Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design

Kim-Anh Tran
Uppsala University
Uppsala, Sweden
kim-anh.tran@it.uu.se

Christos Sakalis
Uppsala University
Uppsala, Sweden
christos.sakalis@it.uu.se

Magnus Själander
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
magnus.sjalander@ntnu.no

Alberto Ros
University of Murcia
Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Uppsala University
Uppsala, Sweden
stefanos.kaxiras@it.uu.se

Alexandra Jimborean
Uppsala University
Uppsala, Sweden
alexandra.jimborean@it.uu.se

ABSTRACT

Out-of-order processors heavily rely on speculation to achieve high performance, allowing instructions to bypass other slower instructions in order to fully utilize the processor's resources. Speculatively executed instructions do not affect the correctness of the application, as they never change the architectural state, but they do affect the micro-architectural behavior of the system. Until recently, these changes were considered to be safe but with the discovery of new security attacks that misuse speculative execution to leak secrete information through observable micro-architectural changes (so called side-channels), this is no longer the case. To solve this issue, a wave of software and hardware mitigations have been proposed, the majority of which delay and/or hide speculative execution until it is deemed to be safe, trading performance for security. These newly enforced restrictions change how speculation is applied and where the performance bottlenecks appear, forcing us to rethink how we design and optimize both the hardware and the software.

We observe that many of the state-of-the-art hardware solutions targeting memory systems operate on a common scheme: the visible execution of loads or their dependents is blocked until they become *safe to execute*. In this work we propose a generally applicable hardware-software extension that focuses on removing the causes of loads' *unsafety*, generally caused by control and memory dependence speculation. As a result, we manage to make more loads *safe to execute* at an early stage, which enables us to schedule more loads at a time to overlap their delays and improve performance. We apply our techniques on the state-of-the-art Delay-on-Miss hardware defense and show that we reduce the performance gap to the unsafe baseline by 53% (on average).

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414640

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures**; • **Software and its engineering → Source code generation**.

KEYWORDS

speculative execution, side-channel attacks, caches, compiler, instruction reordering, coherence protocol

ACM Reference Format:
Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. 2020. Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3410463.3414640

1 INTRODUCTION

Side-channel attacks have been known to the security and hardware communities for years, and they have been demonstrated to be effective against a number of security systems [6, 9, 23]. Among them, attacks that use the memory system as the side-channel, be that the caches, the main memory, the memory bus, or even the coherence mechanisms, have been particularly effective, partly due to how easy it is to exploit them [23].

However, recently, with the introduction of Meltdown [21] and Spectre [17], a new class of side-channel attacks has emerged: Speculative side-channel attacks. These attacks can still exploit the same side-channels but they do so under speculative execution. This makes them especially devastating because *(i)* software countermeasures can be easily bypassed during speculative execution (e.g., Spectre), *(ii)* hardware countermeasures might also be bypassed during speculative execution (e.g., Meltdown), and finally because *(iii)* the speculative execution might be squashed, leaving no trace of anything malicious having ever happened. The reason why these attacks are possible is because while architectural changes, such as writes to architectural registers or the memory, are kept hidden during speculative execution, micro-architectural changes are not. These might include memory reads, which introduce changes in the cache hierarchy [17], instruction execution, which introduces

```
1  uint8_t array[10];
2  uint8_t probe[256 * 64];
3
4  uint8_t victim(size_t index) {
5    if (index < 10)
6      return array[index];
7    else
8      return 0;
9  }
10
11 void attack() {
12   // Train the branch predictor
13   for (...) victim(0);
14   // Flush the probe array from the cache
15   for (...) clflush(probe[i * 64]);
16   // Speculatively load secret data
17   secret = victim(10000);
18   // Leak the secret value
19   _ = probe[secret * 64];
20 }
```

Figure 1: Speculative side-channel attack example code.

resource contention [35], or even changing the frequency of the core [33].

In this work we focus on attacks exploiting the memory system. Figure 1 contains a simplified example that shows how such an attack can be constructed. The exact same principle is used in the Spectre v1 attack [17]. In this example, the attacker wants to bypass the check enforced by the `victim` function (Line 5) in order to perform an out-of-bounds access on `array` and access a memory location containing a secret value (Line 17). The attack is performed by:

(1) The attacker starts by training the branch predictor to assume that the `if` statement in `victim` is always true (Line 13). This can be done by simply calling the `victim` function with valid indexes multiple times. Additionally, a probe array, which will be used later, is allocated and flushed from the caches (Line 15).
(2) The attacker then proceeds to call the `victim` function with an invalid index (Line 17). It will take some time before the `if` condition can be checked but thanks to branch prediction and speculative out-of-order execution the execution can continue speculatively.
(3) Since the branch predictor has been trained to assume that the `if` statement is always true, the execution will continue speculatively by accessing the `array` with the invalid index. The attacker then proceeds to use the secret value as an index in the probe array (line 19). This will cause one cache line from the array to be loaded into the cache; namely the one that is *indexed by the secret value*.
(4) Once the branch misprediction is detected, the speculative execution is squashed without causing any architectural changes. The execution restarts at the `if` statement, this time returning the value `0`, indicating an error.
(5) The attacker can now probe the probe array, by trying each possible index and measuring the time it takes to access the array. Since one cache line was loaded during the attack, and the index of that cache line depends on the secret value read during the attack, the attacker can determine the secret value by finding which cache line that takes less time to access (not shown here).

Many state-of-the-art hardware defense mechanisms will try to either delay or hide the speculative load that leads to the information leakage. In our example, the speculative access to `array[10000]` would therefore not return a value that can be used to leak the secret. While secure, such mechanisms suffer from reduced performance [30, 36, 39]. Being able to execute (speculative) loads in parallel and out-of-order is crucial for performance. It allows memory latencies to be overlapped, which makes better use of existing resources and achieves a high degree of *memory-level-parallelism* (MLP). With MLP, the performance cost of memory operations can be significantly reduced. With the delay of loads, memory accesses have to be serialized and the gap between memory and processor speed widens even more. In a way, disallowing the processor to speculatively execute loads is a restriction on the *out-of-orderness* of the out-of-order processor.

In this paper we look into the possibility to find MLP despite the serializing effects of delaying speculative loads for security. Our goal is to further close the performance gap between the unsafe baseline and the secure out-of-order processor. To this end, we introduce a software-hardware extension that is generally applicable to several existing hardware defense solutions. Our observation is that delay-based mitigation techniques only allow the side-effects of speculative loads to be observable as soon as they are deemed as *safe*. What they all miss is that we can actually *influence* when loads become safe if we can accomplish to *remove the cause for speculation* at an early stage.

Our techniques remove the reason for speculation when possible, and otherwise shorten the period of time in which loads are considered to be speculative. As a result, more loads become *safe to execute* and we unlock and exploit the potential for MLP, and thus for performance. Our contributions are:

(1) The proposal of a generally applicable software-hardware extension to improve the performance of hardware solutions that target speculative attacks on the memory system by delaying or hiding loads and their dependents, including:
   (a) The usage of a coherence protocol that allows loads to be safely, non-speculatively, reordered under TSO [27], thus unlocking the potential for MLP.
   (b) An instruction reordering technique that exposes more MLP through *(i)* prioritizing instructions that contribute to unresolved memory operations and unresolved branches, and through *(ii)* scheduling independent instructions in groups.
(2) The evaluation of our extension on top of the state-of-the-art Delay-on-Miss security mechanism [30], which delays speculative loads that miss in the L1 cache.

Although we select a specific hardware defense to evaluate our ideas, our solutions are not tied to a specific system. They are applicable to any hardware solutions that tackle observable memory-hierarchy side-effects by restricting the execution of loads and their dependents, since this is what we focus on.

Our evaluation shows that our techniques improve over Delay-on-Miss with 9%, and thus reduce the performance gap to the unsafe baseline processor by 53%,

2 SPECULATIVE SHADOWS AND DELAY-ON-MISS

Completely disabling speculative execution would solve all speculative side-channel attacks, but it would come at an unacceptable performance cost. Instead, the selective delay solution proposed by Sakalis et al. [30] reduces the observable micro-architectural state-changes in the memory hierarchy while trying to delay speculative instructions only when it is necessary. Specifically, only loads are delayed, as other instructions (such as stores) are not allowed to cause any changes in the memory hierarchy while speculative. In addition, only loads that miss in their private L1 cache are delayed, as loads that hit in the L1 cause minimal side-effects that can be easily hidden until the load is no longer speculative. Sakalis et al. name their technique *Delay-on-Miss*.

