Understanding Selective Delay as a Method for Efficient Secure Speculative Execution

Christos Sakalis, *Member, IEEE,* Stefanos Kaxiras, *Senior Member, IEEE,* Alberto Ros, *Senior Member, IEEE,* Alexandra Jimborean, and Magnus Själander, *Senior Member, IEEE*

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

Index Terms—speculative execution, side-channel attacks, memory, security

1 Introduction

Since the introduction of Meltdown [1] and Spectre [2], speculative execution, the cornerstone of modern, high-performance CPUs, has been under attack. By abusing the fact that speculative execution, by nature, can execute instructions that would have otherwise not have been executed by the application, a malicious attacker is able to bypass software and hardware barriers and leak sensitive information. These attacks exploit different parts of the system to access and leak the information, but they all consist of two main parts: The *illegal* access to the information, done speculatively, and the *information leakage*. The first part depends on the aforementioned property of speculative execution that allows instructions to execute and access data that they would normally not be allowed to. However, to leak that data, the second part is necessary, which takes advantage of micro-architectural state-changes that are done under speculative execution and that can be observed by the attacker during or after the speculation has been resolved. While there are some practical limitations, these two parts are interchangeable: The way the attack gains access to the sensitive information and the way this same information is leaked are independent of one another, assuming there is a way of transferring the information from the first to the second. Specifically, for the second part, a number of different side-channels are available, capable of leaking information across software and hardware barriers. These might include side-channels such as memory-timing side-channels [3], port-contention side-channels [4], or even a frequency-scaling side-channel [5]. Due to being easy to exploit and also offering great versatility between where the victim and the attacker are placed relative to each other, memory-timing side-channels (including cache-timing side-channels) are particularly popular [6], [7].

Researchers, both in academia and in the industry, have been frantically working on providing solutions for these issues. To solve specific attacks, solutions might focus on preventing the illegal access to the sensitive information in the first place [8], [9], [10], [11]. This approach has the advantage that it can be implemented quickly, sometimes requiring only software or operating system changes, but it does not protect against any future attacks or even variants of the same attack. Instead, some researchers (including ourselves) have focused on preventing the leakage of information through side-channels, and specifically speculative side-channels.

In our previous work [12], we proposed an efficient way for invisible speculative execution by introducing delay-on-miss, a way of selectively delaying load instructions from executing under *unsafe* speculation, preventing any speculative side-effects from being made visible in the memory hierarchy. Specifically, delay-on-miss builds on the insight that if a load hits in its own private L1 cache, it only causes minimal side-effects that can easily be delayed, as they are not part of the critical path of the memory access. To determine when an instruction is *safe* or *unsafe*, we also introduced the concept of speculative shadows [12], [13], a way of determining the earliest point at which an instruction is no longer speculative, reducing the cost of naïvely delaying all speculative instructions. Finally, we combined the delay-on-miss approach with value prediction, as a way of further mitigating the cost of delaying speculative loads.

In this work, we give an in-depth analysis into the results

- *C. Sakalis, S. Kaxiras, A. Jimborean, and M. Själander are with Uppsala University, Sweden. E-mail:first.lastname@it.uu.se.*
- *A. Ros is with the University of Murcia, Spain .*
- *M. Själander is with the Norwegian University of Science and Technology.*

Manuscript received January 13, 2020; revised August 26, 2015.

of our previous work (delay-on-miss and value prediction). Specifically,

- We provide a detailed justification for the performance of the delay-on-miss approach, especially when compared to a delay-all approach.
- We examine the effect that delay-on-miss has on the memory level parallelism (MLP) of the application and how this affects the runtime performance.
- We revisit our results for the value prediction, in light of a new, more accurate implementation that uncovered issues in the original implementation; issues that inadvertently led to an overestimation of the benefits of value prediction.
- We explore the properties of value prediction to further understand what its limiting factors are and how it can be improved. We identify that the accesses required for validating the predictions are the main culprit, as not only they have to be restricted under the same conditions as all the other loads in delay-on-miss, but they also introduce a new type of unsafe speculation, which imposes further restrictions.

One of the main insights of this work is that the delay-on-miss approach performs well with respect to the baseline (when compared with other secure solutions) because, contrary to intuition, it does not completely restrict all the memory level parallelism available in the application. On the other hand, value prediction offers additional *instruction* level parallelism (ILP), but does not contribute to memory level parallelism (Section 3). Due to these effects, the delay-on-miss approach reduces the overhead of delaying speculative loads by two thirds, from -53% under the baseline for the delay-all version to -18%. Value prediction reduces the delay-on-miss overhead by 1 percentage point, and we show that even with an oracular value predictor that can correctly predict any load value, the mean overhead is still reduced by only 3 percentage points. Furthermore, we show that for the value predictor to offer significant performance improvements, we would have to allow the prediction validations to be issued without any restrictions, violating the delay-on-miss guarantees.

2 Threat Model

We consider speculative side-channel attacks that utilize the memory hierarchy as the side-channel. Attacks that use different side-channels, such as port contention, energy usage, etc., are outside the scope of this work; please see Section 4 for other delay based solutions that go beyond the memory system. Additionally, we focus on speculative side-channel attacks that target the core concept of speculative execution. Specifically, we assume that memory permission checks are done fully and correctly as soon as the virtual to physical memory translation has been completed. If this is not the case, then we assume *i)* that loads are delayed until such permission checks are made and *ii)* that the duration of the relevant shadows (Section 5) is also extended accordingly. Attacks that depend on such information not being checked on time (e.g. Meltdown – which is still covered by delay-on-miss) exploit implementation specific bugs that are not inherent in speculative execution, are possible only on specific CPU manufactures/models and can be fixed without restricting speculative execution.

Two components are necessary to successfully mount a speculative side-channel attack: *i)* A way of bypassing software/hardware barriers to access information illegally, and *ii)* a way of leaking that secret data across the speculation boundary. Generally, based on the first component, such attacks can be split into two broad categories, Spectre-style and Meltdown-style attacks [14], depending on how they exploit speculative execution to illegally access information. Our work is orthogonal to this categorization, as we are not concerned with how the attacker manages to execute the malicious code or how the secret data are accessed, we are only concerned with preventing the leakage of information. In addition, we assume that an attack can be launched from any context: From the same thread, from a different thread sharing the core (SMT), or even from a different core. Whether the attacker and the victim share the same virtual/physical memory or not is also not a limitation. We also assume that all data is equally important in needing to be kept secret, solutions that identify subsets of the data and only protect those can be applied to our solutions as well, but evaluating them is outside the scope of our work. Finally, outside the scope of this work are also attacks that are not speculative in nature, i.e., traditional side-channel attacks, even if they utilize the same memory side-channel techniques.

