Pinned Loads: Taming Speculative Loads in Secure Processors

Zirui Neil Zhao
University of Illinois
Urbana-Champaign, USA
ziruiz6@illinois.edu

Houxiang Ji
University of Illinois
Urbana-Champaign, USA
hj14@illinois.edu

Adam Morrison
Tel Aviv University
Tel Aviv, Israel
mad@cs.tau.ac.il

Darko Marinov
University of Illinois
Urbana-Champaign, USA
marinov@illinois.edu

Josep Torrellas
University of Illinois
Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

In security frameworks for speculative execution, an instruction is said to reach its *Visibility Point* (VP) when it is no longer vulnerable to pipeline squashes. Before a potentially leaky instruction reaches its VP, it has to stall—unless a defense scheme such as invisible speculation provides protection. Unfortunately, either stalling or protecting the execution of pre-VP instructions typically has a performance cost.

One way to attain low-overhead safe execution is to develop techniques that speed-up the advance of the VP from older to younger instructions. In this paper, we propose one such technique. We find that the progress of the VP for loads is mostly impeded by waiting until no memory consistency violations (MCVs) are possible. Hence, our technique, called *Pinned Loads*, tries to make loads invulnerable to MCVs as early as possible—a process we call *pinning the loads* in the pipeline. The result is faster VP progress and a reduction in the execution overhead of defense schemes. In this paper, we describe the hardware needed by *Pinned Loads*, and two possible *Pinned Loads* designs with different tradeoffs between hardware requirements and performance. Our evaluation shows that *Pinned Loads* is very effective: extending three popular defense schemes against speculative execution attacks with *Pinned Loads* reduces their average execution overhead on SPEC17 and on SPLASH2/PARSEC applications by about 50%. For example, on SPEC17, the execution overhead of the three defense schemes decreases from 112.6% to 51.3%, from 35.8% to 15.3%, and from 24.8% to 13.2%.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Security and privacy** → **Side-channel analysis and countermeasures**.

KEYWORDS

Speculative execution defense, Processor design, Cache coherence protocol, Memory consistency

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507724

ACM Reference Format:
Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, Josep Torrellas. 2022. Pinned Loads: Taming Speculative Loads in Secure Processors. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507724

1 INTRODUCTION

Speculative execution attacks leak information through the execution of *transient* instructions—i.e., instructions that will get squashed [5, 9, 10, 12, 16, 22–24, 27–30, 34, 35, 41–43, 46]. These attacks are concerning because they exploit fundamental mechanisms of modern processors such as branch prediction, memory dependence prediction, and out-of-order instruction execution [15].

Since these attacks were first disclosed [23, 27], many defense schemes have been proposed. Such schemes range from hardware-based (e.g., [1, 4, 20, 21, 26, 32, 33, 38, 48, 51, 52]) to software-only (e.g., [2, 11, 17, 40]) and hybrid (e.g. [25, 39, 53]). They prevent the early, unprotected execution of *transmitters*—i.e., instructions whose micro-architectural resource usage may reveal secret information [18, 21, 52]. While there are many types of transmitters, the most important one is loads, which, depending on the address they read, exercise different parts of the memory hierarchy.

A central idea in defense schemes against speculative execution attacks is an instruction's *Visibility Point* (VP) [48]. An instruction reaches its VP when it is no longer vulnerable to pipeline squashes that are relevant to the threat model considered. For example, assume a threat model based on Spectre [23] and that transmitters are loads. A load reaches its VP when it can no longer be squashed by any branch misprediction—i.e., when all of its older branches are resolved.

Each instruction transitions from being pre-VP to reaching its VP, and then to becoming post-VP. A transmitter cannot safely execute before it reaches its VP. In the example above, we can prevent the load from executing by inserting a fence before it. Sometimes, a defense scheme provides special protection that allows a pre-VP transmitter to execute. For example, with the InvisiSpec scheme [48], pre-VP loads can be issued invisibly, but need to be followed by a second access later on.

When the transmitter reaches its VP, it can execute without protection. In the example above, once all the branches older than the load are resolved, we can remove the fence. As instructions reach their VPs and execute, they enable younger instructions to

reach their own VPs. Hence, we intuitively say that "the older instructions pass the VP downstream."

The stall or protection of pre-VP transmitters slows down program execution over a conventional, unsafe processor. For instance, in the examples above, pre-VP loads are either delayed by fences or have to be issued twice.

The more aggressive the threat model is, the more costly protecting pre-VP instructions becomes. Consider the Comprehensive threat model [53], where a load L reaches its VP only when it can no longer be squashed for any reason. In this model, reaching the VP requires that: (i) all branches older than L are resolved; (ii) neither L nor any older instruction can suffer exceptions; (iii) there is no unresolved older load or store that L or an older load could alias with; and (iv) neither L nor any older load can cause a memory consistency violation (MCV). Schemes that protect L until all of these conditions are true have substantial overhead.

Based on this discussion, a new approach to reduce the overhead of defense schemes against speculative execution attacks could be to try to *speed-up the advance of the VP* toward young instructions. If a technique could be found to do this, then potentially all the defense schemes could have lower overhead.

In this paper, we propose such a technique, which we call *Pinned Loads*. To conceive it, we first examine, under the Comprehensive threat model, the delay induced to the VP advance by each of the four conditions listed above. We use the Comprehensive model because it is the most general one and covers recent attacks, including MCV-based attacks [29, 37]. In our analysis, we find that what delays VP progress the most is ensuring that no MCV is possible. This condition, therefore, adds the most overhead to safe program execution.

Based on this observation, we design *Pinned Loads* as a microarchitecture that tries to make loads invulnerable to MCVs as early as possible, therefore speeding-up VP progress. In our design, we assume the TSO memory consistency model [36, 44]. Recall that, under TSO, a load is conservatively flagged as causing an MCV and squashed when the core receives a coherence invalidation for the line accessed by the load or when the line is evicted from the cache. Hence, given a load L that has met all the conditions required to reach the VP except for the guarantee of no MCVs, *Pinned Loads* tries to ensure that no invalidation or eviction of L's line is possible anymore. If *Pinned Loads* can ensure this, we say that it *pins* L in the reorder buffer—making L unsquashable and moving L to its VP. If we manage to do this for many loads, the VP makes fast progress and the execution speeds-up.

In this paper, we describe the hardware needed for *Pinned Loads*. Further, we propose two possible designs of *Pinned Loads*, which offer different tradeoffs between hardware requirements and delivered performance. Finally, we extend several popular defense schemes for speculative execution with *Pinned Loads*. As we run the SPEC17, SPLASH2, and PARSEC benchmark suites with them, we observe a substantial reduction of their execution overhead. Indeed, the average execution overhead of defense schemes that (i) either place fences before loads, (ii) or stall speculative loads that miss in the L1 (Delay-On-Miss [26, 33]), (iii) or stall speculative loads whose arguments are tainted (STT [52]) decreases by about 50%. Specifically, on SPEC17, *Pinned Loads* decreases the execution overhead of the fence-based defense from 112.6% to 51.3%, of Delay-On-Miss from 35.8% to 15.3%, and of STT from 24.8% to 13.2%; on SPLASH2/PARSEC, *Pinned Loads* decreases the execution overhead of the defense schemes from 113.1% to 46.4%, from 15.8% to 7.6%, and from 11.3% to 8.1%, respectively.

In summary, the paper makes the following contributions:

- Introduces *Pinned Loads*, a novel technique to reduce the overhead of speculative-execution defense schemes by speeding-up VP progress.
- Presents the mechanisms behind *Pinned Loads*.
- Describes two different designs of *Pinned Loads*.
- Evaluates multiple *Pinned Loads*-extended popular defense schemes on an extensive application set.

2 BACKGROUND

Speculative Execution and Pipeline Squashes. In out-of-order processors, some instructions may execute but later get squashed and not commit. These bound-to-squash instructions are called transient instructions. Some of these transient instructions can create micro-architectural resource usage that may reveal secret information [18, 21, 52]. They are called transmitters. For example, loads exercise different parts of the memory hierarchy depending on the address they read, some floating-point instructions take different times to execute depending on their operand values, and different ALU instructions use different functional units. Attackers use the side-effects of transmitters to mount speculative-execution attacks. In this paper, we focus on loads because they are the most important type of transmitter.

There are multiple reasons why an instruction may be squashed in a modern pipeline. Which speculative threat model we use determines which reasons for squashes are considered relevant. For example, if we use the popular but weak Spectre threat model [23], we only need to consider squashes due to control-flow mispredictions. Alternatively, if we use the Comprehensive threat model [53], we need to consider all possible sources of squashes.

