Securing the Memory Hierarchy from Speculative Side-Channel Attacks

CHRISTOS SAKALIS
Securing the Memory Hierarchy
from Speculative Side-Channel Attacks

Christos Sakalis
christos.sakalis@it.uu.se

January 2020

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Christos Sakalis 2020
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

Modern high-performance CPUs depend on speculative out-of-order execution in order to offer high performance while also remaining energy efficient. However, with the introduction of Meltdown and Spectre in the beginning of 2018, speculative execution has been under attack. These exploits, and the many that followed, take advantage of the unchecked nature of speculative execution and the microarchitectural changes it causes in order to mount speculative side-channel attacks. Such attacks can bypass software and hardware barriers and gain access to sensitive information while remaining invisible to the application.

In this thesis we will describe our work on preventing speculative side-channel attacks that exploit the memory hierarchy as their side-channel. Specifically, we will discuss two different approaches, one were we do not restrict speculative execution but try to keep its microarchitectural side-effects hidden, and one where we delay speculative memory accesses if we determine that they might lead to information leakage. We will discuss the advantages and disadvantages of both approaches, compare them against other state-of-the-art solutions, and show that it is possible to achieve secure, invisible speculation while at the same time maintaining high performance and efficiency.
Acknowledgments

This thesis would not have been possible without the help of my supervisors, my co-authors, my colleagues in the Uppsala Architecture Research Team, and all the administrative and support staff that make sure everything is running smoothly. To everyone, thank you! Special thanks to Magnus for always helping me and putting up with me when I am not listening to him, as well as to Stefanos, without whom I wouldn’t be here at this point today.

Of course, also thanks to all my family and friends, especially my parents that have always supported me in my choices, and Barbara for being a light in my life.
List of Papers

This thesis is based on the following papers


## Contents

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>3</td>
</tr>
<tr>
<td>1.1 Speculative Side-Channel Attacks</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Current State of Affairs and Related Work</td>
<td>9</td>
</tr>
<tr>
<td><strong>2 Securing the Memory Hierarchy from Speculative Side-Channel Attacks</strong></td>
<td>15</td>
</tr>
<tr>
<td>2.1 Paper I: Ghost Loads: What is the Cost of Invisible Speculation?</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Paper II: Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Paper III: Understanding Selective Delay as a Method for Efficient Secure Speculative Execution</td>
<td>21</td>
</tr>
<tr>
<td><strong>3 Conclusions</strong></td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paper I</strong></td>
<td>27</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>33</td>
</tr>
<tr>
<td>1.2 Speculative Shadows</td>
<td>36</td>
</tr>
<tr>
<td>1.3 Non-Speculative Loads</td>
<td>39</td>
</tr>
<tr>
<td>1.4 Ghost Loads</td>
<td>40</td>
</tr>
<tr>
<td>1.4.1 Materialization</td>
<td>41</td>
</tr>
<tr>
<td>1.4.2 Ghost Buffer</td>
<td>41</td>
</tr>
<tr>
<td>1.5 InvisiSpec</td>
<td>44</td>
</tr>
<tr>
<td>1.6 Evaluation</td>
<td>44</td>
</tr>
<tr>
<td>1.6.1 Methodology</td>
<td>44</td>
</tr>
<tr>
<td>1.6.2 Ghost Loads</td>
<td>45</td>
</tr>
<tr>
<td>1.6.3 Memory Behavior</td>
<td>46</td>
</tr>
<tr>
<td>1.6.4 Performance</td>
<td>49</td>
</tr>
<tr>
<td>1.6.5 Energy Efficiency</td>
<td>52</td>
</tr>
</tbody>
</table>
Part A
Chapter 1

Introduction

In order to satisfy the ever-increasing need for more computational power, hardware designers need to constantly modify and optimize their implementations of modern, high-performance CPUs. At the same time, for their customers to be able to adopt the newest, most powerful, and most energy efficient hardware, a stable interface between the CPUs and the rest of the hardware and software components is necessary. This interface consists of multiple different components, such as the instructions and registers available or the memory consistency model, but they can be grouped under the umbrella of the Instruction Set Architecture (ISA). This architecture interface/layer hides the actual hardware implementation, which is referred to as the microarchitecture, from the user. This allows for the hardware designers to change the underlying functionality of the CPU while still maintaining a stable interface for their users. For example, a very common optimization found on modern CPUs is the use of caches to reduce the latency of memory requests. On the architecture level, the user only sees one single, flat memory space, but on the microarchitecture level one can find a number of different caches with different sizes, latencies, and data sharing parameters.

Another common optimization is speculative out-of-order execution. On the architecture level, instructions are executed one after the other, in exactly the same order as they are found in the program. However, hardware designers have realized that this is very inefficient, as most of the time not all instructions need to be executed in program order for the program to be executed correctly and different instructions require different hardware components and incur different latencies during execution. Unfortunately, it is very hard for the hardware to determine with absolute certainty when this is the case or when it is not, so instead speculative execution was introduced. The main idea between speculat-
ive execution is simple: The CPU is allowed to speculate (guess) which instructions should be executed and in which order, as long as the speculation is not made visible to the user until it has been determined to be correct. For example, the CPU is allowed to guess if a branch will be taken or not, as long as any instructions executed based on that guess are kept hidden from the user until the branch target has been verified. To achieve this, the CPU does not permit any speculative instructions to modify the architectural state of the system and, if the speculation proves to have been incorrect, the instructions and any changes they have caused in the pipeline are squashed. However, the same is not true for all microarchitectural changes. For example, any data that were brought into the cache by a speculative instruction will remain there even if the instruction is squashed, as they do not affect the architectural state of the system and the correct execution of the application.

Or so we thought.

As it turns out, it is possible to peer under the architecture layer and observer, or even modify, the microarchitectural state. For example, when an instruction accesses data in the memory, these data are also installed in the caches. Since accesses that hit in the caches are significantly faster than accesses that reach the main memory, it is possible to determine if the data resides in the cache or not. Similarly, by simply executing a memory access, it is possible to install data from the memory into the caches. This behavior does not, by itself, affect the correctness of the application, as the behavior on the architectural level remains unaffected, but it can be abused by a malicious user.

Specifically, a malicious user can abuse this in two ways. First, it is possible to monitor the behavior of an application on the microarchitectural level and based on this extract secrets from the application. For example, it is possible to extract encryption keys from some very common ciphers by monitoring their cache access patterns [6] or the types of instructions they are executing [2]. This is made possible because the architectural behavior of the application, e.g. which memory locations it accesses or what types of instructions it executes depends on the values of the bits in the encryption key, and this behavior directly affects the microarchitectural state and behavior of the system in a deterministic way.

The second way in which a malicious user can abuse the microarchitectural state is by constructing a communication channel between parts of the system that are isolated on the architecture level. For example, it is possible to construct a communication channel between two virtual machines (VMs) residing on the same computer [18], even if they are otherwise isolated from one another. This can then be used to covertly
leak information from one machine to the other, bypassing any software or hardware barriers that are in place.

In both cases, in the encryption example and in the VM example, the malicious user has mounted what is referred to as a side-channel attack. Essentially, side-channel attacks are attacks that aim to gain and transfer information using the microarchitectural details of the system rather than flaws in the architecture or the software. Some side-channel attacks, such as the examples provided here, assume that the malicious user has access to the system and is able to execute software on it, but there are also remote side-channel attacks and even physical side-channel attacks. In this thesis we focus on a specific type of side-channel attacks, known as speculative side-channel attacks.

### 1.1 Speculative Side-Channel Attacks

Before explaining the specifics of speculative side-channel attacks, it is important to understand side-channel attacks in general. The two examples already given cover two different types of attacks: Attacks that depend on permanent microarchitectural state changes and attacks that depend on transient microarchitectural state.

The first category (permanent microarchitectural state attacks) exploits microarchitectural structures that hold data for long (in CPU terms) durations of time. Such a commonly exploited structure is the cache hierarchy. As already explained, a user is able to manipulate the state of the cache hierarchy by performing memory accesses. For example, in order to install a cache line into the cache, all one has to do is access that cache line with a normal load instruction. At the same time, it is possible to determine if a cache line exists in the cache, and even in which level of the cache, by simply timing a load access to that memory location. These methods are of course not 100% reliable, as the microarchitectural state might be modified for other reasons, introducing noise into the side-channel. However, much like on any other noisy communication channel, all the attacker has to do is device an appropriate communication protocol in order to overcome this issue. For side-channels that are built on top of the cache hierarchy, it is not uncommon to simply repeat the transmission (modifying the cache state) and the reception (timing memory accesses) multiple times.

The second category (transient microarchitectural state attacks) does

---

1Some ISAs provide limited functionality for manipulating some parts of the cache hierarchy directly, such as the `clflush` instruction on x86. Attacks that utilize such functionality can still be considered as side-channel attacks.
not exploit structures that hold information even after the transmission of the information has completed, but rather transient (temporary) state that exists only while a specific operation is in flight. For example, integer division requires more time and resources to execute than an integer addition. A side-channel can be constructed by having the transmitter perform an integer addition for transmitting a 0 and an integer division for transmitting a 1. The receiver can detect if the transmitter is performing a division or not by trying to also perform a large number of divisions; if the latency for each division is higher than normal, then it means that someone else (the transmitter) is also performing one at the same time. This is also referred to as contention-based attack, since it uses resource contention as the side-channel. Such a side-channel has been demonstrated in practice (e.g. [2, 67]), but it does have a number of issues: It requires precise synchronization between the transmitter and the receiver, only one bit of information can be reliably sent at a time, it requires the transmitter and the receiver to share the same integer division unit at the same time, and finally it is very susceptible to noise and interference from other operations. Such side-channels usually require a higher amount of repetitions before being able to transmit information, reducing the bandwidth available for the attack.

\textit{Speculative} side-channel attacks still utilize these two types of side-

\footnote{Unlike cache based side-channels, where multiple cache lines can be set and probed at the same time.}
channel attacks, the difference is that they do so under speculative execution. Under speculative execution it is possible for the CPU to execute code and perform memory accesses that would have not been possible otherwise. At the same time, under speculative execution, microarchitectural changes are only allowed if they do not affect the architectural state of the system, and incorrectly executed instructions are squashed before they have a chance to be made architecturally visible. So while it might be possible to bypass software and hardware barriers while executing speculatively, it is not possible, as far as the architecture is concerned, to store and then retrieve any of the information obtained. However, it is possible to do exactly this using the microarchitectural state, i.e. by using side-channels.

Specifically, a speculative side-channel attack consists of two parts: Accessing some information illegally while under speculation and transmitting/leaking that information outside of the speculative domain. The first part depends on the aforementioned fact that during speculative execution the CPU might execute code and perform memory accesses that would have not otherwise happened. For example, permission checks for memory addresses might be delayed or branches might be predicted incorrectly. Listings 1.1 and 1.2 contain two example of such attacks, namely the Meltdown (1.1) and the Spectre (1.2) attacks that were the first to demonstrate how speculative execution can be abused. In the case of the Meltdown attack, the malicious code tries to access an address some kernel memory space that it does not have the appropriate permissions for (line 1). In specific CPU implementations, such access checks are not performed immediately while the instruction is still spec-

```
1 byte probe_array[256 * CACHE_LINE_SIZE];
2
3 leak_secret(byte secret) {
4   x = probe_array[CACHE_LINE_SIZE * secret];
5 }
6
7 probe(byte *cache_line) {
8   for (i = 0; i < 256; i++) {
9       t1 = RDTSCP();
10      x = probe_array[i * CACHE_LINE_SIZE];
11     delay = RDTSCP() - t1;
12   }
13 }
```

Listing 1.3: Side-channel code example.
ulative, allowing for the value to be loaded, thus bypassing the memory protection that is normally enforced by the hardware. On the other hand, in the case of the Spectre attack, it is a software protection mechanism that is bypassed. The attacker can train the branch predictor to always expect the if statement (line 2) to be true simply by calling the victim function multiple times with values that cause the if statement to be true. The next time the CPU reaches that branch, even if this time the condition is false, it will still initially speculate that the condition is true, performing the out-of-bounds memory access (line 3). In both this and the Meltdown attack, the CPU will eventually realize that it had speculated incorrectly and it will squashed any incorrect instructions, and with them any illegally obtained information. This is where the second part of speculative side-channel attacks comes in.

While still under speculation, the malicious code can transmit any information it has obtained using a side-channel, such as the aforementioned cache or contention based side-channels. As long as the receiver is executing non-speculatively, the information obtained while under speculation can then be brought into the architectural level. In Listing 1.3 we can see an example of such a side-channel being utilized. Recall that the leak_secret function is being used by the examples in Listings 1.1 and 1.2. In this example, a cache side-channel is used, but other side-channels can be used as well. With the cache side-channel, all the leak_secret function has to do is encode the secret value that is to be leaked as part of an address and access one single cache line (line 4). In this example, a probe array that can encode 256 values is used and the secret is encoded by simply accessing the nth cache line of the array, with n being the secret value. Then, in order to detect which cache line was the one accessed by leak_secret, the cache lines of the array are probed one after the other, in order to determine which one has a fast access time (i.e. which one has been installed in the cache), thus leaking the secret value. As already mentioned, this is just one example of a side-channel that might be used, and the choice of side-channel is, with some restrictions, orthogonal to how the attacker has managed to gain access to the secret information. However, this specific side-channel is particularly interesting because the transmitter and the receiver do not have to be executed in parallel, which creates the opportunity for both to

---

3In practice, almost all instructions are executed speculatively, but this not an issue for the receiver as long as it speculating correctly.

4For this to work correctly, only the cache line brought in by leak_secret should be present in the cache, so the attacker has to have flushed the cache before mounting the attack. Because of this pattern, flushing and the probing the cache, this specific side-channel is referred to as Flush&Reload [83].
be part of the same application. Let us imagine that the victim function in the Spectre example (Listing 1.2) is JavaScript runtime function used by a web browser to ensure that the JavaScript code being executed does not access any memory outside of the JavaScript VM. An attacker could construct code similar to the one given in the example to bypass the software barriers put in place by the web browser and access information outside of the JavaScript VM. Depending on how the web browser is designed, the JavaScript VM might share the same process for multiple web pages or even the same process as sensitive internal browser state, such as stored login information, all of which could be leaked to the attacker. As a matter of fact, modern web browsers have modified their design since the Spectre attack was announced, to combat exactly this issue.

Overall, speculative side-channels attacks can be particularly devastating because they can be executed even if both the software and the hardware implementation is functionally correct. In addition, since speculative execution is a normal component of modern CPUs and is squashed without any visible architectural side-effects, such attacks might leave no visible traces after they have finished executing. In the next section we will discuss what the current state of affairs is when it comes to speculative side-channel attacks, as well as the solutions that have been proposed.

1.2 Current State of Affairs and Related Work

After Meltdown [40] and Spectre [31], numerous other speculative side-channel attacks, many of which are variants of their predecessors, have been introduced [10]. These attacks fall under two categories: Meltdown-style attacks, and Spectre-style attacks. Meltdown attacks take advantage of the fact that in some implementations some permission and correctness checks for memory accesses are done after the load has already received a value. For example, Meltdown works because the page permissions are not checked before performing the memory access (Section 1.1, Listing 1.1). This is an implementation specific problem that is not an inherent part of speculative execution in general, which is why not all CPUs are affected by it. Meltdown attacks are, in essence, an implementation bug. On the other hand, Spectre-style attacks take advantage of the fundamental behavior of speculative execution in order to perform illegal accesses (Section 1.1, Listing 1.2). Of course, in practice, they might still need to exploit specific microarchitectural implementation details, but conceptually Spectre attacks are not just implementation bugs, instead they exploit speculative execution itself.
Regardless of the method of the attack and what they exploit, all attacks need to use a side-channel in order to leak the information from under speculation. For the most part, this side-channel is orthogonal to the attack itself and several different side-channels can be used interchangeably. This thesis focuses on side-channels that try to exploit the memory hierarchy, such as cache [45], directory [81], and DRAM [58] based side-channels. Specifically, in Paper I we focus on side-channels that work by modifying the cache state, while in Papers II and III we discuss all memory based side-channels. Other observable side-effects, such as port contention [3], frequency scaling [65], or even energy usage [32] and EMF radiation [53] can also be used as side-channels, but they are outside the scope of this thesis. Similarly, non-speculative attacks that might use the same side-channels are also outside the scope of this thesis.

To fix these side-channels, numerous solutions have been proposed. As side-channel attacks in general predate their speculative variants, a lot of the existing solutions focus on preventing the side-channel even under non-speculative execution. Specifically for cache side-channels, the majority of the existing solutions focus around cache partitioning [51] [30] or randomization [42, 43, 59]. While such solutions can prevent attacks originating from other execution contexts (or in some cases just from other cores), they cannot protect from attacks originating from within the same context, such as the Spectre attack in our example. Yan et al. [82] have proposed a way of protecting from directory based attacks using victim directories. Finally, using a close-page policy on DRAM modules has been suggested in some works (e.g. [62]) as a way of preventing some of the DRAM based attacks. Overall, all these solutions focus on cache and memory side-channels in general, not on speculative side-channel attacks.

For speculative attacks, as each new attack was announced, software and hardware vendors released patches to fix as many of them as possible. For example, the Linux kernel contains a number of mitigations for different Spectre variants [1]. The MSVC compiler team has also introduced software mitigations for preventing similar attacks in the compiled software [52]. The Chrome web browser team has altered the architecture of the browser to isolate sites from one another and from the main browser [60], to prevent Spectre attacks. Context-Sensitive fencing [70] proposes modifying the decoding process of modern CPUs to introduce additional fences in order to prevent Spectre-type attacks. All of these measures target specific attacks, trying to prevent the illegal access of information, so if new attacks are introduced new methods need to be devised. Also, since a lot of these methods have been developed and
applied outside of academia, it is not easy to find concrete information regarding the performance overheads they introduce. The same is true for hardware based solutions that vendors such as Intel have introduced, either for older processors, or in some cases only for new processors, where it is not even possible to identify what the exact mitigations that have been implemented are.

So far the solutions discussed either focus on all side-channels, regardless if they are used speculatively or not, or focus on mitigating specific attacks by preventing the illegal accesses from happening in the first place. This thesis fits into neither of these two categories. Instead of focusing on all side-channels, the work presented focuses only on speculative side-channels. At the same time, the goal is not to mitigate specific attacks nor is it to prevent the attacker from accessing the information illegally. Instead, the focus is on preventing the leakage of information during speculative execution by preventing speculative execution from causing observable microarchitectural side-effects in the memory hierarchy. Essentially, we are not interested in if or how the attacker might gain access to information illegally, we are only interested in preventing the attacker for leaking that information outside of the speculative domain. Solutions that try to achieve this can be split into three main categories: Solutions that try to hide the side-effects of speculative execution until the speculation has been resolved, solutions that delay the execution of speculative instructions, and solutions that undo the side-effects in the event of a mispeculation.

