Seeds of SEED: Preventing Priority Inversion in Instruction Scheduling to Disrupt Speculative Interference

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

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

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

***Abstract*—Speculative side-channel attacks consist of two parts: The speculative instructions that abuse speculative execution to gain illegal access to sensitive data and the side-channel instructions that leak the sensitive data. Typically, the side-channel instructions are assumed to follow the speculative instructions and be dependent on them. Speculative side-channel defenses have taken advantage of these facts to construct solutions where speculative execution is limited only under the presence of these conditions, in an effort to limit the performance overheads introduced by the defense mechanisms.**

Unfortunately, it turns out that only focusing on dependent instructions enables a new set of attacks, referred to as "speculative interference attacks". These are a new variant of speculative side-channel attacks, where the side-channel instructions are placed *before the point of misspeculation* and hence before any illegal speculative instructions. As this breaks the previous assumptions on how speculative side-channel attacks work, this new attack variant can be used to bypass many of the existing defenses.

We argue that the root cause of speculative interference is a priority inversion between the scheduling of older, bound to be committed, and younger, bound to be squashed instructions, which affects the execution order of the former. This priority inversion can be caused by affecting either the *readiness* of a not-yet-ready older instruction or the *issuing priority* of an older instruction after it becomes ready. We disrupt the opportunity for speculative interference by ensuring that current defenses adequately prevent the interference of younger instructions with the *availability* of operands to older instructions and by proposing an instruction scheduling policy to preserve the priority of ready instructions. As a proof of concept, we also demonstrate how the prevention of scheduling-priority inversion can safeguard a specific defense, Delay-on-Miss, from the possibility of speculative interference attacks. We first discuss why it is susceptible to interference attacks and how this can be corrected without introducing any additional performance costs or hardware complexity, with simple instruction scheduling rules.

I. Introduction

Speculative side-channel attacks abuse speculative execution to gain access to data that would not be accessible otherwise. The attacks consist of two main parts: One or more speculatively executed instructions that are able to temporarily bypass software and hardware barriers to illegally gain access to sensitive data ("the secret"), and one or more side-channel instructions that leak said secret from the speculative to the non-speculative domain. These side-channel instructions can in turn be categorized into two parts: The transmitter and the receiver. The transmitter is executed within the speculative domain and has access to the speculatively accessed secret data, but it cannot directly communicate the secret outside of the speculative domain. Instead, the transmitter performs one or more actions that change the microarchitectural state of the processor in a way that is dependent on the value of the secret. In turn, the receiver, which is executed outside the speculative domain, can perceive these microarchitectural state changes and, indirectly, infer the secret.

It has generally been assumed that the transmitter side-channel instructions are dependent on the speculative access instructions, and that the receiver instructions are executed either in parallel or after the transmitter instructions, depending on the nature of the microarchitectural state being used as the communication channel. However, a new variant of speculative side-channel attacks, the speculative interference attacks, introduced by M. Behnia et al. [5], breaks this assumption. Under a speculative interference attack, the receiver, or even part of the transmitter, exists before (in program order) the speculative instructions that illegally access the secret data. This breaks the assumptions that many speculative defense mechanisms are designed around, leading to information leakage.

More specifically, speculative interference, as described by M. Behnia et al. [5], uses speculative instructions to interfere with the scheduling of older instructions that are on the correct path of execution and bound to commit. By changing the order in which these instructions are executed (e.g., a different order of loads accessing the cache), it is possible to induce observable microarchitectural changes in the system, such as changes in the cache replacement state [5], [8]. This is achieved by having secret-value-dependent speculative instructions that are *issued before* and *interfere with* earlier (in program order) non-speculative instructions.

While detailed information regarding instruction schedulers in actual microarchitectures is scarce, the job of any instruction scheduler is to create a prioritization, e.g., based on age, of all the ready instructions such that those with the highest priority can be identified and issued. The implemented prioritization specifies an order in which ready instructions will be executed. We can see ***speculative interference as a priority inversion in instruction scheduling***. Assuming priority based on age, an *older, bound to be committed instruction* loses its chance to execute in the earliest cycle it becomes ready, because some *younger speculative instruction* hogs a resource that is needed for its execution. For example, a younger integer division instruction that becomes ready before an older instruction of the same type might occupy the integer division functional unit (FU) for several cycles, preventing any other instructions (including older ones that have become ready in the meantime) from being issued. Normally, without taking security into account, such priority inversions do not matter much — after all, the goal is to maximize performance and if a priority inversion leads to a better utilization of the core's out-of-order resources, so much the better. However, from a security point of view, such priority inversions are the cause of speculative interference.

