ENHANCING REVICE FOR MITIGATING CACHE SIDE-CHANNEL ATTACKS: A SECURE AND PRACTICAL APPROACH

A Thesis

by

YOUNGKI KIM

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2023

Major Subject: Computer Science

ABSTRACT

With more sensitive data being stored on computers, the cyber-attack risk is increasing globally. Therefore, it's crucial to address security vulnerabilities before new threats emerge promptly and potentially cause significant harm in the future. Recently, Spectre and Meltdown attacks known as cache side-channel attacks exploit modern processor characteristics. Despite the seriousness of these attacks, they are under-researched and need more attention to safeguard our data.

This thesis addresses how to optimize and improve the ReViCe, the solution for mitigating vulnerabilities of cache side-channel attacks, a problem caused by characteristics of modern processors. ReViCe enables speculative loads to refresh caches ahead of time while saving any removed line within the victim cache. If mis-speculation occurs, the replaced lines from the victim cache can be returned to the cache for undoing the cache changes, which can effectively isolate the cache changes for protecting us against cache-based Spectre and Meltdown attacks.

We also introduce a more realistic, secure design for ReViCe. We enhance the security design by conducting experiments, tackling related work CacheRewinder, identifying the appropriate size of victim cache and buffer, and incorporating a deadlock-free algorithm. These changes allow us to implement a safer and more practical version of ReViCe, requiring less memory.

DEDICATION

To my wife and our newborn baby Ejun Kim.

ACKNOWLEDGMENTS

First and foremost, I want to express my gratitude to EJ Kim, who offered me the opportunity to study in this fascinating field and guided me to a path of profound insights. I also sincerely thank SungKeun Kim, a doctoral candidate who generously shared his knowledge and provided detailed guidance throughout this journey. Lastly, my deepest appreciation extends to my home country, the Republic of Korea, and the Republic of Korea Army for their endless support during my studies.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Eun Jung Kim and Chia-Che Tsai of the Department of Computer Science & Engineering and Professor Jiang Hu of the Department of Electrical & Computer Engineering.

The ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage was published in SecDev, IEEE 2020 by Sungkeun Kim, a doctoral candidate in the Department of Computer Science Computer Engineering department of Texas A&M.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a fellowship from the Republic of Korea Army.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Rising data security costs

In our digital-driven world, all organizations heavily depend on cyberspace for data storage. This cyberspace reliance stems from the ease and efficiency it provides in managing large populations and administrative tasks. However, the growing importance of data security brings with it numerous threats at the same time. Therefore, the cost of protection has risen significantly, especially for protecting sensitive data since the world has stored many of those in cyberspace. Therefore, while computerized data storage allows for quick and efficient management of necessary information, it poses significant security risks that require continuous attention and resources to mitigate.

1.1.1 Organized cyber-attack

With the escalating value of data held in cyberspace, cyber-attacks are continually evolving, becoming more intricate, systematic, and organized to extract maximum benefit from this rich trove of information. Countries such as North Korea exemplify this trend. They have established advanced, highly-skilled hacking collectives, namely Kimsuky, Lazarus, and APT38, operating under the auspices of the Reconnaissance General Bureau (RGB)[3]. As these hacking tactics mature, becoming ever more organized and sophisticated, the damage they inflict has significantly amplified, causing devastating harm. Moreover, as these attacks are often state-sponsored, they seriously threaten international security and stability.

1.1.2 Danger of national level of threat

Malicious groups increasingly target data containing critical and sensitive information. Some of these groups are even state-sponsored organizations that can pose a threat at the national level, as previously mentioned. The well-known data security threat at the national level is North Korea again. They have demonstrated their capacity to execute damaging data breaches using diverse methods, posing a severe crisis for affected countries.

For example, the U.S. Secretary of Homeland Security recently announced that North Korea had stolen over 1 billion in crypto-currencies and used them to develop weapons of mass destruction[4]. Further illustrating more examples was the 2017 WannaCry data breach, attributed to the Lazarus Group occurred in 2017. This attack affected over two hundred thousand computers across one hundred fifty countries and cost an estimated four billion dollars in recovery efforts. The attackers took advantage of a flaw in Microsoft Windows, encrypted user files, and demanded cryptocurrency in exchange for file restoration. The breach affected various organizations, including hospitals, government agencies, and private companies.[5]. These incidents highlight the critical nature of data breaches and emphasize the importance of promptly identifying and rectifying system vulnerabilities.

1.2 New vulnerabilities in hardware

Modern processors have seen significant advancements, bringing novel features and new vulnerabilities. A key feature is speculative execution; technique processors use to enhance performance by predicting and executing potential instructions ahead of time. This method forecasts the results of conditional or branch instructions, running possible paths, and then discards any unnecessary but proceeded operations if the predictions are wrong.

Out-of-order execution is another technique that reorders instructions based on resource availability instead of executing them in their programmed order. This method allows the processor to use its resources to the fullest by running independent instructions concurrently. This rearrangement leads to faster execution times. However, while these techniques increase speed and efficiency, they can also create potential security risks that must be managed.

1.2.1 Emergence of cache-side channel attack

Cache-side channel attacks[6, 7] use the above processors to extract sensitive information from a victim's computer. The attack works by measuring the time differences it takes to access cached memory on the system. Furthermore, it checks cache changes made by speculative instructions and out-of-order execution to access targeted information. This sequence makes it possible to infer sensitive data stored in the cache, such as encryption keys, without directly accessing the system.

1.2.2 Mitigation against cache-side channel attack

Against these attacks, new concepts such as Invisispec[8], Cleanupspec[9], and ReViCe[1] have been developed to mitigate system vulnerabilities. Those defense methods show promise in mitigating cache side-channel attack vulnerabilities[8, 9]. These above studies use two forms of 'undo' or 'redo' to isolate cache changes caused by speculative execution against attackers. Invisispec is a representative method that does not expose cache changes as a redoing method that reverts cache changes by re-executing them in the same state as before[9]. On the contrary, Cleanupspec and ReViCe are representative methods of Undoing that cancel the action executed to return to the previous state[8].

1.3 Improving secure design

This thesis will explore the ReViCe method's potential for mitigating cache-side channel attack vulnerabilities[1]. ReViCe uses the Undoing method against cache-side channel attacks. ReViCe stores messages from the cache that can potentially be changed by speculative instructions in the victim cache to undo the changed cache state. If the result of speculative execution is correct, the cache state will remain as it has changed, and if the result is not correct, the data stored in the victim cache will be recalled and put back in place.

1.3.1 Target of research

Through several optimization strategies, we will make ReViCe more realistic and secure in this study. Firstly, we aim to identify a suitable buffer size. The original ReViCe uses the Gem5 simulation's primary setting, which assumes an infinite buffer size. Since an unlimited buffer size is not feasible, we will look for an appropriate, limited size that effectively addresses this issue. Secondly, we will address CacheRewinder[10], a related study. CacheRewinder asserts a superior hardware design to ReViCe, as it repurposes an existing write-back buffer for dual functions - simultaneously handling write-back messages and victim cache functions. We will critique this approach, highlighting potential issues with transforming a current structural buffer that serves a single role to a dual-functioning one within an actively progressing operation. Additionally, we will search for a better configuration for ReViCe through comparative analysis. Since ReViCe employs an additional cache structure called victim cache, minimizing its extra memory can reduce memory overhead. Finally, to maintain system integrity, we will address possible deadlocks by introducing additional logic to dismantle deadlock situations. This step ensures the system's smooth functioning, even under potential deadlock circumstances.

1.3.2 Contents of thesis

The remainder of this thesis is organized as follows. Chapter 2 reviews the background on cache side-channel attacks and existing defenses, write-back buffer, and Spectre and Meltdown, which are hardware vulnerabilities that allow unauthorized access to sensitive data by exploiting the speculative execution feature of modern processors. Chapter 3 addresses how to set the realistic buffer size in the ruby network system in the Gem5 simulation. Chapter 4 criticizes the related work, CacheRewinder, which uses a write-back buffer with two functions handling write-back messages and the victim cache function. Chapter 5 proposes optimizing the victim cache size to improve the system performance. Chapter 6 shows deadlock free scheme to ensure system integrity, and Chapter 6 discusses the results and future works.

2. BACKGROUND

2.1 Out-of-order execution

Out-of-order execution is one of the techniques of modern processors to improve program speed. Instructions are not executed in the order in which they appear in the program but in the order in which they can achieve the fastest use of processor resources. This can be possible because the system can dynamically reorder instructions by determining each instruction before it is executed.

The system reorders the instructions after judging whether the commands do not have a dependency relationship, the order in which dependencies can be quickly resolved, or the commands that can be done simultaneously. However, when the instructions are saved to memory, they maintain the correct order. This is because the system accurately adjusts their sequence in the reorder buffer.

2.2 Speculative execution

Speculative execution is also one of the characteristics of modern processors that improve program speed. Speculative execution means the system predicts which branch of the statement is more likely to be taken based on past execution history or other factors.

