A FAIR AND SECURE CACHE ARCHITECTURE FOR MULTI-THREADED PROCESSORS

by

Sercan Sarı

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2019

A FAIR AND SECURE CACHE ARCHITECTURE FOR MULTI-THREADED PROCESSORS

APPROVED BY:

Assist. Prof. Dr. Onur Demir ....[illegible]............

(Thesis Supervisor)
(Yeditepe University)

Assoc. Prof. Dr. Gürhan Küçük ....[illegible]............

(Thesis Co-Supervisor)
(Yeditepe University)

Prof. Dr. Fatih Uğurdağ ....[illegible]............
(Ozyegin University)

Assist. Prof. Dr. Alp Arslan Bayrakçi ....[illegible]............
(Gebze Technical University)

Assist. Prof. Dr. Gökhan Şahin ....[illegible]............
(Yeditepe University)

DATE OF APPROVAL: .... /.... /2019

ACKNOWLEDGEMENTS

First of all, I would like to express my deep gratitude to Professor Onur Demir and Professor Gürhan Küçük, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I also would like to thank jury members for their guidance and interest.

Since the doors of the professors are always open to us, the pursuit of knowledge throughout my life becomes one of the purposes of life.

Most especially, I would like to thank my father Veli, my mother Ayşen, and my wife Burçin, for their endless love and support, which makes everything beautiful and meaningful.

ABSTRACT

A FAIR AND SECURE CACHE ARCHITECTURE FOR MULTI-THREADED PROCESSORS

Hardware security gained more attention due to the widespread use of cloud computing and remote execution, where multiple executions share a computer's resources. It is possible to extract confidential information such as cryptographic keys through cache-based side-channel attacks as in Meltdown and Spectre attacks, and as a result, secure cache architectures have become one of the hot research topics in the computer architecture field, today. These architectures come with an inevitable performance penalty since there is always an overhead for hiding information from the attackers. Subsequently, the performance degradation is traded off with the improvement in security. In this thesis, we analyze cache-based side-channel attacks, and the performance deterioration of the existing architectures and come up with a new solution which improves the fairness of the general framework. We propose a secure cache mechanism that respects fairness among the competing threads within a processor. We evaluate FairSDP architecture in 4-threaded and 8-threaded processors. As a result, we show that we can achieve up to 8.7 percent performance improvement over the baseline and 9.2 percent better performance compared to the static partitioning on the average, in an 8-threaded system. We also achieve almost identical results in terms of the fairness metric compared to a non-secure dynamic cache partitioning scheme.

ÖZET

ÇOKLU İŞPARÇACIKLI İŞLEMCİLER İÇİN ADALETLİ VE GÜVENLİ BİR ÖNBELLEK MİMARİSİ

Bulut bilgi işlemin yaygın kullanımı ve bir bilgisayarın kaynaklarının birden fazla bilgisayar tarafından paylaşıldığı uzaktan çalıştırma nedeniyle donanım güvenliği daha fazla önem kazanmaya başlamıştır. Meltdown ve Spectre saldırılarında olduğu gibi önbellek tabanlı yan kanal saldırıları yoluyla şifreleme anahtarları gibi gizli bilgileri çıkarmak mümkündür. Sonuç olarak, güvenli önbellek mimarileri üzerine yapılan çalışmalar daha derin bir odak noktası haline geldi. Güvenli önbellek mimarileri kaçınılmaz bir performans cezasıyla birlikte gelir, çünkü her zaman saldırganlardan bilgi gizleme yükü vardır. Performansın düşmesi güvenlikteki iyileşme ile birlikte işlem görmektedir. Bu tez ile birlikte, mevcut mimarilerin performans bozulmalarını analiz ediyoruz ve genel çerçevenin adaletliliğini artıran yeni bir çözüm sunuyoruz. Bir işlemci içindeki rakip işparçacıkları arasında adalete saygı duyan güvenli bir önbellek mekanizması öneriyoruz. FairSDP mimarisini 4 dişli ve 8 dişli işlemcilerde değerlendiriyoruz. Sonuç olarak, 8 dişli bir sistemde ortalama bazda yüzde 8.7'ye varan performans artışı ve ortalama statik bölümlemeyle karşılaştırıldığında yüzde 9.2 daha iyi performans sağlayabileceğimizi gösteriyoruz. Ayrıca, güvenli olmayan dinamik önbellek bölümleme şemasına kıyasla, adalet ölçüsü açısından neredeyse aynı sonuçları elde ediyoruz.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

FairSDP	Fair and secure dynamic cache partitioning
H	High
IPC	Instruction per cycle
IQ	Issue queue
L	Low
L2	Level two of cache
LLBC	Low-Latency Block-Cipher
LRU	Least recently used
LSQ	Load/Store queue
QoS	Quality of service
ROB	Re-order buffer
SMT	Simultaneous multi-threading
UDCP	Utility-based dynamic cache partitioning
UMON	Utility monitors

1. INTRODUCTION

In today's world, one of the major concerns for users of the computer system is protecting the confidentiality o f s ecret i n formation. T his c oncern h as r evealed i tself, e specially after the use of cloud computing. The users of cloud computing have to share their hardware with the other parties where there are not enough guarantees over the mutual trust.

When the hardware is isolated, meaning the users do not share common hardware for their computing, this concern might be a research area of cryptographic methods. These methods ensure that even the data has been compromised the contents are still not readable by adversaries [1]. However, when the hardware is shared, there might be breaches of data that depends on hardware-based attacks.

A recent hardware-based attack type is a side-channel attack. Side-channel attacks use information gained from the implementation of the computer system, rather than weakness in the implemented algorithm itself [2]. They collect side-channel information such as electromagnetic leaks, power consumption, timing information or even sound produced by a system. These types of attacks are so strong even cryptographic keys can be extracted. For example, the secret RSA key was extracted from a smart card by looking at the amount of power used during the decryption [3].

While side-channel attacks mostly focused on physical implementation in hardware, it can be argued that cache-based side-channel attacks have a more impact area than physical side-channel attacks. From this point on, we are going to refer cache-based side-channel attacks as cache attacks.While side-channel attacks mostly focused on physical implementation in hardware, it can be argued that cache-based side-channel attacks have a more impact area than physical side-channel attacks. From this point on, we are going to refer cache-based side-channel attacks as cache attacks. One of the reasons is cache exists almost all processors. Second of them is the cache attacks can be performed without physical proximity to the actual hardware. This is because physical side-channel attacks need physical access or proximity. Cache attacks simply use timing information for cache hits and misses. For example, if two processes share L2 cache, a process can validate every single location of the cache. After yielding to other processes, the attacker process can check the whole cache to see if any location is invalidated by measuring the access times of the locations. Since this timing information varies for all caches, cache attacks are hard to eliminate. In Section 2.1, a more comprehensive explanation of these attacks is provided.

There are some countermeasures to prevent this kind of attacks at different abstraction levels. Several recent work [4] propose secure caches against cache-based side-channel attacks. Almost all solutions rely on confiscating the timing information by two alternative methods: either partitioning the whole cache, such as PL cache [5], SecVerilog cache [4] and SecDCP cache [6], or randomizing the timing data, such as RP cache [7], CEASER cache [8], and NewCache [9].

The former method classifies data as secret or public and then restricts the caching locations of secret information so that the timing information is not visible to other parties. The latter method uses extra measures such as new lookup schemes that use encryption or hashing to randomize the memory-to-cache mapping.

Restricting processes on certain cache partitions reduces the performance significantly, in static cache partitioning [6]. To solve this issue, in [10], dynamic cache partitioning is proposed. With this approach, it is possible to change the partition size dynamically according to the demands of processes. However, this approach is still vulnerable to cache side-channel attacks. Previous work [6] proposes secure dynamic cache partitioning. Although this scheme eliminates cache side-channel attacks with dynamic cache partitioning, it still restricts sharing partitions of the cache among processes, even if they have the same security level. Therefore, each process can only interact with its own region of the cache.

While these techniques eliminate cache attacks, none of these techniques consider fairness among the processes. Fairness metric measures the equality of all threads and treating all threads equally is not straightforward while adjusting the partitions on the cache. Partitioning based solutions do a good job of isolating the secret information, while they suffer from performance loss which is visible by fairness metric which explained in detail in Section 4.3.

We re-evaluate and re-design the partition based solutions for the cache attacks by taking fairness as a measure of performance in multi-threaded systems. In this thesis, we present secure and fair cache sharing with dynamic partitioning, a new approach, FairSDP (Fair and Secure Dynamic Cache Partitioning) cache, which considers fairness among processes while eliminates cache attacks.

1.1. MOTIVATION

Due to the recent attacks Meltdown [11] and Spectre [12], it has been shown that it is possible to extract confidential information such as cryptographic keys through cache attacks. To prevent this vulnerability, a lot of secure cache architecture designs are proposed in academic literature. However, in earlier studies there is a lack of evaluation of secure cache architectures in terms of fairness metric. In this thesis, we are going to evaluate some of these secure cache architectures in terms of fairness metric. Also, we propose a secure and fair dynamic cache partitioning scheme. We evaluate the new cache architecture in 4-threaded and 8-threaded systems.

With cloud computing and multi-threading, the cache becomes a shared resource, and it is becoming more important to manage it even more efficiently and safely than before. For example, IBM's Power9 processor [13] supports up to 12 cores with 96 threads (SMT8) or up to 24 cores with 96 threads (SMT4). Consequently, the fairness and the quality of service (QoS) metrics also become invaluable metrics next to performance and power metrics on SMT-based platforms. There are already some studies to improve the fairness metric [10]. However, as we have mentioned earlier these solutions are still vulnerable to cache attacks.

As we have discussed, although there are ample research studies in the literature, none of them considers both security and fairness. Fairness may be ignored while focusing on performance. However, it leads to unfair situations among multi-threads. To cope with this problem we present a new solution, FairSDP.

We consider that confidential threads should have a priority over public threads because they are security-critical threads. We have observed that while adjusting partition size, just looking for public threads will cause unfair situations even in 2-threaded systems. Since observing the demand of confidential threads cause vulnerability to cache attacks, we reserve a static partition for confidential threads and the rest of the cache is adjusted to be shared between public threads. Thus, we achieved fair cache sharing and secure dynamic cache partitioning. We are going to discuss the new scheme in detail in Section 3.3.

1.2. SCOPE AND AIMS

Main objectives of this thesis are as follows:

- We aim to focus on cache attacks and secure cache architectures.
- We aim to perform a survey of current solutions for the cache attacks and model them using a simulator.
- We aim to investigate the impact of partitioning over multi-threaded systems by several different scenarios.
- We aim to propose a fairness based partitioning and analyze its impact over performance.
- We aim to provide our results from several experiments and highlight the use cases where fairness based solution outperform the other solutions.