3 Instruction and Memory Level Parallelism

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

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

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

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

To better understand the effects of delaying speculative loads, we will further divide MLP into two categories: **intra-cacheline MLP** and **inter-cacheline MLP**. By *intra*-cacheline MLP, we refer to multiple in-flight memory accesses that all target the same cache line. Once the line is brought into the cache, then all the memory accesses targeting it can be satisfied. However, even if these accesses were not overlapped, once the first access brings the line into the cache, then the rest would be cache hits, downgrading the intra-cacheline MLP to hit-MLP.

On the other hand, *inter*-cacheline MLP refers to the case when there are multiple independent memory accesses to *separate* cache lines, as in the example above. If inter-cacheline MLP is not exploited in the application, then each memory access to each cache line will introduce significant latency in the execution. Thus, we argue (and demonstrate in Section 8) that inter-cacheline MLP is the most important type of MLP to exploit during execution.

4 The State-of-the-Art

There is a body of work covering side-channel defences for the memory system that predates speculative side-channel attacks [6], [7]. These works focus on the conventional side-channel attacks that leak information between different execution contexts. However, some speculative side-channel attacks, such as Spectre v1 [2], can be exploited from within the same execution context, bypassing the aforementioned defence mechanisms. Instead, we are interested in solution that specifically target speculative side-channel attacks. Unfortunately, there is currently no established metric for comparing how secure each solution is, nor is there a way of comparing the security and cost trade-offs. This makes it impossible to directly compare the different solutions against each other, as each one makes different assumptions about what is secure or insecure, and what falls within its scope. We can, however, categorize the different solutions in three broad categories:

Hiding speculation: This approach includes solutions such as InvisiSpec [15], Ghost loads [13], SafeSpec [16], and MuonTrap [17]. All of these allow speculative execution to continue unhindered, but delay any visible side-effects until after the speculation has been resolved. Specifically, all three solutions focus on hiding the side-effects of loads in the memory system by keeping fetched data in temporary buffers, in order to prevent speculative memory accesses from updating the cache. While such solutions do not fully restrict the MLP of the application during execution, they still segment it and also need to impose restrictions during validation for coherence reasons. They also have the disadvantage that they require modifications throughout the whole system and that, since instructions are allowed to execute and access the memory, it is hard to hide all possible side-effects.

Delaying speculation: Instead of trying to hide the side-effects of speculatively executed instructions, these solutions prevent "unsafe" instructions from being executed in the first place. We have quoted the word "unsafe" here because which instructions are considered as safe and which are not changes depending on the exact threat model and implementation details of each solution. Our current work, delay-on-miss [12], delays all speculative loads until they are certain to be retired (Section 5). Conditional Speculation [18], NDA [19], Speculative Taint Tracking (STT) [20], SpectreGuard [21], and Context [22] keep track of the flow of information during execution and prevent any speculatively loaded data from being used by any "unsafe" instruction. The advantage of such solutions is that the changes required are mostly isolated in the core, instead of being pervasive in the whole memory hierarchy and the coherence protocol. At the same time, since instructions are not executed at all, they prevent a larger array of attacks than solutions that try to hide speculative execution. As a matter of fact, the last three solutions protect more than just the memory system, but at a significant performance loss. Their disadvantage is that by restricting the execution of speculative instructions, they restrict the amount of ILP and MLP that can be exploited during execution. The authors of STT have also proposed Speculative Data-Oblivious Execution (SDO) [23], an extension to STT that utilizes speculation to replace speculative execution that depends on speculatively loaded data with data-oblivious execution. In their work they focus on different forms of prediction, instead of using value prediction, so it is not possible to compare the two directly.

Undoing speculation: During speculative execution, architectural side-effects are kept hidden and any changes made during speculation that might lead to incorrect execution are squashed. CleanupSpec [24] takes a similar approach for the micro-architectural changes as well. Specifically, it utilizes cache randomization to prevent information leakage between different execution contexts, invalidations and re-fetching of data to undo the side-effects in the L1 cache, and delaying some accesses that can cause unsafe transitions in the coherence directories. This solutions achieves the best (reported) performance from all the other solutions, but at the cost of pervasive modification in the whole system. In addition, the energy costs have not been explored.

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

5 Speculative Shadows

As described in our previous work [12], [13], speculative shadows are cast by instructions that might cause a misspeculation and, subsequently, squashing of all instructions

that follow them in the execution window. Essentially, the shadows are a way of defining (and tracking) when an instruction is guaranteed to be retired, without having to wait for it to reach the head of the reorder buffer (ROB) [25]. As long as an instruction is under such a shadow, then it cannot be speculatively executed *safely*. We refer to such instructions as *shadowed*, and to the instructions that cause the shadow as *shadow casting instructions*. As soon as the shadow is *lifted/resolved* and the instruction is *unshadowed*, then it can be safely executed even if it is still speculative. Since we only focus on the memory system, we are only concerned with tracking shadowed loads; other instructions that might be used as part of a non-memory side-channel attack are not considered.

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

- **E-Shadows:** These shadows are cast by instructions that might throw an *exception*, causing the execution to be aborted and redirected to an exception handler. Common examples include memory access operations that might cause memory exceptions and arithmetic operations.
- **C-Shadows:** These are shadows that are cast by instructions that *control* the execution flow directly, most commonly branches. While the target of the branch is unknown, it is not possible to know if the instructions being executed are the correct instructions, or if they will have to be squashed when the branch target has been resolved.
- **D-Shadows:** When a load reads a memory location previously modified by a store, then the load has a *data dependency* to that store. If during execution a store with an unknown store address is encountered, all following loads have the potential of being dependent on that store. Out-of-order processors deploy a memory dependency predictor, which determines if there is a high chance that a load can safely bypass a store. When a misprediction occurs, execution has to be restarted from the offending load.
- **M-Shadows:** This is a special type of shadow caused by the coherence and memory model requirements of the system. If the total store order (TSO) model is employed, the order between loads needs to be observed. Younger loads can speculatively bypass older loads, but if an order violation is detected[1], the execution needs to be restarted from the younger load that caused the violation. This shadow is not present in systems where the memory model allows for loads to be reordered, such as release consistency (RC) based systems.
- **VP-Shadows:** If value prediction is employed, it is necessary to validate the prediction. If a prediction is incorrect, then the execution needs to be restarted, this time with the correct value.