Section 1 defined the Visibility Point of an instruction, and listed the conditions required for a load to reach its VP under the Spectre and Comprehensive threat models. In practice, in conventional processors, by the time a load reaches its VP under the Comprehensive model, the load is very close to the Reorder Buffer (ROB) head.

Memory Consistency Violations (MCVs). In a multiprocessor system, the memory consistency model defines the order in which a processor's loads and stores are observed by other processors. When a store retires from the pipeline, its data is deposited into the write buffer. From there, when the memory consistency model allows, the data is merged into the cache, making it observable by all the other processors. In this paper, we say that a store is performed when its data is merged into the cache; we say that a load is performed when it receives its data. In conventional, unsafe processors, loads can read from the memory hierarchy and be performed before they reach the ROB head, and even out of order—i.e., before older loads and stores in the ROB are performed. These out-of-order loads can lead to memory consistency violations (MCVs) if another processor observes an order not allowed by the memory consistency model.

A processor recovers from an MCV by using the instruction squash and rollback mechanism of speculative execution [13]. We discuss how this is done for the Total Store Order (TSO) memory consistency model [36, 44], which is the one used by the x86 architecture and assumed in this paper. TSO forbids load to load reorderings (load→load), which is when a younger load is performed before an older load to a different address. Implementations of TSO prevent observable load→load reordering by ensuring that the value that a load reads when it is performed remains valid when the load retires. This guarantee is conservatively maintained by squashing a load that has performed, but not yet retired, if the processor receives a cache invalidation for the line read by the load. Moreover, the load is also squashed if the line read by the load is evicted from the cache before the load retires—since, on a subsequent external write, the cache may not receive an invalidation.

Strictly speaking, cache line invalidations and evictions do not need to squash the oldest load in the pipeline—since such load has not been reordered. A reorder can only occur when the line in question has been read by a load L that is not the oldest load in the pipeline, and then the hardware needs to squash L and all its successor instructions. This is the design that we use in our evaluation. However, for ease of explanation of our mechanism, we discuss the simpler implementation where any load that has read a line that is invalidated or evicted triggers a squash. This is the implementation used in Intel processors [29].

TSO also forbids load to store (load→store) and store to store (store→store) reorderings. Implementations of TSO prevent them by merging a store with the cache hierarchy only after the store instruction has retired, and by using a FIFO write buffer, ensuring that stores are drained in program order.

3 PINNED LOADS: MAIN IDEA & IMPACT

3.1 Advancing the VP is Crucial

The Visibility Point (VP) is an important concept in defense schemes against speculative execution attacks. When an instruction reaches its VP, it becomes safe to execute without any protection. Consider a load, which is the focus of this paper. If the baseline defense is to place a fence before a load, then when the load reaches its VP, the fence can be removed. If the defense is to issue the load early invisibly, followed by a second access later [48], when a load reaches its VP, it is unnecessary to issue the load twice anymore.

The conditions that determine when a load reaches its VP depend on the threat model used. However, it is evident that any technique that can help a load reach its VP sooner will help speed-up execution under practically any defense scheme against speculative execution attacks: loads will execute sooner or with lower overhead, and will in turn enable subsequent loads to reach their VPs sooner. Intuitively, the hardware will be "moving the VP to younger loads" faster.

There are some defense schemes that, using certain assumptions, allow the unprotected issue of some loads that have otherwise not reached their VP—e.g., loads that have reached the Execution Safe Point in InvarSpec [53] or loads whose arguments are not tainted by transiently-read data in STT [52]. Even in these cases, enabling loads to reach their VP sooner is useful: the conditions that enable such unprotected early load execution depend on older loads actually reaching their VP. In this paper, to keep the discussion simple, we will not discuss such "early safe" loads.

3.2 Focus on Memory Consistency Violations

For the Comprehensive threat model, Section 1 listed the four conditions necessary for a load to be free of potential squashes and, therefore, reach its VP. To design an effective "VP-advancing" technique, we need to understand how performance-limiting each of these conditions is in practice. To this end, we take a processor that places a load-stalling fence before each load, and consider four possible times when to remove the fence—from typically earlier to later times. The times are when no squash is possible due to: (i) branches (*Ctrl Dep*), (ii) branches or aliasing (*Alias Dep*), (iii) branches, aliasing, or exceptions (*Exception*), and (iv) branches, aliasing, exceptions, or memory consistency violations (*MCV*). Figure 1 shows the resulting execution overhead of the environments (in a stacked manner) over a conventional unsafe processor. The processor is the one shown in Table 1, and the programs are those in the SPEC17 [8] suite (single-threaded), and in the SPLASH2 [47] and PARSEC [6] suites (with eight threads).

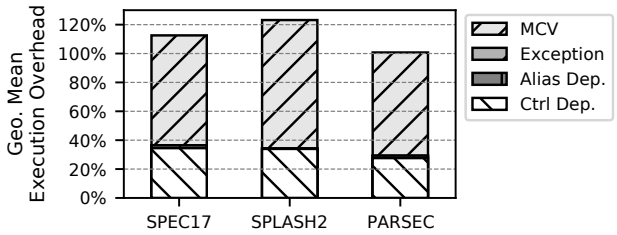

Figure 1: Effect of each reason that delays reaching the VP.

Of all the conditions, ensuring that no MCV is possible is, by far, the one that delays reaching the VP the most and, therefore, slows down execution the most. Waiting for branch resolution also has a substantial, yet smaller impact, while waiting for alias resolution and exception-free state is much less significant. Overall, as intuition suggests, while the absence of squashes due to *Ctrl Dep*, *Alias Dep*, or *Exception* is determined relatively soon, the absence of squashes due to *MCV* remains unresolved until the load is close to the head of the ROB. Hence, this condition substantially delays "moving the VP downstream" and, therefore, slows down program execution.

For this reason, in this paper, we focus on making loads invulnerable to MCVs as early as possible. We pick a load L that has met all the conditions to reach its VP except for guaranteeing that L will not cause MCVs. Then, our goal is to *Pin* L in the ROB—i.e., to declare it unsquashable due to MCVs and hence declare that it has reached its VP—as early as possible.

Recall from Section 2 that a load is conservatively identified as causing an MCV and squashed when the core receives a coherence invalidation for the line accessed by the load or when the cache evicts this line. Hence, to declare load L as *Pinned*, the hardware needs to guarantee that none of these two events will occur for L.

Our proposed architecture, called *Pinned Loads*, guarantees it as follows. First, to guarantee that there will be no squash of L due to invalidations, *Pinned Loads* delays incoming invalidations to the

line read by L until L retires. Since invalidations can only be delayed for a limited time period, the hardware also has to guarantee that L will eventually reach the ROB head and retire—at which point, no more invalidation delays will be needed. Consequently, *Pinned Loads* can only declare L as pinned if the core has enough resources to retire L and all the instructions older than L in the pipeline—in particular, the write buffer needs to have enough entries to fit all the yet-to-complete stores older than L.

Second, to guarantee that there will be no squash of L due to cache evictions, *Pinned Loads* pins only loads that access cache lines that it can guarantee are non-evictable. To obtain such a guarantee, *Pinned Loads* needs to reserve, for each core, a minimum number of lines W_d per set in the directory plus last-level cache (LLC). Furthermore, *Pinned Loads* knows the associativity W_{L1} of the L1 cache. With this information, *Pinned Loads* only declares L pinned if the lines accessed by L and by the set of already-pinned older loads: (i) do not overflow W_d for any directory plus LLC set, and (ii) do not overflow W_{L1} for any L1 cache set. In addition, *Pinned Loads* refuses to evict from its L1 cache and from the directory plus LLC any line that has been accessed by a currently-pinned load. Such eviction request may be a self eviction initiated by the local processor or a cross eviction initiated by another processor.

To keep the design simple, *Pinned Loads*: (i) pins all the loads that will eventually retire and (ii) does it in strict program order. Further, no load can be pinned before it has generated its address, since it can suffer an exception during address translation. After address translation, we assume the load cannot suffer exceptions.

3.3 Potential Performance Gains

To understand the performance gains enabled by *Pinned Loads*, consider a ROB with three independent loads. Recall that we assume a baseline processor implementation where even the oldest load in the ROB can suffer an MCV. Figure 2(a) shows the behavior of the conventional, unsafe processor. As denoted by the arrows, all three loads can be issued to memory in parallel. Figure 2(b) shows the behavior of a safe processor. In this case, a load can only be safely issued when it reaches its VP. Generally, this occurs when the load is close to the ROB head. The result is poor performance, as loads are issued late and only one load can be in progress at a time.