The following subjects are out of our scope:

- The physical side-channel attacks are out of our scope in this work.
- The randomness based solutions are out of the scope of this work.

1.3. THESIS CONTRIBUTIONS

We make the following contributions:

- In this thesis, we propose a fair and secure dynamic cache partitioning scheme that prevents timing-channel attacks.

- In this thesis, we evaluate existing secure cache architectures against on cache attacks and evaluates the fairness metric.
- In this thesis, we evaluate our new architecture in 4-threaded and 8-threaded systems. Although recent works claim that their approach is expandable with four applications, these works do not evaluate their architecture in 4-threaded and 8-threaded systems.

2. BACKGROUND

In this chapter, we are going to discuss the background information necessary to understand cache attacks and secure cache architectures. In section 3.1, we will give a detailed understanding of CPU caches. Section 3.2 will explain various techniques of cache attacks. Then we focus on different secure cache architectures, which are closely related to this thesis.

2.1. CPU CACHES

A CPU cache is hardware that reduces the average cost of data accessing from memory. A CPU cache is smaller, faster and closer to the processor core compared to main memory. Frequently used data or the most recent data is stored in the processor cache. The reason for storing frequently used data or the most recent data in the processor cache is to eliminate latency of memory access delays. Since memory operations are a bottleneck compared to CPU operations a hierarchy of memory to make memory operations faster is employed as shown in Figure 3.1.

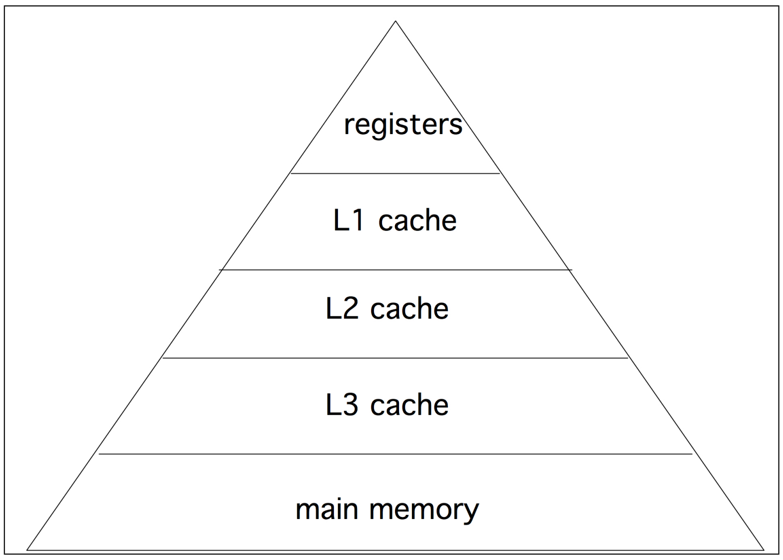

Figure 2.1. Memory hierarchy

The memory hierarchy is divided into multiple levels where each level that is closer to CPU is faster, smaller and more expensive. As seen in Figure 3.1, the first level cache(L1) is connected to the core logic directly because of performance issues. Although a Von Neumann architecture has a single cache for instruction and data, the Hardvard architecture uses two separate buses for instruction and data as seen in Figure 2.2. Since it has two caches, this allows memory operations to be performed on both buses and increases to performance.

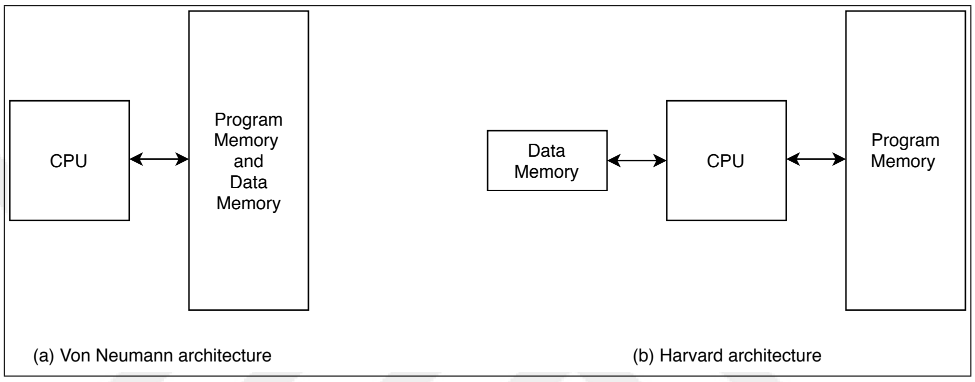

Figure 2.2. Von Neumann and Hardvard architectures

Programs prefer to use data and instructions close or equivalent to the addresses they have previously used. For example, if a program uses a loop, the same code is executed over and over again. Therefore, recent and often used data is stored to make the memory faster. This has significant importance for CPU performance. However, this cannot be possible in a core without caches. The cache just keeps some of the contents of the main memory. Therefore, it must store both data and the address of the data in the memory. When the processor needs to read or write a specific address, it will look into the cache. If the data is not found in the cache, the data is fetched from a lower level in the memory hierarchy which can be L2 cache or the main memory. Because of performance issues cache lines or blocks are moved at the same time since they are needed together due to locality. This reduces access times for subsequent loads and stores.

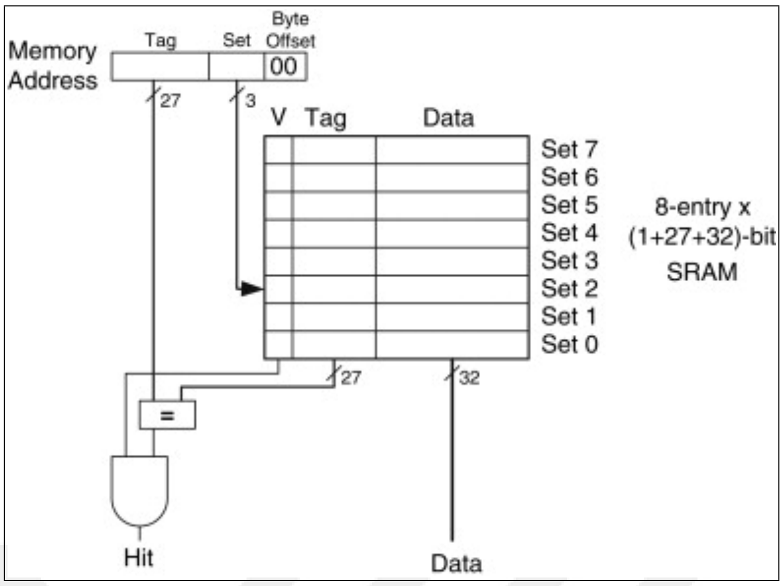

Figure 2.3. Direct-mapped cache [1]

2.1.1. Direct-mapped Cache

Within the cache, there are some basic types of the organization where the simplest one is a direct-mapped cache. Direct-mapped cache has each block mapped to exactly one cache memory location. If a line is previously occupied by a memory block, when a new block needs to be loaded, the old block is flushed.

2.1.2. Set-associative Caches

By providing N blocks in each set where data mapping to that set can be found, the number of conflicts reduces, which is known as N-way set-associative cache. Modern caches are organized in sets of cache lines. If the cache is fully associative, then the replacement policy can choose any entry in the cache to place the data. Set associative caches tends to have lower miss rates than direct mapped caches of the same capacity.

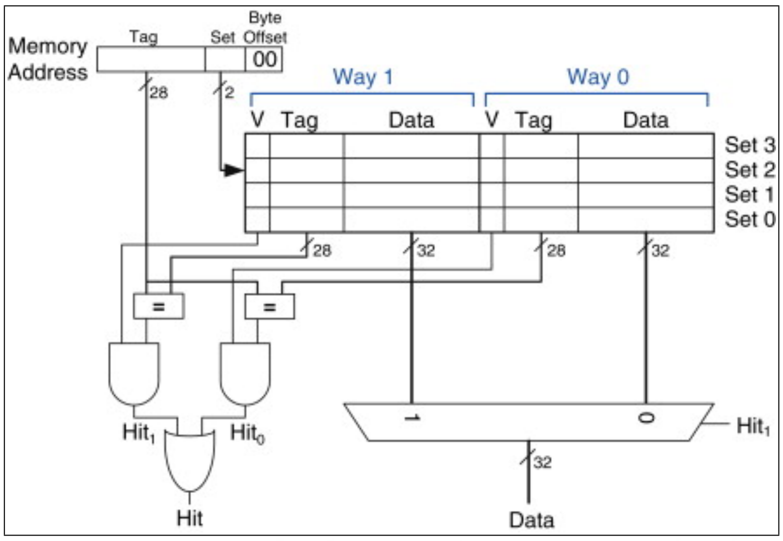

Figure 2.4. 2-way set-associative cache [1]

2.2. CACHE-BASED SIDE-CHANNEL ATTACKS

Cache attacks are serious vulnerability for all processors with caches. This essentially includes all computers from embedded systems to smartphones to cloud servers.

Cache attacks can be performed by using micro-architectural time differences when data is loaded from the cache instead of the main memory. This can lead to severe information leakage such as revealing of secret keys.

2.2.1. Classification of Cache-based Side-channel Attacks

The most common classes of cache attacks are access-driven attacks and timing-driven attacks [14]. These types of attacks use the information which is obtained by measuring hardware-based channels such as the access times. In the access-driven attacks, the attacker measures the impact of the victim's cache accesses to the attacker's own accesses so that the attacker can reveal the confidential information. In the time-driven accesses, the attacker can measure the execution time of the victim process.

In [15], it is argued that this classification can be improved for identifying main factors and countermeasures to prevent them. Based on the method of identifying the memory addresses, the attacks are classified as contention-based and reuse-based attacks. This new classification of all known cache attacks can be seen in Table 2.1 [15].

Table 2.1. Classification of cache-based side-channel attacks

	Contention-based	**Reuse-based**
Access-driven	Prime-Probe	Flush-Reload
Timing-driven	Evict-Time	Cache collision

2.2.1.1. Prime-Probe Attack

In [16] and [17], they proposed detailed explanations of memory accesses to the CPU cache and formalized two concepts which are Evict-Time and Prime-Probe. The main point of these two attacks is to determine which specific cache sets are accessed by a victim.