The authors introduce the concept of speculative shadows [29, 30] to understand when a load is considered to be speculative. Traditionally, any instruction that has not reached the head of the reorder buffer might be considered speculative, but speculative shadows offer a more fine-grained approach. Speculative shadows are cast by instructions that may cause misspeculation, such as branches. Branches need to be predicted early in the pipeline, as instructions need to be fetched based on the branch target. If the branch is mispredicted, then the wrong instructions might be executed, as seen in the example in Section 1. However, there is no need to wait until the branch reaches the head of the reorder buffer to mark it as non-speculative, instead this can be done as soon as the branch target has been verified. Therefore, the branch will cast a shadow that extends from the moment the branch enters the reorder buffer until the branch target is known.

The authors categorize the shadows into four types, depending on the reason of misspeculation: the E-(exception), C-(control), D-(data), and M-(memory) shadows. If value prediction is used, a fifth type, the VP-(value prediction) shadow is also introduced, but we are not exploring the use of value prediction in this work. Table 1 shows an example for each shadow type. E-shadows relate to instructions that may throw an exception, C-shadows are caused by unresolved branches, D-shadows by potential data dependencies, and, finally, M-shadows exist under memory models where the observable memory order of loads has to be conserved, such as the Total Store Order (TSO) model. Shadows are lifted as soon as the reason for the potential misspeculation is resolved; for example, for memory operations, the E-shadow is lifted as soon as the permission checks can be performed. If a load is under any of these shadow types then it is not allowed to be executed, unless it hits in the L1 cache.

Figure 2 shows the performance degradation of delaying unsafe loads as described by Sakalis et al. on a range of SPEC 2006 benchmarks. Each benchmark is represented by a number of *hot regions* that were identified through profiling (for more information on the selection of regions for evaluation see Section 4). On average the delay of loads incurs a 23% performance degradation compared to the unsafe, unmodified out-of-order core, measured in the number of cycles required to execute the regions.

Table 1: Examples for shadow types identified by Sakalis et al. [30]. In each example, the load instruction in `y = ...` is under a shadow cast by the previous instruction.

Type	Example

E-shadow (**E**xception)	`int x = a[invalid] /* throws */`
	`int y = a[i]`
	`/* E-shadows are cast by any instruction that may throw an exception*/`

C-shadow (**C**ontrol)	`if (test(i)) { /* unknown path */`
	`  int y = a[i]`
	`}`
	`/* C-shadows are cast by unresolved branches.*/`

D-shadow (**D**ata)	`a[i] = compute()`
	`int y = b[i]  /* a == b? */`
	`/* D-shadows are cast by potential data dependencies. */`

M-shadow (**M**emory)	`int x = a[i] /* load order in TSO */`
	`int y = a[i+1]`
	`/* M-shadows conserve the observable load ordering under TSO.*/`

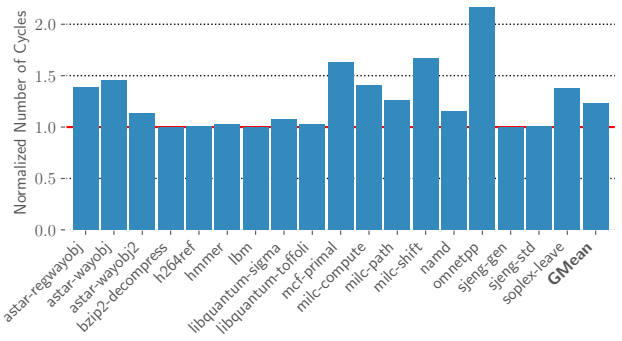


Figure 2: Impact of shadows on performance: the number of cycles required for Delay-on-Miss running the selected regions, normalized to the unsafe out-of-order processor

Figure 3 shows the contribution of loads, stores, control and other instructions to the overall number of shadows that are cast for the selected benchmarks. The largest proportion of shadows is cast by

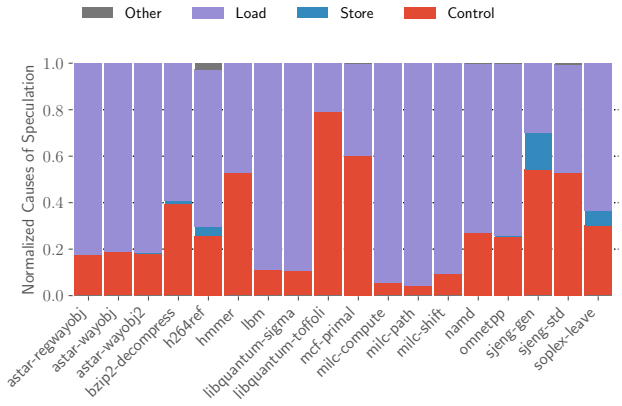


Figure 3: Causes of speculation

```
                                                          Shadows
for (int i = 0; i < 1000; ++i) {                              C
  int addr1 = ..;                                             /
  int l1 = p1->a[addr1];                                      E,M
  params.fullf[0] = l1;                                       E,D
  int addr2 = ..;                                             /
  l2 = p2->a[addr2];                                          E,M
  params.fullf[1] = l2;                                       E,D
  bool cond = l1 < l2;                                        /
  if ( cond ) ..                                              C
}                                                             /
```

Figure 4: Example code showing the type of shadows cast by the instructions (E,C,M,D), and their overlap. Instructions towards the end of the code excerpt are blocked by several overlapping shadows and thus darker.

load and control instructions, only a small proportion is cast by stores, and a minimal amount is cast by the remaining instructions, such as floating point operations. In the following we will discuss how to shorten the shadow duration of those instructions that contribute most to the overall number of cast shadows, namely load, store, and control instructions.

3 EARLY SHADOW RESOLUTION AND ELIMINATION

When the shadow that covers a load is resolved/removed, we refer to that load as *unshadowed*, and the act as *unshadowing a load*. For most loads, removing a single shadow is not enough, because they are covered by *multiple overlapping shadows* and for the load to become unshadowed, all shadows cast by preceding instructions need to be removed. Consider Figure 4, which shows a code example for overlapping shadows. To the right of the code we annotate which types of shadow that line is casting. As an example, the first line (`for (int i = 0; i < 1000; ++i)`) contains a comparison ($i < 1000$) which is used to branch to the loop body. Unresolved branches cast C-shadows, and therefore a shadow (illustrated with a gray box) spans over the succeeding code. As almost all lines cast shadows, an increasing number of shadows end up overlapping.

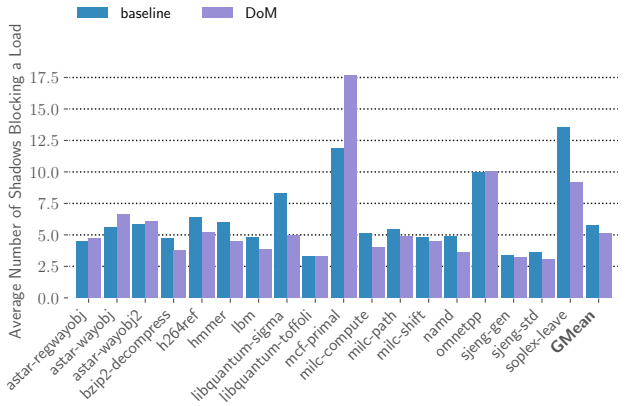


Figure 5: Average number of shadows that are blocking a load at a time.

This example illustrates why simply removing one single shadow does not make any difference: to successfully unshadow loads we need to remove *all overlapping shadows* cast by the instructions that lead up to each load.

For SPEC 2006, an average of 63% of the total number of dynamic instructions are either loads, stores, or branches [25]. This means that at least 63% of the instructions in SPEC 2006 have the potential to cast shadows[1]. In Figure 5 we can see, on every cycle, the average number of shadows that each load is simultaneously under. The results show that there are on average five separate overlapping shadows shadowing each load. Across all benchmarks, the maximum number of distinct shadows that shadow a load at a time is 59.

MLP: The Key to Performance. An important aspect of speculative execution is allowing multiple loads to execute in parallel, which enables faster loads (cache hits) to bypass long latency loads (cache misses) and also multiple long latency loads to overlap with one another. This results in memory-level-parallelism, which benefits performance significantly. Shadows prevent multiple loads from executing ahead of time, since sensitive information may be leaked if a load is executed when it should not have been. Shadows thus handicap the out-of-order processor's capability to speculatively execute instructions (loads) in an out-of-order fashion. The execution of loads is *serialized*, which affects the performance of both memory- and compute-bound applications.