Consider now a safe processor augmented with *Pinned Loads*. We propose two designs, which will be detailed later. To understand them, consider a load L that has met all the conditions to reach the VP except for guaranteeing no MCVs. Our first design (*Early Pinning*) has special hardware that determines whether there is enough space in the cache hierarchy and directory to hold the line that L requests—given that there may already be other pinned loads. If the answer is yes, *Pinned Loads* declares L pinned even before issuing L to memory, and "passes the VP downstream." Our second design (*Late Pinning*) is simpler and has no such special hardware. In this design, L is first issued to memory. If L successfully brings the data to the L1 cache, hence proving that directory and caches have space for the line, *Pinned Loads* declares L pinned and "passes the VP downstream." These two designs offer different tradeoffs between hardware requirements and performance.

The behavior of the safe processor augmented with Late Pinning is shown in Figures 2(c)-(e). As shown in Figure 2(c), the oldest

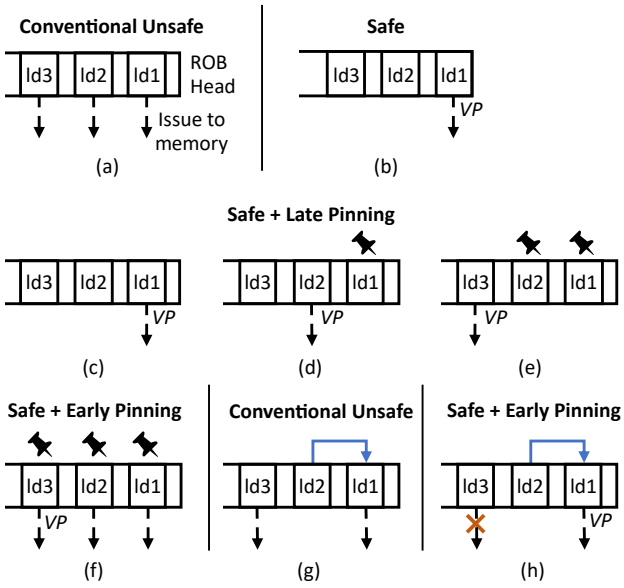

Figure 2: Overlapping of loads in the reorder buffer (ROB).

load reaches its VP (and issues to memory) *earlier* than in the safe processor: when only an MCV could squash it. However, in reality, no MCV-induced squash will occur: while the data has not returned, no MCV can occur by construction; and as soon as the data arrives, *Pinned Loads* will pin it and hence ensure no MCV can occur. Assume that, when the oldest load gets pinned, the second load reaches its VP. The second load then issues to memory (Figure 2(d)) and, on reception of the line, gets pinned. The process repeats for the third load in Figure 2(e). We see that, while loads are not issued in parallel as in the unsafe processor, they are issued much earlier than in the safe processor.

The behavior of the safe processor augmented with Early Pinning is shown in Figure 2(f). In this design, *Pinned Loads* can pin a load (and enable the next load to reach its own VP) even *before* the load issues to memory. Hence, as shown in the figure, the VP "is passed downstream" quickly and all the loads proceed in parallel. The result is safety and high performance.

For completeness, Figure 2(g) shows the case when the second load is dependent on the first one. The unsafe processor can issue the first and third loads in parallel. However, even the Early Pinning design cannot match the performance of the unsafe processor. Indeed, the second load's address depends on the return value V of the first load. Hence, the second load cannot be declared pinned until V is known and the load's address is translated—since there is a risk of an address translation exception. Since the second load is not pinned, the third one, although independent, cannot be claimed as pinned and issue (Figure 2(h)).

If *Pinned Loads* is able to remove most of the stall due to MCVs in Figure 1, the resulting performance may be close to that of a processor only stalling for branch resolution. In that case, the performance of a safe processor under the Comprehensive threat model would be close to that of a safe processor under the Spectre threat model.

If the processor supports the more aggressive implementation of TSO described in Section 2, where cache line invalidations and evictions do not squash the oldest load in the ROB, a more aggressive design of Late Pinning is possible. Specifically, as soon as the oldest load (i.e., *ld1* in Figure 2(c)) is free of all the other sources of squashes (i.e., branches, aliasing, and exceptions), it issues and "passes the VP downstream"—since it cannot be squashed anymore. Hence, *ld2* can be issued while *ld1* is still outstanding. Furthermore, when *ld2* receives the data, it gets pinned, and *ld3* can be issued even if *ld1* is still outstanding. Overall, while the safe baseline can have only one outstanding load, this more aggressive design of Late Pinning can support two outstanding loads, as long as one of them is the oldest one in the ROB—in addition to supporting the sequential issue of multiple loads much earlier, as indicated before. This is the design we use in the evaluation.

4 THREAT MODEL

We assume the Comprehensive threat model and various baseline hardware defense schemes that *Pinned Loads* augments for performance. The Comprehensive model is necessary to cover recent attacks, including attacks related to memory consistency [29, 37] (Section 10). Examples of baseline schemes that *Pinned Loads* can augment are those that protect pre-VP loads with blocked execution [4, 45, 52], execution only if they hit in the L1 [26, 33], or invisible execution that does not change the state of the cache hierarchy [1, 20, 48].

Pinned Loads does not modify the speculative execution security properties of the baseline defense schemes. The reason is because *Pinned Loads* does not modify the definition of VP; it simply enables loads to reach their VPs earlier.

Pinned Loads does not add new speculative side or covert channels. The reasons are: (i) a load is pinned only if it satisfies all the conditions for reaching its VP except for the possibility of causing an MCV, and (ii) a pinned load is guaranteed not to cause an MCV. These combined properties imply that, once a load is pinned, its retirement is guaranteed, and so any side-effects of its execution cannot result in *speculative leakage*. In particular, this argument applies to any new side-effects introduced by *Pinned Loads* itself (e.g., changes in the processing of coherence invalidations or evictions). These side-effects are only a function of the pinned load's operands, and thus do not leak *speculative* information.

Pinned Loads does not address *non-speculative* side channels. It is well known that, in a multi-threaded shared-memory environment, an attacker can exploit cache coherence states for timing-based non-speculative side channels [50].

5 DESIGN OF PINNED LOADS

In this section, we describe the *Pinned Loads* design and present the Late and Early Pinning variations. In Sections 6 and 7, we outline some implementation aspects, and compare to a related design. In the following, we refer to a line that is accessed by a currently-pinned load as a *pinned line*.

At its core, *Pinned Loads*: (i) delays incoming invalidations to pinned lines and (ii) prevents cache evictions of pinned lines. In addition, it has to ensure that the processor has enough resources to pin a load—i.e., enough write buffer entries for yet-to-complete stores, and enough cache and directory space for all the pinned lines. Finally, it has to ensure that delayed stores make progress.

Note that *Pinned Loads* never pins loads younger than in-ROB MFENCE or LOCK instructions because doing so would be incorrect. For example, pinning a load before an older lock is acquired would be equivalent to binding the value returned by the load before the lock is acquired.

5.1 Pinned Loads Mechanisms

5.1.1 Delaying Invalidations to Pinned Lines. Processors that support TSO [36, 44] conservatively avoid MCVs by squashing a yet-to-retire load issued by the processor when the L1 cache receives an invalidation for the line read by the load. When an invalidation is received in L1, the Load Queue (LQ) is snooped and, on finding a matching entry, the corresponding load and its successor instructions are squashed.

Pinned Loads keeps a record of the pinned lines. Such a record only requires one bit in each LQ entry, although other designs are possible (Section 6.1). When an invalidation arrives and the LQ snoop finds it is directed to a pinned line, the hardware denies the invalidation.

Supporting this functionality requires a modification to the write transaction of the cache coherence protocol. Figures 3(a) and (b) show the conventional and the *Pinned Loads* write transaction, respectively. In the figure, Core 1 has brought the line to its L1 cache in state shared (S) with a yet-to-retire load, and Core 2 issues a write to the line (arrow ⓪). In the conventional transaction (Figure 3(a)), Core 2 issues a GetX request to the directory (①). The directory returns the line plus the number of sharers to Core 2 (②), sends an invalidation to Core 1 (②), and enters a transient state that rejects other requests to the line. Core 1 invalidates its local copy of the line, snoops its LQ, squashes its load to the line, and sends an ack to Core 2 (③). Core 2 then sends an Unblock request to the directory (④), which exits the transient state and updates the sharers.