Algorithm 2.1. Prime-Probe

```
1: The attacker fills the specific cache sets with his/her own data.
2: The attacker waits for victim program.
3: The attacker determines which cache sets are still occupied.
```

Algorithm 2.1 explains the operations behind this type of attack. In the first step, one or more specific cache sets are filled by the attacker with his/her own data. In the second step, the attacker waits for the victim program. Finally, the same process is run again and time is measured by the attacker to load each set of attacker's data. Measuring the execution time for accessing the addresses the attacker used to fill the cache set in the first step causes a longer load time. Some of the attacker's cache lines in these cache sets will be evicted if some cache sets are used by the victim process during the interval. This leads to a much longer load time during the final step because of cache misses.

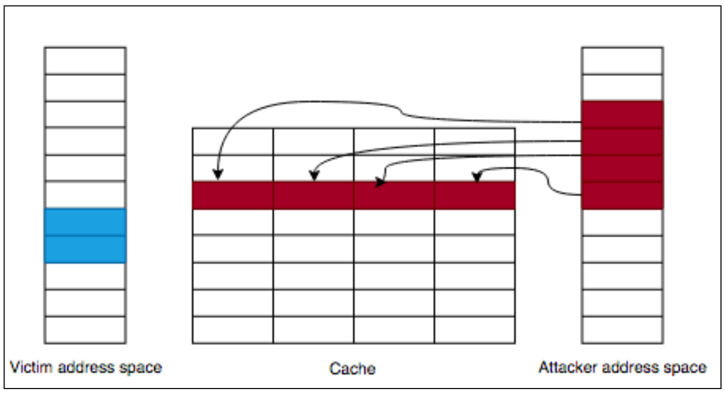

Figure 2.5. Prime: Fills the cache set

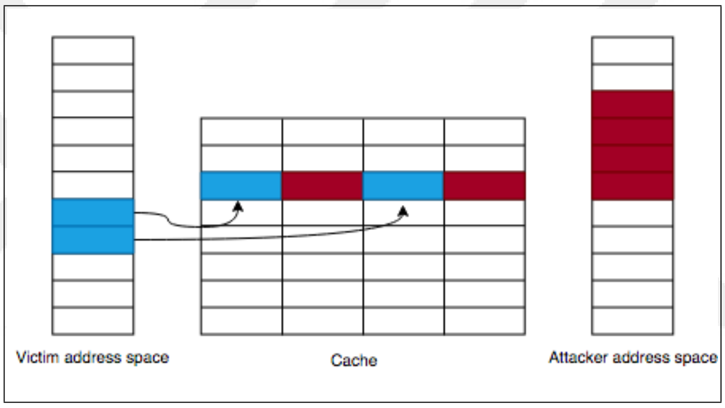

Figure 2.6. Waits for victim program

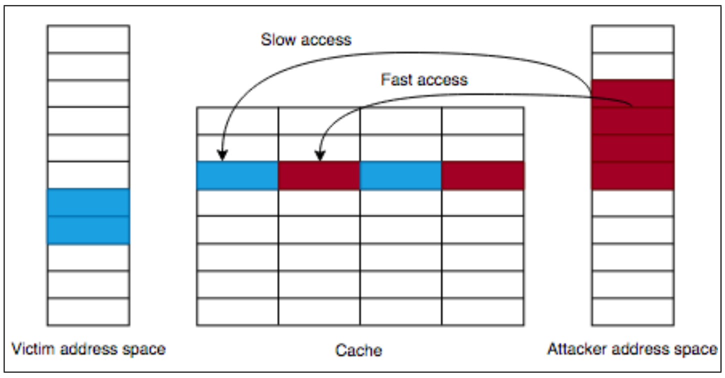

Figure 2.7. Probe: Determines cache occupation

2.2.1.2. Evict-Time Attack

Evict-Time is the second attack described in [17]. The key point is to determine which cache set is used during the victim's accesses. Therefore, these attacks considered contention-based attacks in [15].

Algorithm 2.2. Evict-Time

```
1: The attacker measures execution time of victim program.
2: The attacker evicts a specific cache set.
3: The attacker measures execution time of victim program again.
```

Algorithm 2.2 outlines the steps of this kind of attack. Firstly, the execution time of the victim process is measured by the attacker. In the second step, one specific c ache s et w ith the attacker's own data and victim's data is evicted in that cache set by the attacker. In the third step, the execution time of the victim program is measured again and used the timing difference between the two measurements by the attacker. Therefore, it can be inferred how much the specific cache set is used while the victim's program is running.

2.2.1.3. Flush-Reload Attack

In contrast to Prime-Probe and Evict-Time attacks Flush-Reload [18] attack technique requires shared some address space between the attacker and the victim process.

Algorithm 2.3. Flush-Reload

```
1: The attacker flushes a cache line(security-critical data from the cache).
2: The attacker waits for victim program.
3: The attacker checks whether the corresponding cache line from step 1 has been loaded
   by the victim's program.
```

Algorithm 3 summarizes the Flush-Reload attack technique. In the first step, the attacker flushes a cache line. In the second step, the attacker waits for the victim's program and the third step attacker accesses the same cache line flushed in the first step. Execution time is measured by the attacker to decide whether the access has been loaded from the cache or the main memory. Reload time of the attacker depends on whether the victim program accesses some security-critical data during the interval. The attacker will get a much lower reload time, because of the cache hit.

Gullasch et al. [18] proposed this attack technique to attack OpenSSL implementation of AES. Yarom et al. applied the first last-level cache attack using the Flush-Reload technique [19].

2.3. SECURE CACHE ARCHITECTURES

Designing secure caches is one of the solutions to mitigate cache attacks. In [15], it is argued that this solution provides much higher performance and often greater security than software solutions. While designing secure caches, there can be different approaches such as preserving victim cache lines, randomizing memory-to-cache mapping and encrypting memory-to-cache mapping.

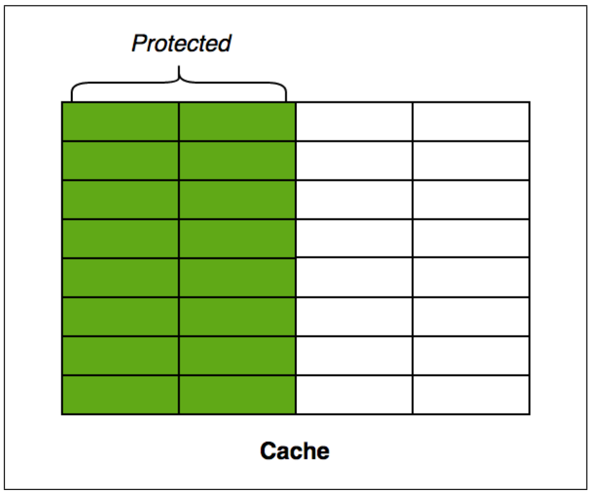

Figure 2.8. Partitioning the cache

2.3.1. Partitioning the Cache

Partitioning is intended to remove the cache contention between the attacker and the victim processes. For different processes, the cache is partitioned into different zones, and each process can access only the cache blocks in its zone. It is possible to achieve partitioning statically or dynamically.

2.3.1.1. Partition Locked (PL) Cache

Rather than statically partitioning, PL cache [5] locks a protected cache line into the cache and prevents it from being evicted through another process. By adding a process ID and a lock status bit to the tag-store entry, it extends each cache block. Load instructions are extended as (ld.lock/ld.unlock) and store instructions are extended as (sd.lock/sd.unlock) to control process ID and the lock status bit. This allows the compiler to control what data to lock. Replacement policy of PL cache as follows, for a cache hit, the routine procedure is performed and the process ID and the lock status bits are updated by the new instructions with locking or unlocking capability while the hit occurs. When a cache miss occurs, data that is locked cannot be evicted by data that is not locked, and even locked data cannot be evacuated from each other between different processes. In this situation, it either loads or stores the new data without caching.

2.3.1.2. Non-Monopolizable (NoMo) Cache

NoMo cache [20] reserves cache lines for active threads and prevents the eviction of reserved lines by other co-executing threads. They proposed a simple cache replacement policy modification that restricts an attacker to use no more than a predetermined number of lines in each set of an associative cache. As a result, the victim's data cannot be replaced by the attacker in the protected cache lines of each cache set. This prevents the attacker from observing those memory accesses through the side-channel. They guarantee that at least Y lines are reserved exclusively in each cache set for each running thread. Y's possible values are in the N/M range, where N is the cache's associativity, and the M is number of the simultaneous multi-threading (SMT) threads.

2.3.1.3. SecVerilog Cache

SecVerilog cache [4] statically divides cache lines between Low (L) and High (H) security levels. There is a timing label associated with each source program statement and represents confidential information and this label communicates with the hardware level. In [4], it says high partition cannot affect the timing label of L instruction, H instruction cannot change the low partition, and H partition cache lines cannot affect L partition cache lines. Basically, it statically divides cache ways between security levels L and H. H instructions are therefore unable to write lines to L partition. When the line is in H partition, it will result in cache miss for L instructions in cache read, but this line will be moved from H to L partition in order to preserve consistency. On the other hand, if the timing label of instruction is H, both H and L partitions will be searched. For cache writing, there is strictly a partition that H instruction can write only to H partition, and L instruction can write only to L partition except if the actual cache line is in H partition. Moreover, to avoid inconsistency, these are only one copy of data in the cache and TLB.

2.3.1.4. SecDCP Cache

SecDCP [6] relies on SecVerilog Cache [4]. Instead of statically partitioning, it partitions the cache ways dynamically. At least two security classes Low (L) and High (H) can be supported. SecDCP adjusts the ways assigned to L by monitoring percentage of cache misses that is reduced or increased when the partition size of L is increased or decreased. When adjusting cache ways, if there is a change from L's to H's, cache line is flushed before reusing it. However, if there is a change from H's to L's, H lines remain unmodified. This will not allow L to deduct the cache lines that are previously allocated to H.

2.3.2. Randomizing Memory-to-cache Mapping

The cache partitioning disadvantage is under utilization of the cache. Other processes cannot use cache lines that are locked or belong to a private partition, even if these cache lines are unused. Randomizing memory-to-cache mapping may prevent under utilization of the cache. The approach to randomization still allows cache contention, but no useful information can be extracted from the contention by the attacker. One of the drawbacks of randomizing memory-to-cache mapping is that it can cause huge mapping tables. Because in some cases it should be needed to map all entries in the cache.