It is possible to efficiently track these shadows in hardware [12] but, for brevity, we will not repeat the mechanism here in detail. With the exception of the VP-Shadows, which

1. In the context of the system's memory model.

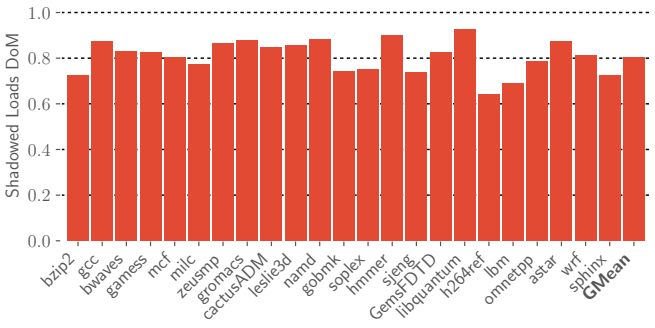


Fig. 1. Ratio of loads executed under a speculative shadow when delay-on-miss is enabled.

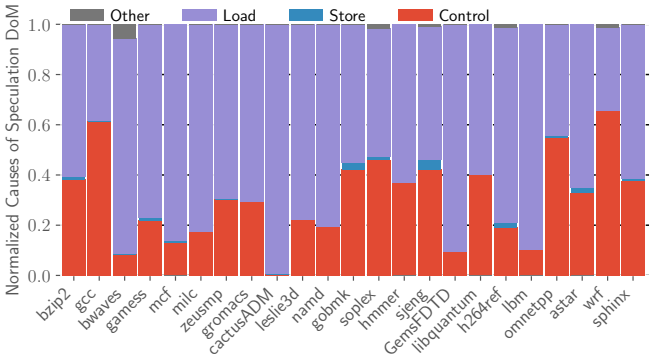


Fig. 2. Oldest shadow covering each shadowed load.

start once a load is predicted, the rest of the shadows start when an instruction is dispatched and put into the ROB. All of the shadows last until the reason for the shadow has been resolved. For example, C-Shadows can be removed once the target of the branch instruction is known. On the other hand, M- and VP-Shadows require the shadow casting load to be executed/validated before being lifted. This restricts the number of loads that can be unshadowed to one at a time, which makes solutions that delay all shadowed loads indiscriminately impractical.

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

Figure 1 contains the number of loads executed under a speculative shadow in the delay-on-miss version for the SPEC2006 benchmarks, normalized to the total number of loads executed. This number ranges from 64% (`h264ref`) to

93% (`libquantum`), with a mean of 80%. In Figure 2 we can also see the types of instructions that cause these shadows. Specifically, each time we need to check if an instruction is under a shadow or not, we record the oldest shadow-casting instruction. As we have modified our shadow-tracking implementation to lift shadows caused by stores earlier, the number of store shadows is greatly reduced. In our experiments, removing just one shadow type does not significantly affect the performance, as each load is usually covered by multiple shadows, but this new implementation does indicate that the most important shadows (for optimizations) are the ones caused by load and by control instructions.

In addition to misspeculation, the execution can also be interrupted and squashed by interrupts. As, generally speaking, an interrupt can happen by external reasons and at any point during the execution, a generic "interrupt shadow" would always be present regardless of the current state of the instructions in the pipeline. In this work, we assume that interrupts can be delayed until the speculation in the pipeline has been resolved but, as not all interrupts are the same, we can handle different interrupts in different ways.

To begin with, there are interrupts that are uncorrelated to the underlying execution and do not have precise timing requirements, such as context switches from the OS scheduler or IO related interrupts. As these interrupts can happen at any time, squashing earlier or a bit later does not play any role in the correctness of the execution. For these reasons, these interrupts can be left unaltered, as any (delayed) squashing caused by them cannot be part of a speculative side-channel attack.

On the other hand, interrupts that are caused by the execution of the program directly, such as exceptions or explicit interrupt instructions, require precise timing and squashing, as they do affect the correctness of the execution. Such interrupts need to be treated as any other shadow caused by an instruction. In our evaluation, interrupts caused by exceptions are covered by the E-shadows, but explicit interrupt instructions (e.g. the x86 `int` instruction) are not, as they are treated as non-speculative by our simulator to begin with. The shadow tracking mechanism can be easily modified to include such instructions as well, but since they are rare in the benchmarks we use, there will be no observable performance difference.

Note that some systems, such as real-time systems, have special requirements on how interrupts need to be handled, as well as special use cases for them. Such systems fall outside the scope of this work, as such cases need to be evaluated separately, to determine the security implications together with the system requirements.

6 Delay-on-Miss

Delay-on-miss protects against speculative attacks that use the memory hierarchy as a side-channel by delaying all "unsafe" loads that miss in the L1 cache. As "unsafe" we define all loads that are executed under a speculative shadow. Stores are not delayed, because stores are not allowed to speculatively modify the memory hierarchy in the first place. Other, non-memory instructions are also not delayed, since they fall outside the scope of this work. The main insight behind delay-on-miss is that hiding the side-effects of loads that hit in the private L1 cache is easy, as the data can be returned without making any changes in the cache, and the replacement policy and prefetcher updates can be delayed until later. On the other hand, hiding loads that need to fetch data from other, lower level, caches is not as easy, because it requires mechanisms for getting the data without modifying anything in the hierarchy or the main memory, storing the data temporarily, and handling the coherence implications [13], [15].

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

Delay-on-miss differs from mechanisms such as NDA [19] or STT [20], as both NDA and STT allow for speculative loads to be executed but delay all dependent instructions. This is based on the insight that for a secret to be leaked, it has to be loaded first. The first load, the one that loads the secret into the core, does not itself leak any information, instead the loaded value is fed to other instructions, which in turn construct a side-channel and leak the secret. In STT, only values loaded by a load that is currently speculative are considered unsafe. The implication is that STT does not protect against attacks that leak secret values that are already stored in registers [19]. NDA on the other hand offers different modes, one of which protects all values from being leaked, at additional performance cost.

Figure 3 shows the first four steps of the example in our previous work [12], describing how shadows are tracked, with the addition of how the memory hierarchy is accessed. The example shows the reorder buffer (ROB) and the additional structures required to track speculative shadows, i.e., the shadow buffer and release queue. The example starts with an empty ROB and due to previously executed instructions SB-Head and SB-Tail arbitrarily point to the fourth entry of the empty shadow buffer. For simplicity, and without loss of generality, only the branch C-Shadows are tracked in the example, other shadows are ignored.