In the *Pinned Loads* transaction, when Core 1 receives the invalidation (②), the hardware snoops the LQ before invalidating the cache line. On finding a match with a pinned line, the cache is not invalidated, the load is not squashed, and a Defer message is sent to Core 2 (③). If Core 2 receives a Defer from any sharer of the line, it aborts the write and sends an Abort to the directory (④). The latter exits the transient state and does not change the sharer bits. Core 2 will now retry the write. To ensure that Core 2 is able to eventually write, additional support is added in Section 5.1.5.

5.1.2 Ensuring Enough Write Buffer Entries. Delaying invalidations is a temporary mechanism applied until the pinned load reaches retirement. Hence, before *Pinned Loads* marks a load L as pinned, it has to ensure that there are enough resources for L to reach retirement. One obvious resource required is related to stores: there need to be enough write buffer entries to be able to hold all the yet-to-complete stores that are older than L. This includes stores already in the write buffer and stores not yet in the write buffer. The reason is that, for L to retire, all of its older stores should be pushed into the write buffer.

If this condition is unmet, deadlock may ensue. To see why, consider the two cores in Figure 4. Core 1 has retired a store to line x to its write buffer. Its ROB contains another store and then a

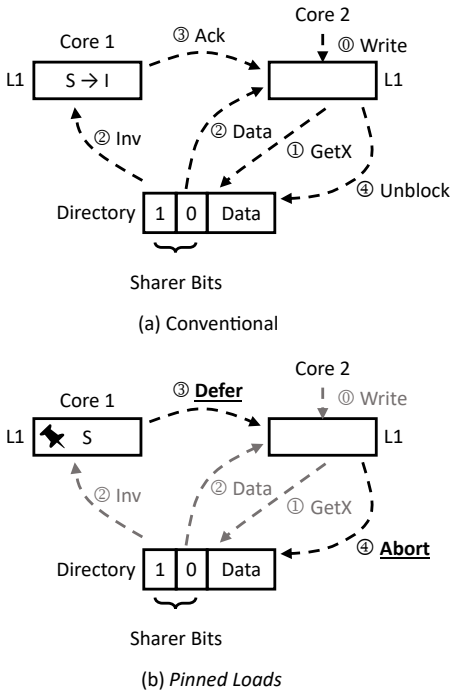

Figure 3: Conventional (a) and *Pinned Loads* (b) write transaction.

pinned load to line y. Core 2 has retired a store to line y to its write buffer. Its ROB contains another store and a pinned load to line x. Assume that *ldx* and *ldy* have loaded their data to the L1 caches, and that the write buffers can hold a single write. In the write buffer of Core 1, *stx*'s attempt to write is denied by Core 2 because line x is pinned by *ldx* (①). Similarly, in the write buffer of Core 2, *sty*'s attempt to write is denied because *ldy* is pinned. To make forward progress, either *ldx* or *ldy* have to retire. Load retirement would remove the pin, which would in turn allow the write in the other core to succeed, and execution to proceed. However, no load can retire (②): both ROBs have an older store that cannot leave the ROB because the write buffer is full.

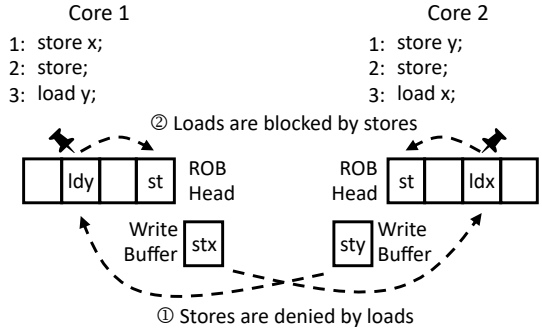

Figure 4: Deadlock due to insufficient write buffer entries.

To prevent this deadlock, before *Pinned Loads* declares a load pinned, it counts the number of yet-to-complete stores older than the load (already in the write buffer or not). The load is not pinned while such count is higher than the number of write buffer entries.

5.1.3 Preventing Evictions of Pinned Lines. Processors that support TSO also conservatively avoid MCVs by squashing a yet-to-retire load when the L1 cache wants to evict the line read by the load. In *Pinned Loads*, the hardware prevents pinned lines from being evicted from L1.

The process is similar to how *Pinned Loads* denies invalidations in Section 5.1.1. Specifically, when the L1 wants to evict a line, the hardware checks whether the line is pinned. The record of what lines are pinned can be kept in the LQ, as described in Section 5.1.1, or in the L1 tags, as we will see in Section 6.1. If the line is found pinned, the eviction is denied. Then, the cache controller updates the replacement algorithm state as if the line had been accessed (to minimize future attempts to evict the line), and then selects a new victim from the same cache set.

The action of evicting a line from L1 may be initiated by a request from the local core, which may need to allocate space in any cache, or from another core, which may need to allocate space in the shared cache. Moreover, it may occur with inclusive, non-inclusive or exclusive cache hierarchies, and with different directory organizations.

Note that this mechanism is not unusual: conventional cache hierarchies sometimes need to deny cache line evictions, as is the case when the victim cache line is in a transient state. More details are given in Section 6.1.

5.1.4 Guaranteeing Space in Cache & Directory. A core cannot pin any number of cache lines. The number of pinned lines that map to a set in a private cache or to a set in a shared directory/LLC cannot be bigger than the associativity of these structures: all the pinned lines need to remain in the caches or directory/LLC, respectively. Consequently, before *Pinned Loads* declares a load L pinned, it has to ensure that the lines accessed by all the currently-pinned loads plus L can co-exist in the private caches and in the shared directory/LLC.

One approach to ensure that L can be pinned is to issue it first and observe whether it attains the cache and directory space needed; if so, it gets pinned. Another approach is to only issue and pin L if *Pinned Loads* can first guarantee that there will be space.

For this second approach, let us assume an inclusive cache hierarchy with private L1 caches and a shared L2 LLC with the directory. We discuss other cache hierarchy organizations in Section 6.2. In this case, *Pinned Loads* needs to know the associativity of L1 (W_{L1}) and, because the directory/LLC is shared by all the cores, the number of entries in each set of each directory/LLC slice that are reserved for each core (W_d). In addition, *Pinned Loads* needs to know the mapping of line addresses to sets in L1 and to slices and sets in the directory/LLC. Finally, *Pinned Loads* needs to have a small hardware-managed table that records, for each pinned load L, the L1 set and the directory/LLC slice and set where the line accessed by L maps. This table is called the *Cache Shadow Table* (CST) and is discussed in Section 6.2.

In this second approach, when *Pinned Loads* wants to pin load L, it first determines the L1 set and the directory/LLC set and slice where the line maps. Then, it accesses the CST and determines whether such sets and slice can hold one additional pinned line—i.e., whether with the addition of L, no more than W_{L1} and W_d pinned lines map to the same set in L1 and directory/LLC, respectively. If

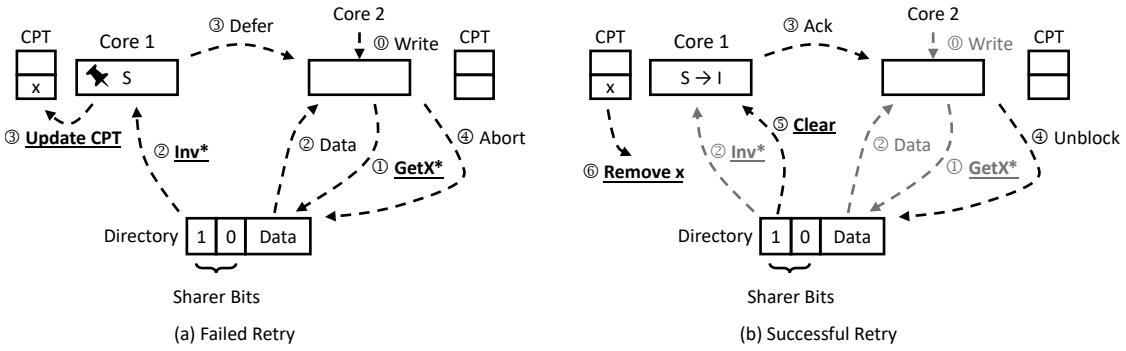

Figure 5: Mechanism to prevent store starvation. In the figure, *x* is the address of the line that the store is trying to update.

these conditions are all met, *L* is declared pinned; otherwise pinning needs to wait.

5.1.5 Preventing Store Starvation. Figure 3 showed that when a core pins a cache line, a write by another core is denied, and the hardware in the writer core has to keep retrying. Unfortunately, it is possible that the reader core (and other additional cores) keep reading and pinning the line. Since the decision to pin a line is made locally in each core, the readers may never know that there is a writer that is starving.