2.3.2.1. RP Cache

RP cache [5, 7] is proposed by Wang and Lee to randomize memory-to-cache mapping [5, 7]. This strategy allows cache storage while randomizing the subsequent interference so that no helpful data about which row of cache has been evicted can be inferred. Permutation of memory-to-cache mapping is an essential procedure performed by the RPcache. In RPcache, a permutation table (PT) stores the memory-to-cache mapping for a process. The number of entries in the table is the same with the number of cache sets. Randomization of list entry data offers full randomization of the mapping of memory-to-cache. A P bit and ID fields are added to each cache line to specify the memory region to be protected from the attacker. When a cache hit, it behaves similar as a normal cache hit but P-bit of the cache line needs to be updated. If a cache miss occurs with data D of a cache set S, normal cache replacement policy chooses a line R in set S. If R and D belong to the same process and have the same protection bit normal replacement policy handles the procedure. If they do not have the same protection bit, a random data of a random cache set S' is evicted and D is accessed without accessing the cache. If they do not belong to the same processes, a random set S' is selected. The new line D replaces R' in S' and the memory-to-cache mapping for S' and S is swapped.

2.3.2.2. NewCache

NewCache [9] proposes dynamic randomization for memory-to-cache mapping. It introduces a ReMapping Table (RMT). While the mapping between this RMT and memory addresses adopts the direct mapped architecture, the mapping between the actual cache and the RMT is fully associative. The mapping from the index bits of the address to a real cache line is stored in ReMapping Table. They also propose the security-aware replacement algorithm to update and randomize the remapping table dynamically. Instead of holding a fixed set of cache lines, the proposed cache stores the most useful cache lines. Each cache block has a protection bit if it is and the process ID which are stored in RMT. It is very much alike to RP cache in terms of the cache replacement policy.

2.3.2.3. Random Fill Cache

Random Fill Cache [15] uses random filling t echnique t o d e-correlate c ache fi lls with memory access. It divides data accesses into three categories and can check whether the requested data belongs to a normal request or a random fill request by using new instructions. If data accesses are not security critical, Normal request will be utilize to execute routine replacement policy. When the confidential d ata accesses of v ictim occurs, a Nofill request is executed, the requested data access is brought without accessing the cache and a Random Fill request is executed and arbitrary data will be brought to the cache from the range of addresses. In [15], the authors claim that random spatial data filling d oes n ot affect performance.

2.3.2.4. CEASER Cache

CEASER cache [8] mitigates conflict-based c ache a ttacks u sing t he e ncrypted address and dynamic remapping. It accesses the cache with encrypted address and changes the encryption key periodically to prevent key reconstruction. CEASER cache employs Low-Latency Block-Cipher (LLBC) to convert physical line address to encrypted line address.

3. DESIGN

This chapter discusses the design of FairSDP architecture which is implemented in the M-Sim simulation suite for evaluation. Moreover, it describes the underlying utility based cache partitioning and utility monitoring in Section 3.1, static cache partitioning in Section 3.2, secure and dynamic partitioning in Section 3.3 and finally, fair and secure dynamic cache partitioning in Section 3.4.

3.1. UTILITY BASED CACHE PARTITIONING

Utility Based Cache Partitioning (UDCP) [10] is a run-time mechanism which partitions a shared cache between multiple threads according to the demands of the running threads. This scheme uses utility monitors (UMON) for each thread at run-time to fairly share cache resources between threads. UMON obtains utility information for each thread and the partitioning algorithm utilizes the information collected by the UMON to decide the number of ways to allocate in cache. Figure 3.1 shows the framework for UDCP cache. In order to monitor the utility information of a thread, the number of misses for all possible number of ways should be recorded. According to this information, optimal number of ways for each cache can be adjusted. In the subsections, building blocks of UDCP is explained in detail.

3.1.1. Utility Monitors

The number of cache misses for all possible number of ways should be tracked in order to monitor the utility information of an application. To compute the utility information for the 16-way cache in [10], they propose a monitoring circuit to track misses for the sixteen different cases according allocated ways of threads. They keep this information by having sixteen tag directories. Each tag category has the same number of sets as the shared cache. They use dynamic set sampling [21] to reduce hardware overhead of UMON. The main concept behind dynamic set sampling is that only a few pairs can approximate the cache's activity by sampling.

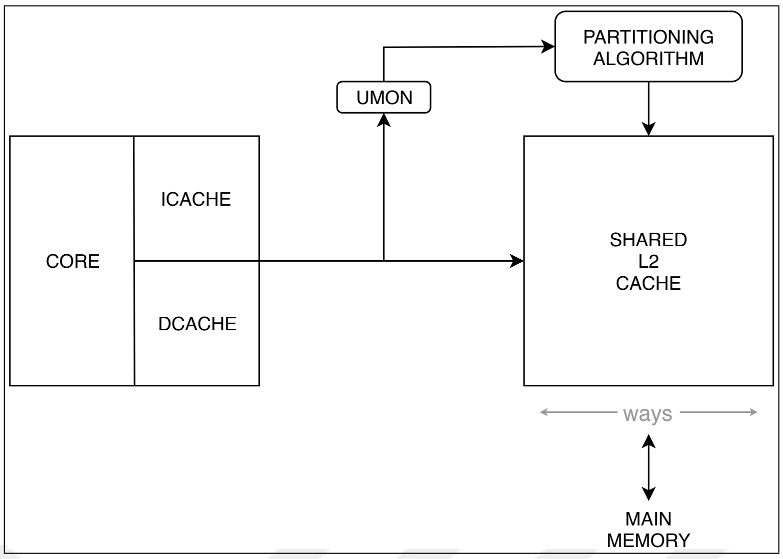

Figure 3.1. Framework for UDCP cache

3.1.2. Original Lookahead Algorithm

Lookahead Algorithm is proposed in [10]. With this algorithm, the marginal utility is considered for all possible number of blocks that the application can receive. Algorithm 3.2 shows the pseudo code for the lookahead algorithm. Here, N represents the cache associativity, the output vector, allocations, stores the maximum number of allocated cache ways to each process. The aim of this algorithm is to optimize the number of ways to be allocated to the threads using run-time statistics.

In [10], they calculate the maximum marginal utility (max mu) and the minimum number of blocks at which the max mu occurs. The calculation is repeated for each application at each iteration. Algorithm 3.1 shows the calculation process. The application with the highest value for max mu is assigned the number of blocks it needs to obtain max mu. Until all blocks are assigned, iterations are repeated. In each iteration, the lookahead algorithm can appoint a different number of blocks.

While UDCP achieves to divide the cache among the competing threads fairly, it does not consider security at all. We are going to propose both secure and fair cache architecture for multi-threaded processors.

Algorithm 3.1. Get_mu value and get_max_mu functions [10]

```
**Function** get_mu_value(p,w,y)
U = when the number of ways assigned to it change in misses for app p
increases from w to y
**return** U/(y-w)
**EndFunction**
**Function** get_max_mu(p,alloc,balance)
max_mu = 0
**for** j $= 0$; j $<$ balance; $j++$ **do**
    mu = get_mu_value(p, alloc, alloc+j)
    **if** mu $>$ max_mu **then**
        max_mu=mu
    **end if**
**end for**
**return** max_mu
**EndFunction**
```

Algorithm 3.2. Original lookahead algorithm [10]

```
balance $= N$
allocations[i] $= 0$ for each process i
**while** (balance) **do**
    **for** each app i, **do**
        alloc = allocations[i]
        max_mu[i] = get_max_mu(i, alloc, balance)
        blocks_req[i] = min blocks to get max_mu for i
    **end for**
    winner_app = app with max value of max_mu
    allocations[winner_app] += blocks_req[winner_app]
    balance – = blocks_req[winner_app]
**end while**
**return** allocations
```

3.2. STATIC PARTITIONING

In the static cache partitioning, the victim and attacker have different cache ways. This approach basically partitions the cache for the victim and the attacker. Although it is a cheap and effective method to mitigate cache side-channel attacks, restricting processes on certain cache partitions significantly reduces processor performance [22].

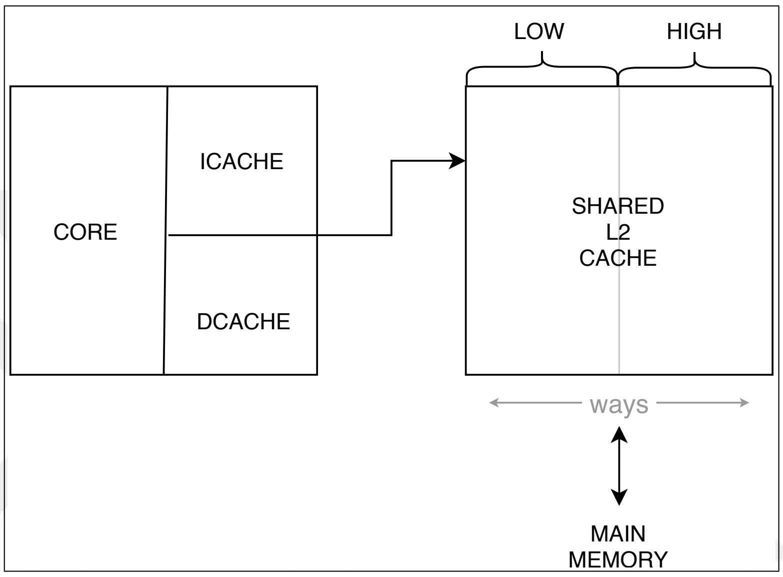

Figure 3.2. Framework for static partitioning

3.3. SECURE AND DYNAMIC PARTITIONING

As we already mentioned earlier, SecDCP relies on SecVerilog. Instead of statically partitioning, it partitions the cache ways dynamically. At least two security classes Low (L) and High (H) are supported. SecDCP adjusts the ways assigned to L by monitoring the percentage of cache misses that are reduced or increased when the partition size of L is increased or decreased. When adjusting cache ways, if there is a change from L's to H's, the cache line is flushed before reusing it. However, if there is a change from H's to L's, H lines remain unmodified. This will not allow L to deduct the cache lines that are previously allocated to H.

Although this scheme eliminates cache side-channel attacks with dynamic cache partitioning, it still restricts sharing partitions of the cache among processes, even if they have the same security level. Therefore, each process can only interact with its own region of the cache. This can create an unfair situation when both public processes and secure processes need the cache extensively. Although it reserves at least one cache way for the secure process, this does not guarantee fairness since each secure process may receive only one cache way. For example, when there are many cache- needed public processes running on the system, SecDCP distributes all the cache ways among those public processes leaving insufficient cache resources for the remaining secure processes. We try to eliminate this problem by the proposed scheme which is explained in Section 3.4. While they said that SecDCP is scalable, they do not evaluate their scheme on SMT systems with more than 2 threads.