1) **Step 1:** When the first load ($\texttt{LD}_0$) enters the ROB the shadow buffer is empty (SB-Head==SB-Tail) indicating that the there are no shadow casting instructions and that the load can be executed as normal ⓐ. *The load can to go all the way to the DRAM to fetch the data.* This is the same behavior as in an unsecured architecture.
2) **Step 2:** The branch ($\texttt{BR}_1$) casts a C-shadow, which is indicated by setting the SB-Tail entry of the shadow buffer ⓑ and marking the ROB entry with SB-Tail ⓒ. The SB-Tail is then incremented. No access is made to the memory hierarchy since it is not a load instruction.
3) **Step 3:** When the second load ($\texttt{LD}_2$) enters the

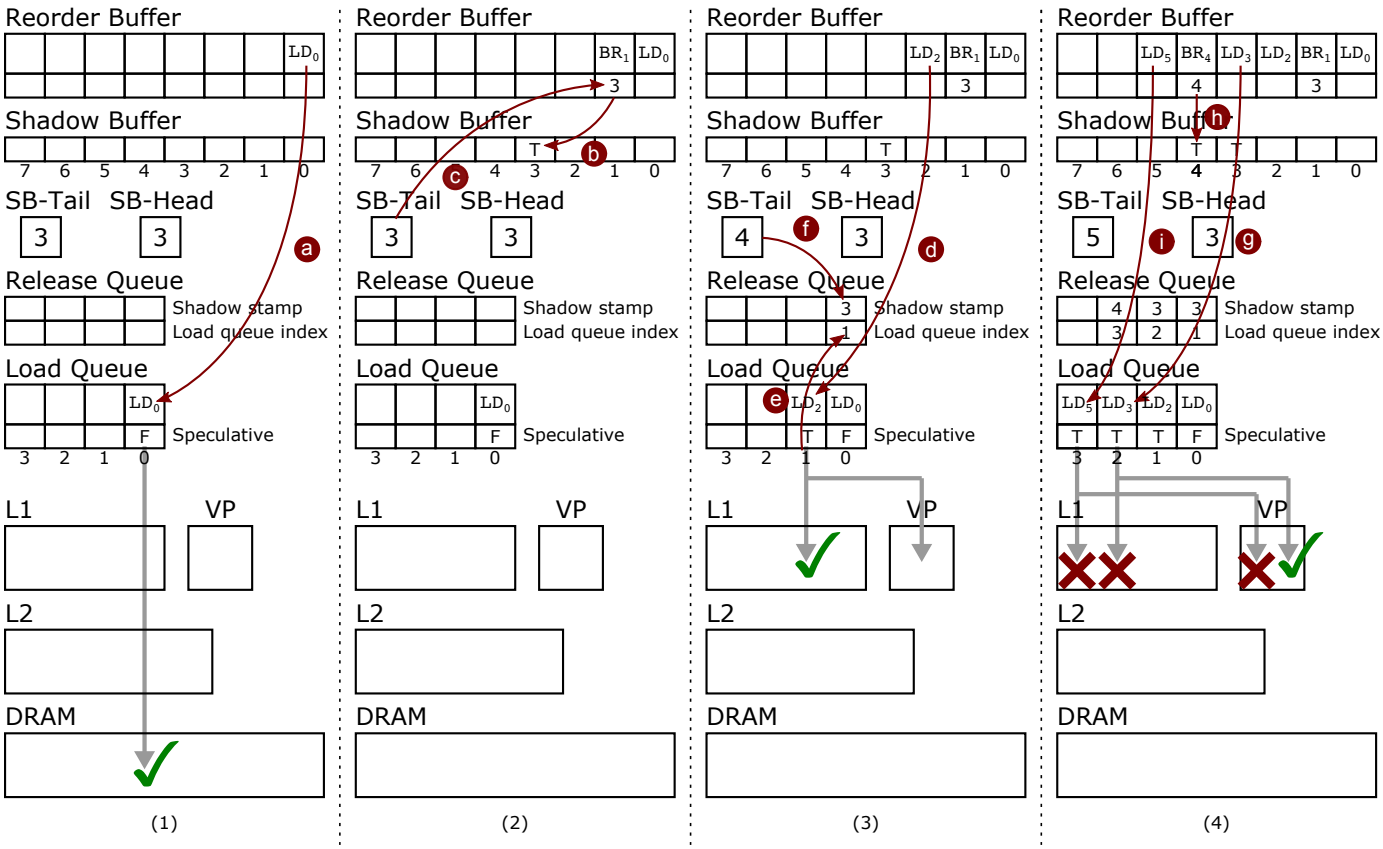


Fig. 3. Example of speculative and non-speculative loads accessing the memory hierarchy and the value predictor.

ROB the shadow buffer is no longer empty. This indicates that at least one older shadow casting instruction exists causing the load to be shadowed/speculative (d). An entry is allocated in the release queue and associated with the load queue entry of the load (e) and the youngest shadow-casting instruction, which is determined by SB-Tail minus one (f). Since the load is shadowed a tentative access is made in the L1 cache that turns out to be a hit. *The data of the load can be returned from the L1 without modifying any state in the cache, e.g., without updating replacement policy state.* Instead, the cache state will be updated once the load is unshadowed, i.e., once $\texttt{BR}_1$ is completed causing SB-Head to be incremented. This results in the first entry of the release queue to not equal SB-Head indicating that no older shadow casting instructions exist and the load can safely update the cache state and exit the release queue. This is not shown in this figure, for a detailed description see steps six and seven in the example in our previous work [12].

4) **Step 4:** Two shadowed loads ($\texttt{LD}_3$ and $\texttt{LD}_5$) miss in the L1 and are prevented from accessing the next level caches or the main memory. *This is in contrast to an unsecured architecture, where the next level cache would have been accessed instead.* We will see in the next section (Section 7) how the value predictor comes in play. The two loads will only be allowed to access the next level caches once they become unshadowed, as is shown in step 1 of the example.

