Ghost Loads: What is the Cost of Invisible Speculation?

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

Mehdi Alipour
Uppsala University
Uppsala, Sweden
mehdi.alipour@it.uu.se

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

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

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

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

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.

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3321558

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

ACM Reference Format:
Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost Loads: What is the Cost of Invisible Speculation?. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3310273.3321558

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) [5], the shared LLC cache (when the attacker can share the LLC) [40], and the coherence protocol (when the attacker can simply be collocated in the same system, under a single coherence domain, with the target) [14]. 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 [19] and Meltdown [24].

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 [35–37].
(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 [35]), 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 [19], 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 [16, 39], 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 [39]. 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 [39], 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%.

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.

Loads: 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[1]. 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 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 whether a load is no longer covered by a shadow it is enough to *i)* mark the load at the time it is dispatched with the current youngest shadow casting instruction and *ii)* compare the load's shadow casting instruction with the head of the SB. If the load's shadow casting instructions is older than the head of the SB, then the load is not under a speculative shadow. This simple mechanism assures that there does not exist any shadow casting instruction older than the load.

[1]Not to be confused with InvisiSpec's Speculation Buffer.

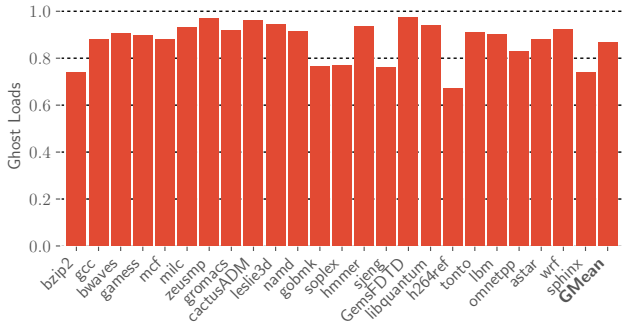

Figure 1: The ratio of loads executed speculatively.

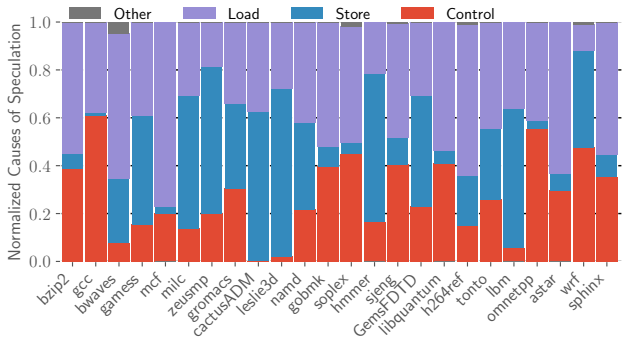

Figure 2: A breakdown of the instructions casting speculative shadows on executed loads.

Figure 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 6.

Figure 2 presents a breakdown of the type of instructions casting shadows over executed load instructions in the applications found in the SPEC2006 [1] benchmark suite. The hardware parameters of the evaluated system can be found in Table 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. [2] 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.

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 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. [32] have already developed a side-channel that exploits the timing of the DRAM system instead of the cache hierarchy.

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 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 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.

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.

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 by 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 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" [16]. 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.

4.2.1 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 [10, 11, 15, 18, 20, 21, 26, 27, 30, 36, 37] 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 multithreading (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. [39]. 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.

4.2.2 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 misspeculated, 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.

5 INVISISPEC

InvisiSpec [39], 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

Table 1: The simulation parameters used for the evaluation.

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

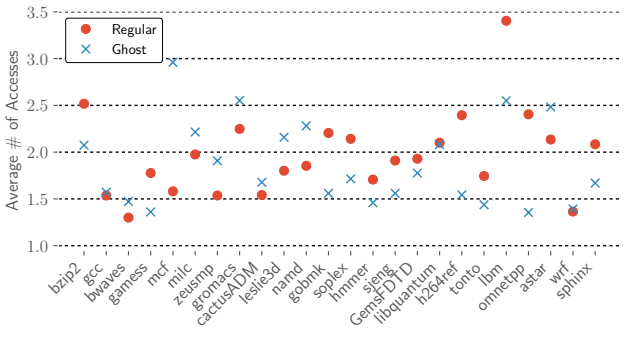

Figure 3: The average number of consecutive, exclusively regular or Ghost accesses to a cache line in the L1. For each cache line we maintain a counter that is incremented every time consecutive accesses of the same type occur and is reset for every access of the opposite type. The deviation, though significant, is omitted for clarity.