For example, if a specific conditional statement is usually 'True,' the system executes it as if it had been determined to be 'True' first. At this point, if the result for the branch is 'True,' we can keep the first executed instruction, which results in the profit for the speed. However, if the result of the branch is 'False,' the instruction that was executed in advance must be discarded unconditionally, which means loss. However, since the system predicts the most probable way of the branch, we usually gain the profit rather than the loss.

2.3 Cache structure

A cache is a memory that aims to reduce the bottleneck caused by the speed difference between a relatively fast CPU and a slow main memory. For example, when a CPU loads data from the main memory, it will store frequently used data in the cache for better performance since the cache can quickly hand the data to the CPU than the main memory. The next time it is needed, it loads it from the cache, not the main memory. These activities have the effect of improving the overall system speed. However, while the cache has the advantage of speeding up the system, on the contrary, it has the disadvantage of being very expensive: the larger the cache capacity, the exponentially greater the cost.

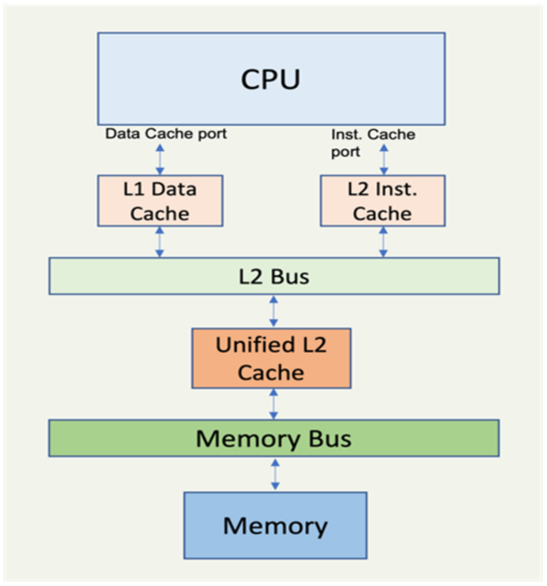


Figure 2.1: Cpu and caches interconnections concepts for single-core system

The cache typically has an L1 and L2 cache. Categorized by speed and size, we can see that the L1 cache is closest to the CPU, and the L2 cache is closest to the main memory if we look at Figure 2.1. On the other hand, in Figure 2.2, we see a multi-core system where the L1 cache is used as a private cache that only that CPU core can use. The L2 cache is a shared cache that other cores can share. Here, the L2 cache is commonly called the last-level cache(LLC).

2.4 Spectre and meltdown

Meltdown[11] and Spectre[12] take advantage of security vulnerabilities found in modern computer processors. They exploit a feature called speculative execution. Speculative execution is a performance optimization technique in which the processor tries to predict which instructions will

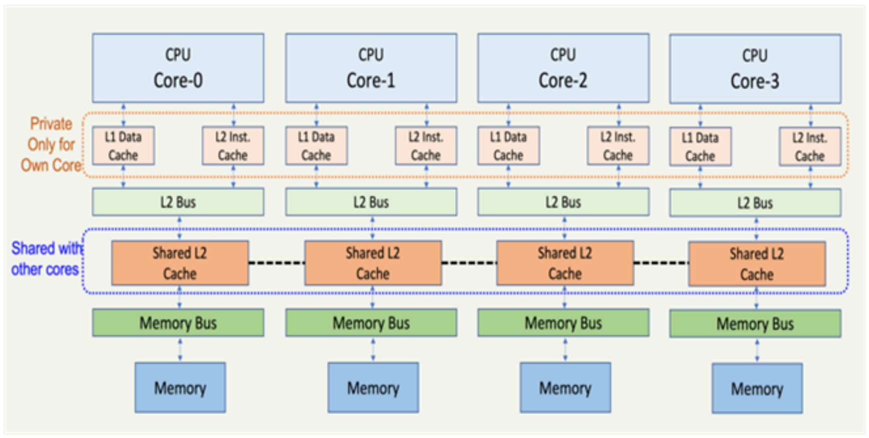


Figure 2.2: Cpu and caches interconnections concepts for multi-core system

be run next and starts processing them before it is certain they are needed. This helps the processor perform better by reducing because the system runs the predicted instructions in advance.

The Meltdown attack allows an attacker to access the user system's protected memory and reveal sensitive information like passwords[11]. To access the operating system, Meltdown uses the fact that during speculative execution, the processor does not fully isolate between user applications and the kernel memory. Therefore, Meltdown temporarily makes a transient attack even if the user side and operating system do not allow to do it.

Spectre exploits the speculative execution feature in processors like Meltdown but in a different method. Spectre is a type of vulnerability that breaks the boundaries between applications. It lets terrible actors trick good software into revealing secrets. It allows harmful software to reach into other programs and take the information it is not supposed to[12].

2.5 Microarchitectural side-channel attacks

Microarchitectural side-channel attacks exploit the physical design and implementation of microprocessors to extract sensitive information[8, 9] By measuring the timing or power consumption of internal components; attackers can deduce confidential data such as passwords or cryptographic keys. Examples of microarchitectural side-channel attacks include cache attacks, branch predictor attacks, and timing attacks.

Defending against these attacks can be difficult because they do not rely on software vulnerabilities. Instead, hardware and software countermeasures are necessary to protect against microarchitectural side-channel attacks. These countermeasures include cache partitioning, instruction randomization, and timing noise generation. It is essential to stay up-to-date with the latest research and to consider the potential for microarchitectural side-channel attacks when designing secure systems.

2.6 Example of possible attack scenario

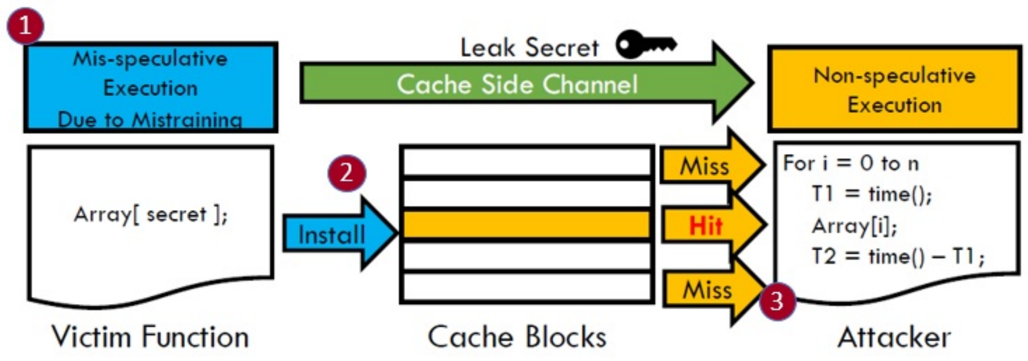


Figure 2.3: Possible attack scenario for cache side-channel attack reprinted from [1]

With the background of a cache side-channel attack, let us illustrate a potential attack scenario as depicted in Figure 2.3[1]. First, mis-speculative execution comes into play due to mistraining. Suppose there is a simple function running on a victim's computer. This function is repeatedly executed, producing a valid result every time. This consistent behavior trains the victim computer's processor to anticipate a similar outcome the next time this function is run. Next, when we use the same function intending to access a secret array, the victim computer's processor predicts this is a valid request due to its prior mistraining. It pre-emptively fetches the secret data from the array into the cache block. Lastly, an attacker can take advantage of this situation. They can steal the secret data by analyzing the timing differences in cache changes. In this way, the attacker exploits the speculative execution feature of modern processors to carry out a cache side-channel attack.

2.7 Invisispec

InvisiSpec[8] is a hardware mitigation technique designed to secure against cache side-channel attacks of security vulnerability that allows attackers to access sensitive information. The basic principle of InvisiSpec works by making changes to offending cache side channels invisible to other processes.

The main idea behind InvisiSpec is the 'Redoing' for the cache changes, which means delaying the update of the cache state until the speculative memory access is confirmed to be non-speculative. This is achieved by storing the speculative loads in an additional buffer called the Invisible Speculative Buffer (ISB). Once the speculative execution is confirmed to be non-speculative, the data from the ISB is committed to the cache, making it visible to other processes. However, it may cause performance overhead since speculative execution is an essential optimization technique in modern processors.

2.8 Cleanupspec

CleanUpSpec[9] is another proposed hardware technique to protect against speculative execution side-channel attacks, such as Spectre and Meltdown. However, this method reduces the performance overhead of previous mitigation approaches like InvisiSpec.

The idea behind CleanUpSpec is to allow speculative memory accesses to proceed as usual but then "Undoing," which means cleaning up any potentially observable information after the speculation is resolved against attackers. This undoing process ensures that sensitive information isn't leaked through side channels because the system deletes any information which can be visible during cache changes due to speculative execution.