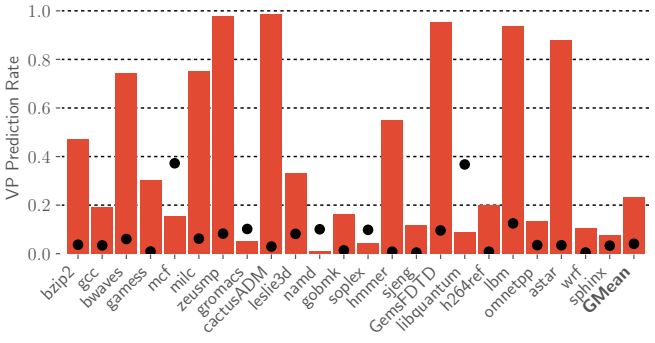


Fig. 4. Prediction rate for the VP (bars) and the L1 miss ratio (circles)

7 Value Prediction

Value prediction is a micro-architectural optimization that aims at reducing the delay and overheads of executing [26] or even scheduling [27] instructions by predicting their value based on previously seen values. Similar to a cache, a value predictor takes advantage of the locality exhibited by the data but instead of focusing on spatial or temporal locality, the value predictor exploits *value locality*. In our work, we use value prediction as a mechanism for hiding the delay introduced by delaying shadowed loads. Evaluating all the different predictor types, their advantages and disadvantages, and how well they perform, is beyond the

scope of this work, but we do have three requirements that the predictors must meet in order to be safely used:

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

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

Figure 4 contains the VP prediction rate for each benchmark, i.e., the number of shadowed L1 misses that the VP is able to predict over the total number of shadowed L1 misses. In the same figure, the L1 miss ratio of the benchmarks can also be seen, as the total number of predicted loads over all loads depends on the number of L1 misses during execution. Overall, the VP achieves a prediction rate of 23%, with benchmarks ranging from as low as 1% (`namd`) to as high as 98% (`cactusADM`). In the performance analysis, Section 8, we will also present an oracular VP, with 100% prediction rate, to evaluate the upper limit of the performance gain that improving the prediction rate and accuracy can offer.

Returning to our example in Figure 3, **Step 4** shows the case where two shadowed loads ($\texttt{LD}_3$ ⓖ and $\texttt{LD}_5$ ⓘ) miss in the L1. Under delay-on-miss, these loads are not allowed to access the next level caches. However, the value predictor (VP), which is accessed in parallel with the L1 accesses, is able to predict the value of the third load with a high confidence. The predicted value is returned without updating any state of the cache nor the value predictor, letting the load execute.

Once the first branch ($\texttt{BR}_1$) is resolved the predicted value is validate by issuing a normal load to the memory hierarchy, which retrieves the value and updates the caches and the value predictor state. The final load has to wait for the second branch ($\texttt{NR}_4$) to also be resolved before a normal memory access can commence.

Note that, for simplicity, the example tracks only C-Shadows. The load instructions seen in the example will themselves also cast E- and M-Shadows, as well as VP-Shadows (explained below). The main difference with the example would be that each load would also allocate an entry in the shadow buffer indicating that the load casts a shadow.

Regarding the VP-Shadows, since the values provided by the predictor are speculative, they need to be *validated*, both to ensure that the value was correct at the time of the prediction, but also to avoid coherence issues on memory models that enforce the load-load order. To do so, a normal, globally visible memory access needs to be issued in the memory system. Since such an access would be unsafe under delay-on-miss, the same rules as for normal loads apply: If the validation access hits in the L1, then it can be proceed unhindered, otherwise it has to be delayed. At the same time, if the validation fails, all instructions that follow the value predicted load need to be squashed[2], hence why the VP-Shadow is introduced. This means that while value prediction introduces some additional ILP, it does not introduce any MLP, as it simply pushes the actual memory access from the execution point to the validation. We will discuss the performance implications of this in detail in Section 8.2.

In addition to providing functional correctness, we also need to ensure that the predictor cannot be used to construct new speculative side-channel attacks. Specifically, a side-channel attack consists of two components, a way of bypassing software/hardware barriers to access information illegally, and a side-channel for leaking this information outside of the speculative execution.

When it comes to the first component, the predictor can be used to guide the execution to a misspeculated path, gaining access to data illegally, or it can even provide sensitive data directly, in the form of a prediction. For example, in the classic Spectre v1 attack, where the attacker trains the branch predictor in order to bypass an array bound check, the value predictor can also be trained to provide an incorrect array bound value. However, as we have introduced the VP-Shadows, all data accessed speculatively due to a value prediction are protected by delay-on-miss, and cannot be leaked.

On the other hand, as the predictor contains micro-architectural state that affects the timing of the execution, it can also be used as the second component of an attack, i.e. it can be used to construct a timing side-channel. To solve this, the predictor is never updated using speculative data, as such micro-architectural state changes under speculation can be used to leak information. In our case, as the predicted value can only be validated when the load becomes unshadowed, all the updates to the predictor are delayed until the speculation has been resolved by default. This prevents attacks from both the same and from different execution contexts from using the value predictor as a side-channel. In addition, to further secure the predictor, any temporary state that needs to be kept for implementation reasons, needs to be either obfuscated or statically partitioned between different execution contexts. This prevents temporary state from being used a side-channel between two execution contexts being executed in parallel. Finally, as the predictor is on the L1 level and does not participate in coherence, it is invisible to the other cores and we do not need to protect it from attacks originating from another core.

8 Evaluation

We use a combination of the Gem5 simulator [28], McPAT [29] with CACTI [30], and the SPEC2006 [31] benchmark suite. The architectural parameters of the simulated system can be found in Table 1. In Gem5, we first fast-forward through one billion instructions using the atomic simple CPU, and then simulate in full detail using the O3 CPU for another three billion instructions. From all the benchmarks in SPEC2006, we have excluded `perlbench`,

2. There is a possibility of selectively squashing and replaying only the load-dependent instructions, but evaluating such approaches are beyond the scope of this work.

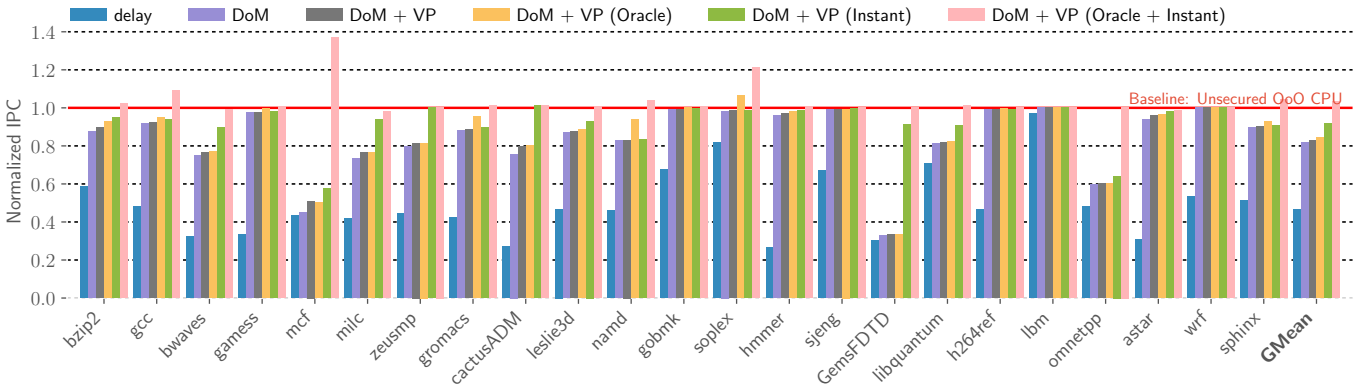


Fig. 5. Performance (IPC) normalized to the unsecured baseline.