In this work, we first characterize the problem of speculative interference attacks in the context of priority inversion. Then, having developed an understanding of the underlying issue, we discuss a generic method for preserving the execution order of instructions and preventing the issue. The key to avoiding such priority inversions is to assure that no lower priority instruction can prevent a higher priority ready instruction from executing, regardless of when the lower priority instruction became ready and was issued. Finally, as a proof-of-concept, we apply our method on an existing state-of-the-art speculative side-channel defense, Delay-on-Miss [14], [16],

```
if(...) // mispredicted branch
    secret = load(...) // illegal access
    if (secret == 1) // secret-value-dependent
        b = load(B_addr)
        a = load(A_addr)
    else
        a = loab(A_addr)
        b = load(B_addr)
```

Listing 1. Typical Spectre attack.

```
a = load(A_addr)
b = load(B_addr)
if(...) // mispredicted branch
    secret = load(...) // illegal access
    if (secret == 1) // secret-value-dependent
        interfere_with_A()
    else
        // do nothing
```

Listing 2. Speculative interference Spectre attack.

such that priority inversion is prevented. Specifically, we discuss which of the speculative interference attacks described by M. Behnia et al. [5] are actually applicable to Delay-on-Miss and changes in the instruction scheduling and miss status handling register (MSHR) allocation policies to counteract them at no additional overhead.

II. Speculative Interference Attacks

Speculative interference attacks [5] are a new variant of existing speculative side-channel attacks. Specifically, they are a new way of making observable microarchitectural changes to the system while executing under speculation, with the goal of leaking speculatively accessed secret data, bypassing some of the current state-of-the-art defenses against such actions [1], [12], [15], [16], [21]. This is achieved by breaking one of the main assumptions that defenses make about speculative side-channel attacks, namely the assumption that the code that modifies the microarchitectural state and leaks the secret follows, *in program order*, the code that illegally (speculatively) accesses the secret, and will be squashed when the misspeculation is detected. Instead, using a speculative interference attack, it is possible for the leaking instructions to precede (in program order) the illegal accesses and to be on the correct, bound to be committed, path of execution.

As an example, we can compare a typical Spectre-type attack (Listing 1) with a variant that utilizes speculative interference (Listing 2). Both attacks utilize a replacement policy side-channel, where the attacker is able to distinguish the order of two loads in the cache by probing the replacement state of the cache [5], [8]. In the typical Spectre example, the malicious code will first access the secret illegally under speculation and then perform a secret-value-dependent modification to the microarchitectural state of the processor. In this case, the order between loading `A_addr` and `B_addr` changes depending on the secret. Existing defences that focus on preventing cache and memory speculative side-channels, such as InvisiSpec [21] or Delay-on-Miss [16], will prevent these two loads from making any visible changes to the replacement state of the cache, therefore preventing any information leakage.

On the other hand, in the speculative interference variant, the two loads appear before the mispredicted branch and the illegal access, and are therefore not affected by defences like InvisiSpec or Delay-on-Miss. For this example, let us assume that (with no interference) (i) the load to `A_addr` is executed before the load to `B_addr` and (ii) both loads will not be issued until after `interfere_with_A` has been issued. Also, let us assume that `interfere_with_A` performs some operation that will prevent the load to `A_addr` from being issued when it becomes ready (but not the load to `B_addr`) and that said operation is not inhibited by the deployed defense mechanism. An example of such an operation would be interfering with the FUs necessary for calculating `A_addr` itself. Since neither InvisiSpec nor Delay-on-Miss try to hide or delay non-memory operations, the interference will not be stopped and the order of the loads to `A_addr` and `B_addr` will be affected in a secret-value-dependent way, thus bypassing the defenses and leaking the secret information.

III. The Root Cause: Instruction Scheduling

In a typical modern out-of-order (OoO) processor, newly decoded and renamed instructions enter the instruction queue (IQ) and wait to become ready and to be issued to an appropriate execution/functional unit (FU). For an instruction to become ready, first all its operands need to be ready. Then, for a ready instruction to be issued, there needs to be an appropriate FU that will be available in the next cycle, where the instruction will then be executed.