2.9 CacheRewinder

CacheRewinder[10] is also a hardware-based defense mechanism against cache side-channel attacks. Like Cleanupspec and ReViCe, CacheRewinder prevents secrets from being leaked due to speculative execution, one of the modern processors' characteristics, through the "Undoing" method. CacheRewinder uses the existing space, the write-back buffer, as a temporary storage space to restore the changes in the cache that have been changed by speculative execution. In other words, in addition to the existing function of handling write-back messages, it also saves blocks evicted from the cache due to speculative execution when speculation fails.

2.10 ReViCe: Reusing victim cache to prevent speculative cache leakage

2.10.1 ReViCe motivation

The ReViCe[1] is a hardware technique to mitigate speculation-based cache side-channel attacks, which are well known for the Spectre and Meltdown[1, 12, 11]. Those cache side-channel attacks steal the information by using the timing of cache changes[6, 7]. Simply speaking, Attackers can take information using this vulnerability because the speed of fetching information from memory and the cache is different, so attackers can know which information is in the cache that stores the more recent data than the memory.

The cache side-channel attack, which intercepts data using cache change, can be solved by Redo and Undo methods. Redo and Undo both approaches ensure that micro-architecture updates are only exposed outside of the transient execution at Visibility Point (VP)[8] when primary operations cannot change the instructions. In the case of Redo, it chooses a method to delay the timing of cache updates by restarting in the right direction after the branch update. Therefore, as much as the time is delayed, overhead occurs as much. On the other hand, Undo uses the method of canceling the work, which is already done if necessary when updating first and then branch resolution[9]. Therefore, Undo can solve the performance overhead problem that Redo had previously. ReViCe uses Undo's method to solve cache side-channel attacks with low-performance overhead.

2.10.2 ReViCe design

ReViCe uses hardware techniques to prevent cache side-channel attacks. It uses a victim cache, which stores evicted cache lines, and the cache replacement policy, which prevents victim cache eviction from causing a side-channel leak. The main idea of ReViCe design is allowing speculative loads to update caches early but keep any replaced line in the victim cache to isolate those changes against the attackers. In case of speculation miss, replaced lines from the victim cache are used to restore the caches, thereby preventing any speculative-based cache attacks[1].

To be detailed, when a load is initially issued on a queue, if that load hits the cache, it will remain as it is, and there will be no change. However, if a cache is missed, the cache state will change as the existing cache line is replaced with the current one. If the speculative load is correct, it is safe that the cache state has changed. However, in the opposite case, if the speculative is incorrect, the attacker can recognize the cache state differently from before and steal the secret. ReViCe stores all changes due to speculative load in the victim cache if the speculative is wrong to prevent this. Therefore, if the speculation is correct, we can keep the cache change as it is, and in the opposite case, we can execute the Undo task to return the status where all saved cache changes.

2.10.3 ReViCe operation

To implement ReViCe design, it runs through the following detailed steps:

1) only loads can speculatively update the cache among the fetched instructions. Therefore, ReViCe adds a flag to each load instruction, whether speculative or not. If the load is speculative, the flag's value is 1, and loads that are not being given a value of 0 as the flag value.

2) In the case of a speculative load with a flag value of 1, the case differs depending on whether it is a cache hit or a miss. If a cache hit happens, the cache state does not change so that data can be delivered immediately, as usual, without any security flaws. However, the case of a cache miss is different. This is because the state of the cache changes depending on the speculative load. ReViCe finds the victim cache line in case of a cache miss, which will be replaced according to the cache replacement policy. Since the speculative load changes the cache state, the replaced cache line is sent to the victim cache. Otherwise, in the case of non-speculative instructions with a Flag value of 0, they are executed as they are without any unique action because they are not speculative instructions. to compensate for security vulnerabilities. These algorithms can be seen in more detail through the graph in Figure 2.4[1].

3) Blocks sent to the victim cache by speculative have the same hardware structure as Fig-

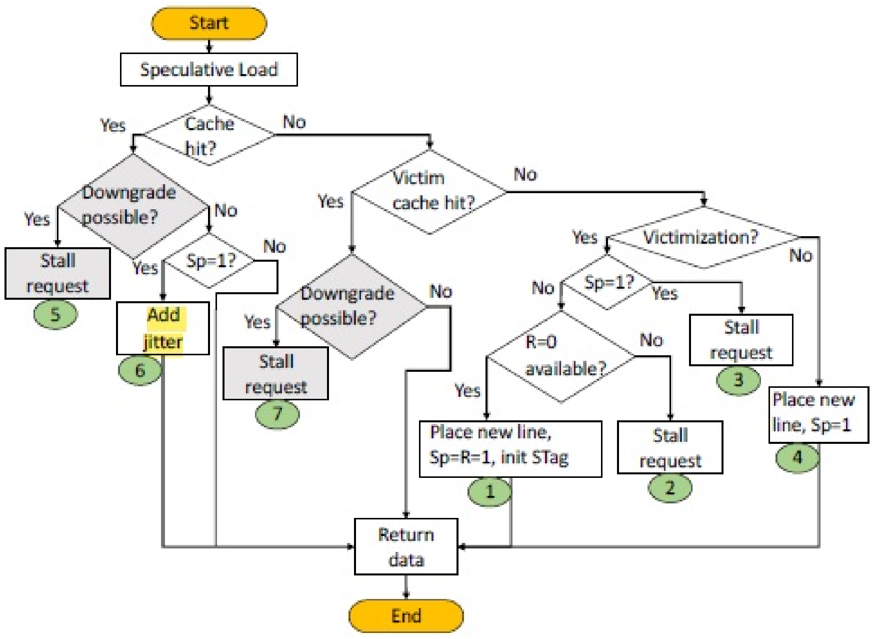
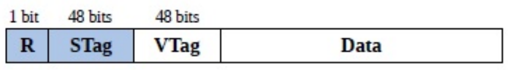


Figure 2.4: ReViCe flow for the speculative load reprinted from [1]

ure 2.5[1]. A VTag identifies each line in the victim cache. Along with this, each line has Restoration Bit(R) and STag. In the case of the R bit, if it has a value of 1, it is ready to go back to the cache, and its destination is the tag included in the STag. Since ReViCe uses the Undo method to cancel the action to the previous state at the time before the speculative is mispredicted, the destination(STag) of each line in the victim cache is the speculative load tag that sent the VTag to the victim cache.



R: Restoration Bit
STag: Tag of the speculative line *s* that victimized *v*
VTag: Tag of the line *v* that is victimized

Figure 2.5: Hardware extended victim cache block reprinted from [1]

2.11 Write-back buffer

When the CPU processes data, it is first written to the cache before going to memory. Unlike the write-through method, which is recorded in memory simultaneously when writing to the cache, the write-back method updates only the cache without updating the memory. In terms of this, the write-back buffer is the buffer that stores the cache changes before writing the memory back.

After being temporarily stored in the cache, the write-back method writes data to the main or auxiliary memory only when the cache replacement policy replaces the cache block. This protocol allows the write-back method to be fast by writing to the main or auxiliary memory only when necessary. However, since the cache is updated and the memory is not updated immediately, there could be a situation in which the cache and memory have different values.

2.12 Deadlock

- Deadlock prevention, Deadlock prevention is a proactive approach to ensure that the system is deadlock-free. For a deadlock to occur, four prerequisite conditions are required[13]. 1. mutual exclusion, 2. hold and wait, 3. no preemption, and 4. circular wait. Since the above four conditions must co-occur for the deadlock, deadlock prevention uses these conditions to prevent deadlock from occurring. The system can prevent deadlocks from occurring. By negating one of these conditions.

    1. Mutual Exclusion: To make deadlock happens, at least one resource must be non-sharable. This is because if all resources are sharable, deadlocks will not occur. After all, there will be no limits on the situation processors can share resources. However, this condition cannot be eliminated because, in modern processors, there will always be the case with limited resources.
    2. Hold and Wait: It is the situation processor that holds at least one resource and waits for additional resources allocated and used by other processes which other processors hold. To prevent this circular waits, a process should request all its required resources at once, or a process can only request new resources when it is not holding any. However,

it would be impractical for most applications in the real world due to the dynamic nature of requesting resources. Moreover, the starvation situation that processors, which require multiple popular resources, may not run forever can happen.

3. No Preemption: It means that the resources which were already allocated to other processes can only be forcibly taken away once they are used up. If we allow preemption, it would be critical since preemption can occur at any point during the execution of the critical section. Then, the system should roll back to a safe state and restart, which severely makes performance overheads.

4. Circular Wait: It is when processors are circularly waiting for resources. For example, when processor A needs resource A', but A' is occupied by processor B. At the same time, processor B needs resource B. However, if it is already occupied by processor A, it is in a situation where each resource is circularly occupied by the other, and the program does not operate. This situation can be solved by setting the order among each processor. However, establishing ordering can be difficult, especially in a natural system with hundreds or thousands of resources and locks.

- Deadlock Avoidance: Deadlock avoidance is an algorithm in which the system knows each process's requirements and allocates its resources. A representative algorithm is Banker's algorithm. It automatically considers requests and checks whether it is safe for the system and also needs to handle future resources required to prevent the deadlock.