- **Hiding speculative execution:** Solutions such as InvisiSpec [80], Ghost loads [63] (Paper I), and SpectreGuard [20] allow speculative instructions, specifically loads, to execute speculatively, but hide all side-effects in the cache until the speculation has been resolved. They achieve this by preventing memory accesses from modifying the cache if they originate from a speculative load, and by utilizing special buffers where the data brought in speculatively can be stored. If the speculation was correct, then the data can be installed from the buffers into the cache, otherwise they are discarded. At the same time, while under speculation, the buffers can serve as small caches for other speculative instructions, reducing the performance cost of such solutions. InvisiSpec also contains a coherence solution for maintaining the requirements of the TSO memory model, while under the Ghost loads and SpectreGuard only more relaxed memory models have been considered. On the other hand, Ghost loads have the advantage that they can allow for speculative prefetching, which, as it can be seen in Paper I, has
a significant performance impact.

- **Delaying speculative execution**: Solutions such as delay-on-miss [64] (Papers II and III), NDA [78], Speculative Taint Tracking (STT) [84], and Conditional Speculation [35] prevent speculative side-channel attacks by restricting speculative execution if it can lead to observable side-effects. Specifically, NDA and STT restrict all instructions, in an effort to prevent all side-channel attacks, but at a significant performance cost, while delay-on-miss and Conditional Speculation only focus on memory side-channels. NDA, STT, and Conditional Speculation track safe and unsafe instructions based on the instruction dependency chains, while delay-on-miss introduces the concept of speculative shadows. It should be noted here that what is considered as safe or unsafe differs between each solution, and sometimes multiple versions are presented for different threat models.

- **Undoing speculative execution**: CleanupSpec [62] takes a different approach than hiding or delaying speculative execution, instead it allows instructions to execute unhindered and, in case of a mispeculation, their side-effects are undone. In practice, while this works perfectly for attacks executed from within the same execution context, it is still necessary to obfuscate and delay speculative execution to prevent attacks where the receiver resides in a different context. Specifically, CleanupSpec proposes undoing the side-effects of speculative execution in the L1 cache by keeping track of which cache lines are installed and evicted by speculative loads and evicting or reinstalling them in the event of a mispeculation. For the L2 cache, a combination of evictions and randomization [59] needs to be used instead, to obfuscate the cache state from other execution contexts. Similarly, the L2 needs to also be protected by randomization, and a closed-page policy is proposed for the DRAM. Finally, to prevent changes caused by coherence, the coherence protocol is modified to introduce invisible accesses and loads are delayed if a state change is necessary.

As mentioned, all of these solutions aim at protecting from a different set of speculative side-channels and make different assumptions about when an instruction can be considered safe or not. Because of these, a direct comparison is impossible and, in fact, some solutions are even orthogonal to one another. For example, the dependency-based tracking from NDA and STT can also be implemented as a mechanism
in InvisiSpec or delay-on-miss. It also remains to be seen which, if any, approach the industry decides to adopt.
Chapter 2

Securing the Memory Hierarchy from Speculative Side-Channel Attacks

This thesis presents two different solutions for protecting the memory hierarchy from speculative side-channel attacks, Ghost loads and delay-on-miss. Ghost loads (Paper I) fall under the category of hiding the speculation, by utilizing a small buffer to keep the caches free from speculative side-effects. Delay-on-miss (Papers II and III) on the other hand secures the whole memory hierarchy by selectively preventing speculative loads from executing in the first place. Both of these solutions are based on the concept of speculative shadows, a way of determining the earliest point at which an instruction is no longer speculative, which we have introduced inspired by the work of Bell and Lipasti [5]. The sections that follow contain short descriptions of each paper, while the full papers can be found in Part B of this thesis.

2.1 Paper I: Ghost Loads: What is the Cost of Invisible Speculation?

In this paper we explore the performance trade-offs between preventing speculative execution overall and only preventing its side-effects in the cache hierarchy by introducing Ghost loads, which are “load operations that are undetectable in the memory hierarchy” [63], specifically in the cache hierarchy. Ghost loads have the following characteristics:
1. They are issued like any other normal memory request, regardless if they are speculative or not. Ghost loads are not based on restricting the execution of speculative instructions.

2. They are allowed to hit in any level of the memory hierarchy, but they are not allowed to modify any state such as performing fills in the cache. One exception is the main memory, where the loads are allowed to open and close rows, although a closed-page policy is assumed.

3. They utilize separate MSHRs than the “normal”, non-ghost loads. Coalescing between Ghosts and normal accesses is allowed only if the normal access precedes the Ghost access.

4. They may notify the prefetcher, but data brought in this way are also marked as Ghosts.

Based on this description, Ghost loads are, in essence, uncacheable accesses. However, based on our evaluation, the majority of memory accesses (87% on average) end up being Ghost accesses; if they are all done as uncacheable accesses then the performance is bound to suffer significantly. In our evaluation, we measured a 61% performance drop under the baseline when Ghost loads are implemented.

To remedy this issue, two optimizations are introduced: Materialization (Mtz) and the Ghost Buffer (GhB). Materialization is a replay mechanism where a load issued as a Ghost will be replayed once it is no longer speculative. Similarly to how a software prefetch instruction might work, Mtz requests bring in data to the cache without passing on that data to a specific instruction. As materialization requests are not used for correctness but only as a performance optimization, the loads do not need to wait for the Mtz request to complete before retiring. This is different from the Validations that InvisiSpec [80] utilizes, which are needed to ensure correctness according to the TSO memory model.

The Ghost Buffer is a small, hidden cache attached to each actual cache. Its purpose is to store the data brought in by Ghost accesses, to provide some locality and also to facilitate faster and more efficient Mtz requests. Unlike a normal cache, only other Ghost loads can access the GhB, and only for data that have been brought in by the same execution context. Additionally, the GhB needs to be flushed in the case of a misspeculation, to avoid information leakage. A more detailed discussion on how the GhB can be made secure from threats can be found in the paper. When used in conjunction with the Materialization, Mtz requests are also allowed to access the GhB and install data from
it to the cache. This reduces the delay between a load becoming non-speculative and its data appearing into the cache and also reduces the energy cost of Materialization. When both optimizations are combined, Ghost loads can achieve 88% of the baseline performance with an energy overhead of only 9% over the baseline. It should be noted here that when this paper was published the performance of the then state-of-the-art solution, InvisiSpec, could offer similar security guarantees at 54% of the baseline performance, significantly worse than Ghost loads. However, the authors of InvisiSpec have released a corrected version which significantly improved performance that approaches that of the Ghost loads. Since we have not evaluated the corrected version, we do not have exact numbers to compare with, but two of the main differences are that InvisiSpec supports the TSO memory model, and thus more hardware, while Ghost loads support hardware prefetching, which affects the performance significantly. A more detailed comparison can be found in the paper.

In addition to the Ghost loads, Materialization, and the Ghost buffers, we also introduce in this paper the concept of speculative shadows, a way of determining the earliest point at which an instruction is no longer speculative. They are inspired by the work of Bell and Lipasti [5] on understanding the conditions under which an instruction can be committed, as well as the work on out-of-order commit [6] done in our research group. The main idea is that while, traditionally, all instructions that have not reached the head of the reorder buffer (ROB) can be considered speculative, in practice we can, under certain conditions, guarantee that some instructions will be successfully retired even if they have not reached the head of the ROB. The speculative shadows are a way of defining and also tracking these conditions. Specifically, we can guarantee that an instruction that has executed successfully will be retired successfully if there are not any older instructions ahead of it that might cause the pipeline to be squashed. For example, if an instruction follows a branch instruction, it can be squashed if the branch instruction has been mispredicted. However, as soon as we can determine that the branch condition and target are correct, then we know that it can no longer cause a squash in the pipeline. Before this happens, we refer to the possibility of squashing caused by the branch as a speculative shadow, cast by the branch instruction and covering all instructions that follow the branch. An instruction might cast more than one type of shadow at the same time and loads might be covered by multiple shadows. Based on this, we define four shadow types.

1Some of the terminology used here, such as the “X-Shadow” types, has actually
• **E-Shadows:** Cast by instructions that might throw an exception, such as integer division or memory instructions.

• **C-Shadows:** Cast by control instructions such as branches.

• **D-Shadows:** Cast by data dependencies between stores and loads with unknown addresses.

• **M-Shadows:** Cast by load instructions on architectures where the load-load memory order needs to be maintained, such as on architectures with the TSO memory model.

As long as a load is covered by at least one shadow, then it is issued as a Ghost load. Similarly, Materialization requests can only be sent out safely after the load is not covered by any shadows. This reduces the total number of speculative loads while at the same time allowing for faster (from time of issue) Materializations, making Ghost loads an efficient way for hiding the side-effects of speculative execution in the memory hierarchy.

### 2.2 Paper II: Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction

In this paper we propose an alternative solution to the Ghost loads, based on restricting the execution of speculative loads instead of trying to hide their side-effects. The motivation for this is that while Ghost loads perform well, they have three disadvantages:

1. They require invasive modifications in the whole system, from the core to the memory hierarchy, potentially including the coherence protocol.

2. Since loads are still traversing the memory hierarchy, it is not possible to completely hide all of their side-effects, such as the need to open and close pages in the DRAM.

3. The TSO memory model, used by the x86 architecture, is not supported, at least not without introducing more system modifications and overheads.

_not been introduced until the next paper, Paper II, but it is used here to avoid confusion._
To efficiently solve this issues, especially (1), a different approach is necessary. Based on our work with Ghost loads, and from the conclusions drawn from InvisiSpec, we had gained the following insight: Hiding the side-effects of speculative loads in the memory hierarchy is hard not because of all memory accesses but because of misses specifically. On a miss, multiple levels of the memory hierarchy need to be accessed, data needs to be fetched into the cache(s), and the coherence state needs to be modified. On the contrary, on a hit, specifically an L1 hit, no data needs to be fetched, and the coherence state needs to be modified only under certain conditions (when there is sharing of data). Other updates to the L1 cache, such as updating the replacement information or triggering the prefetcher, are outside of the critical path and can be delayed until later. Based on this insight, in Paper II we propose the delay-on-miss solution, where shadowed loads are allowed to execute if they hit in the L1 cache, otherwise they are delayed until they become unshadowed. Specifically, we suggest allowing loads to execute in the pipeline regardless of an L1 hit or a miss, and instead delaying the request from the L1 to the L2 cache if the loads is shadowed. This simplifies the architecture significantly, as it requires only small modifications to the core, small modifications to the L1 cache, and no modifications to any other part of the system, including the coherence protocol. As memory requests never leave the L1 cache while under a shadow, no side-effects can be caused in any other part of the system. Finally, modern out-of-order
CPUs are already capable of supporting loads with different latencies while maintaining their memory model, so TSO is supported by default, fixing all three disadvantages that we have identified in the Ghost loads.

In this paper we also expand on our work on speculative shadows, by providing more detailed descriptions and introducing a mechanism for efficiently tracking them in hardware. More details can be found in the paper, but the main insight is that speculative shadows can be tracked without the need for expensive content-addressable memories (CAMs). This is due to the fact that it is not necessary to determine the exact shadows under which a load is under, instead it is enough to know if the load is under at least one shadow. This can be done by checking if the oldest shadow that is currently active is older than the load in question. If it is, then it shadows the load, while if it is not then no other shadows can be covering that load.

Finally, in order to reduce the cost of delaying L1 miss and to further improve the delay-on-miss approach, we introduce a value predictor. If a load is executed under a shadow and misses in the L1, the value predictor tries to provide a value, preventing the load from stalling. A high-level architectural overview of this can be seen in Figure 2.1. The key insight is that while value prediction is in itself a form of speculation, it can be an invisible form of speculation, local to the core and isolated between execution contexts. However, because they are speculative, loads that have been value predicted cast a new shadow, the VP-Shadow, and need to be validated by performing the actual memory request and comparing the predicted with the actual value. As such a memory request would cause observable side-effects in the memory hierarchy, we delay them until after the load has been unshadowed. In the results presented in Paper II, delay-on-miss achieves secure, invisible speculation with 81% of the baseline performance and value prediction increases that to 89%.

This results are close to those of the Ghost loads, but delay-on-miss is more secure, supports more memory models, and is simpler to implement, giving it the advantage.

\(^2\)We will see in the next paper, Paper III, that the results for the value prediction are not as accurate as we initially thought and that the performance benefits of it are not as significant, but delay-on-miss does maintain the same performance.
2.3 Paper III: Understanding Selective Delay as a Method for Efficient Secure Speculative Execution

Paper III is an extension to Paper II, involving a more detailed and accurate performance evaluation, as well as an analysis on why delay-on-miss performs so well when compared to other delay-based solutions. Specifically, after discovering some inaccuracies in our results for the value predictor, we performed a more accurate evaluation that resulted in the value predictor providing a significantly smaller benefit to delay-on-miss. The reason for this, as well as the reason why delay-on-miss performs so well, is memory level parallelism (MLP).

Exploiting MLP (in the positive sense) is necessary for achieving high performance on modern CPUs, as multiple long-latency loads can be overlapped, reducing the overall delay introduced in the pipeline. Delay-on-miss, by design, allows memory coalescing (overlapping) between shadowed and unshadowed loads, as the latter can safely modify the state of the cache. In addition, shadowed loads can also be coalesced with other, non-speculative non-load instructions, such as stores. Finally, shadowed loads can be coalesced with other shadowed loads, although they are not allowed to initiate a memory access from the L1 to the L2 before at least one of them is unshadowed. These conditions allow for a large amount of MLP to be exploited, especially compared against solutions that delay all loads, not just those that miss in the L1. However, as delay-on-miss does impose restrictions on the execution of the loads, some of the MLP exploited in the baseline is still lost, leading to delay-on-miss achieving 82% (updated from Paper II) of the baseline performance. In the detailed evaluation presented in the paper, we can see how the benchmarks that suffer most in performance are all benchmarks where a significant reduction in the amount of exploited MLP can be observed.

By introducing value prediction, we can regain some instruction level parallelism (ILP) but not any MLP. This is because, as explained above, the predicted values need to be validated and these validations are memory accesses performed under the same delay-on-miss restrictions as any other access. So while the load is able to receive a value from the value predictor and execute quickly, it cannot be retired before it has been validated, moving the need for MLP and the cost of reducing it from the execution to the validation stage. Because of this, with the updated simulation model, value prediction provides minimal performance improvements over delay-on-miss, at 83% under the baseline.
Instead, for the value predictor to significantly improve the delay-on-miss (at 92\% of the baseline) approach, the validations would have to be sent out immediately, which would compromise the security guarantees of delay-on-miss. This makes the usefulness of the value predictor questionable, while delay-on-miss still performs well on its own, providing security with low complexity, making it an ideal stepping stone for future research.
Chapter 3

Conclusions

In this thesis we see how speculative execution can be abused with malicious code to bypass software and hardware barriers and to leak sensitive information. While there are a lot of details and different parameters on how this can be achieved, the fundamental issues that enable this are i) that speculative execution, by definition, might execute incorrect code and ii) that microarchitectural changes made in the system during speculative execution can be observed by the malicious code through side-channels. Our work focuses on preventing speculative side-channel attacks on the memory hierarchy by reducing the amount of microarchitectural state exposed during speculation, thus tackling the second fundamental issue enables such attacks.

Specifically, we discuss two different approaches for tackling this issue, one based on hiding the side-effects of speculative execution in the memory hierarchy (Ghost loads, Paper I) and one based on selectively delaying unsafe instructions (delay-on-miss, Papers II and III):

- **Ghost loads** are invisible memory accesses that can be used to prevent memory loads from causing observable microarchitectural changes in the cache hierarchy. Essentially, Ghost loads are memory loads that are not allowed to modify the cache state, operating similar to uncacheable accesses. As the majority of loads are issued speculatively, we also introduce the Ghost buffer, and Materialization, two techniques for improving the performance of Ghost loads by caching them (Ghost buffer) and replaying them in the memory hierarchy when they are no longer unsafe (materialization).

- **Delay-on-miss** is a technique for making speculative execution invisible in the memory hierarchy by simply delaying the execution of instructions that might cause observable microarchitectural
side-effects. Specifically, it is based on the observation that it is easy to hide the side-effects of a memory accesses when it hits in the private L1 cache, but not so easy when it misses. With delay-on-miss, we selectively delay only the memory accesses that miss in the L1 cache, which leads to significantly reduced overheads, when compared with a solution that delays all memory accesses. In Paper II we present the delay-on-miss idea, alongside an optimization technique where a value predictor is used to provide values for the delayed memory accesses, in an effort to further reduce the performance overheads. In Paper III, we revisit delay-on-miss in more detail, offering a more in-depth view in the effects it has to the system’s memory level parallelism and, by extension, to the system’s performance. We also revisit the value predictor, showing that it is, in fact, not the correct optimization for delay-on-miss based systems, and explain why.

To define when a speculative instruction is safe or unsafe, we also introduce the concept of speculative shadow, a way of determining the earliest point at which an instruction can no longer be squashed. Other than the performance gains they offer, speculative shadows also have the advantage that they can be easily tracked in the hardware. By combining them with the Ghost loads and the delay-on-miss, we are able to achieve secure speculative execution while keeping the performance, energy, area, and complexity costs low.
Part B
Ghost Loads: What is the Cost of Invisible Speculation?

Christos Sakalis
Mehdi Alipour
Alberto Ros
Alexandra Jimborean
Stefanos Kaxiras
Magnus Själander

In Proceedings of the
16th ACM International Conference on Computing Frontiers (CF)
New York, NY, USA, 2019
Abstract

Speculative execution is necessary for achieving high performance on modern general-purpose CPUs but, starting with Spectre and Meltdown, it has also been proven to cause severe security flaws. In case of a misspeculation, the architectural state is restored to assure functional correctness but a multitude of microarchitectural changes (e.g., cache updates), caused by the speculatively executed instructions, are commonly left in the system. These changes can be used to leak sensitive information, which has led to a frantic search for solutions that can eliminate such security flaws. The contribution of this work is an evaluation of the cost of hiding speculative side-effects in the cache hierarchy, making them visible only after the speculation has been resolved. For this, we compare (for the first time) two broad approaches: i) waiting for loads to become non-speculative before issuing them to the memory system, and ii) eliminating the side-effects of speculation, a solution consisting of invisible loads (Ghost loads) and performance optimizations (Ghost Buffer and Materialization). While previous work, InvisiSpec, has proposed a similar solution to our latter approach, it has done so with only a minimal evaluation and at a significant performance cost. The detailed evaluation of our solutions shows that: i) waiting for loads to become non-speculative is no more costly than the previously proposed InvisiSpec solution, albeit much simpler, non-invasive in the memory system, and stronger security-wise; ii) hiding speculation with Ghost loads (in the context of a relaxed memory model) can be achieved at the cost of 12% performance degradation and 9% energy increase, which is significantly better that the previous state-of-the-art solution.
I.1 Introduction

Side-channel attacks rely on shared microarchitectural state and behavior to leak information. Side-channel attacks on the cache system have been practically demonstrated in many forms for the L1 (when the attacker can share the same core as the target) [8], the shared LLC cache (when the attacker can share the LLC) [83], and the coherence protocol (when the attacker can simply be collocated in the same system, under a single coherence domain, with the target) [25]. While side-channel attacks have been known to the architecture and the security communities for years, a new type of speculative side-channel attacks has recently surfaced, with the most well known ones being Spectre [31] and Meltdown [40].