Since the processor is able to issue only a limited number of instructions per cycle (issue width), regardless of how many ready instructions might be waiting in the IQ, a *policy* is needed to decide which instructions should be prioritized. When it comes to commercial processors, exact details on the subject are scarce, but one well established policy (used in actual architectures) is age-based or pseudo-age-based [4], [9], [11], [17] prioritization. As an optimization, a (pseudo-)age-based policy can give higher priority to load instructions over other instructions, while still maintaining the age-order between loads. In essence, regardless of optimizations between different instruction types, we typically expect ready instructions to be roughly ordered by age, with respect to the order in which they are issued to an FU. This is natural, as commit has to happen in program order, where older instructions need to be committed before younger ones. If we did not prioritize the older instructions at the issue stage, then we would be inhibiting their timely commit and risk not having enough commit bandwidth when they do become ready, thus losing performance [17]. The current known state-of-the-art implementations include shifting queues [10], where the instructions are inserted into the queue in order and gaps in the queue are filled by shifting the instructions, and age-matrix based queues [17], where the queue itself is not kept in order but instead a matrix holding the age dependencies between instructions is used to order the ready instructions. Comparing the different queue implementations is beyond the scope of this work and we will discuss how our proposed solutions can be implemented in either of these.

However, this age-based prioritization only happens between ready instructions, as we do not want older instructions that are *not* ready delaying younger ready instructions. Even if a younger instruction were to temporarily prevent (by occupying a resource) an older instruction from being issued, it would not be an issue, since the main concern of the scheduler is to utilize the pipeline to its fullest, in an effort to achieve the best performance possible. However, this changes when we also consider security and speculative interference attacks.

A. Priority Inversion in Instruction Scheduling

First, let us assume that the instruction scheduler has a well defined and deterministic policy to select ready instructions, which defines a total order (**priority**) between all the ready instructions currently waiting in the instruction queue. For simplicity and without loss of generality, we will consider this order to be strictly based on instruction age, where "age" is defined by program order and an

instruction is considered "older" than another instruction if it appears earlier in the program order. At each cycle, as the execution of the program progresses, the priority of the instructions changes, with new instructions becoming ready as dependent instructions complete their execution. We can now define an instruction as **priority issued** (where by *issued* we mean – more broadly – that an instruction has obtained all the resources[1] it needs for unimpeded execution) if:

1) its ancestor instructions (i.e., the instructions producing its operands and their ancestors, as well as other instructions that directly affect the execution of the instruction, such as branches) were also priority issued and
2) when ready, it cannot be prevented from being issued and executed by an instruction with a lower (if considered in the same cycle) priority.

Expressing this differently, a ready instruction is not priority issued if either it or any of its ancestor instructions were prevented from being issued and executed by an instruction that would have lesser priority if both instructions were ready *at the same time*. Therefore, instructions that become ready earlier can still violate the priority issue of instructions that become ready later, if the later instructions would have had higher priority if both sets of instructions were ready at the same time. Note also that the definition is applied recursively to the whole dependency chain of instructions needed before the instruction can become ready, including non-data dependencies such as control dependencies. This is necessary to cover the cases where it is possible to indirectly interfere with an instruction by interfering with older instructions that precede it. Furthermore, the definition covers not just the uninhibited issue but also the execution of the instruction. We focus mostly on issue because that is typically the earliest out-of-order stage in the pipeline, where instructions begin to be re-ordered. Finally, note that the definition only applies when instructions become ready and depends on the exact scheduling policy; it can thus be different than the actual program order between all instructions in the program.

Now (assuming, without loss of generality, an age-based policy) consider two instructions, `OLDER` and `YOUNGER` (in program order), both of which require the same non-pipelined resource for several cycles (e.g., integer division). If both `OLDER` and `YOUNGER` become ready at the same time, then `OLDER` will be issued first, as it has higher priority than `YOUNGER`. However, if `YOUNGER` becomes ready even one cycle before `OLDER`, it will be issued first and proceed to occupy the non-pipelined resource. This does not affect the priority issue of `OLDER`, as older is not ready yet. At the next cycle, `OLDER` becomes ready but can no longer be issued, because the non-pipelined resource is now occupied. Hence, the `OLDER` (higher issue-priority) instruction is now being prevented from being issued by the `YOUNGER` (lower issue-priority) instruction and is no longer priority issued, due to a **priority inversion in the scheduling priority of instructions.**