TABLE 1
The simulated system parameters.

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

`dealII`, `povray`, `calculix`, `tonto`, and `xalancbmk` due to baseline simulation issues. In total, we evaluate seven different hardware versions:

- **baseline:** The unmodified, *unsecured* out-of-order CPU that Gem5 provides.
- **delay-all:** A secured out-of-order CPU where all shadowed loads are delayed until they have been unshadowed.
- **DoM:** Delay-on-miss, without any value prediction.
- **DoM + VP:** Delay-on-miss with a VTAGE value predictor.
- **DoM + VP (Oracle):** Delay-on-miss with the oracular value predictor, capable of correctly predicting all speculative L1 misses. Even thought the Oracle VP is always correct, validations are still being sent out once the load has been unshadowed.
- **DoM + VP (Instant):** Delay-on-miss with the VTAGE predictor, but validation are sent as soon as the value has been predicted, instead of waiting for the load to be unshadowed. This is an *unsecured* version that is used to evaluate the cost of delaying and serializing the validations.
- **DoM + VP (Oracle + Instant):** Delay-on-miss with the oracular predictor and instant validations. This is another unsecured version used to demonstrate the cost of validation.

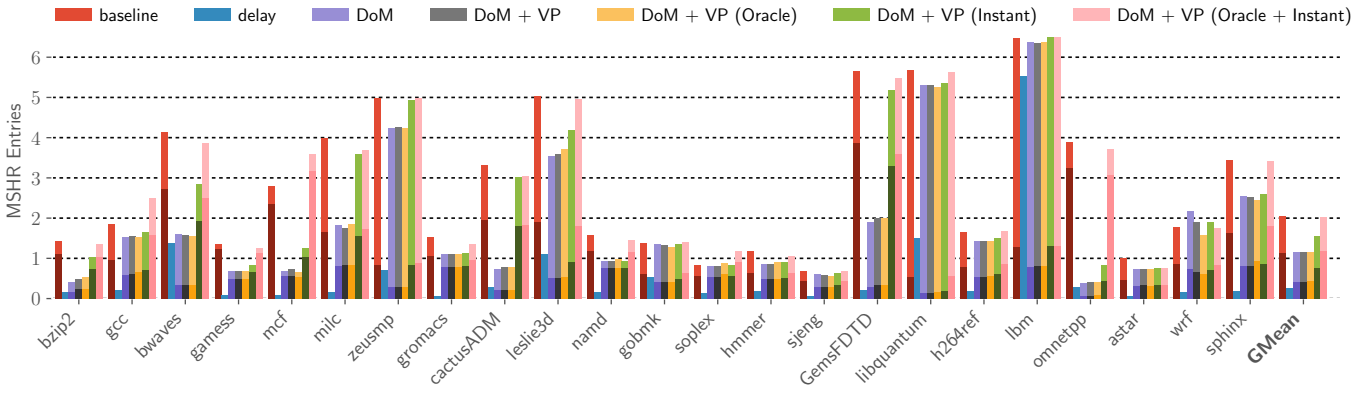
McPAT can model the energy usage of the CPU components but it does not offer the capability of adding new components without modifying its source code. In order to calculate the energy usage of the value predictor, we model it as a branch target buffer (BTB) and a branch predictor (BP) in McPAT. While the implementation details of these units differ from the VP, they are the two components that are the closest to the VP. In addition, McPAT does not model the DRAM, for this we rely on the integrated DRAMPower [32] model in Gem5.

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

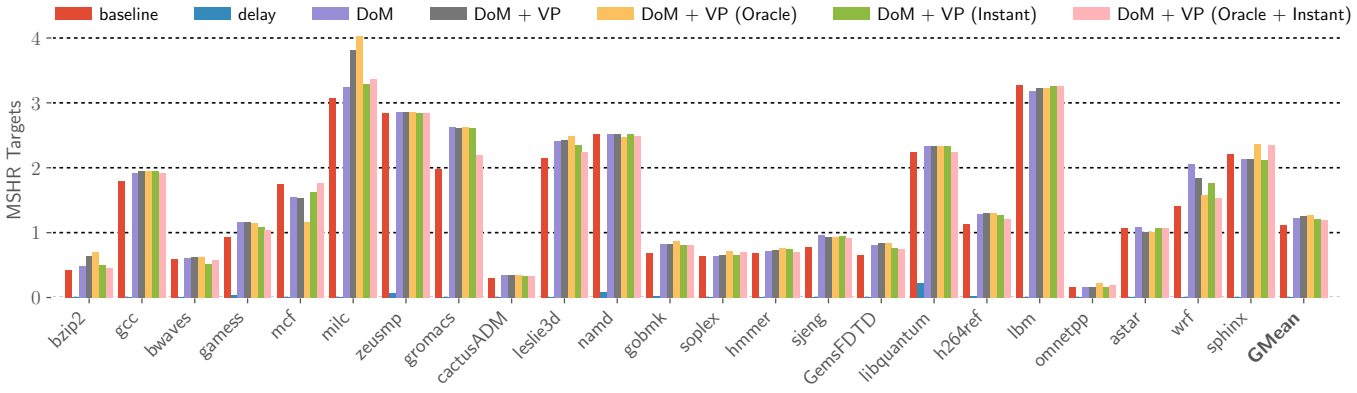
8.1 Performance

In this section we will establish the performance of the different versions, before further diving into the different runtime characteristics in the sections that follow. Figure 5 contains the average number of retired instructions per cycle (IPC) for each benchmark and for each version, normalized to the unsecured baseline version. First, we have the delay-all approach, where all shadowed loads are delayed until they are unshadowed. The performance of this version ranges from -74% (`hmmer`) to -3% (`lbm`) under the baseline, with a mean performance loss of 53%.