- Deadlock detection: Deadlock detection periodically checks whether the system is in a deadlock situation. When the system enters the deadlock state, it does not proceed further, and the exact part is executed cyclically. To solve this problem, Detecting this cycle is called deadlock detection. There are various methods, but one popular way is the system breaking the cyclic connection by setting the order between processors or the messages when a certain threshold is reached.

2.13 Gem5 simulation

The gem5 is an open-source simulator for researching computer processors[2]. In addition, it is a platform for studying computer architecture, encompassing System-level and Micro-architectures. The simulator is used in academia and industry and initially went through 15 years of development at the University of Michigan and the GEMS project in Wisconsin. 2011 the GEMS and m5 projects merged, creating the current gem5.

The gem5 has several features: It has an event-driven memory system. To see the impact of the memories right away, it provides an event-driven memory system, including caches, crossbars, and more. It supports homogeneous and heterogeneous multi-cores. From the number of cores to the stats of the cache to the topology, multi-core systems in many forms can be arbitrarily set up as the researcher wants. The simulator can elastically track the performance of the CPU. For example, this simulator allows researchers to analyze and compare the performance of the cache layer and main memory quickly and accurately. The gem5 is an open-source simulator, as mentioned earlier. For the various analyses and comparisons required for the study, we were free to modify the various pieces of code in Gem5 without copyright issues.

2.13.1 Garnet network

Garnet is an interconnection network model implemented within the gem5 simulation framework[14]. Garnet is designed to simulate on-chip networks for providing a cycle-accurate micro-architectural implementation of an on-chip network router. Garnet is a network method that communicates between cores, caches, and memory controllers. Using it, we can change various network topologies and understand the impact of the changed network. After all, we can get various traffic patterns that affect system performance through Garnet.

2.14 SPEC benchmarks

The SPEC CPU 2017 benchmark package contains SPEC's next-generation, industry-standardized, CPU-intensive suites for measuring and comparing compute-intensive performance, stressing a system's processor, memory subsystem, and compiler. SPEC created these benchmarks to allow researchers to compare and measure the compute-intensive performance of the hardware across hardware using workloads developed in their applications.

Table 2.1: SPEC cpu 2017 benchmark suites

Suites	Purpose of Uses
SPEC speed	To analyze the time for a computer to complete a single task
SPEC rate	To analyze the throughput and work per unit of time

Depending on the purpose, this benchmark is divided into ’Speed’ and ’Rate’ suites. The former compares and analyzes the time it takes for a single computer to complete a task. The latter is intended to analyze the amount of throughput and work per unit of time.

Table 2.2: SPEC cpu 2017 benchmark that we used

Benchmarks	Application Area
perlbench	Perl interpreter
gcc	GNU C compiler
mcf	Route planning
cactuBSSN	Physics: relativity
xalancbmk	XML to HTML conversion via XSLT
x264	Video compression
Leela	Artificial Intelligence: Monte Carlo tree search (Go)
nab	Molecular dynamics
exchange2	Artificial Intelligence: recursive solution generator (Sudoku)

3. REALISTIC BUFFER SIZE

3.1 Background and motivation

We always strive to find the middle ground between cost and performance. As a representative example, we consider the correlation between memory price and performance when choosing a computer. The more efficient memory we use with more cost, the better performance we can achieve. Furthermore, if a situation arises in which unnecessary memory is used, it would be an extremely inefficient choice. Consequently, we are constantly thinking and researching diligently to find the optimal balance that can yield maximum efficiency by using only as much memory as needed in real life.

ReViCe's current buffer size is unrealistic and needs optimization. Seeing Figure 3.1, ReViCe uses the Garnet network for experiments in Gem5 simulation to analyze the impact of changes in network topologies. As the original version of ReViCe uses Garnet's message traffic pattern itself, it follows the basic settings of the network. The problem is that the basic configuration of buffer size is infinite, which makes it difficult to obtain realistic results. For example, It is impossible for this situation where numerous messages are in a particular buffer because of a bottleneck to happen in the real world. However, that situation can happen if the size of the buffer is infinite. To address this issue, we aim to determine an appropriate buffer size to make ReViCe more realistic and naturally cause stalls due to message concentration when the buffer size reaches the limit.

3.2 Methodology

3.2.1 Current situation

ReViCe, utilizing the Garnet network's basic configurations, operates with an infinite buffer size to manage messages. However, this unrealistic setting overlooks potential bottlenecks in network areas, significantly impacting overall performance and causing delays and congestion. The current setup must address these issues to assess performance under real-world conditions.

Obtaining accurate performance measurements requires revising buffer size settings and incor-

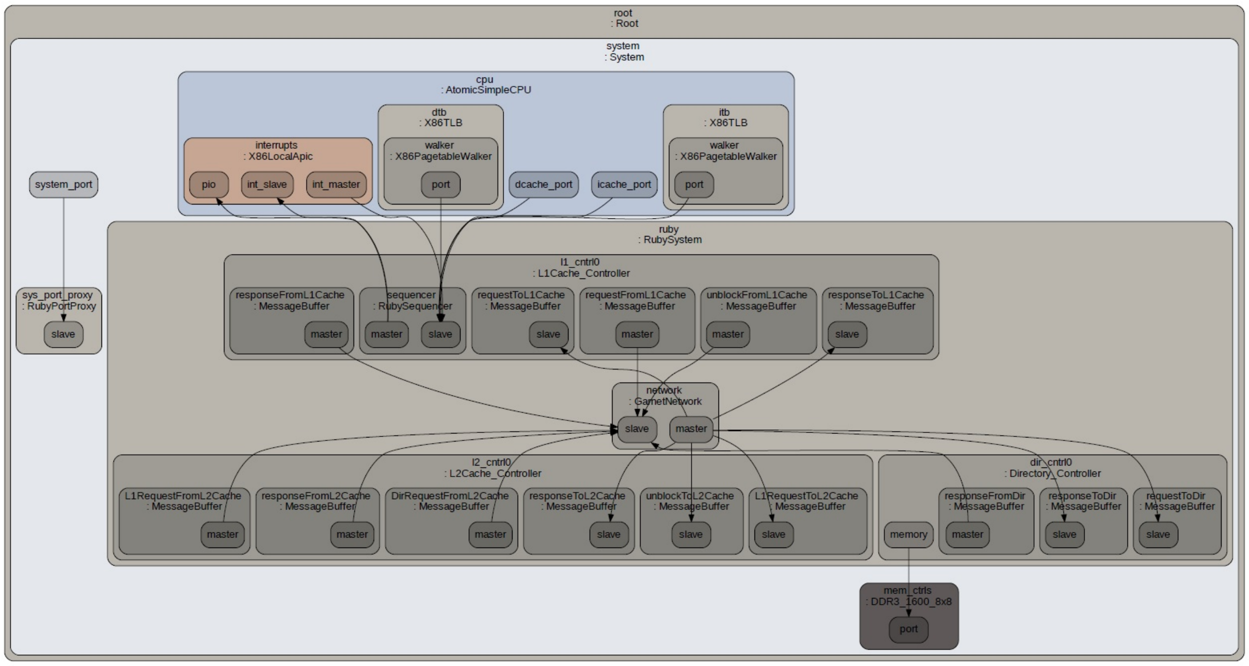


Figure 3.1: Gem5 garnet network[2]

porating constraints reflecting actual limitations. This process involves researching and simulating network conditions, identifying bottlenecks, and modifying the system's configuration. By making these adjustments, ReViCe's performance can be effectively evaluated, aiding users in optimizing their network configurations with a better understanding of its capabilities and limitations.

3.2.2 Required end state

Consequently, our objective is to identify the most optimal buffer size. As previously stated, we strive for maximum efficiency with minimal cost, which can guide our buffer size determination. For instance, incrementally increasing the buffer size from its minimum value can help us pinpoint the size at which resource stalls no longer occur in each network segment depicted in Figure 3.1. By discovering this balance, we can minimize both cost and performance trade-offs. To sum up, we can say the proper size of the buffer is when it does not show any stalls, but the no performance degradation due to the buffer size.

3.3 Experiment configuration

Table 3.1: Parameters of the simulated architecture[1]

Parameter	Value
Architecture	8 ARM cores for PARSEC, 1 X86 core for SPEC & PoC
Core	2GHz, Out-of-Order, no SMT, 32 Load Queue, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache (Private)	32KB, 64B line, 4-way, 1-cycle round-trip lat, 1 port
L1-D Cache (Private)	64KB, 64B line, 8-way, 1-cycle round-trip lat, 3 Rd/Wr ports
L2 Cache (Shared)	inclusive, Per core: 2 MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max),
Cache Coherence	Directory based MESI
Cache Replacement	Pseudo LRU
Network	4x2 MESH, 128 link width, one cycle latency per hop
DRAM	Built-in memory model in Gem5
Victim Cache	64B lines, fully associative, 16 blocks in L1-D & L2 for PARSEC, 32, 64 blocks in L1-D & L2 for CPU2017, respectively.
buffer size	8, 12, 14, 16, 32, and 64 entries

We implemented a comprehensive series of experiments focusing on the incremental expansion of the buffer size. Our initial approach involved adjusting the number of entries within the buffer and exploring various values, including 8, 12, 14, 16, 32, and 64. This step-by-step progression allowed us to closely examine the impact of each modification on the system's performance. While we are increasing the size of the buffer, we will find the size which is the maximum efficiency with minimal cost.