To avoid starvation, *Pinned Loads* uses the following idea: if a write request is denied, its retry works in a slightly different way, which prevents the indefinite repeated pinning of the line by other cores before the write succeeds. To support this idea, *Pinned Loads* adds a small hardware table in each core called the *Cannot-Pin Table (CPT)*. The CPT records the lines that the core cannot pin at the moment.

Figure 5 illustrates how the algorithm works. After the first write by Core 2 in Figure 3 was denied, Core 2 now retries with a new variant of GetX called GetX* (①in Figure 5(a)). After the directory receives GetX*, it sends a special invalidation, Inv*, to Core 1 and all the other current sharers (②). Upon receiving Inv*, Core 1 and all the other sharers add the address of the line (x) to their CPTs (③), meaning that they will not be able to pin the line again until the write succeeds. The sharers then reply to Core 2: if a sharer has the line pinned (as in Core 1), the sharer replies Defer to Core 2 (③); otherwise it replies Ack to Core 2 and invalidates its copy of the line. If Core 2 receives at least one Defer, it knows the line is pinned; hence it sends an Abort to the directory (④), which does not change its state. To minimize hardware modifications, the directory is not modified to record that a write is being denied.

From now on, none of the cores with x in their CPTs can pin the line—although they can read it. Other cores can still read the line and pin it. However, every single retry of the write will insert x in the CPTs of the sharers. In the worst case, all cores but the writer end up with x in their CPTs.

Eventually, all the reader cores will retire the pinned loads, and a retry by the writer will find that all the responses are Ack and there is no Defer (Figure 5(b)). Such Acks come from all the sharers recorded in the directory. The write has now succeeded. Hence, Core 2 sends the Unblock message to the directory (④). On reception of the Unblock message, the directory sends an extra Clear request to all the sharers (⑤) so they remove x from the CPT (⑥), and then updates the sharer information.

5.2 Late and Early Pinning Approaches

We propose two variations of *Pinned Loads* that offer different tradeoffs between hardware requirements and performance: *Early* and *Late Pinning*. In both designs, loads are pinned in program order, all loads that will eventually retire are pinned, and a load can only be pinned when it has met all the conditions to reach its VP except for guaranteeing no MCV.

5.2.1 Late Pinning (LP). This design does not include the CST of Section 5.1.4. A core does not know, at the point of issuing a load, whether the private caches and the shared directory/LLC will have space to hold the line—given all the older pinned loads. Hence, when a load meets all the conditions to reach the VP except for guaranteeing no MCVs, and *Pinned Loads* concludes that there are enough write buffer entries, *Pinned Loads* issues the load. If the core receives a response with the data, it means that private caches and shared directory/LLC have the space for the line; then, *Pinned Loads* declares the load pinned. Otherwise, *Pinned Loads* has to wait until the line can be loaded to declare the load pinned.

This design has two advantages. First, it is simpler because it has no CST. Second, cores can ignore the limitation of only pinning at most W_d lines per set and slice in the shared directory/LLC. A core can issue many loads that attempt to allocate lines in the directory/LLC; if they succeed, the loads are declared pinned. It is possible that a core ends up pinning more than its share of lines in a given directory/LLC set. Such a situation often improves performance and only infrequently ends up temporarily starving other cores.

This design's shortcoming is that the load's response from the memory system is in the critical path of declaring the load pinned and, hence, of "passing the VP to the next load". The result is that all loads access the memory system sequentially (Figures 2(c)-(e)), even if they are independent. Hence, performance is low in programs with bunched-up cache misses.

5.2.2 Early Pinning (EP). This design includes the CST. When a load L meets all the conditions to reach the VP except for guaranteeing no MCVs, and *Pinned Loads* ascertains that there are enough write buffer entries, the CST is checked. If the CST decides that the new line will find space in the private caches and the shared

directory/LLC, L is declared pinned and the VP is "passed down"—potentially even before issuing L to memory.

The pluses and minuses of this design are the opposite of those of the previous one. The advantages are that independent loads are issued to memory with great parallelism (Figure 2(f)), even out of order, and that the VP "is passed to younger loads" faster. The result is high application performance. Recall, however, that if a load cannot be issued to memory due to a dependence, then subsequent, independent loads cannot be issued to memory either (Figure 2(h)).

The shortcomings of this design are the need for the CST hardware and the fact that a core will not attempt to pin more than its W_d share of lines per slice and set in the directory/LLC. This is because loads are pinned before being issued, and hence *Pinned Loads* has to guarantee space for their data in advance.

Note that the assignment of W_d maximum *pinned* lines per slice and set in the shared directory/LLC to each core is an agreement among cores; it does *not* require fixed set partitioning of the directory/LLC. Also, once the load that pinned a line retires, the line gets unpinned, and the line can remain in the directory/LLC for potential future use without counting toward the W_d maximum pinned lines allocated to the owner core.

6 KEY IMPLEMENTATION ASPECTS

In this section, we describe the implementation of key aspects of the *Pinned Loads* hardware.

6.1 Recording Pinned Lines

Pinned Loads provides hardware to record the currently-pinned lines. On any attempt to invalidate or evict a line, the hardware is checked to either allow or prevent the operation. Note that such hardware can be placed very close to the core. The reason is that, if a pinned load has already obtained its data, it has brought the line to L1, and any invalidation or eviction request for the line will reach L1. Alternatively, if the pinned load has not brought the line to L1 yet (which may happen in Early Pinning), since the load has not consumed the data yet, the consistency model does not squash the load on invalidation or eviction of the line from other cache levels. We present two possible designs to record pinned lines. Our chosen design is the first one.

6.1.1 Storing the Information in LQ. This design adds one Pinned bit to each LQ entry, indicating whether the load is pinned. When a load gets pinned, the core sets the bit in the load's LQ entry. When the L1 receives an invalidation or attempts to evict a line, the LQ is checked. If the hardware finds a matching entry with the Pinned bit set, the operation is denied. When a pinned load retires, it trivially becomes unpinned.

Most of the mechanisms in this design are already present in conventional processors. For example, in conventional processors, when a line is to be evicted from a cache level, the hardware informs higher levels of caches (i.e., smaller caches) so they also evict the line—with some variations depending on whether or not the cache hierarchy is inclusive. In some proposals, the higher levels may refuse to evict the line for performance or security reasons, prompting the initiating cache level to find another victim [19, 49]. *Pinned Loads* uses the same approach for pinned loads.

In conventional TSO cores, when the L1 wants to invalidate or evict a line, the hardware checks the LQ and, on a match, the corresponding load and its subsequent instructions are squashed. In *Pinned Loads*, the process is different in two ways. First, the invalidation or eviction may be denied. Second, the LQ check and the invalidation/eviction cannot happen in parallel: the check has to be done first in case the operation is denied.

6.1.2 Storing the Information in L1 Tags. This design adds a Pinned bit to each cache line in L1, to indicate whether or not the line is pinned. At runtime, when a load is to get pinned, *Pinned Loads* accesses the L1 and sets the Pinned bit of the line. When an L1 line receives an invalidation or is picked for eviction, if its Pinned bit is set, the operation is denied.

This design still keeps the Pinned bit in the LQ entries. Recall that a load can become pinned only after all of its older loads are pinned; hence the presence of the Pinned bit in the LQ enables the hardware to find this condition easily. In addition, LQ entries need one additional bit: the *Youngest Pinned Load* (YPL) bit. To understand its functionality, consider multiple pinned loads in the LQ that are accessing the same line. Only when the youngest of them retires can *Pinned Loads* clear the Pinned bit in the cache. Hence, for each pinned cache line, one of the LQ entries has the YPL bit set. When a new load is to be pinned, the hardware searches the LQ for an entry for the same line and the YPL bit set; if the entry is found, the hardware "passes the YPL bit" from the older to the newer entry and there is no need to set the Pinned bit in L1 cache again. When a pinned load with a set YPL bit retires, the L1 cache is accessed to clear the Pinned bit.

When using the Early Pinning of Section 5.2.2, a load may be declared pinned before the L1 receives the data. In this case, since the L1 does not have the line, we add a Pinned bit in the MSHR that the load uses. This is done as soon as a pinned load is issued. When the requested line is received and placed in the L1, the Pinned bit in the MSHR is copied to the L1.

The advantage of this design is that it decides whether to invalidate or evict an L1 line quickly, without waiting for an LQ access. This reduces the latency to respond to requests. However, a disadvantage is that this design requires extra requests from the pipeline to L1 to unpin lines. This fact puts extra pressure on L1 and increases the unpinning latency. Overall, because load pinning/unpinning operations are much more frequent than L1 invalidations or evictions, we do not use this design.