Essentially, at each cycle, the ready instructions in the IQ are in a total order, typically age based. The highest priority of these instructions (in our case, the oldest) is always priority issued if it is not prevented by a *resource conflict*, i.e., it is always priority issued if all the necessary resources (see footnote 1) exist in the system at that cycle. Some resources, particularly single-cycle or fully-pipelined resources, can always be considered as "free," as the only way for them to be occupied by a lower issue-priority instruction would be if said instruction has been issued over a higher issued priority in the same cycle, in which case the scheduler is broken as it failed to select the ready instruction with the highest priority. Thus, between all the ready instructions, instructions that only depend on single-cycle or fully-pipelined (single-cycle repeat) units will always be priority issued.[2] On the other hand, multi-cycle non-pipelined or partially pipelined (with a repeat of more than a single cycle) resources (**Note:** for brevity, we will refer to all these resources simply as **"non-pipelined"**), which once allocated remain occupied for several cycles, may appear to be busy, thereby preventing the highest issue-priority instruction from being priority issued. We discern two cases:

- A resource needed by the instruction `A`, which is the highest-priority ready instruction (i.e., oldest ready instruction in the IQ) is busy because it is held by instruction `B` that is older than `A`. This does not violate the second requirement for an instruction to be priority issued, because if `A` and `B` were considered for issue at the same time, `B` would have a higher priority and would still get the resource. Hence, no priority inversion happen in this case.
- When `B` is younger than `A`. In this case, if `A` and `B` were considered for issue at the same time, `A` would have a higher issue priority and would get the resource. The fact that `B` has it and is preventing `A` from being issued constitutes a **priority inversion** and violates the requirement for `A` to be priority issued. In fact, this is exactly the case in the earlier `YOUNGER`/`OLDER` example.

B. From Priority Inversion to Speculative Interference

Let us now partition all in-flight instructions into two categories: non-speculative and speculative. Non-speculative are all instructions for which, unless prevented by an external factor (such as an interrupt), it can be determined that they are *bound to commit* successfully. In the simplest version, only the head of the reorder buffer (ROB) can be considered as non-speculative, but in practice it is possible to extend this to more instructions, as long as they meet the necessary commit criteria [2], [6], [15], [16]. All other instructions, which are either bound to be squashed or of unknown state, are considered as speculative. As the execution progresses and speculation (e.g., branch predictions) is resolved, more and more instructions in the ROB transition from speculative to non-speculative. Since one of the conditions for an instruction to be successfully committed is that all the instructions that precede it (in program order) in the ROB have also been committed successfully, non-speculative instructions always precede speculative instructions and, under an age-based scheduling policy, the *non-speculative instructions will always have a higher issue-priority than the speculative ones.* However, the scheduler is designed to only focus on ready instructions, in an effort to avoid unnecessary bubbles in the pipeline.

During a speculative interference attack, one or more younger speculative instructions become ready before an older (eventually non-speculative) instruction and occupy a non-pipelined shared resource, causing a priority inversion.[3] Due to this priority inversion, the now ready old non-speculative instruction loses its priority issue and is delayed, which causes observable microarchitectural side-effects, thus leaking sensitive information. Since this is a direct result of the issue priority inversion, we can conclude that (i) the priority inversion is the root cause for speculative interference attacks and (ii) we can prevent speculative interference attacks by preventing such priority inversions from happening.

[1]This includes execution resources beyond FUs, such as TLBs, MSHRs, etc.

[2]Note that this refers to ready instructions, assuming that their ancestors have already been priority issued, to meet the full definition.

[3]Note that while the attack does not necessitate age-based issue priority, it is always necessary to be able to make an assumption about the issue priority of instructions, as the goal of the attacker is to reverse the normal execution order of instructions. In practice, the scheduler has to have a way of resolving conflicts, so there will always be some natural ordering between instructions, regardless of how sophisticated or random that might be.

C. Preventing Priority Inversion

Priority inversion is a known problem, especially in the fields of operating and real-time systems. Solutions such as preemption, assigning higher priority to specific tasks that hold resources, or pre-allocation of resources already exist, but we cannot apply existing solutions blindly, as we have very strict requirements:

- Some solutions for the priority inversion problem assume that the priority inversion does not need to be solved immediately, as long as it is solved in a timely manner. However, depending on the nature of the side-channel being utilized, even a single cycle difference in the scheduling of the instructions can lead to long lasting side-effects in the system. For example, delaying a load instruction by one cycle can lead to changes in the state of the caches, which can be easily observed for several cycles afterwards. Therefore, the solutions we use need to prevent priority inversion immediately and with no additional latency.
- Some solutions would come at a too high cost when implemented in hardware. For example, preemption-based solutions might require a lot of additional storage, which would be prohibitive due to area and latency costs, or might require flushes or replays in the pipeline. Therefore, the solutions we use need to take into consideration the characteristics of the resources involved and the additional hardware cost they might introduce.