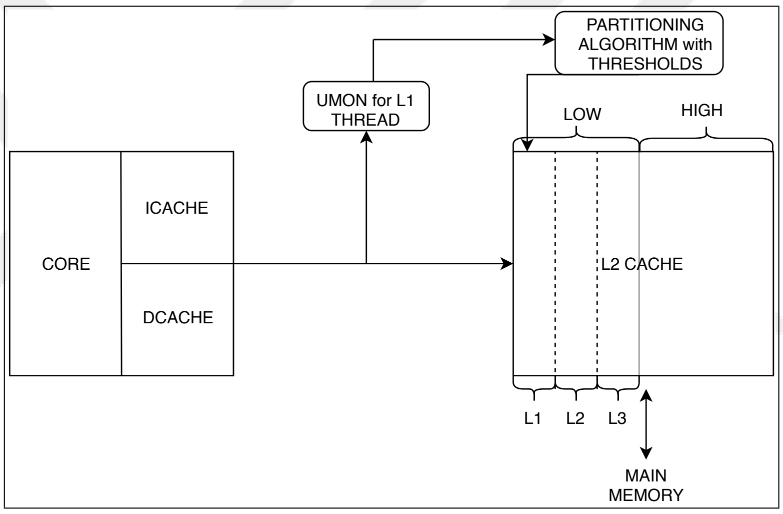

Figure 3.3. Framework for SecDCP cache

3.4. FAIR CACHE SHARING AND SECURE DYNAMIC PARTITIONING

FairSDP aims to eliminate cache side-channel attacks by using dynamic cache partitioning while considering the fairness among processes in a multi-threaded environment. As shown in Figure 3.1, FairSDP divides the shared cache into two regions. We assume the public data, which does not require any confidentiality, is labeled L (LOW) and secret data is labeled H (HIGH). The cache is partitioned such that no L data and H data can coincide and reside on the same cache line. Hence, it will be impossible for a thread to deduct the timing information on secret data. Besides, the partitioning has to be done cleverly so that both L and H data regions have a fair amount of space.

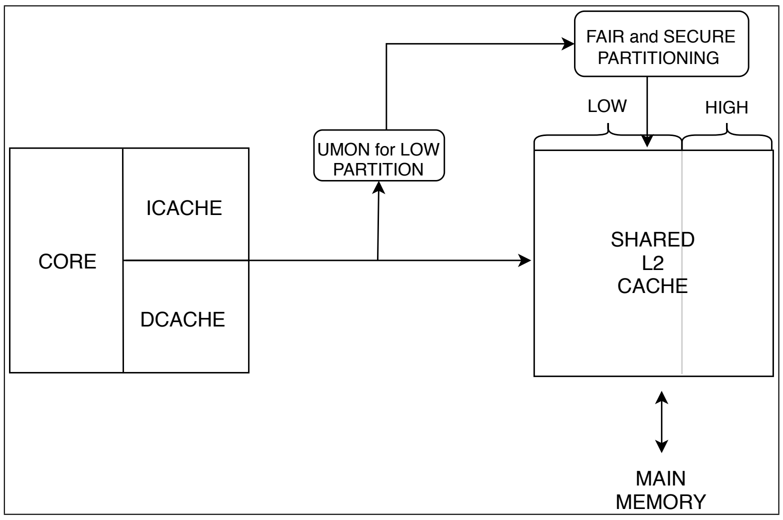

Figure 3.4. Framework for FairSDP cache

We assume processes either have H data or L data exclusively. We can collect cache statistics and predict the demand of L processes, however, it is not possible to do the same for H processes due to security reasons. L processes can adjust their partitions according to their instantaneous demand on the cache. Meanwhile, we reserve a fixed partition for an H process and for measuring the demand of L processes, we use utility monitors (UMON) described in [10]. Utility curves for each thread are generated by dedicated UMON structures. These curves enable us to calculate the number of misses for all possible cache sizes in cache-way granularity. Then, we use this information to adjust the partition space to make a fair sharing of resources.

This framework tries to eliminate the unfair situation that can occur when L processes and H processes are both memory-intensive. Adjusting cache partitioning only by collecting the demand of L processes and by disregarding the cache requirements of security-critical H processes could cause performance problems on running secure applications.

3.4.1. Proposed Lookahead Algorithm

Our algorithm is derived from the lookahead algorithm proposed in [10]. As seen in Algorithm 3.3, we reserve an m-way static partition for each H process, and, then, we apply the original lookahead algorithm for the remaining L processes to adjust optimal partitioning on the rest of the cache. Here, N represents the cache associativity, Hcount represents the number of H processes and Lcount represents the number of L processes. The output vector, alloc, stores the maximum number of allocated cache ways to each process. Note that the algorithm starts by allocating at least one cache way to each L process as in the original algorithm.

Algorithm 3.3: The proposed lookahead algorithm

```
1: balance = $N - (m * Hcount + 1 * Lcount)$
2: alloc[i] = $m$ for each high process i
3: alloc[j] = $1$ for each low process j
4: **call** original lookahead algorithm for all low processes
5: **return** alloc
```

3.4.2. Proposed Replacement Policy

We also made some changes on the replacement policy to ensure secure and fair partitioning, as shown in Algorithm 3.4. We add a bit to the tag-store entry of each line to define the core that installed the line in the cache to implement the way partitioning. On a cache miss, first, we count the number of cache lines of the miss pending process i on the current cache set, and store it in the occupancy vector. If the process is an L process and its occupancy on that set is less than the number of lines allocated to the process, then the Least Recently Used (LRU) line among all the lines that do not belong to the process is evicted. If this number reaches its maximum value or if it is an H process, the LRU line among all the lines of the miss causing process is evicted.

Algorithm 3.4 : The proposed replacement policy

```
1: occupancy[i] = $Count(i, set)$
2: **if** all cache lines are valid **then**
3:     **if** process = L **and** occupancy[i] $<$ alloc[i] **then**
4:         evict LRU line j which belongs to other L processes
5:     **else**
6:         evict LRU line j which belongs to current process i
7:     **end if**
8: **else**
9:     **if** occupancy[i] $<$ alloc[i] **then**
10:        select an invalid line j
11:    **else**
12:        evict LRU line j which belongs to current process i
13:    **end if**
14: **end if**
15: insert new cache line on line j
```

We propose FairSDP cache, which prevents cache timing-channel attacks while still considering fairness among threads. In this study, we focus on the design and evaluation of the partition-based solutions targeting cache attacks by taking performance and fairness as two major metrics. Although our proposed mechanism has similarities with the SecDCP, our main differences over the current state of the art [6] are as follows:

- We evaluate our new mechanism in a 4-threaded and in an 8-threaded SMT system. Although recent works claim that their approach is scalable, they do not evaluate their scheme on SMT systems with more than 2 threads.
- Although SecDCP scheme eliminates cache side-channel attacks with dynamic cache partitioning, it still restricts sharing partitions of the cache among processes, even if they are at the same security level. Therefore, each process can only interact with its own region of the cache. In FairSDP, we relax this restriction by allowing threads in the same security level to share cache regions among them.

- We try to eliminate the unfair situation that can occur when both public processes and the secure process need the cache extensively. SecDCP reserves at least one way for the secure process. However, this does not guarantee fairness since each secure process may receive only one cache way. For instance, when there are many cache-hungry public processes running on the system, SecDCP distributes all the cache ways among those public processes leaving insufficient cache resources for the remaining secure processes. In contrast, FairSDP reserves a fixed number of cache ways to each of the secure processes to guarantee fairness and, hence, achieves similar fairness results as reported by the UDCP scheme.

4. EXPERIMENTAL METHODOLOGY

Throughout this chapter, we are going to explain the experimental methodology that we use while evaluating the proposed scheme. In Section 4.1, processor specifications that we test our proposed design can be seen. Benchmarks that we use to test the performance of the proposed design are described in Section 4.2. Lastly, in Section 4.3, metrics that we use to evaluate and compare the proposed scheme is described.

4.1. PROCESSOR SPECIFICATIONS

We use an architecture simulator, M-Sim [23], to evaluate secure cache architectures and our design. M-Sim is a micro-architectural simulation environment. It provides to measure processor performance under both single and multi-threaded with a detailed cycle-accurate model for the key pipeline structures. It is based on SimpleScalar [24]. SimpleScalar is a set of tools that model the CPU, cache and memory hierarchy of a virtual computer system. Using SimpleScalar tools, users are able to create architecture systems for modeling that simulate real benchmarks running on a range of modern processors and systems.

4.2. BENCHMARKS

We use 8 SPEC CPU2006 benchmarks to form our test cases. The selected benchmarks are hmmer, libquantum, mcf, milc, namd, omnettpp, sjeng and zeusmp.

We tested all possible combinations of 8 benchmarks(70 workloads for 4-thread mixtures). We fast-forward the simulation 10 million instructions and the simulation terminates after 250 million instructions.

Table 4.1. Processor specifications

Decode / Issue / Commit bandwidth	8 instructions / cycle
Register file	256 int point, 256 floating point
Reorder buffer (ROB) size	64 entries
Issue Queue (IQ) size	40 entries
Load / Store queue (LSQ) size	32 entries
Number of integer ALUs	6
Number of integer multiplier / dividers	3
Number of floating point ALUs	3
Number of floating point multiplier / dividers	3
L1 instruction cache	2-way
L1 data cache	4-way
L2 unified cache	8-way
L1 cache hit time	1 cycle
L2 cache hit time	20 cycles
Main memory access time	300 cycles

4.3. METRICS

We use some metrics to evaluate and compare the proposed scheme. These metrics are IPC, Weighted Speedup and Harmonic Mean of IPC. IPC metric is used to indicate the throughput of the processor but IPC alone may not be fair to a low IPC thread. The formula calculating IPC of N threads can be seen in Equation 4.1.

$$IPC = \sum \frac{Number\ of\ instructions\ committed\ by\ thread_i}{Number\ of\ cycles} \tag{4.1}$$

The harmonic mean of IPC is the second metric that we use in our study. It takes the harmonic mean of IPCs as shown in Equation 4.2. This metric provides a fair balance of throughput because both the speed of threads and the standalone performance of each thread affect it.

$$Harmonic\ mean\ of\ IPC = N/\sum \frac{IPC_i\, of\, UDCP}{IPC_i\, of\, FairSDP} \tag{4.2}$$

The third metric that we use is Weighted speedup. With this metric, reduction in execution time is indicated.

$$Weighted\ Speedup\ = \sum \frac{IPC_i}{singleIPC_i} \tag{4.3}$$

5. RESULTS AND DISCUSSION

In this chapter, we are going to present and discuss the test results. In Section 5.1, performance on throughput metric will be discussed, we will also argue about performance on fairness metric in Section 5.2, lastly, in Section 5.3 we will present weighted speedup metrics.