6.2 Optional Cache Shadow Table (CST)

The CST is a per-core hardware structure only used in Early Pinning. It records the mapping of each line pinned by the core—i.e., which set in L1 and which slice and set in the shared directory/LLC (Section 5.1.4). The hardware checks the CST before pinning a load to determine whether, with the addition of this load, all the pinned lines still have enough guaranteed space in the cache hierarchy.

A core has one CST for the directory/LLC and one for the L1 cache. Each CST is a hash table. Figure 6 shows the CST for the directory/LLC. Assume that *Pinned Loads* wants to pin a load L that accesses address A. First, *Pinned Loads* generates the set and slice numbers where A maps, hashes them, and uses the result to access a CST entry. An entry contains M records, each corresponding to a

line. Each record has the hash of the line address, the LQ ID of the youngest pinned load that reads from the line, and a Valid bit. M is equal to or less than the maximum number of lines that can be pinned by the core in the same set and slice (i.e., W_d in Section 5.1.4).

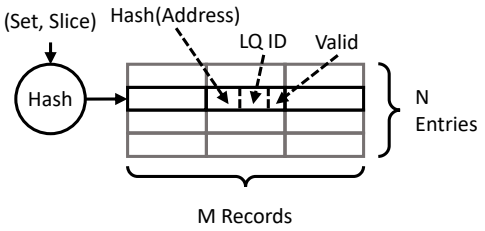

Figure 6: Cache Shadow Table (CST) for the directory/LLC.

At the indexed table entry, the hardware performs a CAM read to find if there is a valid entry for A. If there is, the line is already pinned by older loads, and hence the directory/LLC has enough resources to pin L. Then, the LQ ID field of the record is updated to L's LQ ID, and L is declared pinned.

If the CST entry does not contain a record for A, the hardware checks whether the entry has enough room for a new record. If so, a new record for A is created, its LQ ID field is updated to L's LQ ID, and L is declared pinned. Otherwise, the pinning is denied as there are not enough resources.

To reduce overhead, when a pinned load retires, we do not access the CST to potentially remove its entry. Instead, we let the potentially stale entry remain and remove it only when the hardware attempts to pin a new line. At that point, the hardware discovers if any of the records in the chosen entry has an LQ ID that is outside of the currently-used LQ entries. If so, the record is expunged.

One corner case that we handle is LQ ID wraparound, which could lead to using stale CST entries. We solve this problem by using a longer LQ ID tag in both the CST and LQ. For example, if the LQ has 64 entries, rather than using 6 bits for the LQ ID, we use 24 bits. Then, we use the modulo operation to map an LQ ID to a physical LQ entry. With this longer LQ ID tag, wraparound happens infrequently. When it happens, *Pinned Loads* stops pinning loads until all the pinned loads retire. During this time, loads reach their VPs and issue as they would on a safe scheme without *Pinned Loads*. Once all the pinned loads retire, the CST is cleared, and normal *Pinned Loads* execution resumes. Because wraparound is infrequent, the performance impact of this design is negligible. Other designs to handle LQ ID wraparound are possible.

The use of hashes in the CST may cause hash collisions. One class of collisions occurs when two different $\{set, slice\}$ pairs hash to the same CST entry. This is safe, as it only underestimates the capacity of a $\{set, slice\}$ combination. Another class of collisions occurs when the hashes of two different line addresses in a $\{set, slice\}$ match the same record. This collision needs to be detected. *Pinned Loads* detects it by always using the second field of the record (i.e., the LQ ID) to access the LQ entry and check whether the line address of the existing entry is indeed the same as the one we want to pin. If it is not, then we cannot pin the new load and *Pinned Loads* handles it as if there was not enough space in the $\{set, slice\}$.

The CST for the L1 cache operates similarly, except that there is no slice number, and that the number of records per entry M can be as high as the cache associativity.

Supporting Different Types of Cache Hierarchies. Cache hierarchies can have different organizations, which may affect how *Pinned Loads* designs its CSTs. In particular, cache hierarchies typically have multiple levels of private caches (e.g., an L1 and an L2 level). In nearly all cases, there is no need to have a CST for a private L2 cache. This is true if the L2 is exclusive with respect to the L1: whether the L1 has enough space to hold a line does not depend on L2's organization. It is also almost always true if the L2 is inclusive with respect to the L1: L1 caches typically have a set count and an associativity that are lower than or equal to those of the L2 caches. Hence, it is almost always the case that, if a line has space in L1, it also has space in L2. If such statement is untrue, *Pinned Loads* would need a CST for a private L2. Finally, if the L2 is non-inclusive with respect to the L1, a more subtle analysis of the data flows allowed is required to determine the CST needs.

6.3 Cannot-Pin Table (CPT)

The CPT is a per-core hardware structure that records the addresses of lines that the core is not allowed to pin at the moment (Section 5.1.5). The CPT is placed near the LQ, which checks it before attempting to pin a line. A line's address is inserted in the CPT when the core receives an Inv*; the address is removed when the core receives a Clear. Our CPT can hold up to four addresses although, on average (Section 9.2), it only needs to hold one. If the CPT fills up and a request to insert an address cannot be serviced, the core stops pinning loads until the CPT is half empty.

We expect that a core that tries to write to a pinned line like Core 2 in Figure 5(a) will eventually succeed in inserting an entry in the CPT(s) of the reader core(s). However, there is a corner case when every time that Core 2 attempts to write, the CPT(s) in the reader core(s) are full and do not accept new entries. In this very unlikely case, Core 2 would never succeed in inserting its entry in the CPT(s).

To prevent this case, a more advanced design can add a small FIFO queue to the CPT with the IDs of writer cores that visited the node but found no space in the CPT. Then, when one CPT entry is released, it is reserved for a write from the core whose ID is at the head of the queue.

6.4 Effect of Limited-Sized Hardware Structures

Pinned Loads uses certain key hardware structures such as the CST, CPT, and extended LQ ID tag. Their limited size may sometimes cause *Pinned Loads* to operate with slightly lower performance, but never incorrectly. Specifically, when the CST cannot find space to pin a load, either because there is no space or because of a hash conflict, the core stops pinning loads until space can be found. Similarly, when the CPT fills up, the core stops pinning loads until the CPT is half empty. Finally, when LQ ID tag wraps around, the core stops pinning loads until all the pinned loads retire. In all cases, in the meantime, loads reach their VPs and issue as they would on a safe scheme without *Pinned Loads*. The execution is not as fast but it is correct and does not have deadlocks or livelocks.

7 COMPARISON TO A RELATED SCHEME

Our design to guarantee early that a load will not cause MCVs uses a mechanism to temporarily delay invalidations to a line. Ros et al. [31] proposed a mechanism with a similar goal in their WritersBlock protocol. Their purpose was to improve performance by allowing load-load reordering in TSO without squashes. Later, Tran et al. [39] applied the design to a speculative processor to allow loads to execute early without risking MCVs—the same goal as *Pinned Loads*.

We did not want to use Tran et al.'s aggressive design because its hardware is complex. In this section, we compare *Pinned Loads* to their design.

In the WritersBlock protocol, any load that has been issued speculatively causes its core to (i) reject an incoming write to the line and (ii) send a request to the directory, causing the directory to enter a new transient state for the line called WritersBlock. The rejected write is buffered and blocked in the directory. Other readers that arrive to the directory while in WritersBlock state can read the data. However, to prevent starvation, they get a "tear-off" copy of the data: a copy that is uncacheable, does not get recorded in the directory, and can be used only once. Moreover, a directory entry in WritersBlock state cannot be evicted from the directory. Hence, if a read for a different line arrives to the directory/LLC and cannot allocate space because it would have to evict WritersBlocked entries, the read gets a tear-off copy of the data from main memory and does not allocate a directory entry.

This is an aggressive design that allows any speculative load in any order to get the data—irrespective of how many older loads exist in the ROB, and without needing to guarantee that there is space to hold the line in caches or directory. However, the hardware changes required are major: a new transient directory state, which buffers the write and allows reads; writes that transition between three- and four-hop transactions; reads that dynamically turn uncacheable based on directory state; and new read transactions that directly grab data from memory and skip directory entry allocation. The result is challenging hardware. Perhaps more importantly, adding these no-directory-allocation and uncached paths creates *parallel paths* for transactions, namely cached and uncached paths, which are hard to verify for correctness.