As a general rule, if we can preempt a resource so that it can be used by a higher issue-priority instruction, without introducing additional latencies or costs, then this would be the preferred solution. When this is not possible, we fall back to a pre-allocation based solution, where resources are allocated to instructions when they appear in the IQ, even before they become ready. In essence, instead of allocating resources to ready instructions as they appear, the non-ready instructions are also taken into consideration. This is not the optimal approach as, unlike preemption, it restricts instruction execution at all times, regardless if during execution an instruction would actually cause a priority inversion or not. Preemption on the other hand is only applied dynamically when the need arises.

IV. Proof-of-Concept Use Case

In this section, we describe how speculative interference can be prevented in the context of a state-of-the-art mitigation technique. We first describe the general principles behind Delay-on-Miss, which delays speculative loads if they miss in the L1 cache to prevent speculative side-channels in the memory hierarchy. We also explain how and where priority inversions due to speculative interference can arise, as well as how these can be corrected. Specifically, we will discuss how interfering with either non-pipelined instructions or MSHR allocation can lead to information leakage and what modifications we can make to the issue and MSHR allocation policies to prevent this.

It should be noted that while we focus on Delay-on-Miss as our proof-of-concept use case, other defense mechanisms are susceptible to the exact same issues [5] and our example solutions are not case specific and can be applied to more than just Delay-on-Miss.

A. Delay-on-Miss

Delay-on-Miss [16] (DoM) is designed to protect against speculative side-channels that target the memory hierarchy as the side-channel mechanism. Other side-channels, such as non-speculative side-channels or side-channels that do not target the memory hierarchy, are outside the scope of DoM and are generally not affected by it. This is a trade-off between security and performance, as memory hierarchy based side-channels have a number of advantages over other side-channels, such as (i) observability across several cores and contexts, (ii) the fact that caches and memories are typically not fully associative, and thus it is possible to abuse the indexing to leak several bits of information at once, and (iii) that changes in the memory hierarchy can persist until overwritten, even after the memory operation has finished execution. However, as we have seen in the speculative interference attack example (Section II), it is possible to use speculative non-memory operations to affect the order of non-speculative memory operations, therefore bypassing DoM and leaking sensitive information. We will discuss how this issue can be handled efficiently by changing the issue policy for non-pipelined FU instructions in Section IV-B.

DoM works under two basic principles. First, it is hard to hide the side-effects of loads in the memory hierarchy on a cache miss, because reading data into a cache requires complex interactions between the rest of the system, as seen in previous solutions such as InvisiSpec [21] and Ghost Loads [15]. Instead, DoM opts to delay misses from speculative loads until they are non-speculative. On the other hand, a load that hits in the L1 cache requires only small modifications to the cache state, e.g., updating the replacement state, which are outside the critical path of the access operation. Thus, DoM only delays loads that miss in the L1 cache, and permits loads that hit in the L1 cache to execute speculatively, only delaying their non-critical side-effects.

In order to help reduce the performance overhead of delaying L1 misses, DoM tries to allow for as much memory level parallelism (MLP) as possible, while remaining secure. To achieve this, DoM will treat accesses to the L1 cache that miss in the cache but hit in an MSHR as a hit [14]. DoM will prevent speculative loads from allocating an MSHR, so an existing MSHR indicates that the data has already been requested by a non-speculative access. Any side-effects caused in the memory hierarchy is due to the initial non-speculative instruction that in the first place allocated the MSHR. However, the L1 cache needs to know where to send the data for the speculative load, which can be achieved by setting a target for the speculative load in the MSHR. Since each MSHR has a limited number of targets and loads that cannot allocate a target are delayed, allocating targets from speculative loads can lead to observable side-effects. Separating the MSHRs between speculative and non-speculative accesses would not work, as it is not possible to know in advance if a speculative access will eventually become non-speculative or not, while also introducing area and energy overheads. Technically, on a typical implementation the lack of available MSHR targets will not prevent a memory operation from being issued by the scheduler, but the instruction will not be able to execute and will have to be rescheduled at a later time. Therefore, even if the scheduling mechanism is not directly involved in delaying the instruction (due to how memory operations are implemented in the pipeline), the effect is the same as if the instruction was prevented from being issued. For this reason, we consider this as the same case of priority inversion as for the FUs, and we will discuss how this issue can be handled efficiently by altering the MSHR allocation policy in Section IV-C.