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 6) how these differences affect the performance of the two approaches.

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.

6.1 Methodology

We evaluate the different solutions and the suggested improvements using the SPEC2006 benchmark suite [1], from which we exclude five applications due to simulation issues encountered in the baseline simulation. We use the Gem5 [4] simulator combined with McPAT [22] and Cacti [23] 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 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 [39].

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 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.

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 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 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

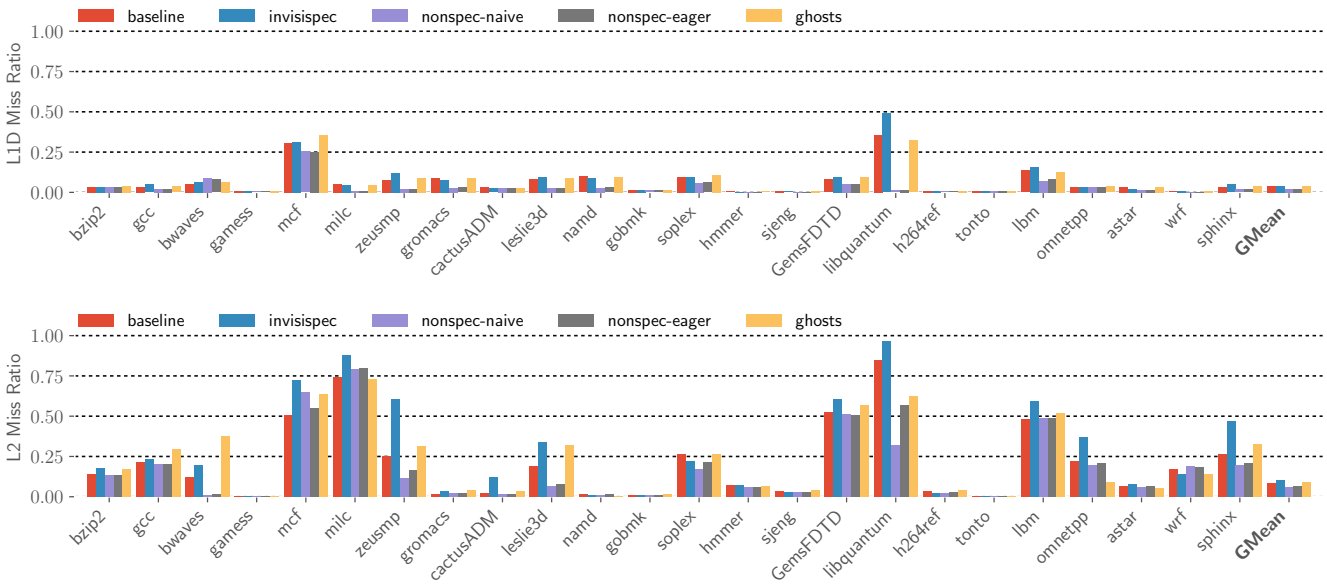

Figure 4: L1-data & L2 cache miss ratios.