To successfully narrow the performance gap between the unsafe and the secure out-of-order processor, we need to find ways to increase MLP while maintaining the same security guarantees. But how can we achieve MLP? Loads, stores, and branches are usually interleaved in the code, and so are their shadows. For us to successfully unshadow loads, overlap their latencies and thus increase MLP, we need to find solutions that consider all shadow types. In the following sections we detail how this can be done. Table 2 gives an overview on the shadow-casting instructions and their shadow types, as well as the techniques that we apply to remove them. Where necessary, i.e., strong-consistent systems, we propose

[1]Other instructions, like floating point operations, may also cast shadows due to exceptions. However, these exceptions can often be disabled through software.

Table 2: Overview on shadow-casting instructions, the shadows they cast (✗), and the solutions in this work to address them. The percentage (%) shows the average number of shadows for which the instruction is responsible (for SPEC 2006 [25]). We exclude shadow-casting instructions that are not memory or control instructions, as their total share is negligible (see Section 2). By excluding them, D- and E-shadows can be combined into one category.

Shadow (%)	Load (70%)	Store (1.9%)	Branch (28%)	End of Shadow when..	Unshadowing Technique
E-shadow	✗	✗		Target address known	Early Target Address Computation (Section 3.2)
C-shadow			✗	Branch target address known	Early Condition Evaluation (Section 3.2)
M-shadow	✗			Load has executed	Non-speculative Load-Load Reordering (Section 3.1)

changing the coherence protocol to *completely remove* M-shadows. We also propose applying compiler techniques to shorten, or in some cases completely eliminate, the duration of E- and C-shadows. D-shadows overlap with their respective E-shadows (both resolve as soon as the address is known) and will therefore not be explicitly mentioned in the following sections.

3.1 Non-Speculative Reordering of Loads (M-shadows)

Among all shadows the M-shadows are the most restrictive on MLP. They are cast by every single load, and even if all other shadows could be magically lifted, the M-shadows would still enforce program order for all loads. Without the security concerns, an out-of-order processor may speculatively bypass an older load if the younger load is delayed (e.g., if its operands are not yet available). A reordering is observed if two reordered loads obtain values that contradict the memory order in which these values have been written. Consider two loads `ld x`, `ld y` that are executed on one core and two stores `st y`, `st x` that are executed on another core. Let x_1 be the old value and x_2 the updated value of x after the store (similarly for y_1 and y_2). An illegal reordering under TSO would be one in which the first load `ld x` loads x_2 (the updated value), but the second load `ld y` loads y_1 (the old value). This reordering can happen if `ld y` bypasses `ld x`.

Since the M-shadows disallow reordering, loads are serialized, which restricts our ability to improve MLP. To solve this, we propose applying a method for non-speculative load-load reordering [27] that allows reordering of loads while effectively hiding it through the coherence protocol. In other words the execution of younger loads before older loads is allowed (given that they they are independent and both are valid accesses to memory), but not revealed to other cores. Consider the following scenario on the previous example: a core bypasses `ld x` (e.g., because loading x misses) and executes `ld y` ahead of time. Another core now performs the store operations to the same memory locations `st y`, `st x`. Since the loads have been reordered, this would normally lead to an invalidation and therefore the squashing of the speculated load `ld y`. Instead of squashing, the coherence protocol delays acknowledging the invalidation, such that both loads can finish execution and the reordering cannot be detected any longer, thus eliminating the possibility of a misspeculation.

Note that the M-Shadows is an artifact of systems that require the aforementioned load-load order to be enforced, such as on x86 systems utilizing the TSO memory model. On systems where this is not the case, such as on the numerous ARM systems utilizing a Release Consistency (RC) memory model, the M-Shadows do not exist and there is no reason to implement a non-speculative load-load reordering solution, such as the one described above, to eliminate them.

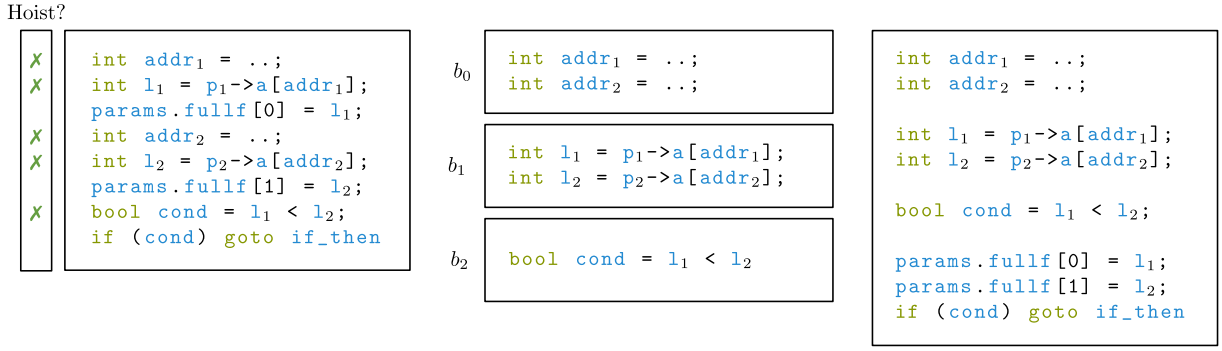
3.2 Early Evaluation of Conditions and Addresses (C- and E-Shadows)

Both C- and E-shadows are lifted as soon as the physical target addresses of memory operations and branch targets are known (for branches, that means an early evaluation of their condition). To shorten their shadow duration, we need to compute the target addresses as early as possible. Unlike on a traditional out-of-order processor, where we want to keep the address computation close to the instruction to reduce register pressure, on the secure out-of-order processor we want to *hoist* and overlap the computations feeding loads and branches as much as it is necessary for all addresses to be ready, to be able to execute them in parallel and ultimately gain MLP.

To this end we reorder the instructions to prioritize target address computation of memory operations and condition evaluation of branch conditions. To keep the problem tractable, we focus on local reordering within basic blocks, as hoisting and lowering beyond basic block boundaries is problematic for three reasons: While the secure out-of-order processor cannot rely on branch prediction to execute loads past unresolved branches it can still execute non-load instructions past branches (safe, as they do not change the cache, and therefore squashing them does not leave traces). As branch prediction is very accurate these safe-to-be-executed instructions will be executed whether or not they are hoisted across the branch. Second, lowering memory operations and their uses to successors would risk to delay execution more than necessary. Ideally we would like to delay loads only as much as needed for the address computation to be ready. Finally, on the compiler side, remaining within the same basic block simplifies the analyses and reduces the overhead introduced by hoisting these instructions.

The idea is to overlap address computation and branch condition evaluation, such that they are ready as soon as possible to allow the hardware to remove the C- and E-Shadows as early as possible. The algorithm consists of two parts, the generation of *buckets* (Algorithm 2) from the original code, followed by the *reordering* of instructions (Algorithm 1).

The idea behind the bucket generation is to find a representation that groups the independent instructions and orders the dependent



(a) Original Code (b) Buckets (c) Reordered Code

Figure 6: The original code and the selected instructions to hoist (address computation for memory operations and branch target, marked with ✗) are shown in Figure (a). The selected instructions and their dependencies are ordered into buckets as in Figure (b). Instructions within a bucket are independent of each other. An instruction of a bucket has at least one dependency on its preceding bucket. The buckets determine the order in which they will be hoisted to the beginning of the basic block. The remaining instructions are kept in their original order. Figure (c) shows the resulting reordering.

instructions, with the goal of finding a legal reordering that overlaps independent instructions and orders the dependent instructions while maintaining the correct dependency. An instruction i in a bucket b_j is dependent on one or more instructions in b_{j-1}. Other dependencies may reside in previous buckets b_{j-2}, .., b_0) too, but there is *at least* one dependency chain from b_0 to b_{j-1} that forces i to reside in b_j. All instructions within one bucket are independent of each other. In the second step, the actual reordering, we select those instructions that contribute to the address computation and branch condition and hoist them according to the ordering specified by the bucket ordering.