As far as the target program is concerned, leaking information across a covert side-channel is not illegal because it does not affect the functional behavior of the program. The stealthy nature of a speculative side-channel attack depends on the microarchitectural state being changed by speculation even when the architectural state remains unaffected.

In this paper, we are concerned with evaluating methods to defend against these kinds of attacks and their performance impact. We are not concerned with how the target program is coerced into executing code that leaks information, as this is orthogonal to the existence of speculative covert side-channels that leak information to the attacker. Instead, the question we are answering is: What is the cost of shutting down the speculative covert side-channels existing in the cache hierarchy?

The main target is to guarantee that no microarchitectural state throughout the system can be observed changing during speculative execution. The obvious ways to achieve this are:

1. Do not speculate (e.g., wait until memory-access instructions become non-speculative). This is not an attractive solution for general-purpose computing, as speculative execution offers substantial performance benefits.

2. Speculate but obfuscate microarchitectural changes so that an attacker cannot discern microarchitectural changes due to speculation [76, 77, 75].

3. Speculate but do not change microarchitectural state until the speculation can be resolved. The insight behind this idea is that speculative execution by itself is not the problem, rather the problem is speculative execution of instructions that should not have been executed to begin with, i.e., transient instructions.
The first choice is, intuitively, detrimental for performance, as we will have to wait for all memory-access instructions to become non-speculative. Waiting for loads to become non-speculative is similar to disabling speculation in general, as applications contain a large number of loads and all computations depend on loaded values. Still, we evaluate the cost of disabling speculation for loads and compare it against other solutions. Our evaluation indicates that, even though the cost is high (−50% to −74% performance, depending on the implementation), competing solutions (e.g., InvisiSpec) might come at a similar cost.

The second choice is akin to existing proposals for preventing side-channel attacks (for example partitioning or randomization [75]), but to the best of our knowledge, no such solution exists for speculative attacks. Current obfuscation approaches can only protect from side-channel attacks that take place with the attacker on a different address space than the victim. As the authors of Spectre show [31], it is also possible to perform devastating attacks from within the same context, for example using the JavaScript JIT compiler found on all modern web browsers. Furthermore, a lot of the work around obfuscating the access patterns focuses on protecting small regions of code that hold sensitive data such as encryption keys. However, the encryption keys are not the only sensitive data in the system. For example, on a web browser, the user’s passwords are sensitive information, but so are a lot of the rendered web pages. Because of these reasons we do not evaluate this as a viable solution against speculative side-channel attacks.

The third choice is having speculative memory-access instructions that are untraceable. In our proposal, we call such accesses Ghost loads. For example, a speculative load that hits in the cache is untraceable if it does not modify the replacement state. If it misses in the cache, it does not cause an eviction, and if it reaches the coherence domain it does not modify the coherence-protocol state. Any prefetches generated because of that load are also made untraceable, preventing attackers from leaking information by training the prefetcher. Recent works [80, 27], as well as industry have shown interest in this type of solution. We propose our own variation that we evaluate in detail to get insights into its performance and energy cost. Since this is a new type of solution for a new type of problems, we are interested in understanding the behavior of such untraceable accesses in the memory system.

In this paper, we first explore the trade-off between delaying an access (no speculation) and issuing it as a Ghost load. We then explore the performance implications of Ghost loads, and how they can be improved, achieving high security with low-performance cost. We compare both the non-speculative solutions and the Ghost loads with the current
state-of-the-art solution, InvisiSpec \cite{80}. We show that, even though Ghost loads take a similar approach to InvisiSpec, such a detailed performance evaluation is critical, as the added complexity introduced by InvisiSpec is not supported by the performance achieved. In fact, we will show that similar performance can be achieved simply by delaying all speculative accesses, without the need for any modifications to the memory hierarchy and the cache coherence protocol.

In addition to loads, speculative stores must have similar properties with the additional requirement that the stored value remains speculative. This is already accomplished by the use of a store queue, so in this paper we are only concerned with loads. We focus on presenting a detailed evaluation of single-threaded applications, and as such, due to space constraints, coherence implications will not be evaluated in detail.

In summary, we evaluate the following:

- **InvisiSpec**: We evaluate and compare InvisiSpec \cite{80}, the current state-of-the-art solution, with our own proposals.

- **Non-Speculative (Non-Spec)**: Speculative loads are not issued and are instead delayed until they are no longer speculative. We evaluate two versions, one where all loads are delayed until they reach the head of the reorder buffer (ROB) and one where they are only delayed until they are guaranteed to not be squashed by another instruction. We call these two versions **Naive** and **Eager**, respectively.

- **Ghost loads**: Speculative loads are executed as Ghost loads, which are not allowed to modify any architecturally visible state. In practice, this prevents caching for a large percentage of the loads in the system. To mitigate the performance cost, we also evaluate two additions to the Ghosts: The Ghost Buffer (GhB), a small cache used exclusively by Ghost loads, and Materialization (Mtz), which instantiates the side-effects of Ghost loads after the speculation has been resolved.

- **Ghost Prefetching**: We propose and evaluate a method for performing prefetching of Ghosts loads, something that is missing from the current state-of-the-art solution.

Our results reveal that the Non-Spec solutions incur significant costs, with 75\% and 50\% performance loss for the Naive and Eager version, respectively. However, so does InvisiSpec, which shows similar performance to the Eager Non-Spec solution, but with additional hardware
complexity. Ghost loads, with the Ghost Buffer, Materialization, and prefetching, show only 12% performance loss, accompanied by a 9% increase in energy usage. Finally, without prefetching of Ghost loads, the performance loss is increased to 22%.

I.2 Speculative Shadows

Speculative execution works by executing instructions and hiding their architectural side-effects until it is certain that the speculation was correct. In case of a misspeculation, the misspeculated instructions are squashed and execution is restarted from the initial misspeculation point. The instructions that were executed but then squashed are often referred to as transient instructions. In practice, almost all instructions are executed speculatively, with few exceptions. In modern out-of-order (OoO) processors, non-speculative execution is achieved by waiting until an instruction is at the head of the reorder buffer (ROB) before being executed. For our work, we need to be more specific with which instructions are speculative and which are not, so we define the concept of speculative shadows. When an instruction that can cause the CPU to misspeculate is inserted in the ROB, it casts a speculative shadow to all following instructions. The shadow can be lifted either when the instruction leaves the ROB, or if possible, when it can be determined that the instruction can no longer cause a misspeculation. For example, an unresolved branch causes a shadow to all instructions following it, but after the branch is resolved and the branch target can be compared with the speculated branch target, the speculative shadow can be lifted. Essentially, sources of speculation are all instructions that can cause the wrong instructions to be executed speculatively, which will then have to be squashed. We have categorized the causes of speculation into four major classes:

**Control:** If the target of a branch is not known, then it may be mispredicted, causing a misspeculation. This includes not only branches but also all instructions that the branch predictor and the branch target buffer (BTB) might identify as a branch.

**Stores:** Stores can cast a speculative shadow for three reasons. Since they are memory operations, they might try to access either memory that is i) unmapped or ii) memory that the current execution context does not have write permissions for. In that case, an exception will be thrown and execution will have to be diverted. Additionally, iii) if unknown addresses are involved, the store might be conflicting with another store or a load on the same location.
Loops: Much like stores, they cast speculative shadows because of exceptions or conflicts with other memory operations. Additionally, the coherence protocol can dictate that a speculatively loaded value has to be invalidated, to enforce the CPU’s memory model.

Operations causing exceptions: This includes every floating point operation, and integer division. For floating point operations, exception throwing can usually be controlled by the programmer, allowing the system to know in advance if floating point operations can throw exceptions or not. Other instruction types that can cause exceptions are rare in benchmark suites like SPEC so we do not consider them for this work, but they can be handled the same way we handle arithmetic operations.

We make the observation that, in order to keep track of whether a load is under a speculative shadow or not, it is enough to know if the oldest shadow casting instruction is older than the load in question. We leverage this to track shadows in a structure similar to a reorder buffer (ROB) but much smaller as only a small identifier instead of the complete instruction is needed to be stored. We call this structure the shadow buffer or SB for short. Shadow-casting instructions are entered into the shadow buffer in the order they are dispatched. Once an instruction no longer causes a shadow, e.g., once a branch has been resolved, then the SB entry is updated. Only when an instruction reaches the head of the SB and no longer casts a shadow does it get removed from the SB. This assures that the oldest shadow-casting instruction is al-

\[1\] Not to be confused with InvisiSpec’s Speculation Buffer.
Figure I.2: A breakdown of the instructions casting speculative shadows on executed loads.

Figure I.1 displays the ratio of loads executed speculatively in each benchmark, based on the conditions discussed above. The ratio of speculatively executed loads is high for all benchmarks, ranging from 67% (h264ref) and up to 97% (GemsFDTD), with a mean of 87%. This strongly indicates that any proposed solution will have to have low overhead, as the majority of the load operations will be affected. We will further discuss this subject in the evaluation, see Section I.6.

Figure I.2 presents a breakdown of the type of instructions casting shadows over executed load instructions in the applications found in the SPEC 2006 [68] benchmark suite. The hardware parameters of the evaluated system can be found in Table I.1. Note that only the oldest shadow is taken into consideration and eliminating one of the shadow types will not necessarily lead to an equal decrease in the number of speculatively executed loads. Previous work by Alipour et al. [4] discusses...
the implications of eliminating different causes of speculation in modern out-of-order processors. We observe that the majority of the speculation is caused by the first three categories: control (branches), stores, and loads. For applications that have frequent and irregular control flow, such as gcc, the branches cause the majority of the speculation. On the other hand, in applications that utilize more regular operations, such as the mathematically heavy bwaves and cactusADM, the speculation is caused mostly by loads or stores. Overall, for the majority of the applications, we observe that not just one type of operation is responsible for causing speculative execution of loads in each benchmark.

I.3 Non-Speculative Loads

An intuitive solution for hiding speculative loads is to not perform speculative loads in the first place. We evaluate two versions of this solution, the Naive and the Eager Non-Speculative (Non-Spec). In the Naive version, all loads in the application are delayed until they reach the head of the ROB, ensuring that they cannot be squashed by any other instructions. In the Eager version, loads are only delayed until they are no longer covered by a speculative shadow. With the eager approach, loads are delayed for a smaller period of time and some loads are not delayed at all (Figure I.1). With these two versions, we provide both an upper and a lower bound for the cost of disallowing the execution of speculative loads.

An interesting property of the solutions that simply delay speculative loads is that no additional steps are necessary in order to support the TSO memory model. Out-of-order CPUs that provide TSO already have all the necessary mechanisms to ensure that instructions being scheduled and executed out-of-order do not break the guarantees of the memory model. Since the non-speculative solutions described only affect the scheduling of the load instructions, TSO can be supported out-of-the-box. Expectedly, more relaxed memory models, such as the popular Release Consistency (RC), are also supported without any modifications.

Another benefit of the non-speculative solutions is that they prevent visible side-effects not only in the caches but also in the TLBs, the main memory, the coherence state, and any other part of the system a data fetch operation might affect. Other solutions have to either explicitly provide ways of hiding the side-effects in the memory system or risk leaking information. For example, Pessl et al. [58] have already developed a side-channel that exploits the timing of the DRAM system instead of the cache hierarchy.
I.4 Ghost Loads

The principle behind invisible speculation we focus on is performing speculative loads as uncacheable accesses that do not alter the cache state. In our work, we call these uncacheable accesses “Ghosts”. A Ghost load is a load operation that is undetectable in the memory hierarchy, specifically in the cache hierarchy. Ghost loads have the following characteristics:

1. They are issued like any other memory request.

2. They can hit on any level of the memory hierarchy including private caches, shared caches, and main memory, in which case the response data are returned directly to the core. The replacement state in the cache remains unchanged.

3. In case of a miss, no cache fills are performed with the response data, and no coherence states are modified.

4. They use a separate set of miss status handling registers (MSHRs) that are not accessible by regular loads. Coalescing between Ghosts is allowed only if they belong to the same context, and so is coalescing Ghosts into in-flight regular loads. Coalescing regular loads into Ghosts is not allowed.

5. Any prefetches caused by Ghost loads are also marked as Ghosts. This assures that an attacker will not be able to train the prefetcher and abuse it as a side-channel. How Ghost prefetches are made possible is discussed in Section I.4.2.

6. Similarly to the data caches, the relevant translation lookaside buffers (TLBs) are also not updated during the lookups performed by Ghost requests.

In practice, it is not possible to have memory operations that are completely side-effect free. For example, even if we disregard any updates to the state of the system, simply by performing a memory request it is possible to introduce detectable contention in the system. Ghost loads and similar techniques aim to balance the exposure of the side-effects of speculation while also limiting the performance and energy costs.

As Ghosts do not interact with the coherence mechanisms of the memory system, only memory models that by default do not enforce any memory ordering are supported, such as the popular RC model. Under
I.4. Ghost Loads

RC, memory ordering is enforced through explicitly placed fences, while all other (non-synchronizing) instructions are free to execute with any order. When a special instruction, such as a memory fence or an atomic operation is detected, it acts a speculation barrier, preventing loads that proceed the fence in program order to be issued before it. This way, no Ghost loads can be reordered with the fence and the memory order is enforced through the underlying coherence mechanism. More restrictive memory models, such as TSO, require additional mechanisms (e.g. Validations in InvisiSpec) that can lead to additional performance overheads. Evaluating such mechanisms is beyond the scope of this work, so we will assume that the Ghost loads mechanism only supports RC or other similarly relaxed memory models.

I.4.1 Materialization

Performing the majority of load accesses as Ghosts can lead to a significant performance degradation, caused primarily by the disruption of caching. To regain some of that lost performance, the data used by a Ghost load can be installed in the cache after the load is no longer speculative. Materialization (Mtz) is a mechanism for achieving that, by performing all the microarchitectural side-effects of the memory request after the load is no longer speculative. When a load is ready to be committed, an Mtz request is sent to the memory system. The request will act as a regular load request, with the difference that it will not load any data into a CPU register. As such, it will install the cache line into the appropriate caches and update the replacement data. However, in order to limit the number of Mtz requests sent into the memory system, when a cache receives an Mtz request for data it already contains, it will not forward that request to any other caches in the hierarchy. Finally, if an excessively large number of requests is generated, the older requests will be discarded. An additional, alternative form of Materialization that does not act as a normal memory request will be discussed in the next section.

I.4.2 Ghost Buffer

The Ghost Buffer (GhB) is a very small (e.g., eight entries for the L1), read-only cache that is only accessible by Ghost or Mtz requests. Multiple Ghost buffers exist in the system, each attached to its respective cache. Any data returned by a Ghost request are placed in the GhB instead of the cache. It is also possible to facilitate prefetching of Ghost requests, by modifying the prefetcher to recognize Ghost requests and
tag prefetches initiated by them as Ghosts. The prefetched cache lines can later be installed by the GhB into the cache when the speculation has been resolved.

While introducing the GhB by itself improves the performance of the Ghosts, it is when combined with the Materialization mechanism that the GhB really excels. Specifically, when an Mtz request misses in a cache, it then checks the GhB. If the data are found, then they are installed in the cache, eliminating the need to fetch them from somewhere else in the memory hierarchy. It is even possible to not let the Mtz packets reach the main memory, which is what the evaluation in Section 1.6 is assuming.

Since the GhB is itself a small cache, it can be susceptible to the same side-channel attacks that regular caches are, in this case referred to as “transient speculative attacks” [27]. These are attacks that specifically target the structures used to hold the data for transient instructions. To prevent this, the design and behavior of the GhB needs to be adapted accordingly, both for attacks originating from a different execution context and for attacks originating from the same context.

**Different Execution Context**

For attacks involving a different execution context, we need to make sure that the entries in the GhB belonging to different contexts are isolated. Previous works [77, 30, 33, 19, 34, 76, 43, 26, 51, 42, 22] have already identified solutions to achieve this in regular caches, but given the special characteristics of the GhB, we propose the following:

**For L1 caches** We suggest flushing the L1 GhB every time there is a context switch. Additionally, to support simultaneous multi-threading (SMT), the L1 GhB is statically partitioned between the different threads. This assures that it is not possible for one execution context to access the L1 GhB of another context. Since the GhBs are read-only, no write-backs are required during a flush operation, which can be achieved simply by resetting all the valid bits in the GhB metadata.

**For other caches** We instead suggest using a solution that randomizes the cache placement based on the execution context. This can be achieved by associating a random bit mask with each context and then XORing the address bits with that mask. By changing the mask during flushing, we can prevent an attacker from deciphering the access pattern of the application or the mask. Since the GhB needs to be efficiently flushed only for a specific context, without flushing the rest of the data,
we propose associating each cache line with its context ID and an epoch timestamp, as proposed by Yan et al. [80]. Each time the pipeline is squashed due to a misspeculation, the epoch is increased. By only allowing Ghost requests to access data from the GhB when the context ID and the current epoch match, we are effectively flushing the cache without the need to wait for the GhB to actually be flushed, which would introduce delays for GhBs other than the L1.

**Same Execution Context**

The solutions described above protect the GhB from attacks from a different execution context, but it is still possible to orchestrate an attack from within the same context. For example, a JavaScript JIT compiler running on the same thread as the main browser process can potentially leak sensitive user information. These attacks are harder to defend against, since we do not want to isolate the accesses from the same context from one another. To solve this issue, we flush the GhB every time a misspeculation is detected and the transient state needs to be squashed. This prevents an attacker from first using speculative execution to load data in the GhB and then initiating a separate speculative region to extract the previously loaded data.

For example, an attacker could use a Meltdown variant to read a secret value from privileged memory, and then use it to index a probe array. After the misspeculation has been corrected and the execution has been restored, the attacker can trigger a second speculative region where the probe array is probed. By timing the second region, the attacker can identify if the probe was a hit or a miss in the GhB, and by extension extract the secret value. By flushing the GhB between speculative regions, this is no longer possible. Instead, the attacker has to incorporate everything in one speculative region. This makes the attack very hard for two reasons: First, to prevent the program from crashing, the speculative region needs to misspeculate. This means that the only information that can be extracted from the speculative region is how long the execution took. Second, the execution time of the speculative region depends only on the misspeculated instruction that initiates it. The moment the instruction is determined to have been missspeculated, execution is aborted and squashed, unaffected by the timing of any other instructions in the region. The only possible way to change the time the region takes to execute is to affect the timing of the initial speculative instruction, which is not easy to do in a way that depends on the loaded secret value, as the instructions that read and use the secret value succeed (in program order) the initial speculative instruction.
I.5 InvisiSpec

InvisiSpec [80], much like Ghost loads, blocks speculative side-channel attacks in the cache hierarchy by hiding the side-effects of speculative loads until the speculation has been resolved. This is achieved by preventing speculative loads from disturbing the cache state in any way and instead installing the data in a small, temporary buffer in the core. After the speculative shadow has been resolved, the data are then verified and installed in the L1 cache.