As shown in Table 3.1[1], the table provides a detailed overview of the parameters employed in our simulated architecture, which is the same as the previous ReViCe model[1] except for the buffer size. By meticulously analyzing the data obtained from these experiments, we aim to identify the optimal buffer size that balances efficiency and resource utilization.

3.3.1 Result of experiment

Figures 3.2, 3.3, and 3.4. show the relationship between the number of resource stalls and buffer size. We observed that resource stalls ceased to occur after increasing the buffer from the buffer size to sixteen entries. Moreover, beyond the sixteen entries threshold, further performance improvements were not detected, as shown in Figures 3.5, 3.6, and 3.7.

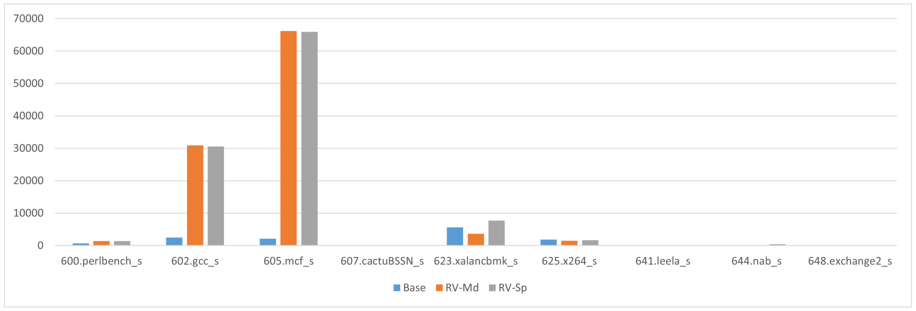


Figure 3.2: Number of resource stalls happen in 8 buffer entries

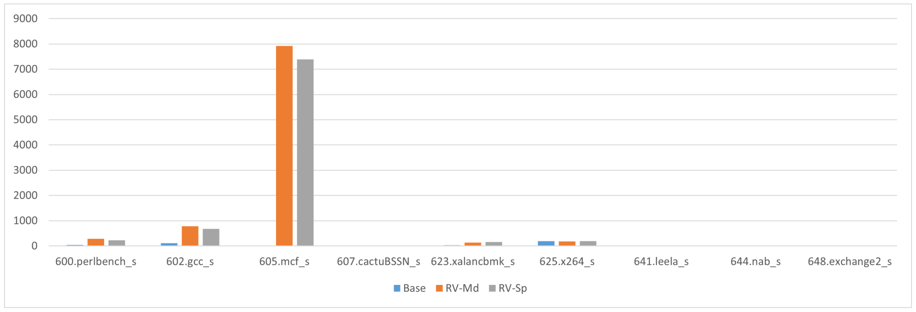


Figure 3.3: Number of resource stalls happen in 12 buffer entries

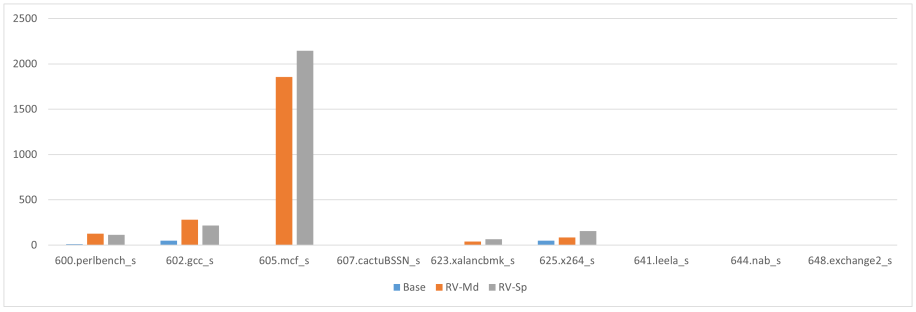


Figure 3.4: Number of resource stalls happen in 14 buffer entries

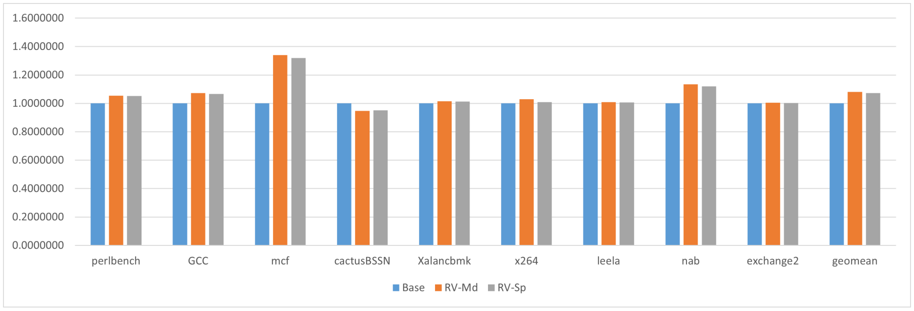


Figure 3.5: Performance comparison in 16 buffer entries

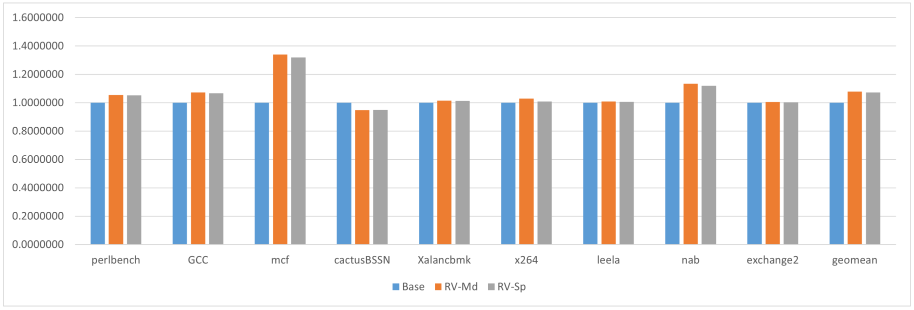


Figure 3.6: Performance comparison in 32 buffer entries

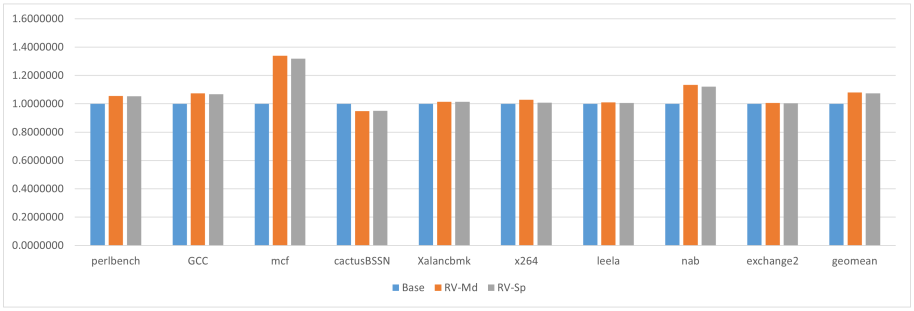


Figure 3.7: Performance comparison in 64 buffer entries

3.4 Summary and evaluation

Our thorough experimentation revealed that a buffer size of sixteen entries effectively eliminated stalls within the Garnet network while it achieves no performance degradation. This optimal value presents a more practical approach than the original unlimited buffer size implemented in ReViCe, ensuring that the system adheres to the constraints imposed by limited memory resources.

As a result, our experiment demonstrates that a suitable finite buffer size can be identified through methodical testing, with a value of sixteen entries proving ideal for both the Garnet network and the ReViCe experiment. By employing this optimized buffer size, ReViCe can more accurately simulate real-world scenarios, thereby enhancing the precision and reliability of its results.

4. COMPARATIVES STUDY WITH CACHEREWINDER

4.1 Motivation

As discussed in the Background section, numerous defense strategies against transient attacks target cache side channels. In this situation, a qualitative and quantitative comparison is essential to evaluate these methods effectively. One Undo method, CacheRewinder, has asserted its superiority over ReViCe. However, in this chapter, we will investigate the validity of this claim and identify any potential advantages ReViCe has over CacheRewinder.

4.2 Background

4.2.1 Purpose of CacheRewinder

CacheRewinder offers a hardware-based defense mechanism against transient execution attacks that leak sensitive data through cache timing side channels [10]. Like other undo schemes, CacheRewinder keeps confidential information by revoking cache updates made during speculative executions and restoring the original cache state. A highlighted idea in CacheRewinder is using underused write-back buffer space as temporary storage for cache blocks evicted during speculative executions. The advantage, which they claimed, effectively protects against transient execution attacks while minimizing performance overhead and storage costs.