Figure 6 shows an example of the bucket creation. The code in Figure 6 (a) is the first basic block of the code in our previous example in Figure 5. To only have one operation per line (which is closer to the code the compiler sees in its intermediate representation), we split some lines into two, and use the `goto` keyword to represent the branch instruction at the end of the basic block. The instructions to hoist (i.e., if they contribute to any memory target address and branch condition computation) are marked with ✗. Figure 6 (b) shows the buckets created for the code, and Figure 6 (c) has the final reordered code. If several instructions are to be hoisted that are within the same bucket, we reorder the non-memory-operations such that they precede the memory operations of the same bucket, such that the address is ready by the time the memory operation is issued (not shown in Figure 6).

Algorithm 1 shows our algorithm to reorder instructions. We go through the basic block and collect all instructions that are of interest for hoisting (Line 2). We then find all the instructions that need to be hoisted along with them, since they are dependencies that are required for correct execution (Line 3). These dependencies are data dependencies, aliasing (may- or must-aliasing) memory operations, or instructions with side-effects that may change memory. Afterwards we apply the bucket creation on the collected instructions (Line 4, detailed below) and hoist them according to their order within the buckets (Line 5). The result is the reordered basic block.

We apply a top down approach for creating the buckets, see Algorithm 2. Starting with the first instruction in the basic block,

```
Input: BasicBlock BB
Output: Reordered BasicBlock
1  begin
2      instsToHoist ← FindInstsToHoist(BB)
3      targetInsts ← FindDepsRecursive(instsToHoist)
4      buckets ← SortInstIntoDepsBuckets(BB, targetInsts)
5      BB_reordered ← HoistInsts(buckets, BB)
6      return BB_reordered
7  end
```

Algorithm 1: Algorithm to identify and reorder the instructions of interest

```
Input: BasicBlock BB, InstsToHoist Hoist
Output: Buckets
1  begin
2      b ← 0
3      instToBucket ← {}
4      foreach inst in BB do
5          if inst ∉ Hoist then
6              continue
7          deps ← GetDeps(inst)
8          depBucketNumber ← GetHighest(deps,
             instToBucket)
9          b ← depBucketNumber + 1
10         instToBucket [inst] ← b
11         buckets [b] ← inst
12     end
13     return buckets
14 end
```

Algorithm 2: A top down approach to create the buckets containing the instructions to hoist and all their dependencies

we first check if it is selected for hoisting (Line 5). If it is, we collect its dependencies, namely its operands, any preceding aliasing stores if we encounter a load, and any preceding aliasing loads and stores if encountering a store (Line 7). For each dependency we look up which bucket it belongs to and record the highest found bucket

number (Line 8). If a dependency does not belong to the basic block in focus we do not consider it. Since we go through the basic block from from top to bottom, all dependencies have already been taken care of in previous iterations, and their bucket number can be looked up using a map (Line 8, Line 10). The bucket number of the current instruction is the highest number of all its dependencies plus one (Line 9). Finally, we add the current instruction to its corresponding bucket (Line 11).

In our example we hoisted instructions that compute addresses for memory operations or conditions for branch instructions. While this is the most intuitive solution for the removal of E- and C-shadows, we also evaluate a version that chooses *all instructions* within the basic block for reordering. The intuition behind this is the following: allowing independent instructions to be issued in-between increases the chance that the required addresses and the branch condition are ready to be consumed as soon as they are needed. In addition, by grouping and reordering all instructions, we also schedule independent loads together, which may further increase MLP. In Section 4 we evaluate both versions and will see that *choosing all instructions* indeed turns out to be better for performance in many cases.

3.3 Discussion on Security Guarantees of Our Approach

Our paper makes use of three main components, *(i)* Delay-on-Miss, *(ii)* non-speculative load-reordering, and *(iii)* early shadow resolution through instruction reordering. In this section we discuss how Delay-on-Miss is effective against speculative side-channel attacks and how our proposal maintains the security guarantees of Delay-on-Miss.

3.3.1 Delay-on-Miss. Speculative loads can have visible side-effects on the memory hierarchy, which can be exploited by attacks such as Spectre to reveal secrets. Delay-on-Miss prevents speculative side-channel attacks by delaying such speculative loads. Under Delay-on-Miss, instructions that may cause a misspeculation are said to *cast a shadow* on all instructions that follow them. When such a shadow is cast by an instruction, it can be lifted only when it is known that no misspeculation can originate from said instruction. Loads that are under such shadows are categorized as speculative and unsafe and, if they request data and the request misses in the cache, are not allowed to proceed (i.e., they are *delayed*) until it is deemed safe to do so (i.e., until they are *unshadowed*). If, however, the request leads to a cache hit, the data is served, and instead only actions that may cause side-effects (such as updating the replacement state) are delayed. These restrictions ensure that there are no visible side-effects in the memory hierarchy that can be exploited by speculative side-channel attacks.

Now that we have established that Delay-on-Miss protects against Spectre and other similar attacks, we show that the components added on top of Delay-on-Miss do not open up new security vulnerabilities.

To begin with, our instruction scheduling technique is conservative and does not reorder instructions speculatively. The scheduling technique selects the set of instructions that contribute to either the address computation of memory operations or the computation of the branch target, and hoists them to the beginning of a basic block (see Section 3.2 for more details). In order to make sure that hoisting does not access data speculatively (which would open up a security hole), we hoist along all preceding *may- and must-aliasing* operations, as well as other operations that may have side effects (such as function calls) when encountering memory operations.

Figure 7 shows a reordering example, where the set of instructions to hoist includes a memory operation. Figure 7(a) shows the original code and the instructions that we initially select for hoisting. Note that one of the instructions that are selected performs a load from memory ($p_1 \rightarrow a[addr_1]$), which follows a store to memory ($p_1 \rightarrow a[addr_1] = x$). Figure 7(b) shows the bucket creation for the instructions to hoist to the beginning of the basic block. In this case, the two memory operations may or must alias; and since we want to be conservative, we include the store operation when hoisting and respect the potential dependency (the load operation has to follow the store operation). Figure 7(b) depicts the case, where at compile-time we know that these two operations are independent of each other. In that case, the load operation may be scheduled earlier than the store operation in focus (as it does not access stale data), and the store operation is therefore not included in the bucket creation.

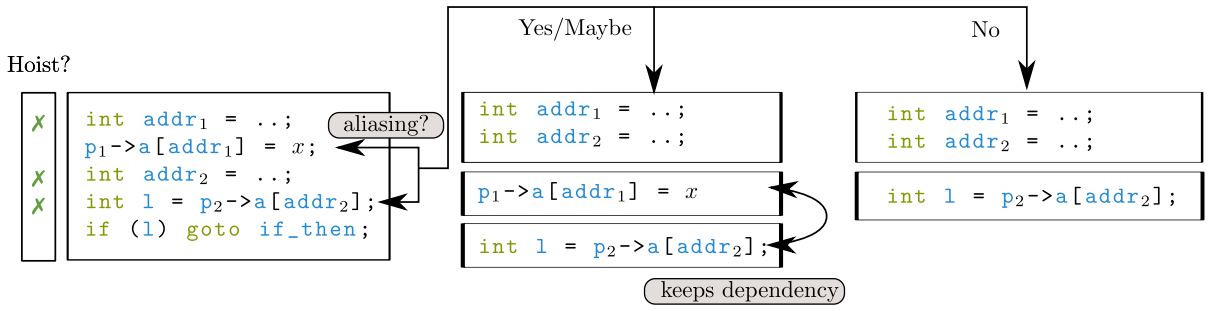
The last component of our approach is the non-speculative load-load reordering, which does contain mechanisms than can cause observable timing differences in the system. Specifically, it makes use of *lockdowns* when a younger load is performed to *delay acknowledging* incoming invalidations. When a cache line is in lockdown, writers to that cache line are delayed, and this delay can potentially be observed by the writers and used as a side-channel. In our case, we do not introduce a new speculative side-channel, because of the following:

While under a shadow other than an M-shadow, the rules of Delay-on-Miss apply and no speculative loads are allowed to make any visible changes to the memory hierarchy. This includes loads that would need to get non-cacheable data or go into lockdown. As the loads are covered by other overlapping shadows, removing the M-shadow at this stage would not help in regaining MLP anyway. Instead, a load is allowed to go into lockdown only after all other shadows have been resolved and the load is shadowed by nothing other than an M-shadow. At this stage, the M-shadow can be safely removed, as the load reordering is now non-speculative and it will not be squashed by an invalidation.