InvisiSpec takes a similar approach to our Ghost loads but with two major differences. First, the buffer utilized by InvisiSpec to hide the speculative data has a one-to-one correspondence with the entries of the load queue (LQ). In contrast, the Ghost Buffer functions as a read-only cache that might contain any random set of cache lines. Because of this, Ghost can support prefetching that is triggered by speculative loads, while InvisiSpec can only safely prefetch non-speculatively.

The second difference is that, in order to support total store order (TSO) coherence, InvisiSpec needs to validate the data from the speculative buffer before installing them in the cache. This means that load instructions need to wait for the validation to succeed before being committed, potentially increasing the pressure on the ROB and the LQ. Additionally, only one validation at a time (per execution context) can be in-flight in the system. This limits the amount of memory level parallelism (MLP) that InvisiSpec can take advantage of, even when optimizations to convert some of the validations to what they call exposures, which are not constraint by the same limitations. Ghost loads only support release consistency (RC) and are not constrained by any of these issues. We will see in the evaluation (Section I.6) how these differences affect the performance of the two approaches.

I.6 Evaluation

We start by discussing the characteristics of the Ghost loads and then proceed to the effects that the various evaluated solutions have on the memory behavior of the applications, as well as the performance and energy implications.

I.6.1 Methodology

We evaluate the different solutions and the suggested improvements using the SPEC2006 benchmark suite [68], from which we exclude five
I.6. Evaluation

Table I.1: The simulation parameters used for the evaluation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology node</td>
<td>22nm</td>
</tr>
<tr>
<td>Processor type</td>
<td>out-of-order x86 CPU</td>
</tr>
<tr>
<td>Processor frequency</td>
<td>3.4GHz</td>
</tr>
<tr>
<td>ROB/IQ/LQ/SQ entries</td>
<td>192/64/32/32</td>
</tr>
<tr>
<td>Decode/Issue/Commit width</td>
<td>8</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 private cache size</td>
<td>32KiB, 8-way, 8 entries GhB</td>
</tr>
<tr>
<td>L1 private cache access latency</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 shared cache size</td>
<td>1MiB, 16-way, 256 entries GhB</td>
</tr>
<tr>
<td>L2 shared cache access latency</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>

applications due to simulation issues encountered in the baseline simulation. We use the Gem5 [7] simulator combined with McPAT [36] and Cacti [37] for the performance and energy evaluation. Each Ghost Buffer (GhB) is modelled as a small cache in McPAT, on the same level of the hierarchy as the cache it is attached to. For the DRAM, we use the power model built into Gem5, as McPAT does not provide one. We perform the simulation by first skipping one billion instructions in atomic mode and then simulating in detail for another three billion instructions. The characteristics of the simulated system can be found in Table I.1. We simulate a system with a private L1 and a shared L2 cache. As the baseline we use a large, unmodified OoO CPU. For InvisiSpec we only simulate the TSO version because, according to its authors, the performance is not improved significantly in the RC version [80].

I.6.2 Ghost Loads

Before discussing the performance and energy implications of the proposed solutions, we need to first understand the behavior of the Ghost loads, and how they interact with regular memory accesses.

Figure I.3 presents the number of consecutive Ghost loads to a cache line between two regular loads, and vice versa. To simplify the figure, only the average is presented. We observe that for all benchmarks, the average number of accesses is very small, around two consecutive accesses for most, both for Ghost and for regular loads. We also know from our data (not shown) that the number of cycles between consecutive loads, either Ghosts or regular, is very small, which is not surprising given how common loads are in the instruction mix. These numbers indicate that the data stored in the GhB will be short-lived, as when a
regular access installs data in the cache the GhB data becomes obsolete. In addition, Materialization requests need to be fast, as regular accesses to the same cache line follow closely after the Ghosts. These observations indicate that large buffers holding all speculative data are unnecessary, as quite often some other load instruction will install the data in the cache before the speculative load has a chance to.

We have also observed that, with the exception of Non-Spec, all solutions increase the number of loads that are executed as Ghosts, because the introduced delays in the execution introduce, in turn, more speculation in the pipeline.

I.6.3 Memory Behavior

With the exception of the Non-Spec solutions, the proposed methods alter how the cache hierarchy works. Hence, the behavior of the cache is what primarily affects the performance and energy characteristics of the system.

Figure I.4 features the L1-data and L2 cache miss ratios for all the different solutions. Both Naive and Eager Non-Spec, which we present as alternatives to invisible speculation, reduce the number of L1 and L2 misses, as well as the number of DRAM reads. This is due to the memory
### Figure I.4: L1-data & L2 cache miss ratios.
Figure I.5: Normalized DRAM reads. The number of DRAM writes is not affected by the different solutions.

Figure I.6: Normalized performance (IPC).
accesses and the overall execution being slowed down, which provides more time for the memory system to respond to requests. Additionally, without speculative execution, only the data that are actually needed by the applications are read and brought into the caches. However, whether Non-Spec reduces the amount of cache misses or not is irrelevant, as it does not change how the cache system works. Instead, we will see in the next section that the cost of the Non-Spec methods is observed in the performance of the benchmarks. For this reason, we will only focus on the Ghosts and InvisiSpec in this section.

Both Ghost loads and InvisiSpec leave the mean L1-data miss ratio unaffected. If we examine each benchmark individually, we will see that there are some benchmarks were the miss ratio is increased, with the worst case being libquantum for InvisiSpec, but overall there are no significant differences. The same is not true for the L2 cache, where InvisiSpec shows increased miss ratios in a number of benchmarks, with the most prominent ones being zeusmp, leslie3d, and sphinx. Ghosts also see an increase in the miss ratio, but not as significant, with the most problematic applications begin bwaves and leslie3d. Overall, we observe a bigger variation in the L2 miss ratios than we do in the L1, with mean miss ratio increases of 25% for InvisiSpec and 13% for the Ghosts.

We can observe these differences more prominently in the number of DRAM reads performed in the system, as seen in Figure I.5. We only focus on the reads because the number of writes are not significantly affected by any of the evaluated solutions. InvisiSpec features a mean increase of 31%, while Ghosts are at 27%. The worst applications are zeusmp and libquantum for InvisiSpec and mcf and gromacs for Ghosts, with all four featuring more than 2 × reads when compared to the baseline.

Overall, we can conclude that both InvisiSpec and Ghosts introduce memory system overheads, with the Ghosts outperforming InvisiSpec by a few percentage points. On the other hand, the Non-Spec solutions do not have such negative side-effects.

I.6.4 Performance

Figure I.6 presents the relative performance of the various simulated solutions, in the form of instructions per cycle (IPC) normalized to an unmodified out-of-order CPU. As anticipated, the Non-Spec solutions, where loads are executed non-speculatively, suffer from a steep performance loss. We observe a mean performance loss of 75% for the Naive version, and 50% for Eager. Load instructions are very common in applications, and all computation depends on the values loaded from
memory. We also know that the large majority of loads in the SPEC2006 benchmarks are speculative (Figure 1.1). The combination of these two facts means that with the Non-Spec solutions not only are most loads delayed, but that these loads also constitute a large and latency-critical part of the applications. In essence, by using the Non-Spec solutions, we force the large and power hungry out-of-order CPU to execute with similar constraints to that of a slower, strict in-order CPU but without the accompanying area and power reduction benefits. In the Naive Non-Spec in particular, which makes it impossible to have more than one load in flight in parallel, the CPU cannot take advantage of the memory level parallelism (MLP) available in the applications.

However, we can also observe quite similar results with InvisiSpec, which reaches a mean performance loss of 46%, just 4% points better than the Eager Non-Spec version. InvisiSpec is even outperformed by the Eager Non-Spec in five benchmarks, namely zeusmp, leslie3d, libquantum, lbm, and omnetpp. Given the reduced complexity, additional security, and reduced area overhead, these performance results indicate that simply delaying speculative instructions is in fact a better alternative to InvisiSpec.

However, neither Eager Non-Spec or InvisiSpec can compete with the Ghost loads when it comes to performance, because the latter is featuring a mean performance loss of only 12%. In addition, Ghosts consistently offer good performance, outperforming InvisiSpec in every single benchmark, and with only two applications, bwaves and leslie3d, dropping below the −25% mark. For both of these applications, we observe in Figure 1.4 that they suffer from an increase in the L2 miss ratio. However, that in itself does not explain the performance loss, as other applications have the same problem. Instead, by analyzing the detailed statistics made available from Gem5, we observed that they also suffer from a large increase in the number of MSHR misses, both in the L1 and the L2, and particularly MSHR misses for Ghost accesses. Other applications also suffer from an increase in MSHR misses, but without a similar increase of the L2 miss ratio. This indicates that not only are bwaves and leslie3d suffering from an increased miss ratio, but also that their available MLP is not fully harnessed. Since regular accesses cannot be coalesced with in-flight Ghosts due to security, and we know from Figure 1.3 that regular accesses and Ghosts are tightly interleaved, not all of the available MLP in the application can be taken advantage of.
Figure I.7: Normalized energy usage. The bottom (shaded) part represents the static (leakage) energy of the system.

Figure I.8: The contribution of each Ghost mechanism to performance (IPC).
I.6.5 Energy Efficiency

Figure I.7 presents the results of the energy usage evaluation. Both versions of Non-Spec affect the execution time of the benchmarks negatively, but not the number of cache misses and accesses to the main memory, thus affecting more the static energy usage of the system. We observe a mean energy increase of $2.5 \times$ for the Naive version, and 49% for Eager. The energy usage increases are not directly proportional to the execution time because i) the dynamic activity is not increased proportionally (the same number of instructions is still executed) and ii) the static power increase is reduced due to power gating, as modelled by McPAT. Given that a much smaller percentage of loads can be in-flight at the same time (Figure I.1), the resources of the system (e.g., load queue) can be scaled down to help reduce the energy usage, but we do not take this into consideration in the evaluation.

As one would expect, for Non-Spec, there is a direct and clear correlation between the benchmarks that perform badly in terms of performance and the benchmarks with the highest energy increase. On the contrary, the remaining solutions also negatively affect the memory access patterns during execution, which leads to large changes in the dynamic energy usage of the system as well. For InvisiSpec, we see an mean energy usage increase of 46%, which is very close to the energy usage in the Eager Non-Spec version, further supporting our view that the latter is a better solution.

Finally, Ghosts outperform both InvisiSpec and the Eager Non-Spec version significantly, with a mean energy increase of 9% over the baseline. A large part of this low overhead is due to the small execution time overhead, while also keeping the GhB sizes small.

I.6.6 Contribution of each Ghost Mechanism

The proposed Ghost loads solution consists of a combination of different mechanisms, namely the Ghost Buffer, Materialization, and Ghost prefetching. When discussing the Ghosts in the rest of the paper we assume that all of these mechanisms are used, in order to achieve the best possible performance. However, it is important to understand how much each of these mechanisms contributes to the final result, and if all of them are necessary.

Figure I.8 contains the performance results for different Ghost configurations. In addition to the baseline and the full Ghost load solution, it contains results for four additional Ghost versions, one with neither the GhB nor Mtz (ghosts-nothing), one without Mtz (ghosts-nomtz),
one without the GhB \( (\text{ghosts-noghb}) \), and finally one without Ghost prefetching enabled \( (\text{ghosts-nopref}) \).

With the Ghost version that uses neither the GhB nor Mtz \( (\text{ghosts-nothing}) \), we observe a mean performance loss of 61% under the baseline, which is worse than both InvisiSpec and the Eager Non-Spec version. The benchmark that is hurt the most by this version of the Ghosts is \text{bwaves}, a benchmark that is already sensitive to the other solutions, reaching a performance loss of 93%. Introducing the GhB \( (\text{ghosts-nomtz}) \) leads to a significant performance improvement, with a mean performance loss of 33%, outperforming both InvisiSpec and the Eager Non-Spec version. Since we have support for prefetching Ghost loads, this version benefits from it even without Mtz support, as the latter is not necessary for training and triggering the prefetcher. Similar results can be seen when Mtz is introduced (but without a GhB – \text{ghosts-noghb}), featuring a mean performance loss of 37%. Note that without a GhB, Ghost prefetching is not possible, which additionally hurts the performance of this version. We can easily conclude from these results that both mechanisms are necessary in order to achieve good performance.

Finally, we have evaluated the performance of the Ghost loads when Ghost prefetching is not available \( (\text{ghosts-nopref}) \). The prefetcher is instead trained and triggered by the Materializations sent once the speculation has been resolved, much like in InvisiSpec. Note that both the GhB and Mtz are used in these results, only the mechanism for prefetching based on Ghost loads has been disabled. With this version, we observe a performance loss of 22% under the baseline, 10% points more than Ghosts with prefetching \( (\text{ghosts}) \). This demonstrates the importance of considering prefetching when proposing such solutions, something that is overlooked by InvisiSpec (and SafeSpec).

### I.7 Related Work

This work was inspired by the Meltdown \cite{meltdown} and Spectre \cite{spectre} attacks published in the early 2018. However, as we explain in the introduction, our goal is not to solve just these attacks but to provide and evaluate a solution that prevents information leakage from cache memory accesses during speculative execution in general. For Meltdown and Spectre, CPU vendors have promised specific solutions in future microcode updates. Software solutions also exist, both for operating systems \cite{os-solutions} and for compilers \cite{compiler-solutions}. These solutions can incur very high costs, especially for applications that perform numerous system calls. Unfortunately, since these solutions are based on the existing attacks, they might
not work for the new attacks and variants that have been released since the initial Meltdown and Spectre attacks were discovered.

Non-speculative cache side-channel attacks have existed for some time [25, 24, 50, 23, 6, 75, 83]. These attacks focus on observing the difference in execution time caused by the cache behavior in order to leak information from the target application. A lot of these attacks focus specifically on attacking cryptographic functions that utilize S-boxes or S-box style encryption tables, such as AES. By detecting the access pattern of the cryptographic algorithm to the S-box, the secret encryption key can be identified. Since such keys are extremely sensitive data, numerous solutions have been proposed [43, 41, 16, 51, 29, 77, 30, 22, 42, 26, 76, 34, 19, 85, 18, 59], usually utilizing either partitioning, cache locking, or obfuscation through random noise introduced to the access patterns of the application. These solutions focus on preventing the side-channel attacks by either preventing or hiding the timing differences observed by the cache accesses. In our work, we focus instead on preventing or hiding the side-effects of speculative execution that, in combination with the traditional cache attacks mentioned above, could otherwise be used to leak sensitive information. Additionally, many of these methods focusing on AES and similar algorithms only protect against attacks that try to determine the access pattern to the S-box. This means that they only work for small amounts of explicitly specified data, which is different from our solution, which secures the whole address space.

When it comes to protecting against speculative attacks, in addition to InvisiSpec by Yan et al. [80], Khasawneh et al. [27] have also been working on a similar solution, named SafeSpec, but their approach differs from our in a number of different aspects. First of all, they discuss instructions caches, something that both we and Yan et al. have left as future work. However, their approach only considers branches as the source of speculation. Instead, both Ghost loads and InvisiSpec consider all instructions that might cause a misspeculation and lead to squashing in the pipeline. Additionally, prefetching is not discussed, except in the context of the prefetching effect that previously squashed loads have in the system. As we have shown in the evaluation disregarding prefetching can lead to a significant performance degradation. Furthermore, similarly to InvisiSpec, SafeSpec requires a buffer large enough to hold all in-flight loads. The exact buffer size not specified in their work, but it is implied that it is larger than the 8-entry L1 GhB the Ghost loads utilize in our evaluation. Unfortunately, we are not aware of the paper having been published to a peer-reviewed venue and as we cannot ascertain the implementation details of their solution, we cannot compare the designs
and the performance differences.

Finally, attacks and defences for the rest of the memory system also exist \[58, 79, 75, 86, 17, 17\]. These focus on different areas from our proposal and should be considered as complementary solutions.

### I.8 Future work

Coherence is an integral part of the caches in modern CPUs, so developing a solution and evaluating it in detail is important. Similarly, other parts of the cache hierarchy, such as the TLBs and the instruction caches, need an equally in-depth evaluation.

In parallel to further reducing the side-effects of speculative execution that are exposed to the system, we need to also investigate ways of further improving the performance and reducing the energy cost. In the current implementation, all Ghost loads that are successfully committed issue a Materialization packet to the cache. This is not efficient, as we have determined that a large number of times this results to an L1 hit, which leads to the cache simply discarding the Mtz packet. If we could know in advance (or predict) if a Materialization is necessary, we could avoid the unnecessary L1 lookups. Furthermore, not all speculative loads need to be executed as Ghosts. For example, not all data in a system are sensitive and need to be secured. If these data constitute a large enough part of the memory accessed during an applications execution, the performance and energy costs of our proposed solutions can be reduced. Finally, the number of speculative loads can be reduced if speculative instructions are disambiguated in advance, using either hardware or compiler techniques.

### I.9 Conclusion

We have evaluated in detail the performance and energy costs of different solutions for the problem of speculative execution leaking information through microarchitectural side-effects in the cache hierarchy. Namely, we have evaluated three different solutions, a non-speculative approach, where speculative loads are delayed until they can be safely issued, Ghost loads, where loads are issued but their side-effects are kept hidden, and InvisiSpec, the current state-of-the-art solution. We have shown that while the cost of the non-speculative solution is, expectedly, high, it is similar to that of InvisiSpec. At the same time, the non-speculative solution is simpler, as it requires no modifications to the cache hierarchy, has lower area and energy overhead, and protects from a wider
range of speculative side-channel attacks. We have also shown that it is possible to reduce the cost of hiding speculation even further using our Ghost loads, a solution similar to InvisiSpec but with key design differences that lead to significant performance improvements. Overall, we have not only provided more efficient solutions than the current state of the art, but we have also shown, through our detailed evaluation, that a more thorough understanding of the problem and the performance implications is necessary in order to formulate effective solutions.