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



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



(b) MSHR Targets.

Fig. 6. Average number of MSHR entries (inter-cacheline) and targets per entry (intra-cacheline).

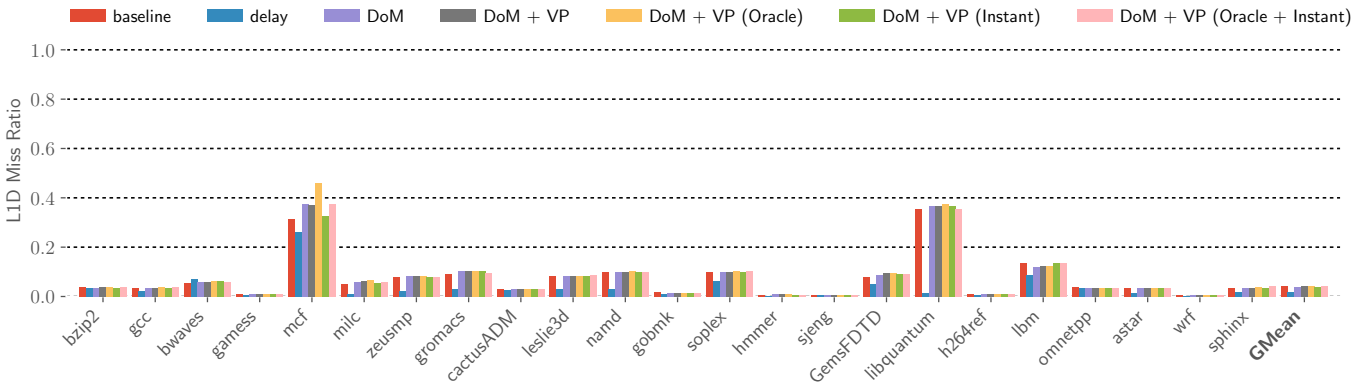


Fig. 7. L1 miss ratio.

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

8.2 Memory Level Parallelism

Having established the performance that the different versions exhibit, it is time to understand why. As we have explained, memory level parallelism plays an important role in the performance of the applications. We will use the

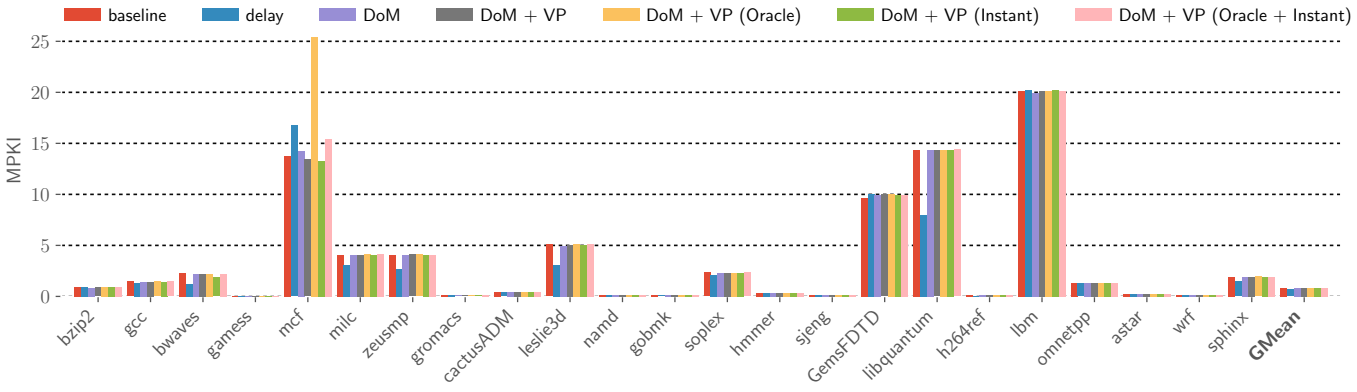


Fig. 8. LLC (L2) misses per one thousand instructions.

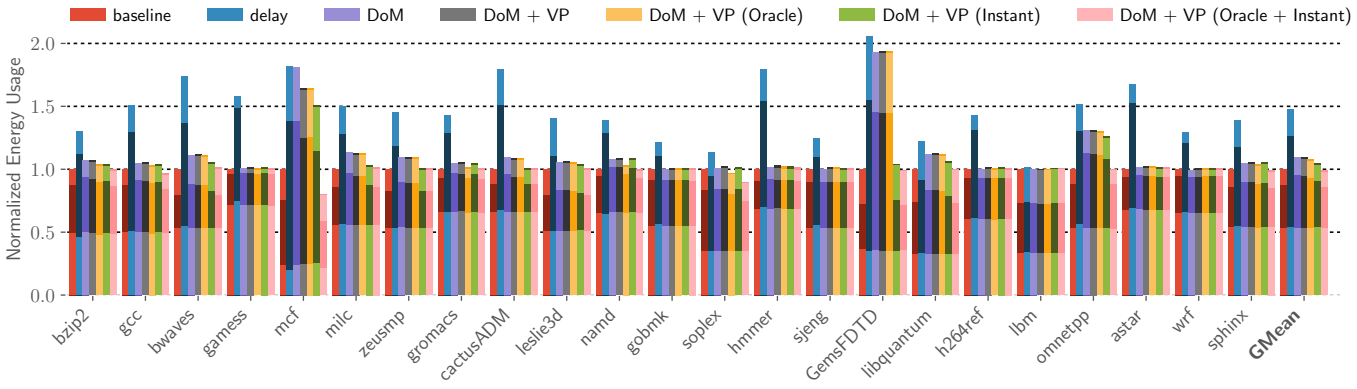


Fig. 9. Energy usage normalized to the unsecured baseline. Each bar consists of four parts (bottom to top): The dynamic energy usage of the CPU (bottom, light colored), the static energy usage of the CPU (middle, dark colored), the dynamic energy usage of the DRAM (middle, light colored), and the VP overhead, both static and dynamic (top, dark colored).

MSHRs in the L1 cache as a proxy metric for measuring the amount of MLP during execution. For each cache line requested in a cache, an **MSHR entry** is allocated. Each entry in turn has several **targets**, each of which corresponds to a memory access, whether that access is a load or a store[3]. The number of entries indicates how many different cache lines are fetched in parallel (inter-cacheline MLP) while the number of targets indicates how many accesses benefit from each fetch (intra-cacheline MLP).