In essence, Delay-on-Miss itself prevents the possible speculative side-channel that would have been introduced by the non-speculative load-load mechanism. At the same time, non-speculative load-load reordering can also be used as a non-speculative side-channel, when the attacker and the victim share physical memory. Under such conditions, simpler, pre-existing, related side-channel attacks, such as Invalidate+Transfer [14], can be exploited. Solutions for such non-speculative attacks already exist and can be applied for the lockdown side-channel, but they fall outside the scope of this work.

4 EVALUATION

We implement our ideas on top of the Delay-on-Miss proposal [30]. Next sections highlight our experimental set-up (Section 4.1) and the performance results (Section 4.2).



(a) Original Code (b) Buckets if: may- or must-aliasing (b) Buckets if: no-aliasing

Figure 7: Non-speculative reordering of instructions. Figure (a) shows the original code. Initially all the instructions that contribute to address computation for either memory operations or branch target computation are chosen for hoisting (marked with ✗). Figure (b) shows what buckets are created if the write to $p_1 \rightarrow a[addr_1]$ and loading from the value $p_2 \rightarrow a[addr_2]$ may (or must) alias, i.e. loading the data before writing may lead to retrieving stale data (and would thus leak secrets). Note that apart from the selected instructions for hoisting, the store operation is also included in the bucket creation, as we need to make sure that we do not load stale data if $p_1 \rightarrow a[addr_1]$ and $p_2 \rightarrow a[addr_2]$ were to alias. However, if we know at compile time that these memory operations do not alias, the write operation does not need to be included in the set of instructions to hoist, see Figure (c).

Table 3: Simulation parameters used for Gem5

Parameter	Value
Technology node	22 nm
Processor type	Out-of-order x86 CPU
Processor frequency	3.4 GHz
Address size	64 bits
Issue width	8
Cache line size	64 bytes
L1 private cache size	32 KiB, 2-way
L1 access latency	2 cycles
L2 shared cache size	512 KiB, 8-way
L2 access latency	20 cycles

4.1 Experimental Set-up

The compiler analysis and transformation is implemented on top of LLVM (Version 8.0) [19]. We use Gem5 [4] with the Delay-on-Miss implementation from Sakalis et al. [30] as our simulator. Table 3 shows the configuration chosen for simulation (i.e. a large out-of-order processor, the same set up as for the Delay-on-Miss work). The baseline is always compiled with the highest possible optimization (-O3). For evaluation we have chosen the SPEC CPU 2006 [12] benchmark suite. We focus on the C and C++ workloads which we were able to compile and run out-of-the-box using both LLVM and Gem5.

Since our evaluation is based on simulation, we need to identify relevant phases of the benchmark that can be simulated. On top of this, we also need to make sure that each region is well-defined, such that different simulation runs using different binaries can be compared.

We compare the different binaries by focusing on the comparison of statistics on *hot regions* that are identified using profiling. Table 4 lists the selected regions for each benchmark. For each region we state *(i)* how many dynamic instructions it responds to in Gem5 (on average), *(ii)* the percentage of runtime all executions of that region would be attributed to relative to the whole program run, and *(iii)* the total percentage when considering all regions of a benchmark. The regions do not add up to 100% and there are several reasons for this: the main loop may be recursive (thus too large to cover as a whole within one simulation run), or the code may have a lot of very small regions whose contribution to the overall execution time is negligible. For (Gem5) practicality reasons we also do not capture regions that start beyond three billion instructions.

The performance numbers in our work do not match (and cannot be compared to) the performance numbers that were presented for Delay-on-Miss [30]. Sakalis et al. have a different selection of benchmarks, and second, we focus our evaluation on hot regions to be able to compare our versions fairly, and therefore the simulated regions do not match.

Evaluated Versions: Our baseline is the Delay-on-Miss running on a large, unmodified out-of-order processor. Our extensions are implemented on top of it. Table 5 shows the evaluated versions and their respective names that will be used in the following.

4.2 Performance

As we compare different binaries, we use the *total number of cycles* as a metric for performance (IPC is not a good fit because the number of instruction varies for each binary). It reflects the number of cycles that were required to finish the same amount of work, i.e. the regions that we identified in Table 4.

Figure 8 shows the number of cycles normalized to DoM to show the improvement relative to DoM. In the following we will mainly focus on comparing our extensions with DoM, however, we will give some insight on the performance differences of our work to the unsafe out-of-order in Section 4.3.

Table 4: Benchmarks and the selected regions of interest (ROI). For each region, we list the average number of micro dynamic instructions of *one region run*, and the total percentage of program runtime each region was contributed to (if we were to run the whole program from start to end).

Benchmark	Region of Interest	Average Number of Instructions	% of Runtime	Total %
401.bzip2	BZ2_blockSort BZ2_decompress	118,562,460 1,711,253	54.7% 15%	69.7%
429.mcf	primal_net_simplex	87,334	78.6%	78.6%
433.milc	path_product u_shift_fermion compute_gen_staple	133,629,807 24,156,916 279,922,620	26.4% 26.9% 21.2%	74.5%
444.namd	doWork	3,310,837	64.6%	64.4%
450.soplex	leave	1,289,630	31.7%	31.7%
456.hmmer	P7Viterbi	6,314,862	95.1%	95.1%
458.sjeng	gen std_eval	1841 3182	18.5% 32.0%	50.5%
462.libquantum	quantum_sigma_x quantum_toffoli	14,680,142 23,540,617	18% 60.2%	78.2%
464.h264ref	encode_one_macroblock	2,199,007	98.6%	98.6%
470.lbm	LBM_performStreamCollide	393,922,845	97%	97%
471.omnetpp	do_one_event	1810	92.3%	92.3%
473.astar	regwayobj::makebound2 wayobj::fill way2obj::fill	3062 75,885,242 607,874,212	18.6% 48.1% 28.2%	94.9%

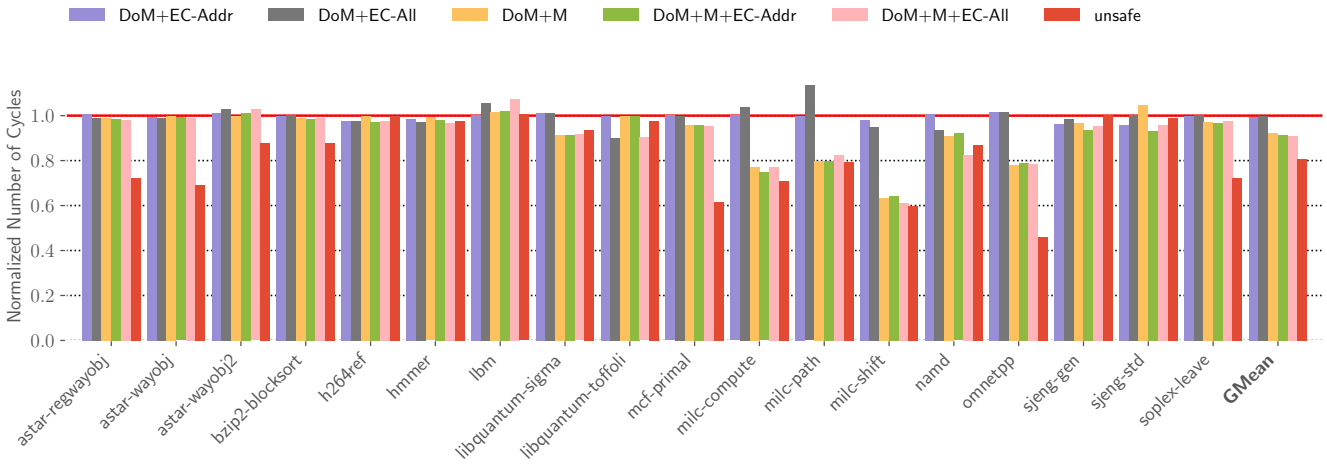


Figure 8: Normalized number of cycles for Delay-on-Miss with our extensions (DoM+M, DoM+EC-All, DoM+EC-Addr, and their combinations DoM+M+EC-Addr, DoM+M+EC-All), and the unsafe out-of-order (unsafe). Baseline is Delay-on-Miss as in Sakalis et al. (see red line).