Acknowledgments

This work was funded by Vetenskapsrådet project 2015-05159. The computations were performed on resources provided by SNIC through UPPMAX and by UNINETT Sigma2.
Efficient Invisible Speculative Execution
Through Selective Delay and Value Prediction

Christos Sakalis
Stefanos Kaxiras
Alberto Ros
Alexandra Jimborean
Magnus Själander

In Proceedings of the
46th International Symposium on Computer Architecture (ISCA)
New York, NY, USA, 2019
Abstract

Speculative execution, the base on which modern high performance general-purpose CPUs are built on, has recently been shown to enable a slew of security attacks. All these attacks are centered around a common set of behaviors: During speculative execution, the architectural state of the system is kept unmodified, until the speculation can be verified. In the event that a misspeculation occurs, then anything that can affect the architectural state is reverted (squashed) and re-executed correctly. However, the same is not true for the microarchitectural state. Normally invisible to the user, changes to the microarchitectural state can be observed through various side-channels, with timing differences caused by the memory hierarchy being one of the most common and easy to exploit. The speculative side-channels can then be exploited to perform attacks that can bypass software and hardware checks in order to leak information. These attacks, out of which the most infamous are perhaps Spectre and Meltdown, have led to a frantic search for solutions.

In this work, we present our own solution for reducing the microarchitectural state-changes caused by speculative execution in the memory hierarchy. It is based on the observation that if we only allow accesses that hit in the L1 data cache to proceed, then we can easily hide any microarchitectural changes until after the speculation has been verified. At the same time, we propose to prevent stalls by value predicting the loads that miss in the L1. Value prediction, though speculative, constitutes an invisible form of speculation, not seen outside the core. We evaluate our solution and show that we can prevent observable microarchitectural changes in the memory hierarchy while keeping the performance and energy costs at 11% and 7%, respectively. In comparison, the current state of the art solution, InvisiSpec, incurs a 46% performance loss and a 51% energy increase.
II.1 Introduction

Side-channel attacks that rely on shared microarchitectural state and behavior to leak information have been known to the architecture and the security communities for years. In particular, side-channel attacks on the cache system have been practically demonstrated in many forms for the level-one cache (L1) (when the attacker can share the same core as the target) [8], the shared last-level cache (LLC) (when the attacker can share the LLC) [83, 44], and the coherence protocol (when the attacker can simply be collocated in the same system, under a single coherence domain, with the target) [25]. However, starting with Spectre [31] and Meltdown [40], a new class of speculative side-channel attacks have sent a shockwave through the architecture community. Speculation, one of the fundamental techniques we have for achieving high performance, proved to be a significant security hole, leaving the door wide open for side-channel attacks to “see” protected data. As far as the instruction set architecture (ISA) and the target program are concerned, leaking the information across a covert side-channel is not illegal because it does not affect the functional, architectural behavior of the program. The stealthy nature of a speculative side-channel attack is based on microarchitectural state being changed by speculation even when the architectural state is not.

The need to hide speculation is unwaning even in the face of the latest and most sophisticated proposals to counteract side-channel attacks. Side-channel attacks based on the position of a cache line in the cache (e.g., Flush+Reload [83] or Prime+Probe [44] attacks) can be confronted with techniques such as randomization or encryption of addresses. Qureshi recently proposed an efficient and performant solution that can provide strong security guarantees for such attacks [59]. However, even such solutions can do nothing for timing attacks such as the invalidation side-channel attack [71]. No matter how addresses are randomized or encrypted (and cache lines shuffled around in the caches), in an invalidation-based coherence protocol, a writer still has to locate and invalidate all shared copies. By timing the difference between writes that invalidate and writes that do not, we can “see” speculative loads that accessed specific addresses. In addition, randomization and cache isolation based solutions do not protect against attacks originating from within the same execution context, such as some variants of Spectre [31]. This makes it a priority to search for an approach to make the effects of speculation invisible without unduly affecting performance or energy consumption.
In a breakthrough paper, Yan et al. were the first to propose an approach, referred to as InvisiSpec, to hide changes in microarchitectural state due to speculation [80]. InvisiSpec makes the accesses of a speculative load invisible in the cache hierarchy (by not changing cache and coherence state) and subsequently verifies correctness and makes changes in the memory hierarchy with a visible access when speculation is validated as correct. If the speculation fails to validate, the invisible access (although it occurred and consumed system resources, e.g., bandwidth) leaves no trace behind.

While Yan et al. provide an elegant solution to overlap the visible accesses of the loads [80], the simple fact that the accesses to the memory hierarchy are effectively doubled carries a non-trivial performance and energy cost [63]. The motivation of our work is to examine if a completely different approach to the same problem, one that still maintains the one-request-per-load execution model, can potentially yield better results.

We present a new solution that not only hides speculation, but recovers a significant portion of the performance and energy cost that double-access proposals such as InvisiSpec are bound to pay. To issue just a single access per load we must ensure that the load is non-speculative, otherwise we risk exposing speculative side-effects. Delaying all loads until they are non-speculative proves to be devastating for performance. However, we do not have to. Our solution is based on two simple observations:

1. For speculative loads that hit in the L1, we can allow them to proceed and use the accessed memory, provided that we do not affect the L1 replacement state or perform any prefetches at that time. This keeps speculative hits invisible.

2. For speculative loads that miss in the L1 we use value prediction instead of sending a request deeper in the memory hierarchy. Value prediction is completely invisible to the outside world, thereby enabling the load to proceed with a value while keeping its speculation invisible. When the load is in-the-clear and can no longer be squashed by an older instruction (Section II.2), we issue a normal request to the memory system that fetches the actual value. At that point, no matter if the value prediction was correct or not, the access is non-speculative and cannot be squashed. It is, therefore, safe to modify the memory hierarchy state.

We only focus on loads, because stores are kept hidden in the store buffer until they are committed. In our approach, there is only one
II.2 Speculative Shadows

For correctness purposes, the architectural side-effects of speculatively executed instructions remain hidden until the speculation can be verified. If a misspeculation is detected, the instructions following the misspeculation point—also referred to as transient instructions—are squashed and execution is restarted from the misspeculation point. In

single request per load that traverses the memory hierarchy and fetches data. While to validate value prediction we must serialize these non-speculative requests, we show that: i) the resulting performance cost is relatively low; and ii) value prediction helps reduce the cost of waiting for a load to become non-speculative. We compare against the performance and energy costs of simply delaying loads until they are no longer speculative and delaying L1 misses without performing value prediction. To summarize, we propose and evaluate the following:

- **Delaying Loads**: We delay speculative loads until they are non-speculative, preventing any speculative side-effects from happening. We present two different approaches, one where loads are only executed when they reach the head of the reorder buffer (naïve delay), and one where we only delay loads until we know they can no longer be squashed (eager delay). The latter is based on the Bell and Lipasti conditions for committing instructions [5], which will be discussed in Section II.2. We will also describe a structure for efficiently keeping track of these conditions in Section II.4.

- **Delaying Only L1 Misses**: We limit the eager delay solution only to loads that miss in the L1 cache, while allowing hits to execute. We refer to this solution as delay-on-miss.

- **Value Predicting Delayed Loads**: We augment the delay-on-miss solution with a value predictor (VP), enabling L1 misses to execute using a predicted value. The value predictor is local to the core and does not leak any information during speculative execution, making the loads completely invisible to others.

  We compare our proposed solutions with the current state-of-the-art approach, InvisiSpec [80], which works by executing all speculative loads and hiding their side-effects until the speculation has been verified. Simulation results reveal that our proposed, value prediction based solution can provide secure, memory-side-channel-free speculative execution with a performance cost of 11%, significantly outperforming InvisiSpec.
practice, on modern out-of-order (OoO) cores, all instructions are considered speculatively executed until they reach the head of the reorder buffer (ROB). This alleviates the need to keep track of which instructions are speculative, reducing the hardware cost and complexity. However, in order to achieve high performance while hiding the microarchitectural side-effects of speculatively executed instructions, a more fine-grain approach is required: During dispatch, instructions with potential to cause a misspeculation cast a speculative shadow to all the instructions that follow. For example, a branch with an unresolved condition or unknown target address casts a shadow to all instructions that follow it because if the branch is mispredicted, the instructions that follow it will have to be squashed. We call the instructions under such shadows (speculatively) shadowed instructions. As soon as the condition and the branch target are known, the speculation can be verified and the shadow is lifted. We identify the following types of shadows:

- **The E-Shadow**: E-Shadows are cast by memory operations with unknown addresses, arithmetic operations, and any other instructions that can cause an exception. The shadow starts when the instruction is dispatched and ends when the absence of an exception can be verified. For memory operations, one can check for exceptions once the address translation has completed and the permissions have been verified. For arithmetic and other instructions that throw exceptions on invalid inputs, checking can be done as
soon as the instruction has been executed. Under systems where precise exceptions are not available, this shadow can potentially be ignored, as an exception does not guarantee that the instructions that follow it will not be committed successfully.

- **The C-Shadow**: C-Shadows are cast by control instructions, such as branches and jumps, when either the branch condition or the target address are unknown or have been predicted but not yet verified. The shadow starts when the control instruction enters the ROB, and ends when the address and the condition have been verified.

- **The D-Shadow**: D-Shadows are cast by potential memory dependencies through stores with unresolved addresses (read-after-write dependencies). Loads can speculatively bypass stores and execute out of order, but they might be squashed if it is later discovered that a store points to the same address as that of a speculatively executed load. The shadow starts when the store enters the ROB and ends when the address is resolved, akin to the E-Shadow for memory operations.

- **The M-Shadow**: Under the total store order (TSO) memory model, or any other model that respects the load-load order, the observable memory order of loads needs to be conserved. If the execution of an older load is delayed then younger loads are allowed to execute speculatively but they might be squashed if an invalidation is received, hence causing the M-Shadow. The shadow starts when the older load enters the ROB, and ends when it has finished executing. While there is an overlap with the E-Shadows, the M-Shadow extends further, as the E-Shadow can be lifted after the permission checks. The M-Shadows do not exist under more relaxed memory models, such as release consistency (RC).

In addition to these shadows, interrupts can also halt and possibly divert execution. In this work, we assume that interrupts can be delayed until there are no shadowed loads preceding them. If the opposite is required, e.g. for a real-time system, then all loads would have to remain under a shadow until they reach the head of the ROB.

Figure II.1 presents a breakdown of the type of instructions casting shadows over the executed load instructions. The statistics have been gathered over applications from the SPEC CPU 2006 benchmark suite. The hardware parameters of the evaluated system can be found in Table II.1. We present the instruction types and not the shadow
types, as one instruction can simultaneously cast multiple overlapping shadows. Only the oldest shadow is taken into consideration for these statistics, therefore, eliminating one of the shadow types does not necessarily lead to a proportional decrease in the number of speculatively executed loads. For the E-Shadow, we assume that, other than memory accesses, only integer division operations can throw an exception. Exceptions for the rest of the integer operations are not that common on general-purpose CPUs, so we do not consider them. Additionally, floating point exceptions can usually be configured through special registers or other ISA specific mechanisms. This makes it possible to detect in advance if a floating point operation might throw an exception or not. For our evaluation, we assume that floating point exceptions are disabled. Finally, we assume that the permission bits for memory accesses are verified immediately after the translation has completed. The rest of the causes of exceptions are rare in benchmarks like SPEC CPU, so we can omit them from the simulation without any loss of precision in the evaluation.

Figure II.2 displays the ratio of loads executed while under a speculative shadow in each benchmark, based on the conditions discussed previously. The ratio of speculatively executed loads is high for all benchmarks, ranging from 67% (h264ref) to 97% (GemsFDTD), with a mean of 87%. This strongly indicates that only low-overhead solutions are viable, as the majority of the load operations are affected. A more detailed analysis of the performance implications is provided in the evaluation, Section II.5.
II.3 Delaying Speculative Loads

An intuitive solution for hiding the side-effects of speculative loads is to avoid speculation on loads altogether, i.e., delay the loads until they are no longer speculatively shadowed. We evaluate four different versions of this approach, starting from a very aggressive and then slowly relaxing the delay conditions while still keeping the speculation hidden:

- A naïve version where loads are only executed when they reach the head of the ROB.
- An eager version where loads are only executed when they are no longer shadowed by another instruction.
- A delay-on-miss version where only loads that miss in the L1 cache are delayed. Loads that hit in the L1 cache are satisfied by the cache, and only their side-effects in the memory system (e.g., updating replacement data or prefetching) are delayed.
- A value predicted version where, instead of simply delaying them, the delay-on-miss version is extended to provide values for L1 misses through value prediction.

II.3.1 Naïve Delay

By requiring loads to reach the head of the ROB before they are issued, we can ensure that all loads are executed non-speculatively without the need for additional hardware complexity. On the other hand, this causes all loads to be serialized, thus, making it impossible to take advantage of any memory-level parallelism (MLP) available in the application. Instruction-level parallelism is equally crippled, as loads are necessary in order to feed data to the instructions. In fact, we are only presenting this solution to show that naively executing all loads non-speculatively is not viable in OoO CPUs, and to provide a worst-case baseline for the evaluation.

II.3.2 Eager Delay

Instead of delaying every load until it reaches the head of the ROB, we can take advantage of the Bell and Lipasti conditions [5] and our understanding of the different shadow types (Section II.2) to release them early. Loads are delayed only until they are no longer shadowed by other instructions, at which point they can be safely issued and executed out-of-order. While this approach still comes with a steep performance cost,
it improves significantly over the naïve delay solution (Section II.5). Additionally, it performs comparably to InvisiSpec, without the hardware complexity cost of modifying the memory hierarchy or the coherence protocol.

II.3.3 Delay-on-Miss

Overall, we propose delaying speculative loads to prevent any side-effects from appearing in the memory hierarchy. In comparison, other solutions, such as InvisiSpec, execute the loads and instead try to hide the side-effects until the speculation has been verified. While such an approach works, it requires modifications to the whole cache hierarchy and it incurs a significant performance cost. We observe, however, that we can reduce this cost by reducing the scope of the side-effects that need to be hidden. By delaying L1 misses (but not hits), we eliminate the need for an additional data structure that hides fetched cache lines until they can be installed in the cache. Additionally, since the cache line already exists in the L1 on a hit, no additional coherence mechanisms are required. Instead, we only need to delay the side-effects of the hits, such as updating the replacement data and notifying the prefetcher, which exist outside the critical path of the cache access. The L1 behavior for shadowed loads is as follows:

- In case of an L1 hit, the cache responds to the request but delays any operations that might cause visible side-effects, such as updating the replacement state. The CPU will signal the cache to perform these operations after the speculation has been successfully verified.

- In the case of an L1 miss, the request will simply be dropped. We refer to these L1 misses caused by shadowed loads as **shadowed L1 misses**.

If a load has received data from the L1 while under a speculative shadow, and after it has left that shadow, it will send a release request to the cache, signaling that the cache can now perform any side-effect causing operations it might have delayed. On the other hand, if a load has not received any data after executing under a speculative shadow, it will simply repeat the initial memory request, this time triggering the normal miss mechanisms in the cache.

While in this work we focus on the data caches, the translation lookaside buffers (TLB) can also be utilized as a potential side-channel
II.3. Delaying Speculative Loads

Figure II.3: VP prediction rate for shadowed L1 misses (bars), combined with the L1 miss ratio of each benchmark (dots).

in an attack. In the SPEC CPU benchmarks that we use for the evaluation, TLB misses are rare, with the majority of the benchmarks having a miss ratio of less than 1%. Therefore, for the remainder of this paper, we will assume that TLB misses are handled with the delay-on-miss mechanism that has been described in this section.

II.3.4 Value Predicting Delayed Loads

The L1 miss ratio of the applications used to evaluate the solution proposed in this paper ranges from less than 1% to 25% (Figure II.3). While the delay-on-miss proposal reduces the number of delayed loads significantly, it still incurs stalls when encountering a shadowed L1 miss. To solve this problem, we propose to continue the execution by predicting the value of L1 misses instead of delaying them.

Value prediction is not a new idea, and it has been used, among other things, to help improve the number of instructions-per-cycle (IPC), to handle L2 misses, to design a new architecture that can bypass the expensive OoO pipeline, and even to secure processors from fault attacks. We are only interested in using value prediction to predict loads, and specifically L1 misses. In this context, value predictors have two interesting properties:

1. The predictor can be local to the core, isolated from other cores or even other execution contexts. The visible state of the predictor is only updated after the prediction is validated.
2. Because the predicted value needs to be validated with a normal memory access, the predictor can be incorporated into an OoO pipeline with only small modifications to the L1 cache and no modifications to the remaining memory hierarchy and the coherence protocol [46].

The first property implies that the value predictor itself can be used during speculative execution without any visible side-effects. By delaying the memory access used to perform the validation of the prediction, we can value predict loads that miss in the L1 while remaining completely invisible to the rest of the system. In addition, we do not need to introduce any modifications to the cache coherence protocol, as the validation of the predicted values happens only after the speculation shadows have been resolved and it can be issued as a normal load. Figure II.4 presents an overview of how this can be achieved. The value predictor is added as part of the L1 cache and is queried in parallel. In case of an L1 miss on a shadowed load, the value is returned by the value predictor instead, if possible. Regardless if the access was resolved from the cache or the value predictor, the cache controller delays any side-effects until the CPU indicates that the speculation has been verified. Since the value predictor is accessed in parallel with the L1, and multiplexing is already necessary for associative caches, we do not introduce any additional delay in the critical path of the cache.
II.3. Delaying Speculative Loads

To the best of our knowledge, this is the first solution that takes advantage of the aforementioned properties of value predictors to hide the side-effects of speculative execution in the memory hierarchy and the cache coherence protocol with only small modifications to the former and no modifications to the latter.

With value prediction in place, a new type of speculative shadow is introduced, the VP-Shadow. All value-predicted loads cast this shadow until the predicted value can be validated. Currently, because our implementation assumes a system implementing the TSO memory model, the VP-Shadows are completely covered by the M-Shadows. If the M-Shadows were to be eliminated, either by relaxing the memory model or through some other approach \[61\], then the VP-Shadows could be tracked to a finer granularity and selective replay can be used instead of fully squashing in case of a value misprediction \[28, 72\]. The VP-Shadows could then be restricted only to the instructions that would have to be selectively squashed and replayed, instead of all younger instructions. With our current proposal, value predicted loads cast a shadow on all instructions until they are validated, and since a validation can only be issued when the load is no longer under a speculative shadow, only one validation at a time can be issued. This restriction can be lifted if the M-Shadow is eliminated and the VP-Shadows are tracked at a finer granularity but evaluating the potential of such a solution is left for future work.