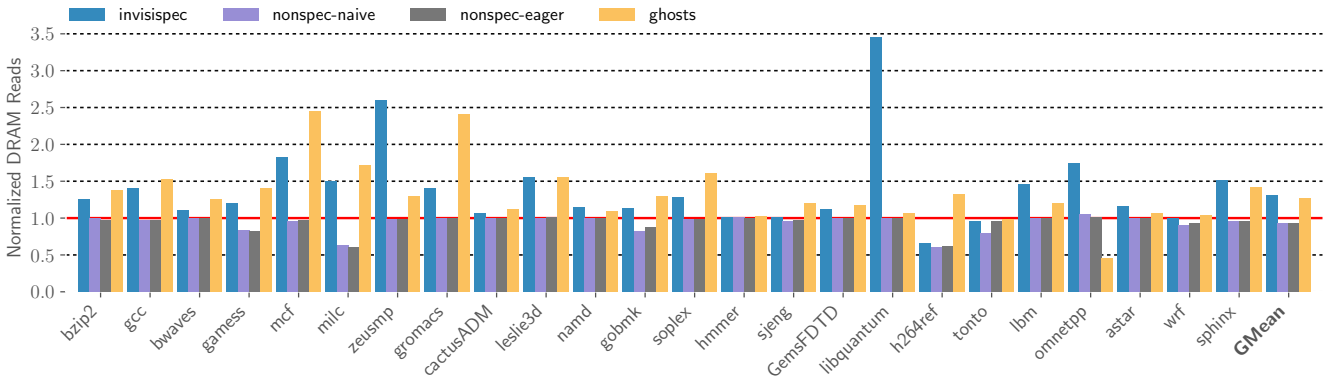

Figure 5: Normalized DRAM reads. The number of DRAM writes is not affected by the different solutions.

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 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 NonSpec solutions do not have such negative side-effects.

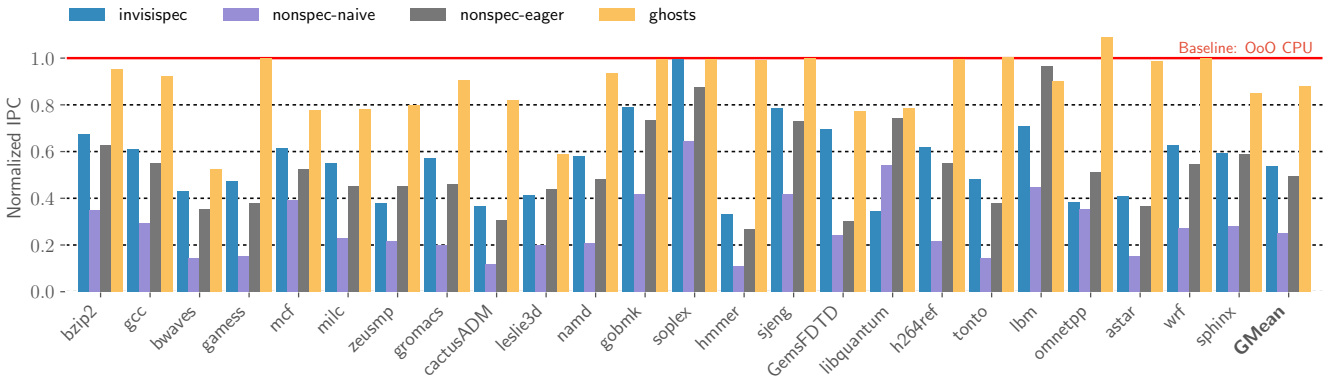

Figure 6: Normalized performance (IPC).