In contrast, with *Pinned Loads*, we seek a simpler and safe design. A key source of complexity reduction is that *Pinned Loads* pins all the loads of a core *in strict program order*, and only when cache and directory *resources are guaranteed*. Further, we limit the complexity added to the coherence protocol as much as we can: we add no new directory states; we create no uncached or no-directory-allocation paths in the protocol; the directory buffers no new state; to attempt to write to pinned lines, we reuse processor retry mechanisms that have been used commercially to access busy directory lines; and, generally, we minimize the changes made to the directory and LLC, moving some functionality to structures that are local to cores.

8 EXPERIMENTAL METHODOLOGY

We model the architecture shown in Table 1 using cycle-level simulations with gem5 [7]. In the simulator, we model all the side effects of transient instructions. The baseline architecture is called Unsafe, because it has no protection against speculative execution attacks. We use loads as transmitters and model the Comprehensive [53] and Spectre [23] threat models. In Comprehensive, squashes can be due to control-flow mispredictions, address aliasing, exceptions, and MCVs. In Spectre, the only relevant squashes are those due to control-flow mispredictions.

We augment the Unsafe architecture with the hardware defense schemes in Table 2. These schemes protect loads until they reach their VP as follows: Fence stalls loads with fences; Delay-On-Miss (DOM) [26, 33] stalls speculative loads that miss in the L1; STT [52] stalls loads whose arguments are tainted by transiently-read data.

We model each hardware defense scheme with the configurations of Table 3. They include Comp and Spectre, which are the defense schemes without extensions under the Comprehensive and Spectre model, respectively. They also include LP and EP, which are the defense schemes augmented with Late Pinning and Early Pinning, respectively, under the Comprehensive model.

Table 1 also shows the area, dynamic read energy, and leakage power of the CST, which is the main *Pinned Loads* hardware structure. The data is obtained using Cacti [3] with 22nm technology.

Table 1: Parameters of the simulated architecture.

Parameter	Value
Architecture	1 (SPEC17) or 8 (SPLASH2 & PARSEC) out-of-order x86 cores at 2.0 GHz
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, LTAGE branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
Private L1-D Cache	32 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
Shared L2 Cache (LLC)	Slice: 2 MB, 64 B line, 16-way, 8 cycles RT latency
Coherence	Directory-based MESI protocol
Network	Ordered, 4×2 mesh, 128b link, 1 cycle/hop
DRAM	50 ns RT latency after L2
L1 CST	12 entries, 8 records/entry; Area: $0.0008mm^2$; Dynamic read energy: $0.6pJ$; Leakage power: $0.17mW$
Dir/LLC CST	40 entries, 2 records/entry; W_d: 2 per slice and set for each core; Area: $0.0005mm^2$; Dynamic read energy: $0.4pJ$; Leakage power: $0.17mW$
CPT	4 entries; Negligible area, energy, and power
LQ ID Tag	24 bits

Table 2: Hardware defense schemes modeled.

Scheme	Description of the defense
Unsafe	No defense: unmodified x86 architecture
Fence	Stall all speculative loads with fences
DOM	Stall speculative loads on L1 miss [26, 33]
STT	Stall loads that are tainted by transient data [52]

Table 3: Extensions added to the defense schemes.

Config.	Description
Comp	No extension: Unmodified scheme under Comprehensive model
LP	Comp + *Pinned Loads* with Late Pinning
EP	Comp + *Pinned Loads* with Early Pinning
Spectre	No extension: Unmodified scheme under Spectre model

We run SPEC17 applications [8] on a single core, and SPLASH2 [47] and PARSEC [6] applications on 8 cores. For SPEC17, we use the *reference* input size. For each application, we use SimPoint [14]

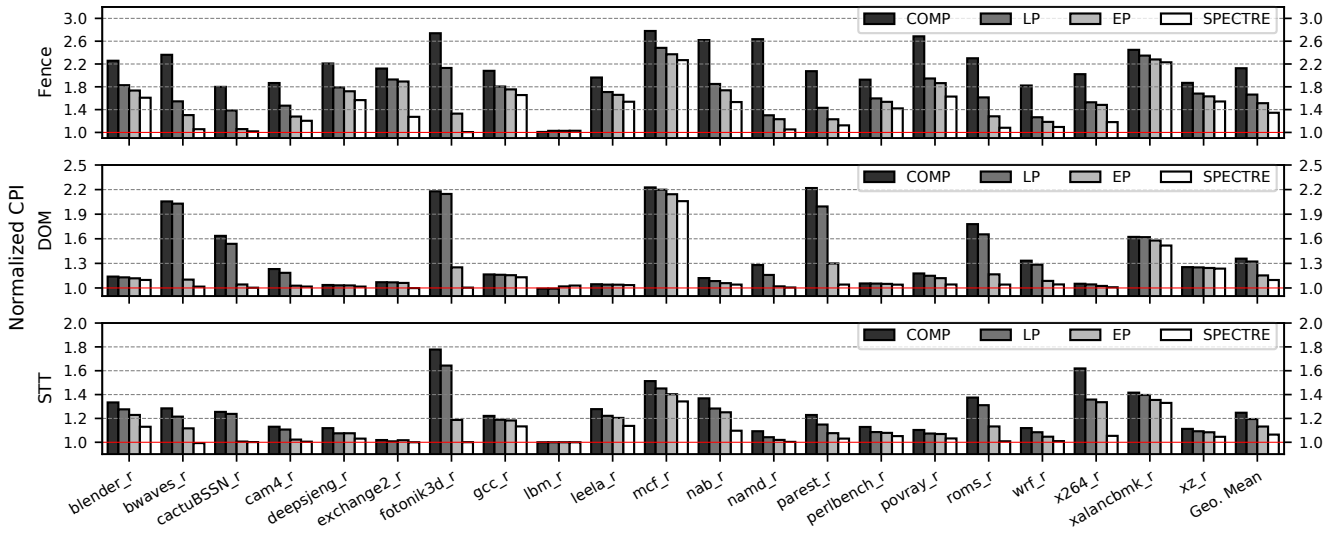

Figure 7: Normalized CPI of SPEC17 programs on different architecture configurations, all normalized to Unsafe. The three plots correspond, from top to bottom, to configurations built on the Fence, DOM, and STT defense schemes. Each plot has a different Y-axis range.

to generate up to 10 representative intervals that accurately characterize the end-to-end performance of the application. Each interval consists of 50M instructions. We run Gem5 on each interval with system-call emulation mode with 1M warm-up instructions. For SPLASH2 and PARSEC, we use the *simmedium* input size and run full-system simulation for the region of interest (ROI).

9 EVALUATION

9.1 Overall Performance Results

9.1.1 Performance on SPEC17. Figure 7 shows the normalized CPI of SPEC17 programs on all the defense schemes with the extensions listed in Table 3 (Comp, LP, EP, and Spectre). The three plots correspond, from top to bottom, to the Fence, DOM, and STT defense schemes. Each plot has a different Y-axis range. All bars are normalized to Unsafe. Each plot shows each SPEC17 program and the geometric mean of all.

Going from top to bottom, we see that Fence has the highest execution overhead among all the schemes evaluated. On average, it has a geometric mean execution overhead of 112.6% with the Comprehensive model. With Late Pinning (LP), we reduce the execution overhead to 66.4%. By using Early Pinning (EP), we further reduce the execution overhead to 51.3%, which is close to the execution overhead with the Spectre threat model (34.5%).

DOM has a moderate execution overhead on SPEC17. On average, it has a geometric mean execution overhead of 35.8% under Comprehensive. Because DOM only delays speculative loads that miss in L1, it usually has high execution overhead on applications that have poor L1 hit rate, in which case LP cannot effectively pin the loads and "pass the VP" (Section 5.2.1). With LP, the average execution overhead is only reduced to 32.3%. EP, on the other hand, can better handle cache misses (Section 5.2.2), and reduces the average execution overhead to 15.3%. This is close to Spectre's (9.7%). EP provides huge speedups to benchmarks with high L1 miss rates, such as `bwaves` and `fotonik3d`.

STT's average execution overhead is 24.8% on SPEC17 under Comprehensive. It is the fastest scheme evaluated. LP reduces the average execution overhead to 19.5% and EP to 13.2%. The execution overhead under the Spectre model is 6.4%.

Overall, we see that augmenting existing defense schemes with EP substantially reduces the execution overhead of the schemes.

9.1.2 Performance on SPLASH2 and PARSEC. Figure 8 shows the normalized CPI of SPLASH2 and PARSEC applications. The figure is organized as in Figure 7. We see that Fence's geometric mean execution overhead is 113.1% under Comprehensive. Because of the relatively high L1 hit rate of SPLASH2 and PARSEC applications, both LP and EP offer good speedups: they reduce the execution overhead to 51.2% and 46.4%, respectively. The execution overhead under Spectre is 31.1%.