Figure II.3 contains the prediction rate for a 13-component VTAGE predictor \[55\], with 128 entries per component. While we use all loads for training the predictor, it is only queried for shadowed loads that miss in the L1 cache. Other types of value predictors also exist \[11, 38, 56, 49\], but a thorough evaluation of value prediction mechanisms is beyond the scope of this paper. With the VTAGE predictor we have implemented, the prediction rate varies significantly between applications, ranging from less than 1% to more than 98%, with a mean value of 16%. Our data indicate that \(i\) the prediction rate for the shadowed L1 misses is lower than the potential prediction rate of all loads (40%) and that \(ii\) delaying the validation of the predictions negatively affects the prediction rate. However, we will see in the evaluation (Section II.5) that even with a prediction rate of 16%, significant gains can be achieved. Improving value prediction algorithms is beyond the scope of this work; instead, in order to explore the benefits provided by improving the predictor, we also present results with an oracle predictor that can achieve prediction rates of 50% and 100%. 
II.4 Tracking Speculative Shadows

To efficiently delay loads or know when a value predicted load can be validated it is necessary to know when the load is no longer covered by a speculative shadow. We make the observation that to answer this question it is enough to know if the oldest shadow-casting instruction is older than the load in question. We leverage this to track shadows in a structure similar to a reorder buffer (ROB) but that is much smaller as it only needs to track a single bit per instruction. We call this structure the shadow buffer or SB for short. Shadow-casting instructions (Section II.2) are entered into the shadow buffer in the order they are dispatched. Once an instruction no longer causes a shadow, e.g., once a branch has been resolved, then the SB entry is updated. Only when an instruction reaches the head of the SB and no longer casts a shadow does it get removed from the SB. This assures that the oldest shadow-casting instruction is always at the head of the SB and that they exit the SB in program order, similar to how the ROB assures that instructions are committed in order. To determine when a load is no longer covered by a shadow it is enough to i) know which was the youngest shadow-

\[\text{(1) (2) (3) (4) (5) (6) (7) (8)}\]

Figure II.5: Example of tracking speculative shadows. The values “T” and “F” indicate “true” and “false”, respectively.

\[\text{(Not to be confused with the Speculation Buffer utilized in InvisiSpec.}\]
casting instruction when the load entered the ROB and \( ii \) check for when this shadow-casting instruction has exited the SB. Assume the following code:

\[
\begin{align*}
\text{LD0: } & \text{ ld } r2 \ [r1] \\
\text{BR1: } & \text{ br } r2 \ #0024 \\
\text{LD2: } & \text{ ld } r6 \ [r3] \\
\text{LD3: } & \text{ ld } r3 \ [r3] \\
\text{BR4: } & \text{ br } r4 \ #0096 \\
\text{LD5: } & \text{ ld } r1 \ [r5]
\end{align*}
\]

Figure II.5 shows an example of how the shadows are tracked when executing the code above. The shadow buffer (SB) is implemented as a circular buffer; to avoid using index 0 and 1 for all the structures in the example it is assumed that previous executions have left the SB empty with its head and tail set at 3. The ROB, release queue, and load queue are also assumed to be implemented as circular buffers such that an entry of these structures can always be identified by its index\(^2\). The SB has as many entries as the ROB and the release queue has as many entries as the load queue to avoid any structural hazards. It might be possible to reduce the area overhead by optimizing these based on the likelihood of how many instructions cause shadows and how many shadowed loads exist. While the example shows a number of distinct steps to clearly illustrate the functionality, in reality many of these can be performed in parallel, similar to how multiple instructions can be committed from the ROB in a single cycle. To further simplify the example, we assume that only branches cast shadows, i.e., we are not considering the E- and M-Shadows of the loads. The example develops as a sequence of nine (one not shown) steps:

1. When the first load (LD0) enters the ROB the SB is empty (SB-Head==SB-Tail), indicating that there are no shadow-casting instructions and that the load can be normally executed \( @ \).

2. The branch (BR1) causes a shadow until its condition and target have been resolved and is, therefore, inserted into the SB \( @ \). The ROB entry of the branch is marked with the index of the SB entry, i.e., the SB-Tail \( @ \). The SB-Tail is then incremented.

\(^2\)The head and tail pointers of the reorder buffer, release queue, and load queue are not shown in Figure II.5 to simplify the illustration.
3. When the second load (LD₂) enters the ROB the SB is no longer empty and the load therefore shadowed. It is still entered into the load queue but it is marked as speculative (\(\text{d} \)), which means that it also needs to be entered into the release queue. There, its index in the load queue (\(\text{e} \)) and the youngest shadow-casting instruction (identified by SB-Tail minus one) (\(\text{f} \)) are marked.

4. The steps just described are repeated for the following load (\(\text{g} \)), branch (\(\text{h} \)), and final load (\(\text{i} \)).

5. Once a shadow-casting instruction, in this example the second branch instruction (BR₄), stops casting a shadow, it updates the SB index that is indicated by its ROB entry (\(\text{j} \)). Nothing happens at this point of time, since the SB-Head is still pointing to an older shadow-casting instruction that has not been executed.

6. Once the first branch (BR₁) is resolved the SB entry is updated (\(\text{k} \)).

7. This triggers a number of events since it is the oldest shadow-casting instruction, as indicated by the SB-Head (\(\text{l} \)). Before the SB-Head is incremented to indicate that this is no longer the oldest shadow-casting instruction, it is compared with the first entries of the release queue (\(\text{m} \)). When these match, the load queue entry are updated to indicate that the loads are no longer under a speculative shadow (\(\text{n} \)). This repeats for every entry in the release queue until a shadow stamp that does not match the SB-Head is encountered.

8. Once the SB-Head is incremented the entry in the SB is checked, causing the events in step (7) to be repeated and the final load to be marked as non-speculative (\(\text{o} \)).

9. Finally, the SB-Head is incremented once more, leaving the head equal to the tail, indicating that the SB is empty and that there are no shadow casting instructions in the ROB (not shown in the illustration).

One big advantage of tracking shadows this way is that it avoids the use of content-addressable memories (CAMs), which are both complex and costly to implement.
II.5 Evaluation

We start by explaining our evaluation methodology and then proceed to discuss the memory behavior, performance, and energy usage characteristics of the evaluated solutions. We end by discussing the implications of improving the value-predictor’s prediction rate.

II.5.1 Methodology

We have implemented our proposal and also the current state-of-the-art solution, InvisiSpec, on Gem5 [7], combined with McPAT and CACTI [36, 37] for the energy evaluation. We use the SPEC CPU 2006 benchmark suite [68], with six of the applications excluded due to baseline simulation issues. We model the speculative buffers from InvisiSpec as small caches on McPAT, and the value predictor as a large branch target buffer (BTB). Both are sized appropriately, taking into consideration the amount of storage required both for data and for metadata. The VTAGE predictor is sized pessimistically, assuming that a proper program counter and branch history are stored for each entry. In practice, the storage overhead for metadata can be reduced without compromising the prediction rate [66]. For DRAM, we use the power model built into Gem5, as McPAT does not provide one. We perform the simulations by first skipping one billion instructions in atomic mode and then simulating in detail for another three billion instructions. The characteristics of the simulated system can be found in Table II.1. We simulate a system with a private L1 and a shared L2 cache, without L3. The reason why

Table II.1: The simulation parameters used for the evaluation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology node</td>
<td>22nm</td>
</tr>
<tr>
<td>Processor type</td>
<td>out-of-order x86 CPU</td>
</tr>
<tr>
<td>Processor frequency</td>
<td>3.4GHz</td>
</tr>
<tr>
<td>Address size</td>
<td>64bits</td>
</tr>
<tr>
<td>Issue width</td>
<td>8</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 private cache size</td>
<td>32KiB, 8-way</td>
</tr>
<tr>
<td>L1 access latency</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 shared cache size</td>
<td>1MiB, 16-way</td>
</tr>
<tr>
<td>L2 access latency</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Value predictor</td>
<td>VTAGE</td>
</tr>
<tr>
<td>Value predictor size</td>
<td>13 components × 128 entries</td>
</tr>
</tbody>
</table>
we chose this unusual configuration is to increase the cost of misses in the L2 cache, since the SPEC CPU 2006 workloads are otherwise not memory intensive enough to properly evaluate the cost of the proposed solutions. As the baseline we use a large, unmodified OoO CPU.

II.5.2 Memory Behavior

Since the memory behavior plays a significant role in the performance and energy usage of the application, we begin the evaluation with analyzing the memory behavior of the benchmarks.

Figure II.6 presents the L1 data cache miss ratio. Overall, the miss ratio is small, with a baseline mean value of 4%. This is a strong argument for our delay-on-miss solution: While almost all instructions are shadowed when they are issued (Figure II.2), only a small percentage of them will have to be delayed or value predicted. On average, the miss ratio of the applications is not affected negatively by any of the evaluated solutions, and it is even improved by the more aggressive delay versions. This is due to the reduced amount of misspeculated memory accesses, which would bring in unneeded data, and also due to the slower execution of the program allowing time for the memory system to respond. InvisiSpec on the other hand incurs a large increase in the miss ratio in some applications, especially in the case of libquantum. The increase is caused by the fact that libquantum is a streaming application that benefits greatly from prefetching, which is disrupted by InvisiSpec. However, these results can be misleading, as when it comes to the actual number of misses and overall memory accesses happening in the application, the absolute number might change while the ratio remains the same. Instead, Figure II.7 features the number of DRAM reads for every application. We can now observe that InvisiSpec incurs a 32% increase in the number of DRAM reads. The effect is particularly prominent in mcf, zeusmp, and as one might expect, libquantum. In contrast, the rest of the evaluated solutions do not have that problem, only marginally exceeding the baseline in one application, GemsFDTD. There is only one case where InvisiSpec behaves better than the delay-on-miss solutions, h264ref, where all approaches reduce the number of DRAM reads below the baseline.

In addition to the potential performance and energy cost, DRAM accesses can also be utilized as a side-channel for attacks [58]. Such attacks are outside the scope of InvisiSpec, but they are covered by our delay-based solutions.
### Table II.5 Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Baseline</th>
<th>Invisispec</th>
<th>Delay-naive</th>
<th>Delay-eager</th>
<th>Delay-on-miss</th>
<th>Delay-on-miss+vp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gcc</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gamess</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>mcf</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>milc</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>zeusmp</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gromacs</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>cactusADM</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>leslie3d</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>namd</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gobmk</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>soplex</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>hmmer</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>GemsFDTD</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>libquantum</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>h264ref</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>tonto</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>lbm</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>omnetpp</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>astar</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>wrf</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>sphinx</td>
<td>1.0</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Figure II.6: L1D cache miss ratio.**

**Figure II.7: Number of DRAM reads, normalized to the baseline.**
II.5.3 Performance and Energy

Figures II.8 and II.9 present the instructions per cycle (IPC) and the energy usage of the benchmarks, respectively. Both are normalized to the baseline. In Figure II.9, the bottom shaded part represents the leakage energy of the system, while the top lighter part represents the dynamic energy. The first obvious observation we can make is that the naïve delay version is not a viable solution for high performance CPUs. With a mean performance loss of 74% and an energy increase of $2^{3.4}$, it consistently performs worse than any of the other solutions we have evaluated.

For the eager delay case, we can see a significant improvement in performance, with a mean loss of 50%, as well as in the energy usage, with an mean increase of 49% over the baseline. While still far from the baseline, the eager delay version already performs similarly to InvisiSpec, which exhibits a mean performance loss and energy increase of 46% and 51%, respectively. Both solutions have benchmarks in which one outperforms the other, but the eager delay solution is simpler and more secure than InvisiSpec.

Introducing the delay-on-miss optimization, we see a performance improvement of 31 percentage points over the eager delay, reducing the mean performance loss to 19%. Energy usage is also improved, to a 13% cost over the baseline. At this point, delay-on-miss, even without value prediction, is already consistently better than both eager delay and InvisiSpec, with the exception of mcf and GemsFDTD.

Finally, delay-on-miss combined with value prediction performs the best out of all the evaluated solutions. With a mean performance loss of only 11% (8 percentage points better than delay-on-miss), and a mean energy usage increase of 7%, it outperforms all the other evaluated solutions. Part of this energy cost is due to the value predictor and the need to validate the predicted results, as well as the need to update the cache metadata after the speculation has been verified.

II.5.4 Improving the VP Prediction Rate

We have implemented an Oracle predictor with a variable prediction rate to evaluate the effects that an improved value prediction rate would have on the delay-on-miss solution. We evaluate two different rates, 50% of all shadowed L1 misses, and 100%. Note that the Oracle predictor decides randomly every time it is queried, and therefore, the prediction rate is not tied to the PC or any other characteristics of the instruction being predicted. As seen in Figure II.10 having perfect value prediction would lead to significant performance improvements, with the majority
II.5. Evaluation

Figure II.8: IPC, normalized to the baseline.

Figure II.9: Normalized energy usage. The bottom (shaded) part represents the static (leakage) energy of the system.
Figure II.10: Normalized IPC with two Oracle value predictors, with prediction rates of 50% and 100%.

of the benchmarks approaching or reaching the baseline. There is one exception, \texttt{GemsFDTD}, in which the VTAGE predictor already provides almost perfect value prediction, so the Oracle predictor can not provide any significant improvements. The missing performance in all cases can be explained by the overhead introduced due to only allowing one validation at a time.

Overall, we observe performance improvements of 2 and 9 percentage points over the VTAGE predictor, for prediction rates of 50% and 100% respectively. Given that the VTAGE predictor has a mean prediction rate of 16\%, an increase of more than 2\times in prediction rate yielding only 2 points improvement brings into question whether improving the value predictor is worth the effort and potential hardware cost. However, the 9 points increase provided by the perfect predictor is still encouraging, although achieving perfect value prediction is of course impossible.

II.6 Security Evaluation

Listings II.1 and II.2 contain pseudocode for a Spectre attack example. Listing II.1 contains the victim code that is going to be exploited by the attack. It consists of two parts, \textit{i}) the program data, which contains some user accessible storage and a secret key, and \textit{ii}) a function (\texttt{access\_data}) that provides access to the storage, making sure that no out-of-bounds accesses are allowed. While this is a simplified example,

Listing II.1: Victim code for the Spectre attack example.

```
struct program_data {
    char storage[10];
    char secret_key; /* = 7 */
}

char access_data(int index) {
    if (index <= 9)
        return program_data.storage[index];
    else
        return -1;
}
```

similar patterns can be found, for instance, in web browsers where the JavaScript runtime needs to prevent the code for accessing other parts of the browser’s memory. Listing II.2 shows the attack code and how it can be used to bypass the victim’s checks in five simple steps:

1. Line 16—The attack starts by training the branch predictor to always expect the if statement in the access_data function (Listing II.1 Line 7) to be true. This is done by simple calling the access_data function multiple times in a row with an index value that is within the storage bounds.

2. Line 17—To prepare for the Flush+Reload side-channel [83], the probe array is flushed from the cache.

3. Line 18—Now that the system has been set up for the attack, the Spectre code calls the access_data function with an out-of-bounds index of “10”. This index points to the element directly after the end of the array, which is where the secret key resides in the memory. From a software point of view, we expect the access_data function to detect the out-of-bounds access and return a value of “−1” but, due to speculative execution, this is not exactly what happens. Because we have trained the branch predictor to always expect the if statement to be true, i.e., to always predict the index to be inside the storage bounds, the OoO CPU will speculatively perform the memory access to storage[10] (Listing II.1 Line 8), and return the loaded value (i.e., the secret key).
void train() {
    for (i = 0; i < 1000; i++)
        access_data(0);
}

void probe(char probe_array[]) {
    for (i = 0; i < 16; i++) {
        t1 = RDTSCP();
        x = probe_array[i * CACHE_LINE_SIZE];
        delay = RDTSCP() - t1;
    }
}

void attack() {
    char probe_array[16 * CACHE_LINE_SIZE];
    train();
    flush(probe_array);
    secret = access_data(10);
    x = probe_array[secret * CACHE_LINE_SIZE];
    probe(probe_array);
}

Listing II.2: Simplified Spectre attack example.

4. Line 19—The attack code will then use the value returned by the `access_data` function to index into the probe array\(^3\), bringing one of its cache lines into the cache.

5. Line 20—Eventually, the CPU will realize that the branch was mispredicted and the incorrect execution will be squashed, eventually leading to the attack code receiving the correct value of “−1”, but any data brought into the cache in the meantime will remain there. The attack code can then scan through the probe array, measuring the delay (time) of accessing each cache line. As one cache line was brought in during the speculative part of the attack, we can deduce, based on its position in the probe array, what the value at `storage[10]` was, thus deducing the secret key (Figure II.11).

Note that both the victim and the attacker are part of the same execution context, which is why cache partitioning or randomization solutions (Section II.7) are ineffective against Spectre. In contrast, when delay-on-miss is enabled, the speculative access to the probe array will

---
\(^3\)For simplicity we use a probe array that can encode 16 different values; in practice, if one byte is read during the attack, 256 values would be required.

Figure II.11: Probe access times (from rdtscp) for the Spectre example in [Listing II.2]. The secret value of seven can be easily deduced in the baseline.

be prevented from loading any data into the cache and probing the array later will not lead to any measurable differences in execution time (Figure II.11).

This variant of Spectre presented here takes advantage of the branch predictor and speculative loads to bypass software-enforced bound checks and gain access to secret information. However, this is not the only variant that has been discovered. For example, the Spectre v1.1 and v1.2 variants [14, 15] take advantage of speculative stores to redirect the execution stream and to bypass sandboxing. Our solution does not prevent the redirection of the execution stream, after all how the victim is coerced into executing the attack code is beyond the scope of this work, but it does prevent the attack code from leaking any sensitive information. Regardless of how the attack code is executed, it still needs a side-channel to leak any sensitive information it accesses, and our solution blocks such side-channels in the cache and memory system. What our solution does not protect against is side-channels outside of the memory hierarchy. For example, the NetSpectre attack [65] uses the slowdown caused by the AVX engine on some modern Intel CPUs. Since we do not delay computational instructions, only loads, that side-channel is left exposed. Protection against side-channels outside of the cache and memory system is left for future work. Similarly, attacks that require
physical access to the machine, such as power analysis attacks, are also outside the scope of this work.

II.7 Related Work

Cache side-channel attacks \cite{25,24,50,23,6,75,83} have been known for years before their speculative variants emerged. Instead of trying to leak arbitrary memory regions of arbitrary applications, they focus on identifying keys by detecting memory access patterns of cryptographic applications, or on covertly transferring information across software and hardware barriers, such as across virtualized containers. Both hardware and software solutions have been proposed \cite{43,41,16,51,29,77,30,22,42,26,76,34,19,85,18}, using either cache partitioning, cache locking, or access pattern obfuscation through randomization. These solutions do not work with speculative attacks such as Spectre \cite{31}, where the attack can be performed from within the same execution context (Section II.6). Additionally, most of these solutions focus on only protecting small amounts of sensitive data, such as cryptographic keys, while the speculative side-channel attacks we are focusing on can attack the whole memory address range.

When it comes to the currently known speculative side-channel attacks, hardware and software \cite{13,73,52} vendors have already promised or delivered solutions. These solutions are specific to the issue they are trying to fix (e.g., not performing loads before the permissions have been checked, for Meltdown), and do not protect against all existing and potential future speculative side-channel attacks. Our solution instead provides a holistic approach that eliminates the threat of speculative attacks that try to take advantage of the memory hierarchy as a side-channel. It builds on insights provided by our work on Ghost loads \cite{63}, which evaluates, in detail, the implications of trying to hide speculative loads in the cache hierarchy.

