations occur at the same time (no *RandRecon*). The probability of observing the allocation pattern at the same time period t is $Pr[\frac{1}{(\#RandRecon)^n}]$, where *#RandRecon* is the total number of possible reconfiguration intervals.

In summary, we provide quantifiable and information theoretic security guarantees, leveraging DP, against conflict-based and occupancy-based attacks. For allocation changes within the protected range, SCALE provides strong security guarantees while for changes outside of the protected range, only weak guarantees are provided and channel bandwidth is reduced. Furthermore, the probability of observing the same allocation fingerprint across repeated executions approaches zero as the number of reconfiguration periods increases.

2) Security of SCALE enforcement mechanism: SCALE guarantees protection against all timing based side-channel attacks: conflict-based, shared-memory-based and occupancy attacks, by enforcing strict isolation between partitions, in a similar manner to prior work [25], [45].

Conflict-based attacks are prevented since replacement state update and victim selection are only performed among the cache lines of a single partition. Shared-memory attacks are prevented since: i) LLC accesses are allowed to hit only if the line resides within the partition, ii) cache flushes only impact cache lines within a partition and do not affect any duplicate lines for the same address in other partitions, iii) coherence-related invalidations are only allowed within the same partition. Occupancy-based attacks are prevented by SCALE allocation policy, as described in Section IV-B1.

V. Performance Evaluation

A. Simulated Architecture

We simulate a 16-core tiled CMP architecture modeled using Sniper [7]. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and an LLC bank of 512KB per core. The cache latencies assumed have been modelled using CACTI 6.5 [34]. Details of the baseline architecture are shown in Table I.

B. Evaluation Methodology

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [50]. We create 21 workload mixes (each comprising 16 applications) by randomly selecting applications.

We fast-forward for 1-B instructions per application (16-B in total) and then carry out detailed simulation until all benchmarks have completed at least 500-M instructions. Statistics are reported based on the detailed simulation. After this period, the applications continue to run and compete for resources to avoid a lighter load on long running applications.

1) Metrics: We report normalized weighted speedup over baseline for each workload as a measure of performance. This is calculated by $\frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{i,new}}{IPC_{i,baseline}}$, in order to evaluate system performance for multi-programmed workloads.

C. Performance Evaluation

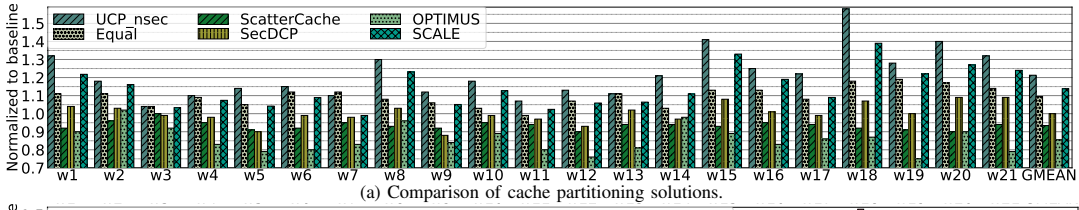
1) Evaluated Configurations: We compare SCALE with secure partitioning solutions, SecDCP and OPTIMUS. For SecDCP, we randomly classify 50% of the applications as confidential and belonging to different security classes and the others as public. To understand the performance potential of dynamic cache partitioning we also evaluate the non-secure UCP. In *Equal* the capacity is equally partitioned among the cores. We also compare against a representative randomization proposal, ScatterCache [65]. To evaluate the suitability of SCALE to commercial designs, we combine SCALEs allocation policy with a coarse-grained (64-way) and non-locality aware enforcement mechanism, similar to Intel CAT.

In total, we evaluate five versions of SCALE to understand the performance impact of the individual defenses when they are applied separately and when they are combined. To ensure a fair comparison we use the partitioned-LRU variant of the SCALE enforcement mechanism for all the evaluated solutions[2]

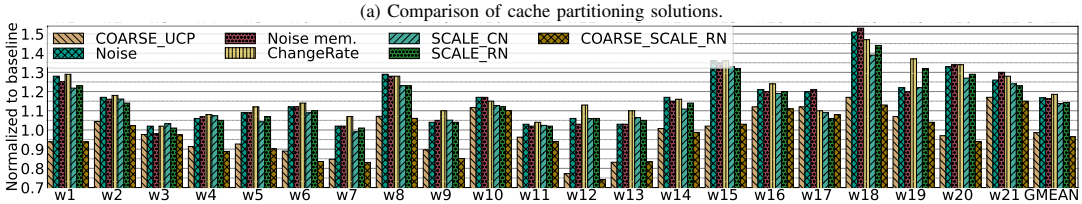
[2]SecDCP and OPTIMUS only propose an allocation policy while UCP's enforcement mechanism only targets monolithic caches.