4.2.2 Using underutilized write-back buffer

To implement Undo methods, a designated space is required to store cache lines evicted due to speculative loads. CacheRewinder utilizes the relatively underused write-back buffer structure in computer architecture. This can be possible because their research claims the average number of occupied entries in the write-back buffer is under one [10]. However, to accommodate cache information within the write-back buffer, it was essential to modify the write-back buffer from its original queue data structure to a cache structure.

4.2.3 Extended write-back buffer in CacheRewinder

Because CacheRewinder uses underused space, which is a write-back buffer, negating the need for additional data storage. However, the write-back buffer must shift significantly from the simple FIFO queue structure to a more complex version to accommodate the victim cache function, which stores lines evicted from the cache. This change injects complexity into the architectural design.

The extended write-back buffer must include two distinct functionalities: maintaining a 'Stag' that identifies evicted lines and a 'V tag' that signals a line's return to the cache. Consequently, the hardware cost of the extended buffer exceeds that of the original write-back buffer by more than twice because it requires two tags per entry. The following equations (4.1 and 4.2) illustrate this cost differential, making it clear that while CacheRewinder repurposes underutilized space, it will lead to imposing notable challenges and hardware costs. Their extended write-back buffer incurs twice the hardware area cost than before, as shown in equations 4.1 and 4.2 above.

$$\text{Original write-back buffer size} = \text{Tag size} * \log_2 \text{Number of blocks} * 2 \tag{4.1}$$

$$\text{Victim cache size} = \text{Tag size} * \log_2 \text{Number of blocks} * 2 * 2 \tag{4.2}$$

4.3 Methodology

4.3.1 Implementing CacheRewinder

Now, we will perform a qualitative and quantitative comparison between ReViCe and CacheRewinder as they described. First, we will follow the configuration of the experiment as CacheRewinder claimed. Next, we mimic the CacheRewinder, which can do two functions to occupy a space simultaneously. The write-back buffer in CacheRewinder can perform two functions simultaneously[10]. 1) Firstly, it can handle write-back messages, and 2) secondly, it can save lines evicted from the cache by speculative loads. Specifically, we implemented it to experimentally create a system from ReViCe where two functionalities verify each other. The Write-back queue and Victim cache check the size of each other and ensure that the sum of the two entries does not exceed a specific limit, similar to how it occurs within a combined one space.

4.3.2 Experiment configuration

We use the configuration detailed in Table 4.1 for our experiment. This configuration does not include the hardware cost implications of CacheRewinder's extended version of the write-back buffer, a crucial aspect we uncovered in our study. This is because we experimented with this setup to recreate the CacheRewinder environment, as they argue accurately. By doing this, we aim to compare CacheRewinder and ReViCe quantitatively. We describe the practical differences between these two defensive designs.

Table 4.1: Configuration for comparison

ReViCe		CacheRewinder
Write-back entries	Victim cache entries	Extended Write-back buffer
4 entries	4 entries	8 entries

4.4 Experiment results and evaluation

Our experiment replicated the setup detailed in Table 4.1, except for CacheRewinder's additional hardware elements. Table 4.2 reveals that without any defensive scheme, our baseline model experienced 6.62% overhead with Specter and 7.22% with Meltdown when using ReViCe. Contrastingly, CacheRewinder displayed 6.60% and 7.18% overhead for Spectre and Meltdown, respectively, revealing it to be marginally more efficient than ReViCe, with a 0.02% reduced overhead in both cases.

However, this slight improvement of 0.02% does not consider other potential factors that could impact performance. The observed benefits are small and may not be worth the effort needed to repurpose existing space. This suggests that such modifications to the system are not significantly meaningful. Going forward, focusing efforts on other aspects of system optimization might be more beneficial.

Table 4.2: Overhead comparison with baseline in ReViCe and CacheRewinder

	Base	Spectre	Meltdown
ReViCe	1	1.0662	1.0722
CacheRewinder	1	1.0660	1.0718

5. OPTIMIZING REVICE

5.1 Motivation

In the previous chapter, we looked at CacheRewinder and compared its performance with ReViCe, as claimed by CacheRewinder. CacheRewinder uses a more complex structure, which takes up more hardware space. This information could help to compare CacheRewinder and ReViCe's performance better and to optimize ReViCe based on this comparison.

Suppose we can find the right balance in the size of the write-back buffer and victim cache and discover the proper ratio between them. In that case, we can cut unnecessary memory use and allocate more space where needed. This experiment could make ReViCe operate more efficiently. So, in this chapter, we will also look at how to find a better, balanced design for ReViCe.

5.2 Background

In the previous chapter, we discovered that the extended write-back buffer has double the hardware cost of the original write-back buffer. Using this fact, we can find out which one has more advantages under the same conditions through a simple equation.

$$WB_{size} + VC_{size} * 2 = EWB_{size} * 2 \tag{5.1}$$

Equation 5.1 compares the memory usage between ReViCe and CacheRewinder. The left side represents the memory used by ReViCe, where WB_{size} denotes the size of the original write-back buffer, and VC_{size} represents the number of entries in the victim cache. Since the victim cache requires double space to store and handle tags than the original write-back buffer, we calculate the total memory usage showing $WB_{size} + VC_{size} * 2$.

On the other hand, the right side of Equation 5.1 represents CacheRewinder's memory usage. The write-back buffer of CacheRewinder serves as both the write-back operation and the victim cache. Therefore CacheRewinder only has one integrated Extended Write-back buffer(EWB) denoted as EWB_{size} in the equation, which is the sum of WB_{size}, and VC_{size} while ReViCe has those separately. However, like the calculation of the victim cache, we need to put double times more hardware cost for EWB.

Simply, while the original write-back buffer size is 2, we can get the size of the extended version of the write-back buffer, and the victim cache size is 1 with a fair comparison. By this relationship, ReViCe can have more capacity under the same conditions under Equation 5.1. Because CacheRewinder uses more hardware area budget, ReViCe can add more entries as much as Cacherewinder uses.

5.3 Experiment methodology

For an equivalent comparison between ReViCe and CacheRewinder, we set all simulation configurations the same without the write-back buffer and victim cache entries. Using the calculated ratio in the previous section, we can make the table the following Table 5.1. As shown in experiment 1, when cache Rewinder uses one integrated space with eight entries, ReViCe can have fourteen entries for the write-back buffer and one entry in the victim cache simultaneously. ReViCe has more space for the simulation because it saves the hardware area cost, not an EWB.

Table 5.1: Configuration for the simulation comparison

	ReViCe		CacheRewinder
	Write-back	Victim Cache	Extended Write-back buffer
Experiment 1	14	1	8
Experiment 2	10	3	8
Experiment 3	4	6	8

5.4 Result and evaluation

We implemented experiments using various ReViCe configurations under the same hardware cost conditions, as shown in Table 5.1. The optimal performance was achieved when the speed of CacheRewinder was set to 1, the write-back buffer held four entries, and the victim cache contained seven entries in ReViCe, as depicted in Figure 5.1. Upon analyzing Figure 5.1, it is clear that

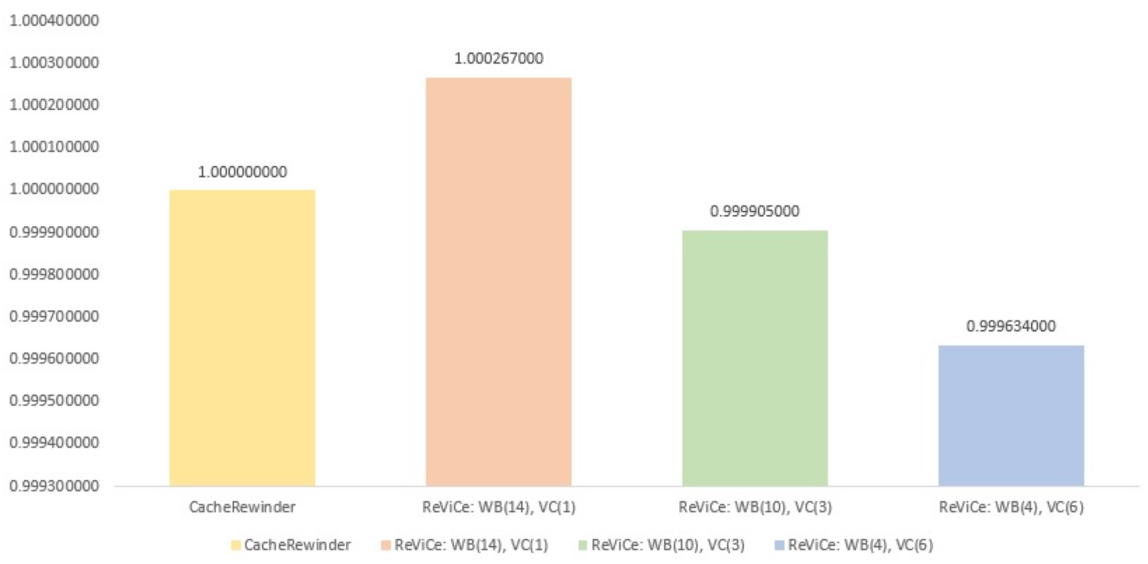


Figure 5.1: Performance comparison between CacheRewinder and ReViCe

increasing the number of victim caches improved performance more than expanding the write-back buffer. However, performance does not improve at certain points along with victim cache size since the secure design does not need extra spaces. As a result, it shows the best performance when we use ReViCe with four write-back buffer entries and six victim cache entries among four configurations. Based on the results, enhancing performance by minimizing unused capacity and giving space to boost structures is more reasonable than complicating the system's architecture by overhauling underused space.