There currently exist, to the best of our knowledge, two proposed solutions for hiding the side-effects of all speculative memory accesses in the cache hierarchy. The first, InvisiSpec by Yan et al. \cite{80} has already been discussed in more details in Sections II.1 and II.5, but we will briefly summarize the differences here as well. First, InvisiSpec does not delay any memory accesses, instead it performs them and does not cache the data. The data are kept in buffers (“Speculative Buffers”) at the L1 and the LLC, and are released after the speculation has been resolved. To avoid attacks on these buffers, they are not kept coherent; instead a second access (“Validation”) is used to validate the loaded value in
II.8 Future work

While we achieve significant performance improvement over the state-of-the-art solution, we can still not recover all the performance lost from the baseline. One of the issues we have encountered is that the delay introduced into the validation of the value predictions negatively affects the prediction rate of the predictor. We would like to investigate ways of reducing this effect, either by changing the design of the predictor or by introducing a new mechanism for earlier validation of the predictions. Additionally, even with the perfect predictor, there is still some perform-
ance missing over the baseline. This performance loss is due to the loss of MLP suffered during the VP validations, as VP-shadowed loads cannot validate until their shadow has been lifted. We would like to further investigate methods for eagerly unshadowing loads, using either more aggressive hardware techniques or compiler information. For example, previous work has investigated a technique for decoupling the calculation of the memory address from the execution of the load [69]. Using such a technique, we can limit the duration of the E- (and maybe D-) Shadows, as the address can be calculated potentially before the load is even dispatched. Furthermore, predicated execution can, in some cases, replace branches, thus eliminating the C-Shadows. Finally, under TSO, Ros et al. [61] have proposed a non-speculative solution for load-load reordering, referred to as WritersBlock, which can eliminate the M-Shadows. By utilizing WritersBlock and by tracking the VP-Shadows at a fine granularity, we can allow for multiple validations to be issued in parallel, restoring the MLP and the performance.

In addition to improving the performance of our current proposal, there is also more work to be done in further reducing the exposure of the side-effects of speculation in the rest of the system. For example, we only focus on data accesses, but the instruction cache needs to be secured as well. Unlike data caches, accesses to the instruction cache cannot be freely delayed, as the pipeline will stall.

Finally, in this work we assume that all data are equally important and need to be kept secret. In practice, there might be data that do not need to be protected from speculative side-channel attacks, significantly limiting the number of speculative accesses that need to be delayed or predicted. Vijaykumar et al. [74] have already proposed a solution for tagging memory regions that can be used exactly for this purpose, assuming that the sensitive data regions have been identified by the programmer.

II.9 Conclusion

In this work, we propose an efficient solution for preventing visible side-effects in the memory hierarchy during speculative execution. It is based on the observation that hiding the side-effects of executed instructions in the whole memory hierarchy is both expensive, in terms of performance and energy, and complicated. At the same time, delaying all speculative loads is also not a viable solution, since most loads are executed speculatively. Instead, it is easier and cheaper to hide the side-effects of loads that hit in the L1 data cache, and prevent all other loads from executing.
To limit the performance deterioration caused by these delayed loads, we augment the L1 data cache with a value predictor that covers the loads that miss in the L1. We observe that one of the limitations that prevents us from reaching the baseline performance, or even exceeding it thanks to the value predictor, is that value predicted loads can only be validated one at a time. However, even with this limitation, we provide secure execution, free of visible side-effects, with a cost of only 11% in performance and 7% in energy.

Acknowledgments

This work was funded by Vetenskapsrådet project 2015-05159, by the SSF Strategic Mobility 2017 grant SM17-0064, the Spanish MCIU and AEI, as well as European Commission FEDER grant RTI2018-098156-B-C53. The computations were performed on resources provided by SNIC through UPPMAX and by UNINETT Sigma2.
Understanding Selective Delay as a Method for Efficient Secure Speculative Execution

Christos Sakalis
Stefanos Kaxiras
Alberto Ros
Alexandra Jimborean
Magnus Själander

Under submission
Abstract

Since the introduction of Meltdown and Spectre, the academic and industry research communities have been tirelessly working on speculative side-channel attacks and on how to shield computer systems from them. To ensure that a system is protected not only from all the currently known attacks but also from future, yet to be discovered, attacks, the solutions developed need to be general in nature, covering a wide array of system components, while at the same time keeping the performance, energy, area, and implementation complexity costs at a minimum. One such solution is our own delay-on-miss, which efficiently protects the memory hierarchy by i) selectively delaying speculative load instructions and ii) utilizing value prediction as an invisible form of speculation. In this work we dive deeper into delay-on-miss, offering insights into why and how it affects the performance of the system. We also reevaluate value prediction as an invisible form of speculation. Specifically, we focus on the implications that delaying memory loads has in the memory level parallelism of the system and how this affects the value predictor and the overall performance of the system. We present new, updated results but more importantly, we also offer deeper insight into why delay-on-miss works so well and what this means for the future of secure speculative execution.
III.1 Introduction

Since the introduction of Meltdown [40] and Spectre [31], speculative execution, the cornerstone of modern, high-performance CPUs, has been under attack. By abusing the fact that speculative execution, by nature, can execute instructions that would have otherwise not have been executed by the application, a malicious attacker is able to bypass software and hardware barriers and leak sensitive information. These attacks exploit different parts of the system to access and leak the information, but they all consist of two main parts: The illegal access to the information, done speculatively, and the information leakage. The first part depends on the aforementioned property of speculative execution that allows instructions to execute and access data that they would normally not be allowed to. However, to leak that data, the second part is necessary, which takes advantage of micro-architectural state-changes that are done under speculative execution and that can be observed by the attacker during or after the speculation has been resolved. While there are some practical limitations, these two parts are interchangeable: The way the attack gains access to the sensitive information and the way this same information is leaked are independent of one another, assuming there is a way of transferring the information from the first to the second. Specifically, for the second part, a number of different side-channels are available, capable of leaking information across software and hardware barriers. These might include side-channels such as memory-timing side-channels [58], port-contention side-channels [3], or even a frequency-scaling side-channel [65]. Due to being easy to exploit and also offering great versatility between where the victim and the attacker are placed relative to each other, memory-timing side-channels (including cache-timing side-channels) are particularly popular [45, 21].

Researchers, both in academia and in the industry, have been frantically working on providing solutions for these issues. To solve specific attacks, solutions might focus on preventing the illegal access to the sensitive information in the first place [52, 60, 1]. This approach has the advantage that it can be implemented quickly, sometimes requiring only software or operating system changes, but it does not protect against any future attacks or even variants of the same attack. Instead, some researchers (including ourselves) have focused on preventing the leakage of information through side-channels, and specifically speculative side-channels.

In our previous work [64], we proposed an efficient way for invisible speculative execution by introducing delay-on-miss, a way of selectively delaying load instructions from executing under unsafe speculative
tion, preventing any speculative side-effects from being made visible in the memory hierarchy. Specifically, delay-on-miss builds on the insight that if a load hits in its own private L1 cache, it only causes minimal side-effects that can easily be delayed, as they are not part of the critical path of the memory access. To determine when an instruction is safe or unsafe, we also introduced the concept of speculative shadows \[63][64], a way of determining the earliest point at which an instruction is no longer speculative, reducing the cost of naively delaying all speculative instructions. Finally, we combined the delay-on-miss approach with value prediction, as a way of further mitigating the cost of delaying speculative loads.

In this work, we give an in-depth analysis into the results of our previous work (delay-on-miss and value prediction). Specifically,

- We provide a detailed justification for the performance of the delay-on-miss approach, especially when compared to a delay-all approach.

- We examine the effect that delay-on-miss has on the memory level parallelism (MLP) of the application and how this affects the runtime performance.

- We revisit our results for the value prediction, in light of a new, more accurate implementation that uncovered issues in the original implementation; issues that inadvertently led to an overestimation of the benefits of value prediction.

- We explore the properties of value prediction to further understand what its limiting factors are and how it can be improved. We identify that the accesses required for validating the predictions are the main culprit, as not only they have to be restricted under the same conditions as all the other loads in delay-on-miss, but they also introduce a new type of unsafe speculation, which imposes further restrictions.

One of the main insights of this work is that the delay-on-miss approach performs well with respect to the baseline (when compared with other secure solutions) because, contrary to intuition, it does not completely restrict all the memory level parallelism available in the application. On the other hand, value prediction offers additional instruction level parallelism (ILP), but does not contribute to memory level parallelism [Section III.3]. Due to these effects, the delay-on-miss approach reduces the overhead of delaying speculative loads by two thirds, from
−53% under the baseline for the delay-all version to −18%. Value prediction reduces the delay-on-miss overhead by 1 percentage point, and we show that even with an oracular value predictor that can correctly predict any load value, the mean overhead is still reduced by only 3 percentage points. On the other hand, if we were able to safely allow the prediction validations to be issued without any restrictions, the value prediction is capable of regaining back more than half of the performance lost with delay-on-miss, with a mean overhead of only −8% compared to the baseline.

III.2 Threat Model

We consider speculative side-channel attacks that utilize the memory hierarchy as the side-channel. Attacks that use different side-channels, such as port contention, energy usage, etc., are outside the scope of this work. Additionally, we focus on speculative side-channel attacks that target the core concept of speculative execution. Specifically, we assume that memory permission checks are done fully and correctly as soon as the virtual to physical memory translation has been completed. If this is not the case, then we assume i) that loads are delayed until such permission checks are made and ii) that the duration of the relevant shadows (Section III.5) is also extended accordingly. Attacks that depend on such information not being checked on time (e.g. Meltdown) exploit implementation specific bugs that are not inherent in speculative execution, are possible only on specific CPU manufactures/models and can be fixed without restricting speculative execution.

Two components are necessary to successfully mount a speculative side-channel attack: i) A way of bypassing software/hardware barriers to access information illegally, and ii) a way of leaking that secret data across the speculation boundary. We are not concerned with how the attacker manages to execute the malicious code or how the secret data are accessed, we are only concerned with preventing the leakage of information. We assume that an attack can be launched from any context: From the same thread, from a different thread sharing the core (SMT), or even from a different core. Whether the attacker and the victim share the same virtual/physical memory or not is also not a limitation. We also assume that all data is equally important in needing to be kept secret, solutions that identify subsets of the data and only protect those can be applied to our solutions as well, but evaluating them is outside the scope of our work. Finally, outside the scope of this work are also attacks that are not speculative in nature, i.e., traditional side-channel
attacks, even if they utilize the same memory side-channel techniques.

III.3 Instruction and Memory Level Parallelism

Instruction level parallelism (ILP) refers to the ability of executing multiple instructions as the same time. Other than the obvious advantage of being able to utilize different functional units in parallel, ILP also prevents long-latency instructions from blocking the pipeline until they are completed, as long as other, independent instructions can be found in the instruction window.

Memory level parallelism (MLP) refers to the ability of having multiple in-flight memory operations at the same time. It is an important capability in modern high-performance CPUs, where the latency of memory accesses can be far greater than any computation instruction. While exploiting MLP offers multiple different advantages, there is one specific advantage that makes MLP crucial for high performance:

Assume two independent loads accessing different cache lines, both missing in the L1 cache, and both being at the head of the ROB, thus blocking other instructions from retiring. Let us also assume that the L2 cache has a round-trip latency of 12 cycles. If both loads are issued in parallel, then their latencies will overlap and the core will receive the data for both of them in (roughly) 12 cycles. So, in this scenario, the head of the ROB will remain blocked for 12 cycles. However, if we cannot issue these loads in parallel, i.e., we need to wait for the first load to retire before issuing the second, then the total latency for retiring both loads will be 24+ cycles. It is easy to see that the more in-flight loads we add the greater the benefit of exploiting MLP, and the higher the cost of not doing so.

In this example, we use L1 misses to demonstrate the importance of MLP, but we could also consider multiple accesses that hit in the L1 as exhibiting a degree of MLP. We will refer to these cases as hit-MLP. Since the latency of an L1 hit is low, the benefits of exploiting hit-MLP are not as significant.

To better understand the effects of delaying speculative loads, we will further divide MLP into two categories: intra-cacheline MLP and inter-cacheline MLP. By intra-cacheline MLP, we refer to multiple in-flight memory accesses that all target the same cache line. Once the line is brought into the cache, then all the memory accesses targeting it can be satisfied. However, even if these accesses where not overlapped, once the first access brings the line into the cache, then the rest would be cache hits, downgrading the intra-cacheline MLP to hit-MLP.
On the other hand, *inter-cacheline MLP* refers to the case when there are multiple independent memory accesses to *separate* cache lines, as in the example above. If inter-cacheline MLP is not exploited in the application, then each memory access to each cache line will introduce significant latency in the execution. Thus, we argue (and demonstrate in Section III.8) that inter-cacheline MLP is the most important type of MLP to exploit during execution.

**III.4 The State-of-the-Art**

There is a body of work covering side-channel defences for the memory system that predates speculative side-channel attacks [45, 21]. These works focus on the conventional side-channel attacks that leak information between different execution contexts. However, some speculative side-channel attacks, such as Spectre v1 [31], can be exploited from within the same execution context, bypassing the aforementioned defence mechanisms. Instead, we are interested in solutions that specifically target speculative side-channel attacks. Unfortunately, there is currently no established metric for comparing how secure each solution is, nor is there a way of comparing the security and cost trade-offs. This makes it impossible to directly compare the different solutions against each other, as each one makes different assumptions about what is secure or insecure, and what falls within its scope. We can, however, categorize the different solutions in three broad categories:

**Hiding speculation:** This approach includes solutions such as InvisiSpec [80], Ghost loads [63], and SafeSpec [27]. All of these allow speculative execution to continue unhindered, but delay any visible side-effects until after the speculation has been resolved. Specifically, all three solutions focus on hiding the side-effects of loads in the memory system by keeping fetched data in temporary buffers, in order to prevent speculative memory accesses from updating the cache. While such solutions do not fully restrict the MLP of the application during execution, they still segment it and also need to impose restrictions during validation for coherence reasons. They also have the disadvantage that they require modifications throughout the whole system and that, since instructions are allowed to execute and access the memory, it is hard to hide all possible side-effects.

**Delaying speculation:** Instead of trying to hide the side-effects of speculatively executed instructions, these solutions prevent “unsafe” instructions from being executed in the first place. We have quoted the word “unsafe” here because which instructions are considered as safe
and which are not changes depending on the exact threat model and implementation details of each solution. Our current work, delay-on-miss [64], delays all speculative loads until they are certain to be retired (Section III.5). Conditional Speculation [35], NDA [78], Speculative Taint Tracking (STT) [84], and SpectreGuard [20] keep track of the flow of information during execution and prevent any speculatively loaded data from being used by any “unsafe” instruction. The advantage of such solutions is that the changes required are mostly isolated in the core, instead of being pervasive in the whole memory hierarchy and the coherence protocol. At the same time, since instructions are not executed at all, they prevent a larger array of attacks than solutions that try to hide speculative execution. As a matter of fact, the last three solutions protect more than just the memory system, but at a significant performance loss. Their disadvantage is that by restricting the execution of speculative instructions, they restrict the amount of ILP and MLP that can be exploited during execution.

Undoing speculation: During speculative execution, architectural side-effects are kept hidden and any changes made during speculation that might lead to incorrect execution are squashed. CleanupSpec [62] takes a similar approach for the micro-architectural changes as well. Specifically, it utilizes cache randomization to prevent information leakage between different execution contexts, invalidations and re-fetching of data to undo the side-effects in the L1 cache, and delaying some accesses that can cause unsafe transitions in the coherence directories. This solutions achieves the best (reported) performance from all the other solutions, but at the cost of pervasive modification in the whole system. In addition, the energy costs have not been explored.

Some of the solutions already mentioned, namely Conditional Speculation and SpectreGuard, also boost performance by not protecting the whole memory space but instead only parts that the user has identified as secret. Such optimization can be applied to the other solutions as well but, to the best of our knowledge, it has not been explored by any of the authors. In our work we also assume that all information is equally sensitive, but there are no inherent limitations preventing us from only protecting parts of the memory space.

III.5 Speculative Shadows

As described in our previous work [63] [64], speculative shadows are cast by instructions that might cause a misspeculation and, subsequently, squashing of all instructions that follow them in the execution window.
III.5. Speculative Shadows

Essentially, the shadows are a way of defining (and tracking) when an instruction is guaranteed to be retired, without having to wait for it to reach the head of the reorder buffer (ROB) \[5\]. As long as an instruction is under such a shadow, then it cannot be speculatively executed \textit{safely}. We refer to such instructions as \textit{shadowed}, and to the instructions that cause the shadow as \textit{shadow casting instructions}. As soon as the shadow is \textit{lifted/resolved} and the instruction is \textit{unshadowed}, then it can be safely executed even if it is still speculative. Since we only focus on the memory system, we are only concerned with tracking shadowed loads; other instructions that might be used as part of a non-memory side-channel attack are not considered.

We have defined four baseline shadow types, plus an additional shadow when value prediction is utilized. These are as follows:

- **E-Shadows**: These shadows are cast by instructions that might throw an \textit{exception}, causing the execution to be aborted and redirected to an exception handler. Common examples include memory access operations that might cause memory exceptions and arithmetic operations.

- **C-Shadows**: These are shadows that are cast by instructions that \textit{control} the execution flow directly, most commonly branches. While the target of the branch is unknown, it is not possible to know if the instructions being executed are the correct instructions, or if they will have to be squashed when the branch target has been resolved.

- **D-Shadows**: When a load reads a memory location previously modified by a store, then the load has a \textit{data dependency} to that store. If during execution a store with an unknown store address is encountered, all following loads have the potential of being dependent on that store. Out-of-order processors deploy a memory dependency predictor, which determines if there is a high chance that a load can safely bypass a store. When a misprediction occurs, execution has to be restarted from the offending load.

- **M-Shadows**: This is a special type of shadow caused by the coherence and memory model requirements of the system. If the total store order (TSO) model is employed, the order between loads needs to be observed. Younger loads can speculatively bypass older loads, but if an order violation is detected\[^{1}\], the execution needs to be restarted from the younger load that caused the violation.

\[^{1}\text{In the context of the system’s memory model.}\]
This shadow is not present in systems where the memory model allows for loads to be reordered, such as release consistency (RC) based systems.

- **VP-Shadows:** If value prediction is employed, it is necessary to validate the prediction. If a prediction is incorrect, then the execution needs to be restarted, this time with the correct value.

It is possible to efficiently track these shadows in hardware [64] but, for brevity, we will not repeat the mechanism here in detail. With the exception of the VP-Shadows, which start once a load is predicted, the rest of the shadows start when an instruction is dispatched and put into the ROB. All of the shadows last until the reason for the shadow has been resolved. For example, C-Shadows can be removed once the target of the branch instruction is known. On the other hand, M- and VP-Shadows require the shadow casting load to be executed/validated before being lifted. This restricts the number of loads that can be unshadowed to one at a time, which makes solutions that delay all shadowed loads indiscriminately impractical.