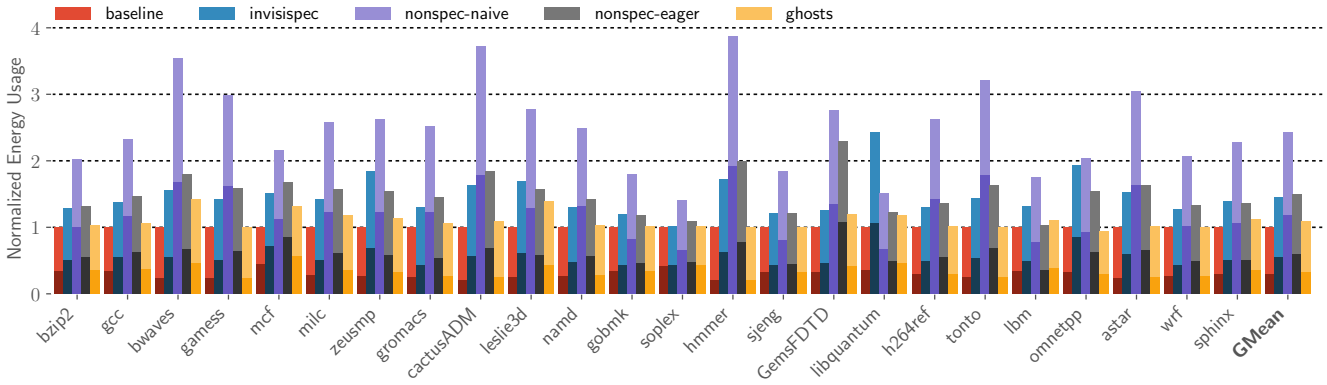

Figure 7: Normalized energy usage. The bottom (shaded) part represents the static (leakage) energy of the system.

6.4 Performance

Figure 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). 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 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 3 that regular accesses and Ghosts are tightly interleaved, not all of the available MLP in the application can be taken advantage of.

6.5 Energy Efficiency

Figure 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× 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 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.

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 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 (*ghosts-noghb*), and finally one without Ghost prefetching enabled (*ghosts-nopref*).

With the Ghost version that uses neither the GhB nor Mtz (*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 `bwaves`, a benchmark that is already sensitive to the other solutions, reaching a performance loss of 93%. Introducing the GhB (*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 – *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 (*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 (*ghosts*). This demonstrates the importance of considering prefetching when proposing such solutions, something that is overlooked by InvisiSpec (and SafeSpec).

7 RELATED WORK

This work was inspired by the Meltdown [24] and Spectre [19] 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 [6] and for compilers [31, 34]. 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 [3, 12–14, 29, 35, 40]. 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 [7, 9–11, 15, 17, 18, 21, 25–27, 30, 33, 36, 37, 41], 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

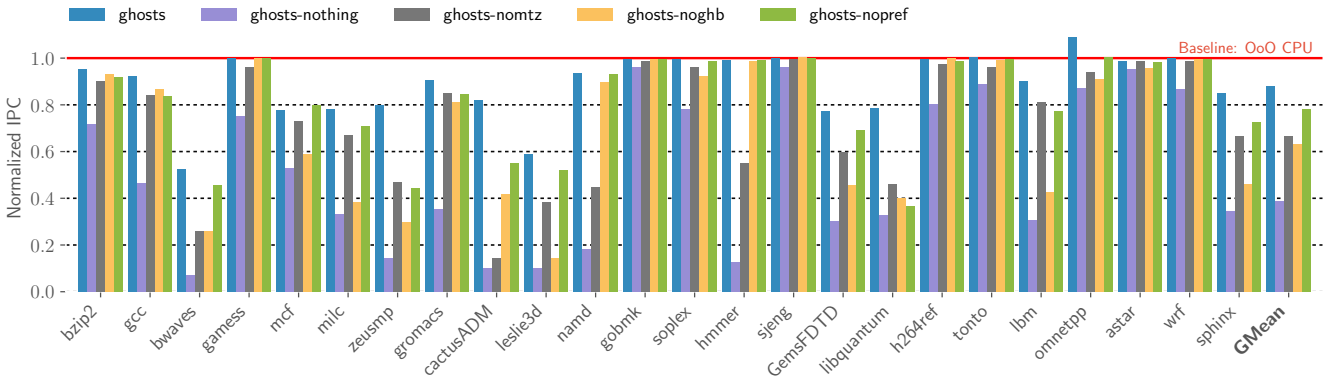

Figure 8: The contribution of each Ghost mechanism to performance (IPC).

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. [39], Khasawneh et al. [16] 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 [8, 28, 32, 35, 38, 42]. These focus on different areas from our proposal and should be considered as complementary solutions.

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.

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 is 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.

REFERENCES

[1] 2006. SPEC CPU Benchmark Suite. http://www.specbench.org/osg/cpu2006/.