5.1. PERFORMANCE ON THROUGHPUT METRIC

Figure 5.1 shows the overall IPC sum in a 4-threaded system. While we provide a secure cache, we show that our proposed mechanism has almost the same performance as the UDCP. As shown in Figure 5.2, results indicate that our mechanism has 8.7 percent performance improvement over the baseline and 9.2 percent better performance compared to the static partitioning on the average, on an 8-threaded system. Workloads can be seen in Appendix A and Appendix B in detail.

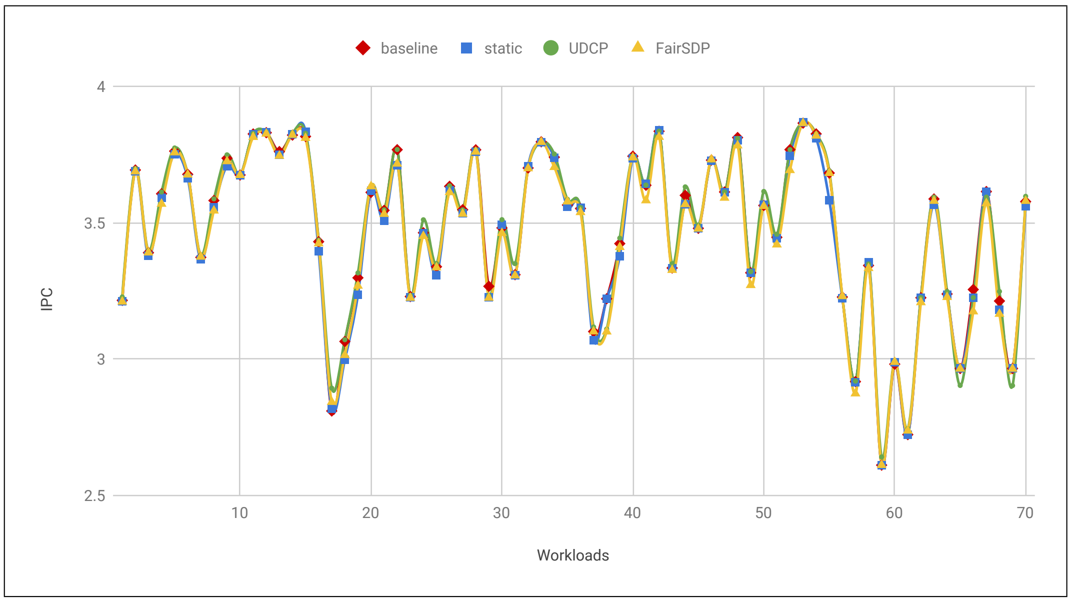

Figure 5.1. Overall IPC sum on a 4-threaded system

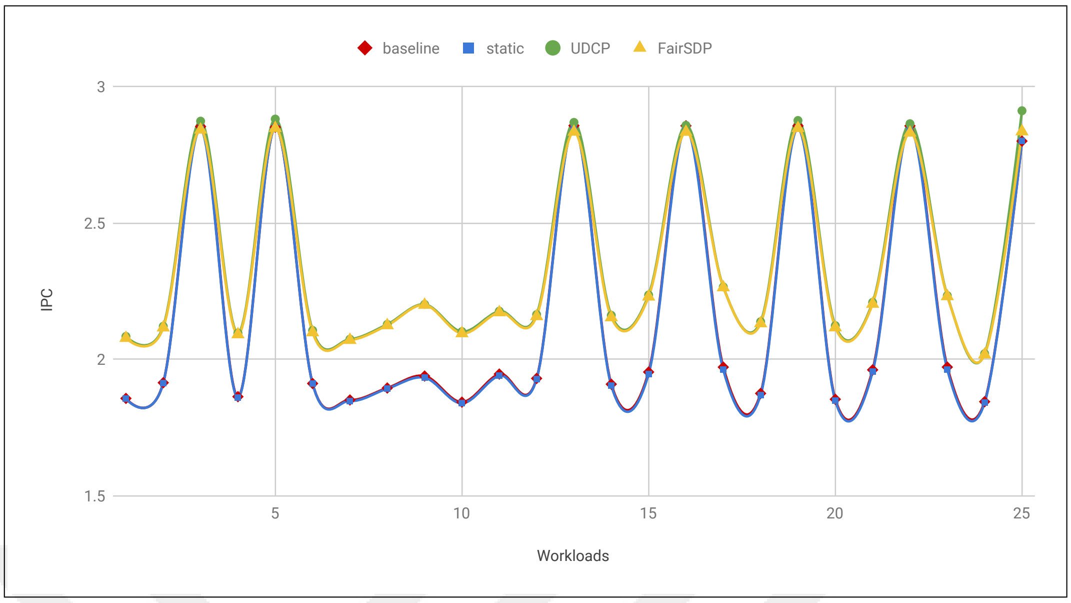

Figure 5.2. Overall IPC sum on an 8-threaded system

5.2. PERFORMANCE ON FAIRNESS METRIC

We compare FairSDP with UDCP[10] in terms of the fairness metric. Figure 5.3 and Figure 5.4 show the performance of FairSDP relative to UDCP on 4-threaded and 8-threaded systems. In [25], the harmonic mean of the normalized IPCs is claimed to consider both fairness and performance. UDCP claims around 10 percent fairness improvement over the static partitioning approach, and our results show that our proposed mechanism is almost as fair as the UDCP scheme while still providing a secure cache.

We also compare SecDCP with UDCP in terms of the fairness metric in order to show how much we can improve fairness among threads. Figure 5 shows the performance of SecDCP relative to UDCP on an 8-threaded system. As we have mentioned in the paragraph above, UDCP claims around 10 percent fairness improvement over the static partitioning approach. However, SecDCP results do not show that it is fair as FairSDP as seen in Figure 5.5.

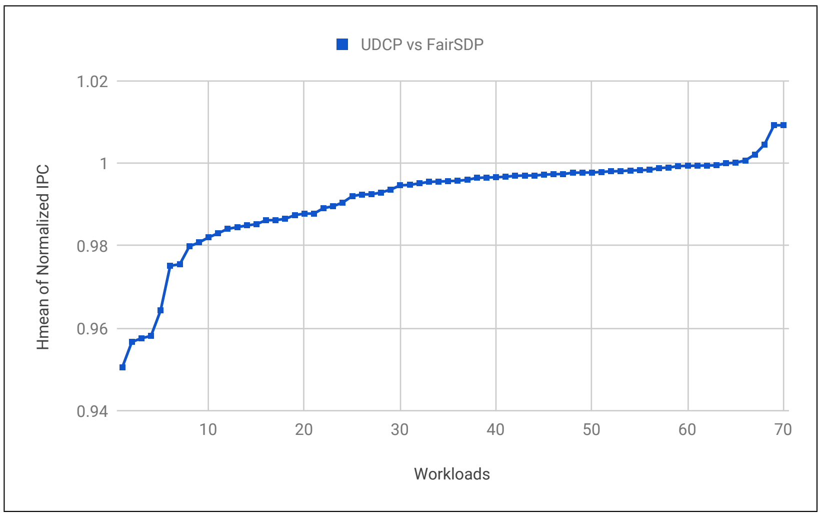

Figure 5.3. FairSDP on fairness metric on a 4-threaded system

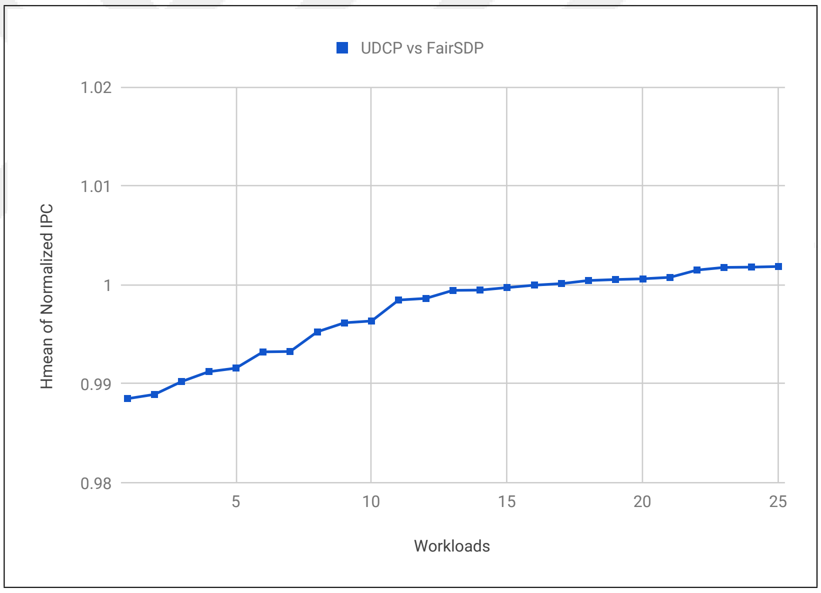

Figure 5.4. FairSDP on fairness metric on an 8-threaded system

5.3. PERFORMANCE ON WEIGHTED SPEEDUP METRIC

Figure 5.5 and Figure 5.6 summarize the performance results across 70 workloads on a 4-threaded system and 25 workloads on an 8-threaded system, respectively. We normalized the throughput results to a baseline processor with an 8-way set-associative cache using the LRU replacement policy. Workloads are sorted according to their performance over the baseline processor.