Cores	16 cores, x86-64 ISA, 4GHz, OOO, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle lat.
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, LRU, inclusive, 9-cycle data and 2-cycle tag latency,
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 MCUs, 1 channel/MCU, latency 80 ns, 16GB/s per channel
SCALE/OPTIMUS/ SecDCP/UCP	*minWays* 2, *reconfiguration_period* 10ms
SCALE (default)	*capacitySetting* 128KB ($b=4,\varepsilon=1$), *change_rate_interval* 0.1-0.25 *reconfig_period_interval* 1-10ms / 100-400ms

TABLE I: Configuration of the simulated 16-core tiled CMP.



(a) Comparison of cache partitioning solutions.



(b) Comparison of SCALE defences and variants.

Fig. 7: Performance comparison.

unless stated otherwise. We use an unpartitioned static non-uniform cache architecture (SNUCA) as the baseline throughout this evaluation. Details about the configuration parameters for each solution are presented in Table I.

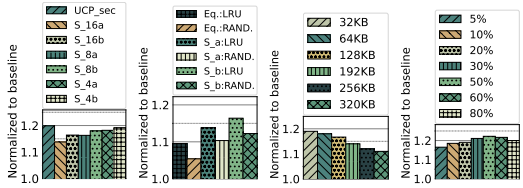
2) SCALE Performance Analysis: Figure 7 shows weighted speedup normalized to unpartitioned SNUCA for each of the workloads. Figure 7a shows a comparison of SCALE against other cache partitioning solutions. *UCP_nsec* is the best-performing configuration since it uses a non-secure allocation policy and enforcement mechanism and improves performance by 21%, on average. In comparison, a secure enforcement mechanism which uses the same non-secure UCP allocation policy improves performance by 20%, on average (not shown). *Equal*, where all data is mapped to the local LLC bank, improves performance by 10% on average. Secure cache solutions *Scattercache* and *OPTIMUS*, perform worse than the baseline by 7% and 14%, respectively, while *SecDCP* performance is equal to the baseline. In contrast, SCALE improves performance for all workloads and provides an average improvement of 14%.

Figure 7b shows the performance of the individual SCALE defences as well as combined variants. *Noise* without and with memoization mechanism improves performance by 17% and 16%, respectively. *ChangeRate* improves performance by 18%. The reason why both *Noise* and *ChangeRate* perform comparably well, is because i) there are often multiple allocation options which give close to equivalent performance and ii) the partition sizes are in the right range, i.e., large vs. small partition sizes cache-friendly applications vs. applications with small working set [20]. Furthermore, slowly changing the allocations is sometimes more beneficial compared to abrupt changes due to the overhead of invalidations, if allocations are later changed again. *SCALE_CN*, which combines both *Noise* with memoization and *ChangeRate* defenses, provides an average improvement of 14%. *SCALE_RN*, combining the defence *RandRecon* with *Noise*, evaluated in a period between 1-10ms yields a performance improvement of 14%, on average, whereas a longer period of 100-400ms leads to a lower performance of 13% (not shown). These results demonstrate that the individual defenses have a small performance overhead and provide most of the performance benefit of UCP. Furthermore, combining these defenses increases the performance overhead slightly but still provides benefit and outperforms the state-of-the-art secure designs while ensuring security.

Coarse-grained enforcement: COARSE_UCP (shown in Figure 7b) uses an enforcement mechanism similar to a secure version of Intel CAT, performs worse than the baseline by 1%. This is due to the lack of locality-aware placement which leads to longer access latencies to cache banks further away from the executing core. Furthermore, coarse-grained enforcement leads to worse fit between applications cache requirements and partition sizes. *COARSE_SCALE_RN* performs worse than the baseline by 4% but improves performance by 6% compared to equal partitioning with the same enforcement (not shown). This demonstrates that SCALE can also be used to secure cache partitioning on commercial systems that support it.