Figure 6 contains the average number of MSHR entries (6a) and targets (6b) in the matching entry already allocated when a load access misses in the L1 cache. For loads that are delayed or value predicted, we only account for their visible access/validation, i.e., we do not count the number of MSHR entries/targets at the point that they were delayed. Similarly, delayed loads are not counted as part of the allocated targets/entries. For the MSHR entries, we also differentiate between entries that were allocated by a load (excluding prefetches – bottom, dark color) and entries that were allocated by another operation (top, light color). In figures 7 and 8 the L1 data-cache miss ratio and the last-level cache (LLC) MPKI can also be seen. By looking at the mean number of entries (6a) we can immediately see how each version affects the inter-cacheline MLP of the application. In the baseline, there is an average of two MSHR entries already allocated when a load misses in the L1, which translates to two cache lines being fetched in parallel. Out of these, roughly half are entries allocated by loads, while the rest correspond to other operations. If we indiscriminately delay all shadowed loads (delay-all), then the number of load-allocated entries falls to zero and the total to a mean value of less than one. Essentially, in the majority of the cases, when a load misses in the L1 there are no other pending memory fetches. This translates to very poor MLP, which in turn leads to very poor performance.

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

Finally, if we look into the number of MSHR targets

3. Other operations (such as prefetching) might also allocate MSHR targets.

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

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

8.3 Energy Usage

As neither version significantly affects MPKI or the instructions executed by the benchmarks, the two main factors affecting the energy usage is the execution time and the overhead of doing value predictions. As the value predictor uses a relatively small amount of energy, compared to the rest of the system, the execution time is the main parameter affecting the energy usage.

The results can be seen in Figure 9. Each bar consists of four parts, from bottom to top: The dynamic energy usage of the CPU (bottom, light colored), the static energy usage of the CPU (middle, dark colored), the dynamic energy usage (including refresh and power-down states) of the DRAM (middle, light colored), and the overhead of the VP, both static and dynamic (top, dark colored).

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

9 Conclusions

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

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

Acknowledgments

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

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," Jan. 2018. [Online]. Available: http://arxiv.org/abs/1801.01207

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, May 2019, pp. 19–37.

[3] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks." in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2016, pp. 565–581.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garcia, and N. Tuveri, "Port Contention for Fun and Profit," in *Proceedings of the IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 2019, pp. 870–887. [Online]. Available: https://ieeexplore.ieee.org/document/8835264/

[5] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," 2018.

[6] Y. Lyu and P. Mishra, "A Survey of Side-Channel Attacks on Caches and Countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, Mar. 2018. [Online]. Available: http://link.springer.com/10.1007/s41635-017-0025-y

[7] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr. 2018. [Online]. Available: http://link.springer.com/10.1007/s13389-016-0141-6

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

[9] C. Reis, "Mitigating Spectre with site isolation in Chrome," 2018, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[10] "The Linux kernel user's and administrator's guide: Spectre side channels," 2019, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[11] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation," *arXiv:1906.01345 [cs]*, Dec. 2019. [Online]. Available: http://arxiv.org/abs/1906.01345

[12] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 723–735. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322216

[13] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and S. Magnus, "Ghost loads: What is the cost of invisible speculation?" in *Proceedings of the ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, 2019, pp. 153–163. [Online]. Available: http://doi.acm.org/10.1145/3310273.3321558

[14] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. v. Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, Oct. 2018, pp. 428–441.

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

[17] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, 2020.

[18] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *Proceedings of the International Symposium High-Performance Computer Architecture*, Feb 2019, pp. 264–276.

[19] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 572–586, event-place: Columbus, OH, USA. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358306

[20] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 954–968, event-place: Columbus, OH, USA. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358274

[21] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," in *Proceedings of the ACM/IEEE Design Automation Conference*. Las Vegas, NV, USA: ACM Press, 2019, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3316781.3317914

[22] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "ConTExT: Leakage-Free Transient Execution," *arXiv:1905.09100 [cs]*, May 2019. [Online]. Available: http://arxiv.org/abs/1905.09100

[23] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, 2020.

[24] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "undo" approach to safe speculation," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 73–86. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358314

[25] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 68–77. [Online]. Available: http://dl.acm.org/citation.cfm?id=1153925.1154589

[26] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 226–237.

[27] A. Perais and A. Seznec, "EOLE: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance," *ACM Trans. Comput. Syst.*, vol. 34, no. 2, pp. 4:1–4:33, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2870632

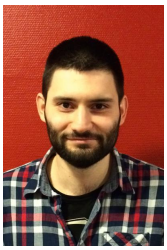
[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, Dec. 2009, pp. 469–480.

[30] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "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*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 694–701.

[31] Standard Performance Evaluation Corporation, "SPEC CPU benchmark suite," http://www.specbench.org/osg/cpu2006/, 2006.

[32] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "DRAMPower: Open-source dram power & energy estimation tool," 2012, http://www.drampower.info.



Christos Sakalis is a PhD student at Uppsala University, Sweden. His research is on secure hardware architectures, and specifically on protecting general purpose systems from speculative side-channel attacks. Before starting his PhD, Christos worked as a compiler engineer at Codeplay, UK, focusing on auto-vectorization for their ComputeAorta toolkit.

Stefanos Kaxiras is a full professor at Uppsala University, Sweden. His research interests are in the areas of memory systems, and multiprocessor/multicore systems, with a focus on power efficiency. He is a Distinguished ACM Scientist and IEEE member.

Alberto Ros is Associate Professor at the University of Murcia, Spain. He was funded by the Spanish government to conduct the PhD studies and received the PhD in computer science from the University of Murcia in 2009. He hold postdoctoral positions at the Universitat Politècnica de València and at Uppsala University. He received a Consolidator Grant from European Research Council. His research interests include cache coherence, memory consistency, and processor microarchitecture.

Alexandra Jimborean is an Associate Professor at Uppsala University since 2019. She obtained her PhD from the University of Strasbourg, France in 2012, was awarded the Anita Borg Memorial Scholarship offered by Google in recognition of excellent research, along with other 30 distinctions, awards and grants. Her research focuses on compile-time and run-time code analysis and optimization for performance, energy efficiency, and security and on software-hardware co-designs.

Magnus Själander is an Associate Professor at the Norwegian University of Science and Technology (NTNU) and a Visiting Senior Lecturer at Uppsala University. He obtained his Ph.D. from Chalmers University of Technology in 2008. Before joining NTNU in 2016 he has been a researcher at Chalmers, Florida State University, and Uppsala University. Själander's research interests include hardware/software co-design (compiler, architecture, and hardware implementation) for high-efficiency computing.