B. Pre-allocation Policy: Multi-cycle Non-Pipelined FU Issue

As explained in Section II, multi-cycle partially/non-pipelined functional units (for brevity: **"non-pipelined FUs"**) can prevent instructions from being priority issued when occupied by a lower priority instruction. This issue does not appear in single-cycle or fully-pipelined units, which significantly limits the scope of the issue. Specifically, other than integer and floating point division, floating point square root (when present), and specialized macro operations (e.g., encryption etc), we expect all other operations to be either single-cycle or fully pipelined when implemented on a modern processor.

At the same time, it is not easy to apply preemption to such units. Even if we were to make the generous assumption that it is in fact possible to reset a unit and issue a new instruction to it within a single cycle, we still run into the problem of having to re-issue the preempted instruction at a later time. This introduces complications into the scheduling of instructions in the pipeline, as instructions can now have arbitrary execution latencies and need to be replayed when preempted. While such issues can be handled (as they already are, for example, for load instructions) this does come with an increased complexity and introduces additional delays [3], which in turn might violate the priority issue of instructions.

Due to these two observations, that only a limited set of instructions need to be handled and that preemption can be costly and ineffective, we handle such cases by modifying the issue policy for non-pipelined instructions to always issue instructions in program order. In practice, instead of just looking at the ready instructions, the scheduler needs to be modified to also consider the non-ready instructions that require the same FU and only issue an instruction if no older instructions (ready or not) exist:

a) Age-matrix implementation: An age matrix (that represents instructions in rows and columns) resolves scheduling conflicts based on age [17]. Briefly, in every cycle, the scheduler picks for issuing a subset of the ready instructions that are *bidding* for FUs. The age matrix allows older instructions to cancel the bids of younger instructions, thus making the latter ineligible for picking. To enforce a pre-allocation policy in an age matrix implementation, we simply modify the age-matrix to allow an older non-pipelined instruction to cancel the bid signals of younger instructions, even when the older instruction is not yet ready and bidding. More specifically, we change the age matrix conflict signal allowing it to be driven not only by the bid/ready signal, which is the normal operation, but also by a one-bit register per instruction that is set for non-pipelined instructions when they enter the instruction queue.

b) Shifting-queue implementation: In a shifting queue, the implementation is even simpler, as we only need to modify the scheduler so that it never checks past the first non-pipelined instruction, regardless if it is ready or not.

With either of these implementations, if the IQ consists of one unified queue for all instructions, then the proposed modifications will also affect single-cycle or fully pipelined instructions, significantly limiting the ILP. For this reason, we propose using a partially split queue implementation, where each non-pipelined instruction type has its own queue[4] and the scheduler picks ready instructions from each of the queues (e.g., [13]).

Intuitively, we might assume that a pre-allocation restriction in the scheduler would lead to significant performance loss, but we will see in the evaluation (Section V) that this is not true. After all, the majority of the instructions (most ALU operations, including branches, and memory operations) fall under the single-cycle or fully-pipelined categories, which remain unaffected. In addition, the restrictions only apply between instructions that require the same FU, older instructions that require a different FU do not prevent younger instructions from being issued when ready. If we could augment the scheduler with knowledge about when an instruction will eventually become ready, it would also be possible to issue lower priority instructions as long as they would be finished by the time the higher priority instructions became ready. However, since we cannot depend on speculative scheduling, such an optimization would require analyzing the data flow during execution and calculating strict lower bounds for when each instruction will become ready, which is both complicated and, as we will see in Section V, unnecessary.

[4]We will see in the evaluation that in practice we only need two queues, one for all the non-pipelined instructions and one for the rest.

C. Preemption Policy: MSHR Allocation

Under DoM speculative loads are never allowed to leave the L1 cache, so we do not need to enforce any order in other levels of the memory hierarchy. In addition, we will assume that the L1 cache itself is fully pipelined, hence preventing port contention from being used as a priority inversion vector. This leaves the problem of MSHR allocations, as discussed in Section IV-A.

There are two possible solutions that can prevent younger speculative loads from delaying older memory operations by allocating all available MSHR targets. The first and most obvious one is to simply guarantee that no speculative load will allocate any MSHR targets before it becomes non-speculative. This is similar to the approach we took for non-pipelined FUs (Section IV-B) and has the disadvantage that it will no longer be possible to coalesce speculative loads with existing non-speculative memory operations, which limits MLP and can have negative performance ramifications (Section V).