5.5 Proof of concept

To know why proper cache size is needed for guaranteeing the performance, first, we set the size of the victim cache as one to see what is happening. As shown in Figure 5.2, when we set the victim cache size as one, numerous stalls happen because we limit its size of it. Due to this, many stalls happen, which makes performance overhead severe to the benchmark simulations. Therefore, based on the experiment results in the Figure 5.2, we know that the proper amount of victim cached is necessary.

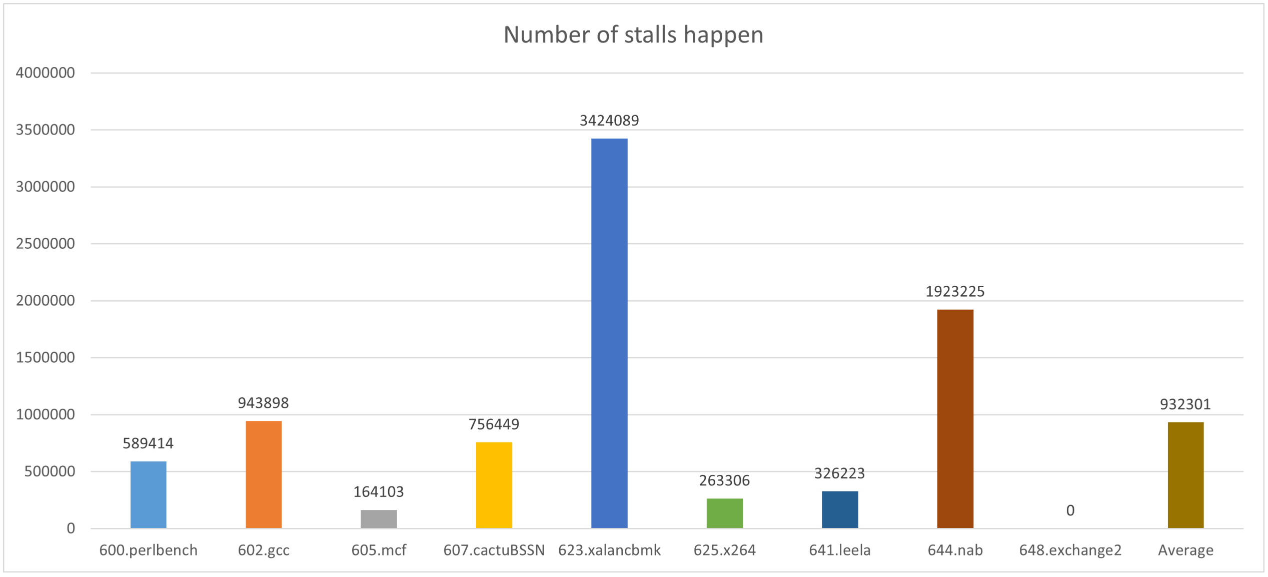


Figure 5.2: Number of stalls happen when limiting the victim cache

6. IMPLEMENT DEADLOCK FREE

6.1 Motivation

During the experiments, we suffered several deadlock issues during the Gem5 simulation related to the secure design integrity. To see how much deadlock frequently happens, we do a simple simulation. In the ReViCe, we set the algorithm which detects if a message is stalled at some point in the process and does not proceed anymore. Figure 6.1 is the latency graph representing how many cycles are needed to resolve the stalled messages.

As shown in Figure 6.1, 96.95% of the total stalled messages were resolved within around 2.8 cycles. However, 3.05% stalled messages hit the threshold, which we set as 800 cycles. Those unresolved messages caused performance degradation and possible deadlock issues furthermore. Therefore, we aim to implement a deadlock-free system to guarantee stability.

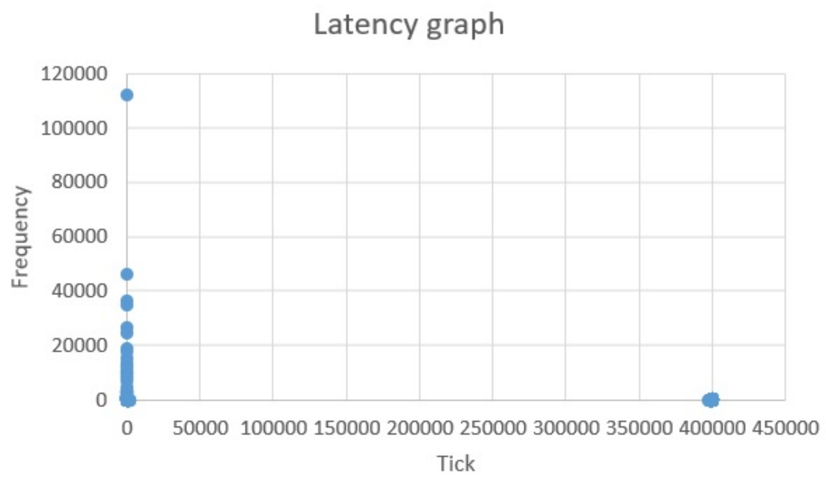


Figure 6.1: Stall frequency graph when we set deadlock threshold as 400,000 tick

6.2 Background

6.2.1 Generalization of typical deadlocks in real world

The following Figure 6.2 is the typical deadlocks we have experienced several times during the simulation. To be detailed processed, 1) Speculative Load A issued from the mandatory queue. 2) SLD A evicts Message B from the cache to the victim cache. 3) Expose(message for branch result true) or undo(message for branch false) A, which can resolve the SLD A, is stalled because the cache and victim cache is full and not available to be replaced, therefore expose or undo message can not be processed, and is stalled. 4) Message B is waiting for SLD A resolution to be replaced. 5) Simultaneously, SLD A is waiting for its resolution message to expose or undo A. This circular hold-and-wait situation makes the deadlocks happen.

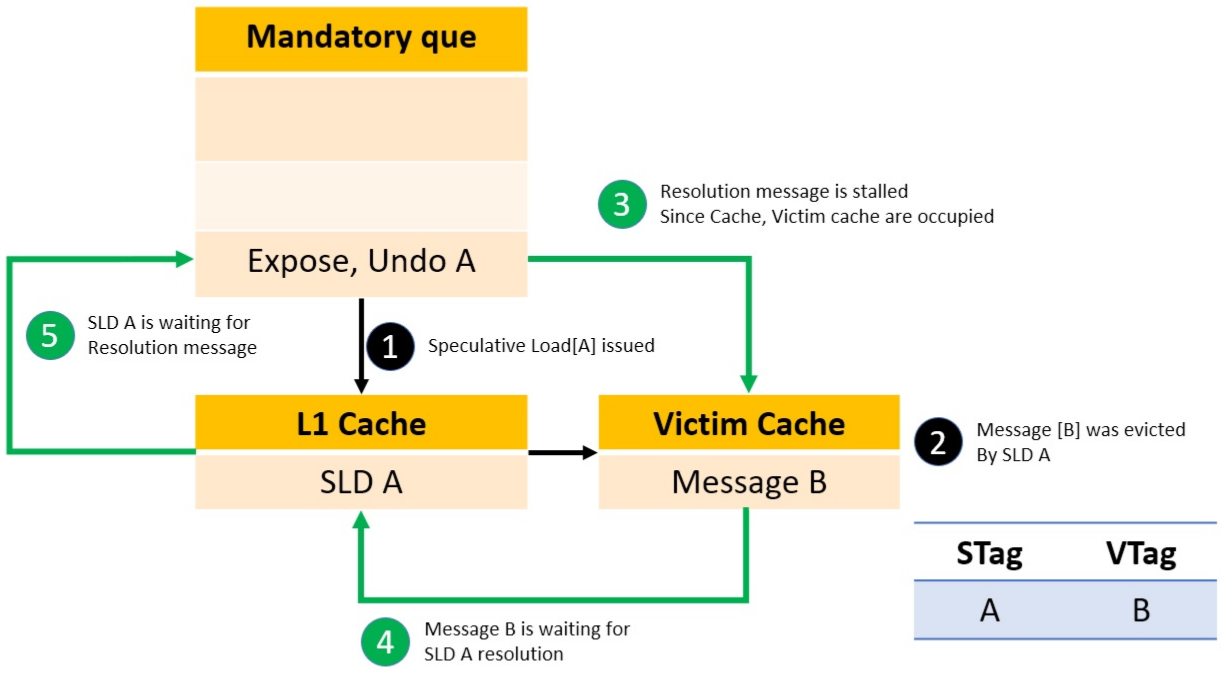

Mandatory que
3 Resolution message is stalled Since Cache, Victim cache are occupied
Expose, Undo A
5 SLD A is waiting for Resolution message
1 Speculative Load[A] issued
L1 Cache
SLD A
Victim Cache
Message B
2 Message [B] was evicted By SLD A
STag
VTag
A
B
4 Message B is waiting for SLD A resolution

Figure 6.2: Typical deadlock description

6.3 Methodology

6.3.1 Scheme for deadlock freedom

We decide to take two steps to solve the previous section's deadlock. Firstly, the system must be able to detect deadlocks. Secondly, the system should be able to solve the problem once a deadlock is detected. We will explore these two steps considering three main ways to deal with deadlock: avoidance, prevention, and recovery[15, 13], which we have explored in the background chapter.