3) Secure and Non-Secure Applications: Figure 8a shows the average normalized weighted speedup over the baseline for the 21 workloads having different configurations with varying number of secure applications in the workload. We randomly pick applications from the workload and mark them as secure.The three different configurations each assume 4, 8 and 16 of the applications in the workload to be secure and the rest to be non-secure. All the secure applications utilize SCALE with *capacitySetting* of either 128KB or 64KB, in version a and b, respectively. *UCP_sec* improves performance by 20%, on average, but is non-secure. *S_16b* assumes all the applications in the workload to be secure, leading to on average 16% performance improvement. *S_8b* with 8 secure applications and *S_4b* with 4 secure applications, show a performance benefit of 18% and 19%, respectively. The results show that having more non-secure applications in the workload mix leads to a higher performance since there is no overhead associated with using the SCALE defenses for non-secure applications.

4) SCALE Enforcement Mechanism Variants: Figure 8b shows the average normalized weighted speedup for all the workloads over the baseline with *Equal* and *SCALE*, when using the partitioned-LRU (LRU) and random replacement variant of the enforcement mechanism. *Equal*, when using the random replacement variant, shows an improvement of 5% as opposed to 10% when using partitioned-LRU. As for *S_a*



(a) Secure and non-secure applications (a:128KB, b:64KB).
(b) Replacement policy comparison (a:128KB, b:64KB).
(c) Sensitivity of Laplace scale b for value of ε + *capacitySetting*) in *Noise*.
(d) Sensitivity (dep. on *ChangeRate*.

Fig. 8: Performance evaluation of SCALE (S) variants.

using the random replacement variant reduces the performance from 14% over partitioned-LRU to 10%, while for *S_b* random replacement reduces the performance from 16% to 12%. This shows that although the random replacement variant has a low hardware overhead in comparison to the partitioned-LRU variant it also provides lower performance.

5) Sensitivity Analysis: Figure 8c shows the average normalized weighted speedup for all the workloads over baseline with different settings for *Noise*. The results show that the overhead increases with more noise, thus enabling a trade-off between performance and security. The performance improvement for *32KB*, *64KB*, *128KB*, *192KB*, *256KB* and *512KB* is 19%, 18%, 17%, 14%, 12% and 11%, respectively. These results indicate that for even larger changes equal partitioning is a better option since it provides an improvement of 10%.

Figure 8d shows the average normalized weighted speedup for all the workloads over the baseline with different values for change rate. The performance for *5%*, *10%*, *20%*, *30%*, *50%*, *60%* and *80%* is 16%, 18%, 19%, 20%, 21%, 21%, and 20%, respectively. The result shows that a change rate below 20% leads to lower performance improvement while an interval in the range of 25% to 60% leads to better performance. We use an interval from 10% to 25%, which gives a reasonable trade-off between security and performance.

In summary, we show that SCALE outperforms prior secure cache solutions and provides a 14% improvement, on average.

VI. Related work

Two broad approaches have been proposed to defend the shared LLC: randomization [23], [30], [31], [42], [43], [46], [55], [58], [64], [65], [67] and isolation [6], [8], [9], [11], [24], [25], [29], [36], [45], [48], [49], [57], [63], [64]. Randomization is achieved by either randomizing the address to cache set mapping, or by randomly inserting and/or evicting data. Recent works have shown that randomization only delays an adversary to infer the secret, especially as smarter and faster attacks that thwart existing defenses are discovered [4], [5], [40], [41], [43], [54], [60]. In addition, randomization cannot protect against occupancy-based attacks. In contrast, SCALE combines isolation with randomization and protects against occupancy-based attacks. SCALE outperforms Scattercache [65], a representative randomization-based proposal.

Isolation based techniques can be grouped into three categories. Firstly, prior works have focused on providing security for selected data. CATalyst and STEALTHMEM [24], [29] statically partition the cache into two regions, i.e. security sensitive and non-sensitive. The sensitive partition is protected by page-coloring, while the non-sensitive partition is shared and insecure. In contrast, SCALE provides protection against timing-based side-channel attacks for all the data in the LLC.