The second solution is to preempt existing speculatively-allocated MSHR targets if an older memory operation (speculative or not) needs to allocate a target. Under DoM, speculative loads need to be replayed in the L1 cache when they become non-speculative, either to re-request data (if the original speculative request was dropped because of a miss in the L1) or to update non-critical-path components of the cache, such as the replacement policy or the prefetcher, which cannot be done while the load is speculative. Since the provision for speculative loads to be replayed exists anyway, we can easily drop a target allocated for a speculative load without the additional overhead of rescheduling the load. Loads that are issued while they are already non-speculative are not replayed, but since we are only interested in preventing speculative loads from interfering with non-speculative instructions, we do not need to preempt existing MSHR targets from non-speculative loads to begin with.

In addition, the targets of each MSHR entry are stored in simple memories which, unlike the complex FUs we were discussing earlier, are easy to reset. As we will see in the next section (Section V), preemption is the better approach, as in practice the cases where we actually need to preempt an allocated MSHR target are very rare.

It should be noted that with the MSHR allocation policy the priority comparison (which memory operation is older) happens outside the IQ, so we can no longer depend on the age-based IQ implementation. Instead we can determine the order of loads using their position in the ROB or the load queue, using an additional bit to handle the cases where the head and tail pointers (assuming a circular buffer implementation) have wrapped around [9].

V. Evaluation

We use gem5 [7] and the SPEC2006 benchmark suite [20], using simpoints collected with ScarPhase [18]. The main simulation parameters can be found in Table I. As the changes in performance are insignificant and we have not introduced any significant hardware changes, there will also be no significant changes to the energy usage of the applications. For this reason, we will focus only on the performance (instructions per cycle – IPC) and we will not include the energy evaluation. We will present the following alternatives:

- **Delay-on-Miss baseline –** As our goal is not to evaluate Delay-on-Miss but ways to prevent speculative interference attacks on existing solutions for secure speculative execution, we use an already secured system as our baseline.

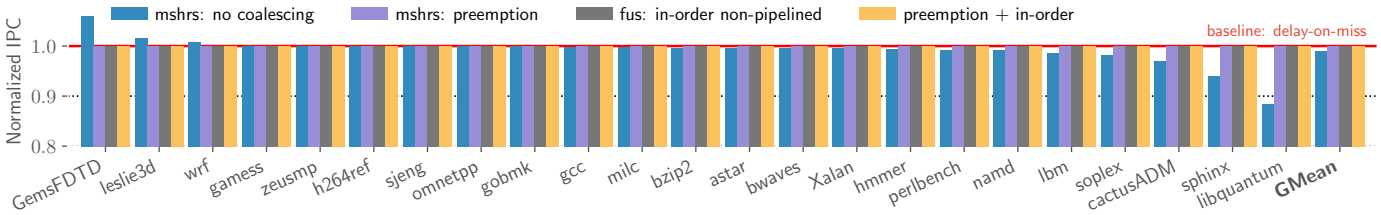

Fig. 1. Performance (IPC) normalized to the baseline DoM implementation. Note that the y-axis has been truncated to the range $[0.8, 1.0]$.

TABLE I
THE SIMULATED SYSTEM PARAMETERS.

Parameter	Value
Technology node	22nm @ 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
Simpoint length	100 million instructions
Fast warmup	990 million instructions
Detailed warmup	10 million instructions

- **MSHRs: no coalescing –** This version changes the MSHR allocation policy under DoM, disallowing coalescing of speculative loads with existing non-speculative loads (Section IV-C).
- **MSHRs: preemption –** This version allows coalescing for speculative loads with non-speculative loads but preempts MSHR targets when necessary (Section IV-C).
- **FUs: in-order non-pipelined –** This version changes the issue policy to make sure instructions to the same non-pipelined FU are always issued in order (Section IV-B). We have evaluated two versions: one were each non-pipelined instruction is issued in order only relative to other instructions of the same type (each non-pipelined instruction type has its own queue) and one where all non-pipelined instructions are issued in order relative to one another (all non-pipelined instruction go into the same queue). There is no observable performance difference between these two version, so we will not discuss them separately.
- **Preemption + in-order –** This version simply combines the policies from the *MSHRs: preemption* and *FUs: in-order non-pipelined* versions, protecting DoM from the interference attacks we have discussed.

Figure 1 contains the results sorted by the performance when load coalescing is disallowed (version *"MSHRs: no coalescing"*). We start with the MSHR allocation policy first as it is the only change that actually introduces any performance overheads. We observe that the majority of the benchmarks are not affected negatively by this, with some, `GemsFDTD` (+6% IPC) and `leslie3d` (+2%), actually benefiting, due to the filtering effect of not allocating MSHR targets for targets that will end up being squashed. However, there are a few benchmarks that are negatively affected, with the worst one being `libquantum` (−12% IPC), which is an MLP sensitive streaming benchmark, followed by `sphinx` (−6%), `cactusADM` (−3%), `soplex` (−2%), and `lbm` (−1%).