[2] M. Alipour, T. E. Carlson, and S. Kaxiras. 2017. Exploring the Performance Limits of Out-of-order Commit. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/3075564.3075581

[3] D. J. Bernstein. 2005. Cache-timing attacks on AES. (2005).

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7. Issue 2. https://doi.org/10.1145/2024716.2024718

[5] J. Bonneau and I. Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems*. Springer Berlin Heidelberg, 201–215.

[6] J. Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/.

[7] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012), 35:1–35:21. https://doi.org/10.1145/2086696.2086714

[8] X. Dong, Z. Shen, J. Criswell, A. Cox, and S. Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 5:1–5:9. https://doi.org/10.1145/3214292.3214297

[9] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. 2018. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust*. 187–190. https://doi.org/10.1109/HST.2018.8383912

[10] A. Fuchs and R. B. Lee. 2015. Disruptive Prefetching: Impact on Side-channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 14:1–14:12. https://doi.org/10.1145/2757667.2757672

[11] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 217–233.

[12] D. Gruss, R. Spreitzer, and S. Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *Proceedings of the USENIX Security Symposium*. 897–912.

[13] D. Gullasch, E. Bangerter, and S. Krenn. 2011. Cache Games – Access-Based Cache Attacks on AES to Practice. In *Proceedings of the IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

[14] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2016. Cross Processor Cache Attacks. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. ACM, 353–364. https://doi.org/10.1145/2897845.2897867

[15] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. 2008. Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* 12, 3 (Sept. 2008), 221–230. https://doi.org/10.1007/s10617-008-9018-y

[16] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv:1806.05179 [cs]* (June 2018). arXiv:1806.05179 http://arxiv.org/abs/1806.05179

[17] T. Kim, M. Peinado, and G. Mainar-Ruiz. 2012. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 11–11. http://dl.acm.org/citation.cfm?id=2362793.2362804

[18] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors.

[19] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203 [cs]* (Jan. 2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[20] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. 2008. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the ACM Workshop on Computer Security Architectures*. ACM, 25–34. https://doi.org/10.1145/1456508.1456514

[21] J. Kong, O. Aciicmez, J. P. Seifert, and H. Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 393–404. https://doi.org/10.1109/HPCA.2009.4798277

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 469–480. https://doi.org/10.1145/1669112.1669172

[23] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. 2011. CACTI-P: Architecture-Level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 694–701. http://dx.doi.org/10.1109/ICCAD.2011.6105405

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown. *arXiv:1801.01207 [cs]* (Jan. 2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[25] F. Liu and R. B. Lee. 2013. Security Testing of a Secure Cache Design. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 3:1–3:8. https://doi.org/10.1145/2487726.2487729

[26] F. Liu and R. B. Lee. 2014. Random Fill Cache Architecture. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 203–215. https://doi.org/10.1109/MICRO.2014.28

[27] F. Liu, H. Wu, K. Mai, and R. B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (Sept. 2016), 8–16. https://doi.org/10.1109/MM.2016.85

[28] R. Martin, J. Demme, and S. Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, 118–129. http://dl.acm.org/citation.cfm?id=2337159.2337173

[29] D. A. Osvik, A. Shamir, and E. Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the RSA Conference*. Springer, 1–20.

[30] D. Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint archive* 2005, 280 (2005).

[31] A. Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/.

[32] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 565–581.

[33] M. K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*.

[34] P. Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[35] Z. Wang and R. B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the Annual Computer Security Applications Conference*. 473–482. https://doi.org/10.1109/ACSAC.2006.20

[36] Z. Wang and R. B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 494–505. https://doi.org/10.1145/1250662.1250723

[37] Z. Wang and R. B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[38] Z. Wu, Z. Xu, and H. Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 159–173.

[39] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*.

[40] Y. Yarom and K. Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *Proceedings of the USENIX Security Symposium*, Vol. 1. 22–25.

[41] Y. Zhang and M. K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*. ACM, 827–838. https://doi.org/10.1145/2508859.2516741

[42] X. Zhuang, T. Zhang, and S. Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. ACM, 72–84. https://doi.org/10.1145/1024393.1024403