The Effect of Removing M-shadows on Performance. Figure 8 shows that by only introducing the coherence protocol on top of Delay-on-Miss we can significantly improve performance (see DoM+M). By allowing loads to be reordered, DoM+M achieves to improve DoM by 7% (on average). M-shadows enforce an ordering on loads which restricts the parallel execution of loads. Using the coherence protocol we can completely disable the M-shadows, which allows the processor to execute loads (if not still shadowed by another instruction) in parallel and therefore overlap their delays. This allows for better resource usage and helps to hide the long latencies that memory accesses introduce. While some benchmarks benefit a lot from removing the M-shadows (such as milc, 37% (-shift), 23% (-compute), 20% (-path), and omnetpp, 22%), others are not affected at all (such as lbm, astar, and h264ref). There are several aspects that play a role in deciding whether or not the removal of M-shadows will have a positive effect on performance.

Benchmarks that benefit from the removal of M-shadows are likely to exhibit *many cache misses* that can be overlapped, to efficiently use the hardware resources and thus gain in performance. On top of this, it is beneficial if there is little control flow (few C-shadows) and little address computation that is required (short

Table 5: Evaluated Versions. All versions are based on Delay-on-Miss (DoM). For the E- and C-shadows we evaluate two versions: one that reorders all the instructions (*All*), and one that only reorders the memory and branch target address computation (*Addr*), see Section 3.2 for more details. If a cell is marked (✗), it is enabled for the version of that row.

M-shadows	E- and C-shadows		Version Name
	All	Addr	
			DoM
✗			DoM+M
	✗		DoM+EC-All
		✗	DoM+EC-Addr
✗	✗		DoM+M+EC-All
✗		✗	DoM+M+EC-Addr

E-shadows). All milc regions fall into this category. Milc is categorized as a memory-bound benchmark [15]. Looking at the hot regions, milc makes use of matrix operations that include a number of independent load operations that access memory using simple, constant indices. Since the address computation is quick to finish, many of the E-shadows are likely to be very short. On top of this, milc has only little control flow, and thus not many overlapping C-shadows that would otherwise block loads from executing. This combination of characteristics makes milc a good fit for DoM+M.

On the other side, benchmarks that have loads that are dependent on each other (i.e. indirection chains, such as $x[y[z]]$), cannot be exploited for increasing MLP as their accesses have to be serialized. Such a dependence chain may also happen if a long latency load feeds the branch condition, since any (missing) load after the branch cannot be executed until the branch target is known (C-shadows). One memory-bound benchmark [15] that cannot profit from the M-shadow removal is astar. Astar's hot regions include tight basic blocks with nested branches and with interleaved loads and stores. Removing just the M-shadows is therefore not enough to achieve higher levels of MLP.

The Effect of Removing E- and C-shadows on Performance. DoM+EC-Addr and DoM+EC-All explore the effect of our instruction reordering technique on top of DoM. Since the M-shadows are not lifted for these two versions, all loads are still serialized and no MLP can be exploited. While the reordering alone does improve performance for a few benchmarks (e.g., 4% improvement on sjeng-std with DoM+EC-Addr, and 4% improvement with DoM+EC-All on bzip2-decompress), they also introduce overhead for others and cancel out the benefit (e.g., 13% decrease for DoM+EC-All on milc-path). On average, both versions do not benefit on their own, since they are designed to increase the degree of MLP, given that MLP can be exploited (which it cannot, if M-shadows are in place).

Although the reordering is intended to be combined with the coherence protocol, there are some cases in which reordering has a positive impact on the performance. Our reordering changes the original code in two ways. First, we (try) to start all independent chains as early as possible, and second, we schedule independent instructions of different chains back-to-back. Shadows handicap the out-of-order processor in its out-of-orderness and it can no longer freely choose the instructions to execute. As a result, it relies more on the schedule determined by the compiler than its unsafe baseline, similar to smaller processors that do not have the ability to look far ahead into the code. By splitting the dependencies and scheduling independent instructions in-between, dependencies are more likely to already be resolved as soon as they are considered for execution.

Reordering all instructions comes however at a risk. DoM+EC-All may introduce an overhead by *keeping many live values* around that may impact performance negatively. This can be the case if a basic block is large and contains many independent instructions that can be overlapped. This is what happens for lbm: lbm's hot region contains a for loop with a big basic block with many independent instructions that are grouped into a bucket and are scheduled together. Naturally, this leads to increased register pressure: the assembly file for DoM counts 26 spills, the one for DoM+EC-All 58 spills. As a result, there is an increased number of instructions required for spilling and reloading (apart from increasing the number of instructions, this also leads to an increased number of shadows). Figure 9 plots the total number of committed instructions normalized to DoM for each benchmark. For most benchmarks the number of instructions is roughly the same, but lbm shows a significant increase in instructions for the two versions DoM+EC-All and DoM+M+EC-All. This increase finally reflects in the decreased performance for lbm (5% performance degradation for DoM+EC-All, and 7% for DoM+M+EC-All respectively).

Putting everything together: The Effect of Removing M-, E- and C-shadows on Performance. DoM+M+EC-Addr and DoM+M+EC-All combine software reordering to tackle E- and C-shadows with load reordering to eliminate M-shadows, and improve DoM on average by 8% and 9% respectively. Most benefit comes from eliminating M-shadows, combining the load reordering with software reordering improves performance for a few single benchmarks (highest are libquantum-toffoli with 10% and namd with 18%). Where does the benefit from software reordering come from? The benefits are achieved when reordering all instructions within the block (i.e. when using DoM+M+EC-All). As mentioned previously, the approach to group independent instructions and schedule them as early as possible may allow enough delay between the branch- and memory operation-feeding instructions to finish just in time. This would make it unnecessary to cast any shadows in the first place, or to at least shorten the duration in which the operation is casting a shadow. On top of that, we may even further increase MLP by grouping independent loads and scheduling them together. Looking at the hot regions within namd, we can identify basic blocks that have many groups of independent loads (that were not grouped before), which is potentially the reason for the performance improvement. However, for the majority of benchmarks the reordering does not help much. The reason is that we are limiting our reordering to the bounds of a basic block. Many times basic blocks only consist of few instructions, or instructions that cannot be moved due to existing dependencies within the block. In these cases, our reordering cannot properly address the early removal of C- and E-shadows and we completely rely on the M-shadow removal.

While DoM+M+EC-Addr was the intuitive solution to eliminate E- and C-shadows and to increase MLP, we find that DoM+M+EC-All performs better overall. The drawbacks of DoM+M+EC-All are

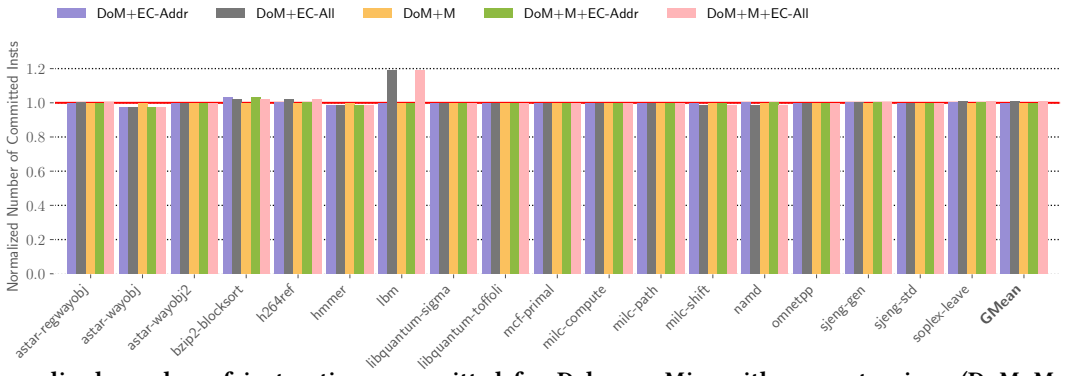


Figure 9: Normalized number of instructions committed for Delay-on-Miss with our extensions (DoM+M, DoM+EC-All, DoM+EC-Addr, and their combinations DoM+M+EC-Addr, DoM+M+EC-All). All numbers are normalized to the unmodified Delay-on-Miss (DoM).

basically the same as for DoM+EC-All, as they both make use of the *exact same binary*, but with a different coherence protocol. As such, DoM+M+EC-All suffers from increased register pressure if too many independent chains of instructions exist, that will all be scheduled right from the start.

Overall, the best version (if one were to select one) is DoM+M+EC-All, which combines load reordering with instruction scheduling targeting all instructions to exploit MLP. On average, it improves DoM by 9%. The unmodified out-of-order processor is better than DoM by 19%, thus, our techniques close the performance gap between DoM and the unsafe out-of-order by 53%.

4.3 More Data to Understand the Performance Benefit

In the previous sections we discussed how our techniques to remove M-, E-, and C-shadows can be beneficial for MLP and thus for performance. In this section we want to show more data to support our previous numbers, and to better understand where the benefit comes from.