In our previous work, the D-Shadows cast by stores were lifted once the store was ready to be executed, including having its value, as this is the point where the virtual to physical address translation happens in Gem5. However, this was too pessimistic, so for this work we decided to modify our implementation to lift D-Shadows once the address register(s) of the store are ready, assuming that the address translation can be performed before the store is ready to be issued. Specifically, we assume that it is possible to split the store address calculation and translation into a separate micro-op and then directly feed the physical address into the actual memory store micro-op. It should be noted here that we are not aware of any systems that actually do this, but there are no fundamental limitations preventing such behavior from being implemented.

Figure III.1 contains the number of loads executed under a speculative shadow in the delay-on-miss version for the SPEC2006 benchmarks, normalized to the total number of loads executed. This number ranges from 64% (h264ref) to 93% (libquantum), with a mean of 80%. In Figure III.2 we can also see the types of instructions that cause these shadows. Specifically, each time we need to check if an instruction is under a shadow or not, we record the oldest shadow-casting instruction. As we have modified our shadow-tracking implementation to lift shadows caused by stores earlier, the number of store shadows is greatly reduced. In our experiments, removing just one shadow type does not significantly affect the performance, as each load is usually covered by
III.6 Delay-on-Miss

Delay-on-miss protects against speculative attacks that use the memory hierarchy as a side-channel by delaying all “unsafe” loads that miss in the L1 cache. As “unsafe” we define all loads that are executed under a speculative shadow. Stores are not delayed, because stores are not allowed to speculatively modify the memory hierarchy in the first place. Other, non-memory instructions are also not delayed, since they fall outside the scope of this work. The main insight behind delay-on-miss is that hiding the side-effects of loads that hit in the private L1 cache is easy, as the data can be returned without making any changes in the cache, and the replacement policy and prefetcher updates can be delayed until later. On the other hand, hiding loads that need to fetch data from other, lower level, caches is not as easy, because it requires mechanisms for getting the data without modifying anything in the hierarchy or the main memory, storing the data temporarily, and handling the coherence implications [80, 63].

In the context of delay-on-miss, we consider loads that miss in the L1 but hit in an MSHR entry that has been allocated by a safe operation as hits, as the data will be brought in by the safe memory operation anyway.
This introduces some additional memory level parallelism (both intra- and inter- cacheline), as it is now possible to coalesce shadowed loads with other unshadowed/non-speculative operations, such as unshadowed loads, validations, stores, or even prefetches, without degrading the security guarantees of delay-on-miss.

Delay-on-miss differs from mechanisms such as NDA [78] or STT [84], as both NDA and STT allow for speculative loads to be executed but delay all dependent instructions. This is based on the insight that for a secret to be leaked, it has to be loaded first. The first load, the one that loads the secret into the core, does not itself leak any information, instead the loaded value is fed to other instructions, which in turn construct a side-channel and leak the secret. In STT, only values loaded by a load that is currently speculative are considered unsafe. The implication is that STT does not protect against attacks that leak secret values that are already stored in registers [78]. NDA on the other hand offers different modes, one of which protects all values from being leaked, at additional performance cost.

III.7 Value Prediction

Value prediction is a micro-architectural optimization that aims at reducing the delay and overheads of executing [38] or even scheduling [57] instructions by predicting their value based on previously seen values. Similar to a cache, a value predictor takes advantage of the locality
that data exhibit but instead of focusing on spatial or temporal locality, the value predictors exploits data with *value locality*. In our work, we use value prediction as a mechanism for hiding the delay introduced by delaying shadowed loads. Evaluating all the different predictor types, their advantages and disadvantages, and how well they perform, is beyond the scope of this work, but we do have three requirements that the predictors must meet in order to be able to be safely used:

1. The predictor must be able to work on the L1 level.
2. The predictor must not require any memory accesses to perform a prediction.
3. The state of the predictor must be able to be obfuscated, any updates to it must be able to be delayed while speculative, and any state kept during speculation must be squashed in the event of a misprediction.

The reason for these requirements is that we cannot allow any memory accesses past the L1 during speculation and we also do not want the predictor itself to leak any information. While the values predicted are speculative, if these criteria are met then it is a safe, invisible form of speculation, isolated from other cores or execution contexts.

Figure III.3 contains the VP prediction rate for each benchmark, i.e., the number of shadowed L1 misses that the VP is able to predict.
over the total number of shadowed L1 misses. In the same figure, the L1 miss ratio of the benchmarks can also be seen, as the total number of predicted loads over all loads depends on the number of L1 misses during execution. Overall, the VP achieves a prediction rate of 23%, with benchmarks ranging from as low as 1% (namd) to as high as 98% (cactusADM). In the performance analysis, Section III.8, we will also present an oracular VP, with 100% prediction rate, to evaluate the upper limit of the performance gain that improving the prediction rate and accuracy can offer.

Since the values provided by the predictor are speculative, they need to be validated, both to ensure that the value was correct at the time of the prediction, but also to avoid coherence issues on memory models that enforce the load-load order. To do so, a normal, globally visible memory access needs to be issued in the memory system. Since such an access would be unsafe under delay-on-miss, the same rules as for normal loads apply: If the validation access hits in the L1, then it can be proceed unhindered, otherwise it has to be delayed. At the same time, if the validation fails, all instructions that follow the value predicted load need to be squashed, hence why the VP-Shadow is introduced. This means that while value prediction introduces some additional ILP, it does not introduce any MLP, as it simply pushes the actual memory access from the execution point to the validation. We will discuss the performance implications of this in detail in Section III.8.2.

### III.8 Evaluation

We use a combination of the Gem5 simulator [7], McPAT [36] with CACTI [37], and the SPEC2006 [68] benchmark suite. The architectural parameters of the simulated system can be found in Table III.1. In Gem5, we first fast-forward through one billion instructions using the atomic simple CPU, and then simulate in full detail using the O3 CPU for another three billion instructions. From all the benchmarks in SPEC2006, we have excluded perlbench, dealII, povray, calculix, tonto, and xalancbmk due to baseline simulation issues. In total, we evaluate seven different hardware versions:

---

2There is a possibility of selectively squashing and replaying only the load-dependent instructions, but evaluating such approaches are beyond the scope of this work.
Table III.1: The simulated system parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology node</td>
<td>22nm</td>
</tr>
<tr>
<td>Processor type</td>
<td>out-of-order x86 CPU</td>
</tr>
<tr>
<td>Processor frequency</td>
<td>3.4GHz</td>
</tr>
<tr>
<td>Issue / Execute / Commit width</td>
<td>8</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 private cache size</td>
<td>32KiB, 8-way</td>
</tr>
<tr>
<td>L1 access latency</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 shared cache size</td>
<td>1MiB, 16-way</td>
</tr>
<tr>
<td>L2 access latency</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Value predictor</td>
<td>VTAGE</td>
</tr>
<tr>
<td>Value predictor size</td>
<td>13 components × 128 entries</td>
</tr>
</tbody>
</table>

- **baseline**: The unmodified, *unsecured* out-of-order CPU that Gem5 provides.

- **delay-all**: A secured out-of-order CPU where all shadowed loads are delayed until they have been unshadowed.

- **DoM**: Delay-on-miss, without any value prediction.

- **DoM + VP**: Delay-on-miss with a VTAGE value predictor.

- **DoM + VP (Oracle)**: Delay-on-miss with the oracular value predictor, capable of correctly predicting all speculative L1 misses. Even though the Oracle VP is always correct, validations are still being sent out once the load has been unshadowed.

- **DoM + VP (Instant)**: Delay-on-miss with the VTAGE predictor, but validation are sent as soon as the value has been predicted, instead of waiting for the load to be unshadowed. This is an *unsecured* version that is used to evaluate the cost of delaying and serializing the validations.

- **DoM + VP (Oracle + Instant)**: Delay-on-miss with the oracular predictor and instant validations. This is another unsecured version used to demonstrate the cost of validation.

McPAT can model the energy usage of the CPU components but it does not offer the capability of adding new components without modifying its source code. In order to calculate the energy usage of the value
Figure III.4: Performance (IPC) normalized to the unsecured baseline.
predictor, we model it as a branch target buffer (BTB) and a branch predictor (BP) in McPAT. While the implementation details of these units differ from the VP, they are the two components that are the closest to the VP. In addition, McPAT does not model the DRAM, for this we rely on the integrated DRAMPower [12] model in Gem5.

A significant difference with our previous work is how the dynamic energy usage of the system is modeled. Previously, while the instruction queue and the ROB were idle, Gem5 would still poll them to try to find instructions ready to issue or commit. This lead to an increase in the dynamic energy proportional to the execution time. We have since modified our energy model to exclude this polling, under the assumption that modern CPUs are capable of clock-gating such structures until an instruction becomes ready.

### III.8.1 Performance

In this section we will establish the performance of the different versions, before further diving into the different runtime characteristics in the sections that follow. Figure III.4 contains the average number of retired instructions per cycle (IPC) for each benchmark and for each version, normalized to the unsecured baseline version. First, we have the delay-all approach, where all shadowed loads are delayed until they are unshadowed. The performance of this version ranges from $-74\%$ (hmmer) to $-3\%$ (lbm) under the baseline, with a mean performance loss of 53%.

The next version is the delay-on-miss version, where we only delay shadowed loads if they miss in the L1. With a mean performance of $-18\%$ under the baseline, delay-on-miss improves significantly over delay-all, reducing the performance loss by two thirds. The five benchmarks that perform the worst with delay-on-miss are GemsFDTD ($-67\%$), mcf ($-55\%$), omnetpp ($-41\%$), milc ($-26\%$), and bwaves ($-25\%$). We will discuss these benchmarks further in the next section (III.8.2), where we discuss the effects of the different versions in the amount of memory level parallelism exploited. For the remaining of the SPEC2006 benchmarks, the performance loss is constrained to less than 25% under the baseline, with five benchmarks reaching 99% (h264ref, sjneg, and gobmk) or even 100% (wrf and lbm) of the baseline performance.

When value prediction is introduced into delay-on-miss, the performance loss is further reduced by 1 percentage point with the VTAGE predictor and by 3 percentage points with the oracle predictor, which contradicts our old value prediction results. The reason for this discrepancy is that in our original implementation the overhead of the validations had not been modeled as accurately, understating the effects of
the validation in the MLP of the application, which led to an overestimation of the performance benefits of value prediction. Instead, in order to achieve results that are similar to our initial simulations, we need to introduce *instant* validations, which can match or even exceed the initial VP results. With instant validations, the VTAGE predictor can achieve a mean performance loss of only 8% under the baseline, while the oracle predictor can even exceed the baseline by 3%.

### III.8.2 Memory Level Parallelism

Having established the performance that the different versions exhibit, it is time to understand why. As we have explained, memory level parallelism plays an important role in the performance of the applications. We will use the MSHRs in the L1 cache as a proxy metric for measuring the amount of MLP during execution. For each cache line requested in a cache, an **MSHR entry** is allocated. Each entry in turn has several **targets**, each of which corresponds to a memory access, whether that access is a load or a store. The number of entries indicates how many different cache lines are fetched in parallel (inter-cacheline MLP) while the number of targets indicates how many accesses benefit from each fetch (intra-cacheline MLP).

![Figure III.5](image.png)

**Figure III.5** contains the average number of MSHR entries (III.5a) and targets (III.5b) in the matching entry already allocated when a load access misses in the L1 cache. For loads that are delayed or value predicted, we only account for their visible access/validation, i.e., we do not count the number of MSHR entries/targets at the point that they were delayed. Similarly, delayed loads are not counted as part of the allocated targets/entries. For the MSHR entries, we also differentiate between entries that were allocated by a load (excluding prefetches – bottom, dark color) and entries that were allocated by another operation (top, light color). In figures III.6 and III.7 the L1 data-cache miss ratio and the last-level cache (LLC) MPKI can also be seen. By looking at the mean number of entries (III.5a) we can immediately see how each version affects the inter-cacheline MLP of the application. In the baseline, there is an average of two MSHR entries already allocated when a load misses in the L1, which translates to two cache lines being fetched in parallel. Out of these, roughly half are entries allocated by loads, while the rest correspond to other operations. If we indiscriminately delay all shadowed loads (delay-all), then the number of load-allocated entries falls to zero and the total to a mean value of less than one. Essentially, in the majority of the cases, when a load misses in the L1 there are no

---

3 Other operations (such as prefetching) might also allocate MSHR targets.
Table III.8: Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>MSHR Entries</th>
<th>MSHR Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bwaves</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gamess</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>milc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zeusmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gromacs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cactusADM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>leslio3d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>namd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>soplex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hmmer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sjeng</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GemFDTD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libquantum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h264ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omnetpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>astart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wrf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sphinx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMean</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) MSHR Entries. The bottom dark part represents entries allocated by memory reads.

(b) MSHR Targets.

Figure III.5: Average number of MSHR entries (inter-cacheline) and targets per entry (intra-cacheline).
Figure III.6: L1 miss ratio.

Figure III.7: LLC (L2) misses per one thousand instructions.
other pending memory fetches. This translates to very poor MLP, which in turn leads to very poor performance.

On the other hand, by introducing delay-on-miss and only selectively delaying loads, we can recover over half (55%) of the baseline MLP. Specifically, the number of load-allocated entries is increased from 0 to 36% of the baseline, but the majority of the MLP comes from the rest (non-load-allocated), at 81% of the baseline. Introducing the value predictor, either the VTAGE predictor or the Oracle, does not significantly change these numbers, which supports our analysis that value prediction only aids in gaining ILP and not any inter-cacheline MLP. We only see a boost in the MLP once the instant validations are used, with the VTAGE version reaching 76% of the baseline, and the Oracle version reaching 100% of the baseline.

Finally, if we look into the number of MSHR targets found in the matching MSHR entry, we see that only the delay version leads to a significant decrease. This is due to the fact that, as explained in Section III.6, we can allow shadowed loads to proceed if a safe memory request is already in flight. The fact that delay-on-miss does not negatively affect intra-cacheline MLP contributes to the reduced performance cost over simply delaying all loads, but as we have already discussed, its effect on the inter-cacheline MLP is also important.

We can look at GemsFDTD as a specific example of this, as it is the application with the biggest performance degradation. In the baseline, there is an average of 5.7 MSHR entries already allocated when a load misses in the L1 cache. Out of these, two thirds are entries that have been allocated by other loads. With delay-on-miss we regain the MLP for the entries that are not allocated by loads (as well as the intra-cacheline MLP), but we only regain 8% for the load-allocated entries. The same is true for the VP and Oracle VP versions, as long as the (unsafe) instant validations are not allowed. On the other hand, if we allow instant validations, then we regain back the majority of the MLP. We can make similar observations for other benchmarks as well, such as bwaves, mcf, milc, and omnetpp, which are the benchmarks we identified in Section III.8 as the benchmarks with the biggest performance loss under delay-on-miss. On the other hand, on benchmarks such as wrf and lbm (the best performing benchmarks), delay-on-miss fully recovers all of the baseline MLP.

### III.8.3 Energy Usage

As neither version significantly affects MPKI or the instructions executed by the benchmarks, the two main factors affecting the energy
Figure III.8: Energy usage normalized to the unsecured baseline. Each bar consists of four parts (bottom to top): The dynamic energy usage of the CPU (bottom, light colored), the static energy usage of the CPU (middle, dark colored), the dynamic energy usage of the DRAM (middle, light colored), and the VP overhead, both static and dynamic (top, dark colored).
usage is the execution time and the overhead of doing value predictions. As the value predictor uses a relatively small amount of energy, compared to the rest of the system, the execution time is the main parameter affecting the energy usage.

The results can be seen in Figure III.8. Each bar consists of four parts, from bottom to top: The dynamic energy usage of the CPU (bottom, light colored), the static energy usage of the CPU (middle, dark colored), the dynamic energy usage (including refresh and power-down states) of the DRAM (middle, light colored), and the overhead of the VP, both static and dynamic (top, dark colored).

We observe that the dynamic-energy usage of the CPU is not significantly affected by the different versions, instead, the static-energy usage, as well as the DRAM-energy usage, are the main factors contributing to the energy overheads. The delay-all version increases the mean energy by 48% over the baseline, while the delay-on-miss reduces this overhead by four fifths, to just 9% over the baseline. With value prediction, the overhead increases insignificantly, due to the added overhead of the value prediction. Finally, the VTAGE and oracle versions with (unsafe) instant validations reduce the overhead to 4% and −1%, respectively.

### III.9 Conclusions

We have performed a detailed evaluation of the implications of delaying speculative loads to improve the resilience of modern, high-performance CPUs against speculative side-channel attacks targeting the memory hierarchy. This evaluation leads to the new insight that the major factor affecting the performance of the applications under such solutions is the amount of memory level parallelism that can be exploited during execution. Our delay-on-miss solution, which builds on the idea that only loads that miss in the L1 cause significant side-effects, is able to recover a significant portion of the MLP lost by indiscriminately delaying all speculative loads, achieving a performance loss of only 18% compared to the baseline. In contrast, the aforementioned delay-all approach that indiscriminately delays all speculative loads suffers from a 53% performance loss.

We have also shown that introducing value prediction does not significantly alter the performance, as value prediction increases the instruction level parallelism of the application but not the MLP. Instead, for value prediction to improve the performance of delay-on-miss, we would have to remove the delay-on-miss restrictions from the prediction
validations, which would lead to information leakage. However, similar to other approaches that face the same issues, such as InvisiSpec, we believe that, under certain cases, future work can remove some of the restrictions on validations, which will lead to significant performance improvements.

Acknowledgments

This work was funded by Vetenskapsrådet project 2015-05159, by the SSF Strategic Mobility 2017 grant SM17-0064, the Spanish MCIU and AEI, as well as European Commission FEDER grant RTI2018-098156-B-C53. The computations were performed on resources provided by SNIC through UPPMAX.
Bibliography


Recent licentiate theses from the Department of Information Technology

2020-002 Ulrika Sundin: Global Radial Basis Function Collocation Methods for PDEs


2019-006 Kristiina Ausmees: Efficient Computational Methods for Applications in Genomics

2019-005 Carl Jidling: Tailoring Gaussian Processes for Tomographic Reconstruction

2019-004 Amin Kaveh: Local Measures for Probabilistic Networks


2019-001 Diane Golay: An Invisible Burden: An Experience-Based Approach to Nurses’ Daily Work Life with Healthcare Information Technology

2018-004 Charalampos Orfanidis: Robustness in Low Power Wide Area Networks

2018-003 Fredrik Olsson: Modeling and Assessment of Human Balance and Movement Disorders Using Inertial Sensors

2018-002 Tatiana Chistiakova: Ammonium Based Aeration Control in Wastewater Treatment Plants - Modelling and Controller Design

2018-001 Kim-Anh Tran: Static Instruction Scheduling for High Performance on Energy-Efficient Processors