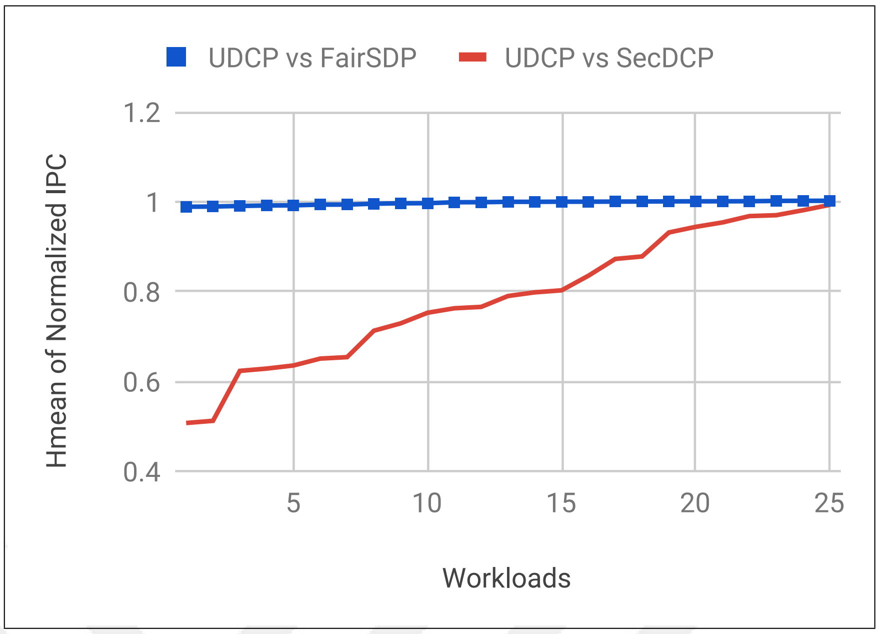

Figure 5.5. FairSDP and SecDCP on fairness metric on an 8-threaded system

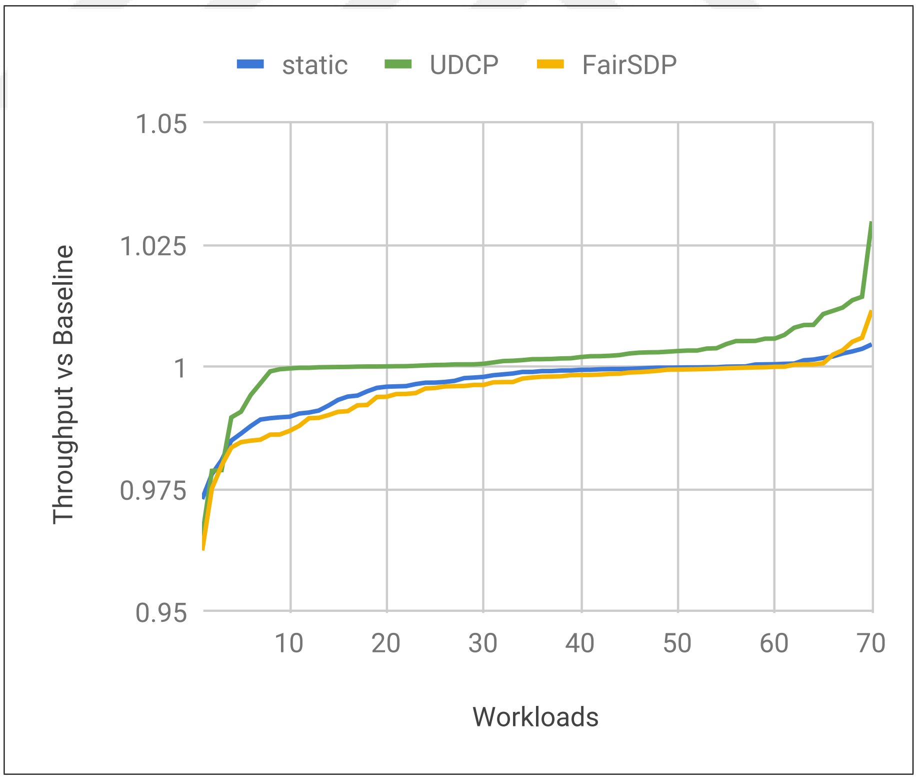

Figure 5.6. Results over all 70 workloads on a 4-threaded system

While there is no significant difference on performance in a 4-threaded system as shown in Figure 5.6, in an 8-threaded processor, we encounter a win-win situation for dynamic partitioning mechanisms. We surpass the performance of the static partitioning while still protecting the cache against time-channel attacks. As a result, we show that we can achieve up to 8.7 percent performance improvement over the baseline and 9.2 percent better performance compared to the static partitioning on the average, on an 8-threaded system.

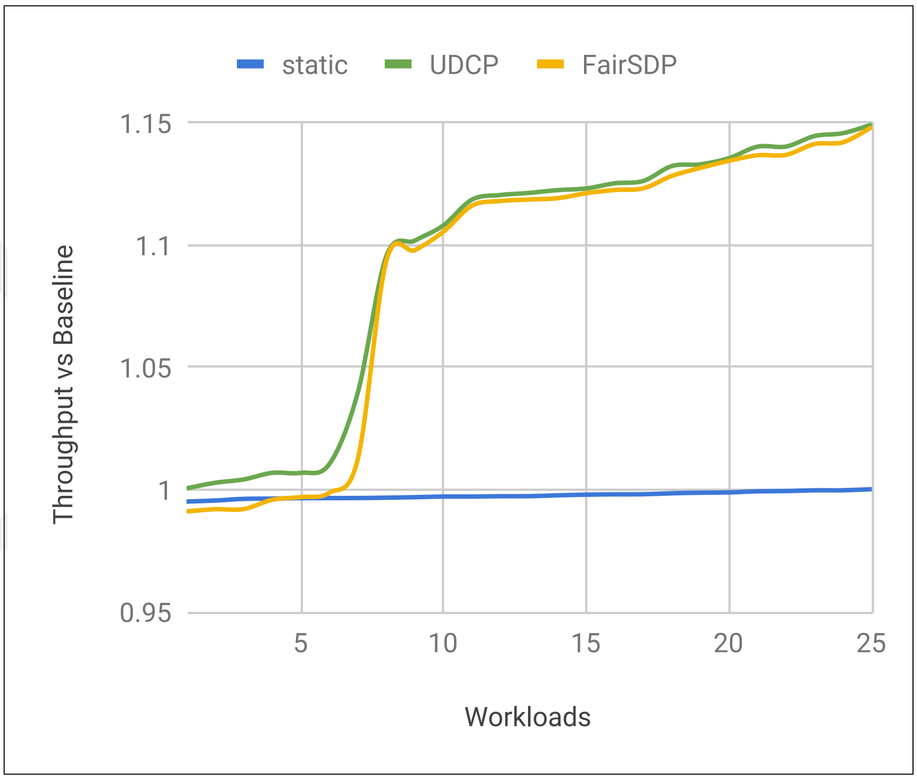

Figure 5.7. Results over 25 workloads on an 8-threaded system

We also applied a t-test to IPC results to make sure whether our sampling workloads represent all benchmarks. Looking at the t-test results, we observe IPC results of workloads on the FairSDP cache is significantly different from the baseline ($p < 0.0001$), static partitioning ($p < 0.0001$) and UDCP ($p < 0.001$). These results support that our sampling workloads represent the whole system.

6. CONCLUSIONS AND FUTURE WORK

Cache attacks are serious vulnerability for most of the processors with caches even in today's popular products. This essentially includes all computers from embedded systems to cloud servers. Cache attacks can be performed by using micro-architectural time differences when data is loaded from the cache instead of the main memory. This can cause serious information leakage such as revealing of secret keys.

There are some countermeasures to prevent this kind of attacks at different abstraction levels. Several recent work [4, 5, 6, 7, 8, 20] propose secure caches against cache attacks. Almost all solutions depend on confiscating the timing information by two methods: either partition the whole cache or randomizing timing data so that it will not reveal any secret information.

It is argued that restricting processes on certain cache partitions reduces the performance significantly, in static cache partitioning [6]. To solve this issue, in [10], dynamic cache partitioning is proposed. With this approach, it is possible to change the partition size dynamically based on the demands of processes. However, this approach is still vulnerable to cache attacks. Previous work [6] proposes secure dynamic cache partitioning. Although this scheme eliminates cache attacks with dynamic cache partitioning, it still restricts sharing partitions of the cache among processes, even if they have the same security level. Therefore, each process can only interact with its own region of the cache.

To solve this issue, we present a fair and a secure cache sharing mechanism with dynamic partitioning, FairSDP cache, which gives a fair way of eliminating cache attacks. Our design is based on a mechanism which dynamically allocates partitions during run-time respecting fairness parameters which are calculated using UMON. The proposed architecture is implemented through a simulation package and its performance is evaluated by using a number of benchmarks.

We tried to approach the performance and fairness problems of the secure cache architectures that dynamically allocate cache to competing threads. We have used the run-time statistics that are collected by UMON to decide how to allocate the cache partitions among competing threads in a more fair manner.

With this framework, we try to eliminate the unfair situation that can occur when public processes and secure processes are both memory-intensive. We achieve up to 8.7 percent performance improvement over the baseline and 9.2 percent better performance compared to the static partitioning on the average.

A non-secure Utility-based Dynamic Cache Partitioning (UDCP) scheme reports around 10 percent improvement over the static partitioning scheme in terms of fairness, on the average. We show that we achieve similar fairness results over the UDCP scheme, and, therefore, we claim that FairSDP cache satisfies our performance and security expectations.

Although this thesis has mentioned about performance problems related to secure cache architectures, different extensions can be worked in the future research. We summarize them below:

- FairSDP can be extended to support more than one high thread.
- Rather than using a simulation environment, secure cache architectures can be implemented on hardware such as RISC-V Rocket Chip architecture.
- Designing secure cache architectures is not the only solution to mitigate cache-based side-channel attacks. To prevent this kind of attacks, other countermeasures can be applied in different abstraction levels.

REFERENCES

1. Harris D, Harris S. *Digital design and computer architecture.* San Francisco: Morgan Kaufmann; 2010.

2. Rebeiro C, Mukhopadhyay D, Bhattacharya S. *Timing channels in cryptography: a micro-architectural perspective*. New York: Springer; 2014.

3. Messerges TS, Dabbish EA, Sloan RH. Investigations of Power Analysis Attacks on Smartcards. *Smartcard*. 1999;99:151–161.

4. Zhang D, Wang Y, Suh GE, Myers AC. A Hardware Design Language for timing-sensitive information-flow security. *ACM SIGARCH Computer Architecture News*. 2015;43(1):503–516.

5. Wang Z, Lee RB. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*. 2007;35(2):494–505.

6. Wang Y, Ferraiuolo A, Zhang D, Myers AC, Suh GE. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM; 2016. p. 74.

7. Wang Z, Lee RB. Covert and side channels due to processor architecture. In: *22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE; 2006. pp. 473–482.

8. Qureshi MK. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE; 2018. pp. 775–787.

9. Wang Z, Lee RB. A novel cache architecture with enhanced performance and security. In: *Proceedings of the 41st annual IEEE/ACM International Symposium*

on Microarchitecture. IEEE Computer Society; 2008. pp. 83–93.

10. Qureshi MK, Patt YN. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE; 2006. pp. 423–432.

11. Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, et al. Meltdown: Reading Kernel Memory from User Space. In: *27th USENIX Security Symposium (USENIX Security 18)*; 2018.