6.3.2 Deadlock detection

We employ a threshold, established in Figure 6.1, to detect deadlock situations. We have integrated an algorithm that identifies messages not processed within a specific cycle. If a message remains unresolved beyond 800 cycles, we consider it a deadlock situation, indicating abnormal program progression. Therefore we focus on these target messages when a deadlock happens to solve the situation.

6.3.3 Deadlock avoidance

The first scheme we can consider as our deadlock-free algorithm is deadlock avoidance. Deadlock avoidance ensures that the system does not enter an unsafe state by requiring information about existing resources, available resources, and resource requests, as well as knowledge of future process resource requests. Based on this information, we can allocate requests to the available resources using the banker's algorithm, one method for implementing deadlock avoidance. However, it is unsuitable for dynamic simulation in the real world due to its reliance on static information. That means we should allocate all the resources to the processor, which is impossible to implement in the existing system.

6.3.4 Deadlock recovery

The second scheme we have looked into is the deadlock recovery scheme. Deadlock recovery solves the problem by restarting the system from scratch after a deadlock or the designated points that were safe to implement and the system saved as backup. However, it could be better for our purposes. Because 1) if we restart the whole system, it will cause colossal system performance overhead, and we can not guarantee that the same deadlock will not happen again. Therefore, it could be an endless loop. 2) Moreover, when we save the backup points while the system is running, we need to figure out which points are safe points to promise deadlock free to us. Due to the above two reasons, we do not choose deadlock recovery as our scheme.

6.3.5 Deadlock prevention

Last, a deadlock-free scheme is deadlock prevention. As we mentioned earlier in the background, Among the four prerequisites for the deadlock issue, we can handle the hold-and-wait situation, as shown in Figure 6.2, which is the typical deadlock pattern in simulation. After we detect the deadlock situation, we add additional code to break the circular dependencies between messages. When circular waits are detected, we forcibly send the undo message for speculative load A in Figure 6.2. These sent messages will break the number four hold-and-waits relationship in the Figure, and also, the numbers three and five will be resolved in a sequence.

6.4 Results and evaluation

6.4.1 Experiment results

In our experiment, we aimed to test the efficiency of our deadlock prevention method. We set the victim cache size to just one to create frequent deadlock situations. After fast-forwarding by ten billion steps, we ran one billion instructions to concentrate on crucial areas. As Figure 6.3 indicates, our deadlock-free algorithm was employed roughly 327,239 times per billion instructions on average. Even when unexpected stalls were frequent, we encountered no further deadlock issues during numerous simulations, demonstrating the effectiveness of our method.

6.4.2 Summary and evaluations

We explored the prevalent deadlock issue in system simulations, leading to an in-depth analysis of three critical deadlock-handling strategies: avoidance, recovery, and prevention. We found deadlock avoidance and recovery to be impractical for dynamic real-world simulations. Avoidance

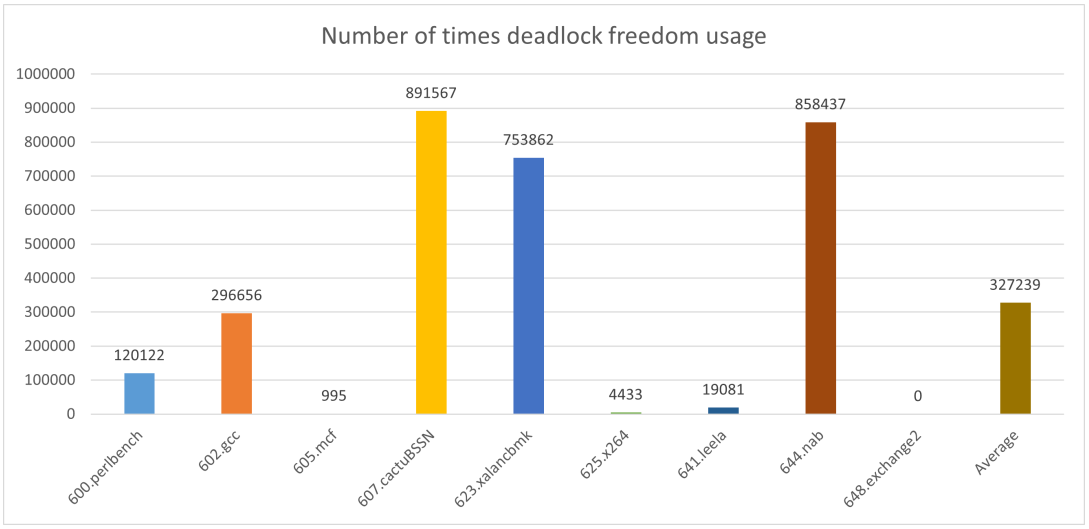


Figure 6.3: Number of times the ReViCe uses deadlock freedom when limiting the victim cache

was limiting due to its reliance on static information, while recovery posed risks of significant performance overhead and the potential for an endless deadlock loop.

In contrast, with the above simple method, we implement the deadlock prevention scheme to our secure design to ensure the system's integrity. We successfully implemented a deadlock prevention scheme, focusing on the 'hold-and-wait' scenario, the typical deadlock pattern in simulation. This approach effectively disrupted the cyclical dependencies, preventing deadlock situations and ensuring system integrity.

7. CONCLUSIONS

7.1 Evaluation of results

In this thesis, our focus was on enhancing security design against cache-side channel attacks. Moreover, the results of our efforts were valuable. We improve ReViCe to be a practical, trustworthy strategy with a strong basis. First, we brought our design to the realities by setting the finite size of the buffer to reflect better how it functions in the real world. Second, during the comparative studies, we compared our approach to other undo-based models and identified areas for improvement based on this comparison. We also proposed the best configuration for optimizing the ReViCe to realize that ReViCe is a superior approach to other ’undo’ based methods of defending against cache side-channel attacks. Last but not least, we introduced and applied a new ’deadlock freedom’ algorithm to strengthen the defense capabilities and the design integrity.

8. Future work

8.1 Finding better performance

While our system is efficient, there is always room for further improvements. Future studies could explore more configuration possibilities for ReViCe to enhance its performance. For instance, expanding the victim cache is more effective than increasing the write-back buffer. We could investigate other existing buffers or architectures that might work with the victim cache to boost overall performance.

8.2 Better deadlock freedom algorithm

The deadlock freedom algorithm in this thesis can be improved through further development. Future research could enhance the algorithm's capability to predict and prevent potential deadlock situations. Currently, we use a certain threshold to identify possible deadlock issues. However, it can be refined by finding better parameters or other ways to get possible deadlock issue alerts.

REFERENCES

[1] S. Kim, F. Mahmud, J. Huang, P. Majumder, N. Christou, A. Muzahid, C.-C. Tsai, and E. J. Kim, "Revice: Reusing victim cache to prevent speculative cache leakage," in *2020 IEEE Secure Development (SecDev)*, pp. 96–107, IEEE, 2020.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[3] M. Raska, "North korea's evolving cyber strategies: Continuity and change," *degruyter*.

[4] J. Da-gyum, "Crypto hacking behind n. korea's renewed nuclear ambition," *TheKoreanHerald*.

[5] E. Nakashima, "The nsa has linked the wannacry computer worm to north korea," *washingtonpost*.

[6] 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*, pp. 368–379, 2016.

[7] 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 fo the 27th USENIX Security Symposium*, USENIX Association, 2018.

[8] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 428–441, IEEE, 2018.

[9] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*,

pp. 73–86, 2019.

[10] J. Lee, J. Lee, T. Suh, and G. Koo, “Cacherewinder: revoking speculative cache updates exploiting write-back buffer,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 514–519, IEEE, 2022.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, *et al.*, “Meltdown: Reading kernel memory from user space,” *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[12] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, “Spectre attacks: Exploiting speculative execution,” *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[13] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts*. Wiley, 10th ed., 2018.

[14] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 33–42, IEEE, 2009.

[15] W. J. Dally and C. L. Seitz, “Deadlock-free message routing in multiprocessor interconnection networks,” 1988.