A second category provides solutions for only securing the enforcement mechanism. DAWG [25] proposes a secure enforcement mechanism which provides isolation using way-partitioning. Scalability with DAWG is limited since it can only support as many partitions as the associativity. BCE [45] provides a secure and scalable enforcement mechanism, using a configurable cache indexing strategy for mapping addresses. The drawback is that even small allocation changes require invalidating and remapping all partition data to accommodate changes to the indexing logic. CC [57] proposes a secure enforcement mechanism based on combining way and set partitioning in HW. MI6 [6] and IRONHIDE [36] provide static partitioning of DRAM and LLC. In contrast, SCALE secures both the allocation policy and the enforcement mechanism, enabling better performance through dynamic partitioning and supports fine-grained partitions.

Finally, the last category provides dynamic cache partitioning using coarse-grain cache allocations to provide security. Jumanji [49] proposes a bank-level secure partitioning between VMs but does not provide security between processes sharing a VM, as opposed to SCALE. Furthermore, the solution is prone to attacks on the cache allocation policy side-channel since the allocation policy is deterministic.

Generalizability: Many prior works on cache partitioning [3], [15], [28], [33], [47], [62] have focused on improving performance and scalability. However, the allocation policy and the enforcement mechanism proposed are prone to side-channel attacks. SCALE can be used together with these cache partitioning solutions in order to protect the cache allocation decisions from being exploited in an attack.

VII. Conclusions

We propose SCALE, a holistic solution for secure and scalable cache partitioning. SCALE leverages randomness to make allocations non-deterministic and protect against conflict-based and occupancy-based attacks on the cache allocation policy side-channel. We present a security analysis using DP and in-depth characterisation of the cache allocation policy side-channel. SCALE supports applications with varying security requirements, as well as a mix of secure and non-secure applications. Our evaluation demonstrates that SCALE outperforms state-of-the-art secure cache solutions. Furthermore, SCALE closes the performance gap with non-secure dynamic cache partitioning, for a mix of secure and non-secure applications.

VIII. Acknowledgements

This research has been funded by the Swedish Research Council (VR) under the PRIME project (registration no. 2019-04929). The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by VR through grant no. 2018-05973.

REFERENCES

[1] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation." New York, NY, USA: Association for Computing Machinery, 2016.

[2] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. USA: USENIX Association, 2010, p. 20.

[3] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*.

[4] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 9–12, 2020.

[5] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1110–1123.

[6] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor." New York, NY, USA: Association for Computing Machinery, 2019.

[7] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.

[8] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," 2019. [Online]. Available: https://arxiv.org/abs/1909.09599

[9] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," 2021. [Online]. Available: https://arxiv.org/abs/2110.08139

[10] B. Ding, J. Kulkarni, and S. Yekhanin, "Collecting telemetry data privately," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3574–3583.

[11] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," vol. 8, no. 4, 2012.

[12] C. Dwork, "Differential privacy," in *Automata, Languages and Programming*, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis." Berlin, Heidelberg: Springer-Verlag, 2006.

[14] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," vol. 9, no. 3–4, 2014.

[15] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," vol. 2018-Febru, pp. 104–117.

[16] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom, "Cachefx: A framework for evaluating cache security," *arXiv preprint arXiv:2201.11377*, 2022.

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.

[18] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., 2015, pp. 897–912.

[19] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 657–668.

[20] N. Holtryd, M. Manivannan, P. Stenström, and M. Pericàs, "Delta: Distributed locality-aware cache partitioning for tile-based chip multiprocessors," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 578–589.

[21] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *Proc. ICPP*, 2009.

[22] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[23] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras, "Non deterministic caches: a simple and effective defense against side channel attacks," *Des. Autom. Embed. Syst.*, vol. 12, no. 3, pp. 221–230, 2008.

[24] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA, 2012, pp. 189–204.

[25] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[26] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[27] W.-C. Kwon, T. Krishna, and L.-S. Peh, "Locality-oblivious cache organization leveraging single-cycle multi-hop NoCs," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS '14*.

[28] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*.

[29] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.

[30] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.

[31] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[33] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-41*.

[34] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO-40*, 2007.