Figure 10 plots the average shadow duration measured in cycles for all versions, with DoM being the baseline. The graph shows clearly that DoM+M, DoM+M+EC-Addr, and DoM+M+EC-All reduce the overall duration over DoM (32 cycles for dom, 14 for DoM+M, 13 for DoM+M+EC-Addr, and 12 for DoM+M+EC-All, on average). With shorter shadows, more instructions can be issued at a time (including loads): Figure 11 shows the average number of instructions that are issued per cycle, for the unsafe, unmodified out-of-order (red) and all evaluated versions (colors consistent with previous plots). A high average number of issued instructions per cycle translates to higher performance, as can be seen comparing Figure 11 and Figure 8. On average, DoM issues 0.98 instructions per cycle. For DoM+M this number is 1.06, for DoM+M+EC-Addr 1.07, and for DoM+M+EC-All 1.09.

Interestingly, for a few benchmarks, such as libquantum-toffoli and namd, DoM+EC-All and DoM+M+EC-All issue more instructions per cycle than the baseline (unsafe out-of-order processor), and require fewer cycles to finish the regions of interest (see Figure 8). One reason may be a fortunate combination of remaining shadows preventing misspeculation penalties and reordering. Shadows prevent the out-of-order processor from speculating, and therefore also from *misspeculating*. Libquantum-toffoli is known to be a memory bound benchmark [15], such that the unsafe processor often needs to speculate past loads. As misspeculation imposes significant overhead if it happens, preventing it may be the better choice. In combination with reordering under these shadows, the processor may (instead of wrongly speculating and squashing) execute useful instructions that are known to be safe.

5 RELATED WORK

Side-channel Attacks. This work focuses on speculative side-channel attacks, which were first introduced in the early 2018 with the announcement of Meltdown [21] and Spectre [17]. Since these two original attacks, numerous variants that exploit different parts of the system have been introduced (e.g. [3, 5, 7, 18, 31, 33, 38]), but they all share the same two parts: Misdirecting execution to speculatively bypass software and/or hardware checks to gain access to secret data and then leaking that data thought a side-channel. The security solutions that we are targeting with this work, such as Delay-on-Miss [30] or InvisiSpec [39] (the future-proof version), are not concerned with how the execution is misdirected, instead they focus on preventing the leakage of information through side-channels during speculative execution. Because of this, these security mechanisms are agnostic to the specifics of the attack and instead try to prevent speculative state from being produced and/or leaked. The solutions we propose are not specific to certain attacks, and instead consider information leakage from speculative execution as a general problem.

Software Mitigations. Software mitigations include speculation barriers and conditional select/move instructions [2, 13]. Barriers prevent speculation altogether and impose a significant restriction on performance. While the compiler may analyze code at risk, static analysis identifying vulnerable code is not complete. Retpotline [10] ("return trampoline") prevents speculation on indirect branches by trapping speculative execution in an infinite loop, by replacing the indirect jump by a call/return combination. Execution only exits the loop as soon as the branch target is known. Attacks targeting conditional branches may be circumvented by introducing a poison

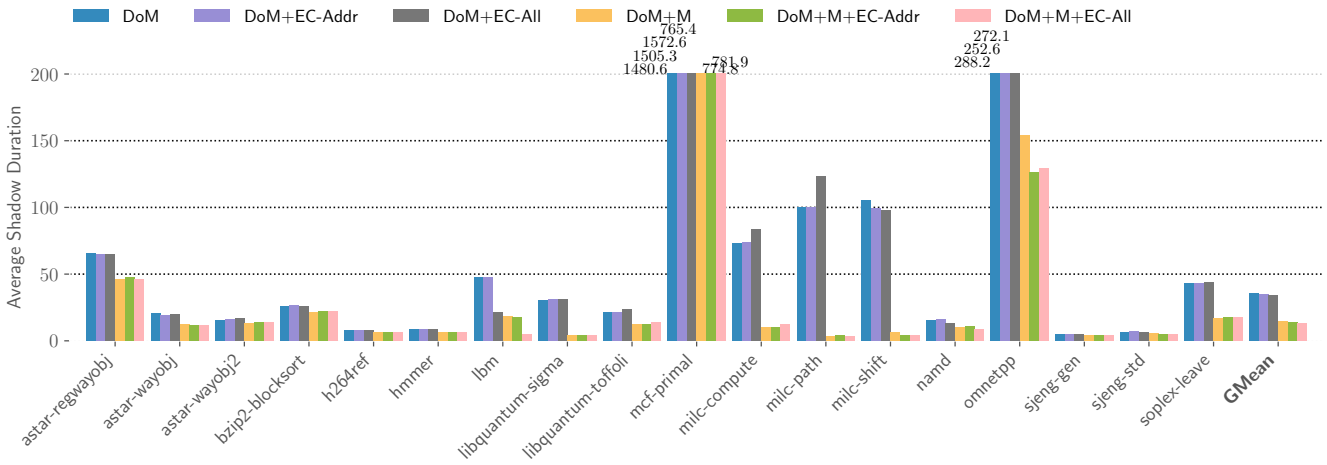


Figure 10: Average shadow duration in cycles for our extensions (DoM+M, DoM+EC-Addr, DoM+EC-All, DoM+M+EC-Addr, and DoM+M+EC-All), with DoM being the baseline

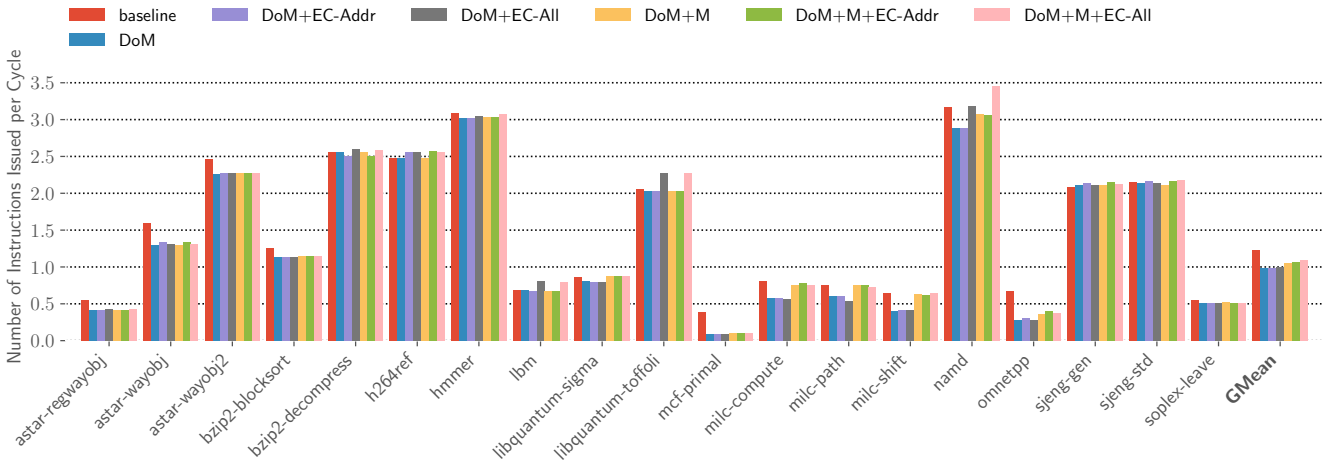


Figure 11: Average number of instructions issued per cycle for the unmodified out-of-order processor (baseline), Delay-on-Miss (DoM) and Delay-on-Miss with our extensions (DoM+M,DoM+EC-All, DoM+EC-Addr, and their combinations DoM+M+EC-Addr, DoM+M+EC-All)

value that poisons loaded values on misspeculated paths [24]. A similar approach is taken by LLVM's speculative load hardening [22], which zeroes out pointers before loading them, if they are mispredicted. KAISER [11] protects against Meltdown by enforcing strict user and kernel space isolation but is ineffective against Spectre. Other software-based mitigations [8, 20, 32] propose annotation-based mechanisms for protecting secret data, as an effort to reduce the overhead, but require additional hardware, compiler, and OS support.