12. Kocher P, Horn J, Fogh A, Genkin D, Gruss D, et al. Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*; 2019.

13. Sadasivam SK, Thompto BW, Kalla R, Starke WJ. IBM Power9 processor architecture. *IEEE Micro*. 2017;37(2):40–51.

14. Page D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*. 2002(169).

15. Liu F, Lee RB. Random fill cache architecture. In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society; 2014. pp. 203–215.

16. Percival C. *Cache missing for fun and profit.* Ottowa: BSDCan; 2005.

17. Osvik DA, Shamir A, Tromer E. Cache attacks and countermeasures: the case of AES. In: *Cryptographers' Track at the RSA Conference*. Springer; 2006. pp. 1–20.

18. Gullasch D, Bangerter E, Krenn S. Cache games–Bringing access-based cache attacks on AES to practice. In: *2011 IEEE Symposium on Security and Privacy*. IEEE; 2011. pp. 490–505.

19. Yarom Y, Falkner K. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*; 2014. pp. 719–732.

20. Domnitser L, Jaleel A, Loew J, Abu-Ghazaleh N, Ponomarev D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*. 2012;8(4):35.

21. Qureshi MK, Lynch DN, Mutlu O, Patt YN. A case for MLP-aware cache replacement. In: *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE; 2006. pp. 167–178.

22. Sari S, Demir O, Kucuk G. FairSDP: Fair and Secure Dynamic Cache Partitioning. In: *4th International Conference on Computer Science and Engineering (UBMK*). IEEE; 2019. pp. 469-474.

23. Sharkey J, Ponomarev A, Ahose K. A-sim: A flexible, multithreaded architectural simulation environment. *Techenical teport, Department of Computer Science, State University of New York At Binghamton*. 2005.

24. Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. *Computer*. 2002;(2):59–67.

25. Luo K, Gummaraju J, Franklin M. Balancing throughput and fairness in SMT processors. In: *International Symposium on Performance Analysis of Systems and Software*. ISPASS. IEEE; 2001. pp. 164–171.

APPENDIX A: IPC RESULTS OF 4-THREADED WORKLOADS

Table A.1. Results of 4-threaded workloads

Workloads	static	UDCP	FairSDP
hmmer libquantum mcf milc	3.2167	3.2274	3.2098
hmmer libquantum mcf namd	3.696	3.6988	3.6879
hmmer libquantum mcf omnetpp	3.3854	3.3975	3.3902
hmmer libquantum mcf sjeng	3.57	3.6127	3.5689
hmmer libquantum mcf zeusmp	3.7652	3.7742	3.7576
hmmer libquantum milc namd	3.6796	3.6777	3.6762
hmmer libquantum milc omnetpp	3.3735	3.3812	3.3746
hmmer libquantum milc sjeng	3.5478	3.5874	3.544
hmmer libquantum milc zeusmp	3.7347	3.749	3.7251
hmmer libquantum namd omnetpp	3.674	3.6762	3.6744
hmmer libquantum namd sjeng	3.8282	3.8272	3.8141
hmmer libquantum namd zeusmp	3.8331	3.8327	3.8263
hmmer libquantum omnetpp sjeng	3.7404	3.7487	3.7451
hmmer libquantum omnetpp zeusmp	3.8227	3.8306	3.822
hmmer libquantum sjeng zeusmp	3.8171	3.828	3.8106
hmmer mcf milc namd	3.4312	3.436	3.4238
hmmer mcf milc omnetpp	2.8062	2.8926	2.8415
hmmer mcf milc sjeng	3.0629	3.0704	3.0137
hmmer mcf milc zeusmp	3.2974	3.3154	3.2681
hmmer mcf namd omnetpp	3.6091	3.6352	3.6331

Table A.1. Continued

Workloads	**static**	**UDCP**	**FairSDP**
hmmer mcf namd sjeng	3.5497	3.5474	3.5316
hmmer mcf namd zeusmp	3.7194	3.7685	3.7161
hmmer mcf omnetpp sjeng	3.2304	3.2332	3.2237
hmmer mcf omnetpp zeusmp	3.4934	3.5106	3.4506
hmmer mcf sjeng zeusmp	3.3088	3.3498	3.3355
hmmer milc namd omnetpp	3.6323	3.6353	3.6114
hmmer milc namd sjeng	3.5497	3.5472	3.5322
hmmer milc namd zeusmp	3.7671	3.7683	3.7602
hmmer milc omnetpp sjeng	3.2305	3.2329	3.2239
hmmer milc omnetpp zeusmp	3.4485	3.511	3.4618
hmmer milc sjeng zeusmp	3.3389	3.3496	3.3059
hmmer namd omnetpp sjeng	3.7009	3.6999	3.6992
hmmer namd omnetpp zeusmp	3.797	3.7955	3.7959
hmmer namd sjeng zeusmp	3.7097	3.7519	3.7038
hmmer omnetpp sjeng zeusmp	3.5321	3.5859	3.5774
libquantum mcf milc namd	3.5554	3.5556	3.5382
libquantum mcf milc omnetpp	3.1015	3.117	3.1004
libquantum mcf milc sjeng	3.1128	3.1095	3.1001
libquantum mcf milc zeusmp	3.4316	3.4432	3.4101
libquantum mcf namd omnetpp	3.743	3.7446	3.7392
libquantum mcf namd sjeng	3.5946	3.6381	3.5816
libquantum mcf namd zeusmp	3.8251	3.8359	3.8146

Table A.1. Continued

Workloads	**static**	**UDCP**	**FairSDP**
libquantum mcf omnetpp sjeng	3.3322	3.3495	3.326
libquantum mcf omnetpp zeusmp	3.573	3.6314	3.5675
libquantum mcf sjeng zeusmp	3.4171	3.4912	3.4776
libquantum milc namd omnetpp	3.7284	3.7293	3.7319
libquantum milc namd sjeng	3.6133	3.6129	3.591
libquantum milc namd zeusmp	3.8079	3.8093	3.7831
libquantum milc omnetpp sjeng	3.3164	3.3204	3.2705
libquantum milc omnetpp zeusmp	3.5911	3.6146	3.5607
libquantum milc sjeng zeusmp	3.4273	3.4573	3.4195
libquantum namd omnetpp sjeng	3.7691	3.7682	3.6925
libquantum namd omnetpp zeusmp	3.8676	3.8683	3.8652
libquantum namd sjeng zeusmp	3.8255	3.8267	3.819
libquantum omnetpp sjeng zeusmp	3.6825	3.6906	3.6818
mcf milc namd omnetpp	3.2271	3.2287	3.2298
mcf milc namd sjeng	2.9168	2.9183	2.8728
mcf milc namd zeusmp	3.342	3.3484	3.3328
mcf milc omnetpp sjeng	2.6104	2.6402	2.61
mcf milc omnetpp zeusmp	2.9876	2.9921	2.9884
mcf milc sjeng zeusmp	2.7295	2.7441	2.7363
mcf namd omnetpp sjeng	3.2258	3.225	3.2075
mcf namd omnetpp zeusmp	3.5866	3.5937	3.5796
mcf namd sjeng zeusmp	3.2119	3.2474	3.2266
mcf omnetpp sjeng zeusmp	2.9645	2.9024	2.9636
milc namd omnetpp sjeng	3.2256	3.225	3.1739
milc namd omnetpp zeusmp	3.5656	3.5938	3.571
milc namd sjeng zeusmp	3.2379	3.2475	3.1649
milc omnetpp sjeng zeusmp	2.9595	2.902	2.9634
namd omnetpp sjeng zeusmp	3.5597	3.5966	3.5794

APPENDIX B: IPC RESULTS OF 8-THREADED WORKLOADS

Table B.1. Results of 8-threaded workloads

Workloads	static	UDCP	FairSDP
hmmer libquantum mcf milc namd omnetpp zeusmp hmmer	1.8549	2.0836	2.0775
hmmer libquantum mcf milc namd sjeng omnetpp hmmer	1.9133	2.1207	2.1158
hmmer libquantum mcf milc namd sjeng zeusmp hmmer	2.8441	2.8731	2.8416
hmmer libquantum mcf milc namd zeusmp omnetpp hmmer	1.8598	2.096	2.0909
hmmer libquantum mcf milc namd zeusmp sjeng hmmer	2.8503	2.8808	2.8474
hmmer libquantum mcf milc omnetpp namd zeusmp hmmer	1.9098	2.1056	2.0981
hmmer libquantum mcf milc omnetpp namd sjeng hmmer	1.847	2.0735	2.0701
hmmer libquantum mcf milc omnetpp sjeng zeusmp hmmer	1.892	2.1278	2.1241
hmmer libquantum mcf milc omnetpp sjeng namd hmmer	1.9323	2.2011	2.199
hmmer libquantum mcf milc omnetpp zeusmp sjeng hmmer	1.8383	2.1012	2.0946
hmmer libquantum mcf milc omnetpp zeusmp namd hmmer	1.94	2.1763	2.1715
hmmer libquantum mcf milc sjeng namd omnetpp hmmer	1.9273	2.1637	2.1573
hmmer libquantum mcf milc sjeng namd zeusmp hmmer	2.8512	2.8686	2.8338
hmmer libquantum mcf milc sjeng omnetpp namd hmmer	1.9032	2.1609	2.1532
hmmer libquantum mcf milc sjeng omnetpp zeusmp hmmer	1.9462	2.2355	2.2291
hmmer libquantum mcf milc sjeng zeusmp namd hmmer	2.8457	2.8577	2.8335
hmmer libquantum mcf milc sjeng zeusmp omnetpp hmmer	1.9613	2.2653	2.2633
hmmer libquantum mcf milc zeusmp namd sjeng hmmer	1.8692	2.1375	2.1312
hmmer libquantum mcf milc zeusmp namd omnetpp hmmer	2.8462	2.8756	2.8473

Table B.1. Continued

Workloads	**static**	**UDCP**	**FairSDP**
hmmer libquantum mcf milc zeusmp omnetpp sjeng hmmer	1.848	2.1232	2.1163
hmmer libquantum mcf milc zeusmp omnetpp namd hmmer	1.9541	2.2085	2.2026
hmmer libquantum mcf milc zeusmp sjeng omnetpp hmmer	2.845	2.8638	2.8304
hmmer libquantum mcf milc zeusmp sjeng namd hmmer	1.9623	2.2334	2.2305
hmmer libquantum mcf namd milc zeusmp omnetpp hmmer	1.8404	2.0204	2.0169
hmmer libquantum mcf milc namd omnetpp sjeng zeusmp	2.8005	2.9113	2.8351