[35] H. Omar, B. D'Agostino, and O. Khan, "Optimus: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks," *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1558–1570, 2020.

[36] H. Omar and O. Khan, "Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 111–122.

[37] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 645–662.

[38] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005, page@cs.bris.ac.uk 13017 received 22 Aug 2005. [Online]. Available: http://eprint.iacr.org/2005/280

[39] C. Percival, "Cache missing for fun and profit," in *http://www.daemonology.net/papers/ htt.pdf*, 2005, pp. 974–987.

[40] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 987–1002.

[41] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache," *CoRR*, vol. abs/1908.03383, 2019. [Online]. Available: http://arxiv.org/abs/1908.03383

[42] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[43] M. K. Qureshi, "New attacks and defense for encrypted-address cache." New York, NY, USA: Association for Computing Machinery, 2019.

[44] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*.

[45] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," 2021.

[46] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[47] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*.

[48] S. Sari, O. Demir, and G. Kucuk, "Fairsdp: Fair and secure dynamic cache partitioning," in *2019 4th International Conference on Computer Science and Engineering (UBMK)*, 2019, pp. 469–474.

[49] B. C. Schwedock and N. Beckmann, "Jumanji: The case for dynamic nuca in the datacenter," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 665–680.

[50] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002.

[51] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage." New York, NY, USA: Association for Computing Machinery, 2018.

[52] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019, pp. 639–656.

[53] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," *IEEE Micro*, vol. 40, no. 3, pp. 91–98, 2020.

[54] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 955–969.

[55] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," in *NDSS*, 2020.

[56] D. Thiebaut, H. Stone, and J. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 665–676, 1992.

[57] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache Side-Channel attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, 2022, pp. 2839–2856.

[58] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems." New York, NY, USA: Association for Computing Machinery, 2018.

[59] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking secrets through compressed caches." New York, NY, USA: Association for Computing Machinery, 2020.

[60] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 39–54.

[61] J. Wan, Y. Bi, Z. Zhou, and Z. Li, "Volcano: Stateless cache side-channel attack by exploiting mesh interconnect," *ArXiv*, vol. abs/2103.04533, 2021.

[62] X. Wang and J. F. Martínez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 113–125.

[63] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[64] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," vol. 35, no. 2, 2007.

[65] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019, pp. 675–692.

[66] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 9.

[67] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks." New York, NY, USA: Association for Computing Machinery, 2017.

[68] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, 2014, pp. 719–732.

Microprocessors are an integral part of modern society. They contain billions of tiny transistors which act as the fundamental building blocks. A large chunk of the transistor budget is dedicated to holding data (caches) and another is reserved for the computational units that process it (cores). In addition, there are separate chips just dedicated to storing data (memory). Think of the data as books, the cache as a desk and the memory as a library. If you need a certain book, finding it on your desk as opposed to visiting the library saves both time and effort.

For an average user higher performance means faster execution and lower cost. In the past this has primarily been achieved by shrinking transistors to fit more in a given area and operating them at higher frequencies. Unfortunately, we're reaching atomic transistor dimensions which changes the physical properties of the transistors making this approach infeasible. This has paved the way for multi-core processing which achieves higher aggregate performance by running applications concurrently.

In a multi-core processor, sharing of cache space by concurrently running applications can lead to conflicts and become a problem, in the same way as your desk would become awfully crowded if shared by too many people. With enough desk intruders you'd be running back and forth to the library. This has fuelled the need to have more optimized caching strategies. One solution to this problem is through partitioning, which is akin to setting up rules for what parts of the desk each person can use, how one can share books that multiple people need and what happens when someone is happy with just a few books while others need troves of them. However, determining appropriate space allocation for each person over time is challenging.

In addition to performance, security is also of paramount importance. Recent discoveries have shown that many microprocessor optimizations which aim to improve performance, such as caching, lead to new security vulnerabilities. Attacks exploiting these vulnerabilities can reveal secrets, such as cryptographic keys, which can have devastating consequences. In the desk analogy, this would be when information about different people's books and reading habits can be gathered by observing the shared desk. The need to protect this information complicates the sharing of desk space.

This thesis tackles the issues of improving the performance and security of multi-core processors with main focus on the cache and the memory system.