Invisible Speculation in Hardware. There are three main approaches when it comes to preventing speculative execution from causing measurable side-effects in the system:

(1) **Hiding the side-effects of speculative execution until speculation is resolved.** This approach is taken by solutions such as SafeSpec [16], InvisiSpec [39], and Ghost Loads [29], and MuonTrap [1]. They hide the side-effects of transient instructions in specially designed buffers that keep them hidden until the speculation is resolved and the side-effects can be made visible. Since these approaches have to wait before they can make the side-effects visible, they incur a performance cost relative to how long the side-effects need to be hidden. Our work can help all of these solutions by reducing and sometimes eliminating the delay before performing an instruction and when its side-effects can be made visible to the system.

(2) **Delaying speculative execution until speculation can be resolved.** Solutions such as Delay-on-Miss [30], Conditional Speculation [20], SpectreGuard [8], NDA [37], and Speculative Taint Tracking (STT) [40, 41] selectively delay instructions when they might be used to leak information. Some, such as Conditional Speculation and SpectreGuard, only try to protect data marked by the user as sensitive, while

others, such as Delay-on-Miss, work on all data. NDA and STT focus on preventing the propagation of unsafe values at their source, based on the observation that a successful speculative side-channel attack consists of two dependent parts, *(i)* an illegal access (i.e., a speculative load) and *(ii)* one or more instructions that dependent to the illegal access and leak the secret. Instead of waking up instructions in the instruction queue as soon as their operands are ready, NDA wakes up instructions as soon as they are safe. This way, NDA prevents secrets from propagating. Similarly, STT taints access instructions (instructions that may access secrets, i.e., loads) and untaints them as soon as they are considered safe (i.e. if all their operands are untainted). While the execution of load instructions is allowed, the execution of their dependents is delayed. In comparison to Delay-on-Miss, NDA and STT therefore only delay the transmit instructions.

The common theme in all of them is that some speculative instructions are considered unsafe under specific conditions and need to be delayed until the speculation has been resolved. Our work can help to reduce the performance overhead caused by delays by reducing and sometimes completely eliminating the duration under which instructions are speculative.

(3) **Undoing the side-effects of speculative execution in the case of a misspeculation.** CleanupSpec [28] takes a different approach to the previous solutions by permitting speculative execution to proceed unhindered and undoing any side-effects in the event of a misspeculation. The main cost comes from having to undo the side-effects after a misspeculation. Our work focuses on detecting correct speculation early, so it would not benefit CleanupSpec significantly. Instead, a similar solution would have to focus on detecting misspeculation early, to reduce the undoing cost. However, such a solution is outside the scope of this work and is left as future work.

Other Designs. Other approaches hoist branch conditions to avoid branch prediction (and thus the necessity of C-shadows) to separate loops [34]. Usually the splitting of condition and branch happen not within the basic block, but spans a bigger code range, since they aim at reordering of conditions that are originally not within the processor's view at a point, the instruction window.

Similar to non-speculative load-load reordering, which modifies the coherence protocol to let reordered loads appear serialized and thus avoid expensive squashes, OmniOrder [26] achieves efficient execution of atomic blocks in a directory-based coherence environment by letting the atomic blocks appear serialized. The main idea behind it is to keep speculative updates in a per-processor buffer, and to leave the basic coherence protocol unmodified. The history of non-speculative updates and their origin is moved along with each coherence transaction, and the receiving processor becomes responsible for merging or squashing the speculative data whenever a transaction is committed or squashed.

6 CONCLUSION

With the discovery of speculative side-channel attacks, speculative execution is no longer considered to be safe. To mitigate the new vulnerability many hardware solutions choose to either delay or hide speculative accesses to memory until they are considered as safe. While sensitive data is safe from being leaked, this approach trades performance for security.

In this work, we take a look at hardware defenses that focus on restricting the execution of loads and their dependents and only reveal their side-effects as soon as they are deemed as *safe*. We analyze the conditions that need to be met for an unsafe load to become safe, and observe that through instruction reordering we can actually *influence and shorten* the period of time, in which a load is considered to be unsafe to execute. In combination with a coherence protocol that enables safe load reordering even under consistency models that require memory ordering, we unlock the potential for memory-level-parallelism and thus for performance. We introduce and evaluate our extension on top of a state-of-the-art hardware defense mechanism, and show that we can improve its performance by 9% on average, and thus reduce the overall performance gap to the unsafe out-of-order processor by 53% (on average).

ACKNOWLEDGMENTS

This work was partially funded by Vetenskapsrådet project 2015-05159, 2016-05086, and 2018-05254, by the European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC) (grant No 801051) and by the European Research Council (ERC) under the European UnionâĂŹs Horizon 2020 research and innovation programme (grant agreement No 819134). The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project SNIC 2019-3-227.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. https://doi.org/10.1109/ISCA45697.2020.00022

[2] ARM. [n.d.]. Cache Speculation Side-channels. ([n. d.]). Online https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability; accessed 27-October-2019.

[3] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. *arXiv:1903.01843 [cs]* (March 2019). http://arxiv.org/abs/1903.01843 arXiv: 1903.01843.

[4] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. *arXiv:1802.09085 [cs]* (Feb. 2018). http://arxiv.org/abs/1802.09085 arXiv: 1802.09085.

[8] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Proceedings of the*

56th Annual Design Automation Conference 2019 on - DAC '19. ACM Press, Las Vegas, NV, USA, 1–6. https://doi.org/10.1145/3316781.3317914

[9] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (April 2018), 1–27. https://doi.org/10.1007/s13389-016-0141-6

[10] Google. [n.d.]. Retpoline: a software construct for preventing branch-target-injection. ([n. d.]). Online https://support.google.com/faqs/answer/7625886; accessed 27-October-2019.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS (Lecture Notes in Computer Science, Vol. 10379)*. Springer, 161–176.

[12] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *Computer Architecture News* 34, 4 (2006), 1–17.

[13] Intel. [n.d.]. Intel Analysis of Speculative Execution Side Channels. ([n. d.]). Online https://www.intel.com/content/www/us/en/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html; accessed 27-October-2019.

[14] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *AsiaCCS*. ACM, 353–364.

[15] Aamer Jaleel. 2010. Memory Characterization of Workloads Using Instrumentation-Driven Simulation. (2010). Online; accessed 06-January-2020. Web Copy: http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[16] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[17] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. 19–37. https://doi.org/10.1109/SP.2019.00002

[18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. https://www.usenix.org/conference/woot18/presentation/koruyeh

[19] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.

[20] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Washington, DC, USA, 264–276. https://doi.org/10.1109/HPCA.2019.00043

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[22] LLVM. [n.d.]. Speculative Load Hardening. ([n. d.]). Online https://llvm.org/docs/SpeculativeLoadHardening.html; accessed 16-January-2020.

[23] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* 2, 1 (March 2018), 33–50. https://doi.org/10.1007/s41635-017-0025-y

[24] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR* abs/1902.05178 (2019).

[25] Aashish Phansalkar, Ajay Joshi, and Lizy Kurian John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ISCA*. ACM, 412–423.

[26] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. 2014. OmniOrder: Directory-based conflict serialization of transactions. In *ISCA*. IEEE Computer Society, 421–432.

[27] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *ISCA*. ACM, 187–200.

[28] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 73–86. https://doi.org/10.1145/3352460.3358314

[29] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Själander Magnus. 2019. Ghost Loads: What is the Cost of Invisible Speculation? 153–163. https://doi.org/10.1145/3310273.3321558

[30] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA*. ACM, 723–735.

[31] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. [n.d.]. ZombieLoad: Cross-Privilege-Boundary Data Sampling. ([n. d.]), 15.

[32] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. 2019. ConTExT: Leakage-Free Transient Execution. *arXiv:1905.09100 [cs]* (May 2019). http://arxiv.org/abs/1905.09100 arXiv: 1905.09100.

[33] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read Arbitrary Memory over Network. (July 2018). https://arxiv.org/abs/1807.10535

[34] Rami Sheikh, James Tuck, and Eric Rotenberg. 2015. Control-Flow Decoupling: An Approach for Timely, Non-Speculative Branching. *IEEE Trans. Computers* 64, 8 (2015), 2182–2203.

[35] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 318–331. https://doi.org/10.1145/3307650.3322228

[36] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*. ACM, 572–586.

[37] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 572–586. https://doi.org/10.1145/3352460.3358306 event-place: Columbus, OH, USA.

[38] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. (Aug. 2018). https://lirias.kuleuven.be/2089352

[39] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*. IEEE Computer Society, 428–441.

[40] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. https://doi.org/10.1109/ISCA45697.2020.00064

[41] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274 event-place: Columbus, OH, USA.