On the other hand, if we allow speculative MSHR target allocations and instead preempt speculative targets when a higher priority target needs to be allocated (version *"MSHRs: preemption"*), we see no observable performance difference across all the benchmarks. Our data (not shown) indicates that the times when a preemption is actually needed are very rare (less than one in a million for some benchmarks), which explains why there are no performance overheads.

We can observe the same results in the case where we enforce in-order issue for all non-pipelined FUs (*"FUs: in-order non-pipelined"*). The cases where a younger instruction is ready to be executed but has to be delayed are rare and do not affect the overall performance of the applications. This included both the case where each instruction type has a separate queue and the case where we have only two queues, one for all non-pipelined instructions and one for the rest. As a worst case scenario, we also evaluated a version (not shown) where we enforce the issue order of all instructions, non-pipelined or not, including all integer ALU operations. This was the only version where we observed significant performance degradation but, since such strict measures are not necessary for security (Section III), we do not need to consider it as a potential implementation.

Finally, by combining *"FUs: in-order non-pipelined"* and *"MSHRs: preemption"* into the *"Preemption + in-order"* version, we can protect DoM from the interference attacks we have discussed, without introducing any additional performance overheads.

VI. Conclusion

The problem of speculative interference attacks, which can bypass existing state-of-the-art speculative attack defenses, stems from the fact that the instruction scheduler, eager to extract as much parallelism from the instruction stream as possible, allows for priority inversions to happen between instructions in the instruction queue. This forces us to revise current defense mechanisms, but as priority inversion is a known problem, we can draw on the existing collective knowledge for solutions, keeping in mind that we are constrained by very strict timing and overhead requirements. Using one of the current speculative defence mechanisms, Delay-on-Miss, we show how the speculative interference problem can be framed as a priority inversion problem and what some of the constraints are when trying to solve it. We propose case-specific solutions and evaluate them, showing that as long as we take into consideration the requirements and characteristics of each interference mechanism, it is possible to shield existing defences against speculative interference without additional overheads.

While we have focused on Delay-on-Miss as a use case, our solutions can be applied on other defenses that also suffer from the same issues. In addition, if more interference mechanisms are discovered in the future, we can use the same principles to devise solutions for those as well, enabling more secure systems in the future.

Acknowledgments

This project was funded by the Swedish Research Council grants 2015-05159 and 2018-05254 and by MRL grant 2021-020. The simulations were performed on IDUN [19], provided by the Norwegian University of Science and Technology.

References

[1] 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*, 2020, pp. 132–144.

[2] M. Alipour, T. E. Carlson, and S. Kaxiras, "Exploring the performance limits of out-of-order commit," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2017, pp. 211–220.

[3] R. Alves, S. Kaxiras, and D. Black-Schaffer, "Dynamically disabling way-prediction to reduce instruction replay," in *Proceedings of the IEEE International Conference Computer Design*, 2018, pp. 140–143.

[4] H. Ando, "Performance improvement by prioritizing the issue of the instructions in unconfident branch slices," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 2018, pp. 82–94.

[5] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, 2021.

[6] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2004, pp. 68–77.

[7] 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.

[8] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 1967–1984.

[9] A. Buyuktosunoglu, A. El-Moursy, and D. Albonesi, "An oldest-first selection logic implementation for non-compacting issue queues," in *15th Annual IEEE International ASIC/SOC Conference*, 2002, pp. 31–35.

[10] J. A. Farrell and T. C. Fischer, "Issue logic for a 600-mhz out-of-order execution microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 707–712, 1998.

[11] A. Gonzalez, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–116, 2010.

[12] 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 *Proceedings of the ACM/IEEE Design Automation Conference*, Jun. 2019, pp. 1–6.

[13] "The issue unit – RISC-V BOOM documentation," 2019, https://docs.boom-core.org/en/latest/sections/issue-units.html.

[14] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Understanding selective delay as a method for efficient secure speculative execution," *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1584–1595, 2020.

[15] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, "Ghost loads: What is the cost of invisible speculation?" in *Proceedings of the ACM International Conference on Computing Frontiers*, 2019, pp. 153–163.

[16] 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, 2019, pp. 723–735.

[17] P. G. Sassone, J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 335–346, 2007.

[18] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient software-based online phase classification," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Nov 2011, p. 104–115.

[19] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure," *arXiv:1912.05848 [cs.DC]*, 2020.

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

[21] 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.