DOM has a moderate execution overhead on SPLASH2 and PARSEC under Comprehensive, mainly because of high L1 hit rate. On average, its execution overhead is 15.8%. LP reduces it to 12.7%, and EP further reduces it to 7.6%. The execution overhead under Spectre is 4.2%. The `lu_ncb` and `raytrace` applications have a high L1 miss rate, but `lu_ncb`'s branches are resolved quickly (hence Spectre performs well), unlike `raytrace`'s. EP reduces `lu_ncb`'s execution overhead from 167.2% to 33.3%.

STT has small execution overheads on SPLASH2 and PARSEC. On average, it has a geometric mean execution overhead of 11.3% under Comprehensive. With LP, it is reduced to 8.7%. EP further reduces it to 8.1%. The execution overhead under Spectre is 5.1%. The `x264` application still has much higher execution overhead under EP than under Spectre. The reason is that it has dependencies between loads, which is a pattern EP cannot efficiently handle.

Overall, for these programs, we observe that LP, and especially EP, substantially reduce the execution overheads of all schemes.

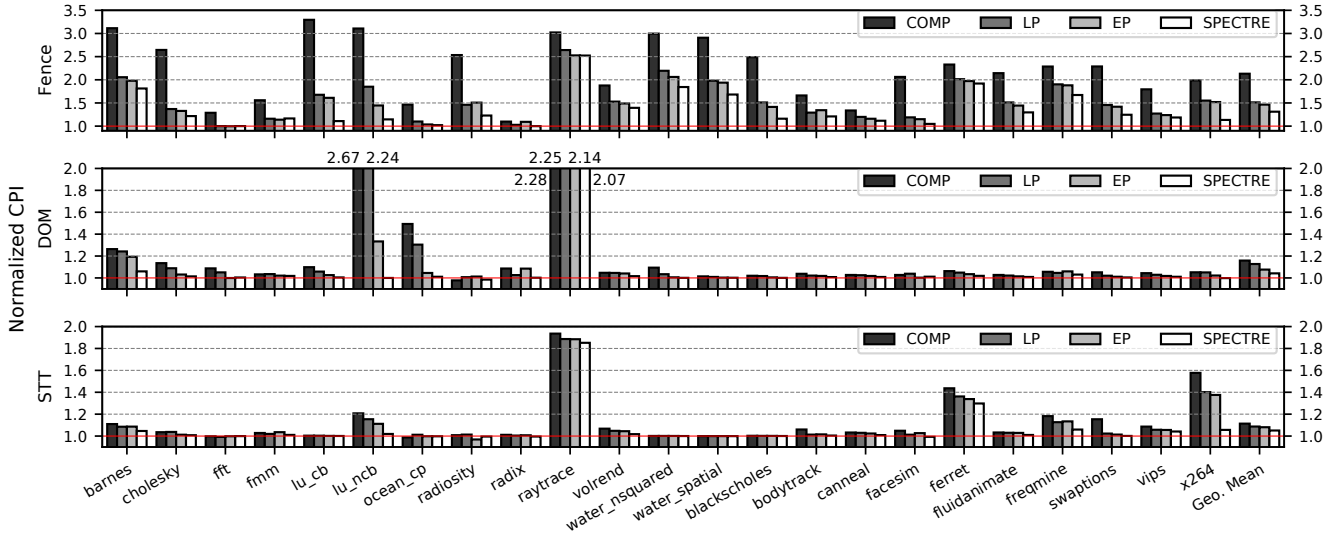

Figure 8: Normalized CPI of SPLASH2 and PARSEC programs on different architecture configurations, all normalized to Unsafe. The three plots correspond, from top to bottom, to configurations built on the Fence, DOM, and STT defense schemes. Each plot has a different Y-axis range.

9.1.3 Network Traffic Overhead. While *Pinned Loads* does not change the network traffic of the SPEC17 applications, it could increase the traffic of the SPLASH2 and PARSEC applications. In practice, we find that enabling *Pinned Loads* on Fence, DOM, and STT has no significant impact on network traffic. The reason is that very few writes and evictions have to retry due to pinning. Even in the worst-case applications, only 14.8 writes and 0.05 evictions are retried per *million* instructions.

9.1.4 Breakdown of the Execution Overhead. We now assess the big-picture impact of *Pinned Loads* on the execution overhead of the defense schemes. Figure 9 combines defense schemes (Fence, DOM, and STT) and applications (SPEC17 and Parallel ones). For each combination, it shows, first, the execution overhead of Comp normalized to Unsafe and broken down into the different sources of speculation. These bars are like those in Figure 1. The second and third bars of each combination are the execution overheads of the same defense scheme augmented with LP and EP, respectively. Note that two of the bars in the graph are cut off.

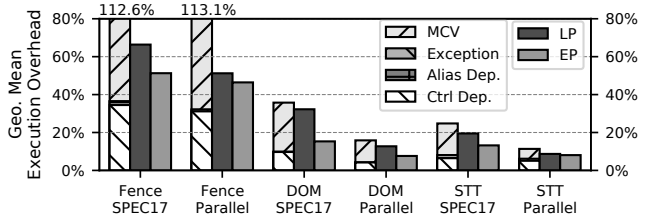

Figure 9: Breakdown of the execution overhead due to different sources and for different schemes, all relative to Unsafe.

We see that, under the Comprehensive model, the execution overhead of every defense scheme is mainly caused by stalls to prevent potential MCVs and, to a lesser extent, control dependencies. LP and EP focus on removing the MCV overhead. We see that LP and, especially, EP eliminate most of the MCV overhead. The upper bound of EP's effectiveness is to eliminate all the MCV overhead. In that case, it would be nearly as if we only had control dependence overheads—which is the Spectre model overhead. From the figure, we see that, in the case of Fence, EP has an absolute 15% higher overhead than Spectre.

9.2 Analysis of the Hardware Structures

9.2.1 Cache Shadow Table (CST) Configuration. Ideally, the CST should precisely track where each pinned line maps in the L1 cache and in the directory/LLC. However, in practice, to minimize area overhead, we reduce the CST's number of entries and number of records per entry. As a result, there are false positive conflicts: the CST claims the load cannot be pinned due to lack of space while, in reality, there is space.

To decide on the sizes of the CSTs, we perform a sensitivity analysis. Our chosen default sizes (Table 1) are 12 entries with 8 records per entry for the L1 CST and 40 entries with 2 records per entry for the Dir/LLC CST. With this design, we find that the average L1 CST false positive rates are smaller than 0.02% on SPEC17, and than 0.01% on SPLASH2 and PARSEC for all the defense schemes with EP. Further, the average Dir/LLC CST false positive rates are smaller than 0.4% on SPEC17, and than 0.02% on SPLASH2 and PARSEC for all the schemes with EP. Hence, false positives are rare.

We measured the execution overhead of the different defense schemes (with EP) and programs for different CST sizes. On average, the execution overhead with our chosen configuration is 3.6% higher than with an infinite CST.

9.2.2 Cannot-Pin Table (CPT) Size. A line is inserted into the CPT only when a write fails a retry after having been deferred. We collect the average and the maximum number of lines that are in the CPT at a time. We use an ideal CPT and run SPLASH2 and PARSEC. On average, the CPT only needs to hold one line, and the

maximum number of lines is 4–7 for all the schemes. Thus, we use a default CPT with 4 entries (Table 1). With this size, we see less than 0.0001 CPT overflows per insertion attempt for a few applications, and no overflows for most.

9.2.3 Smaller Directory/LLC Partition Size. Our default EP design allows a core to pin up to 2 lines per set in the directory/LLC at a time (W_d is 2). We repeat our experiments with W_d equal to 1 while keeping the same CST size. We see that the overhead of the schemes with EP increases: for Fence, it increases from 51.3% to 54.7% on SPEC17 and from 46.4% to 47.0% on parallel applications; for DOM, it changes from 15.3% to 18.5% on SPEC17 and from 7.6% to 8.0% on parallel applications; for STT, it increases from 13.2% to 14.7% on SPEC17 and remains the same on parallel applications. Consequently, keeping W_d equal to 2 is best.

9.2.4 Hardware Overhead. The main storage structure added by *Pinned Loads* is the CST used by EP. The CPT and the extended LQ ID tags are very small. With our default configuration and including the tags, the L1 CST is 444 bytes and the Directory/LLC CST is 370 bytes. We use CACTI 7.0 [3] to estimate the CST area, dynamic read energy, and leakage power at 22nm. As shown in Table 1, these numbers are very small.

10 OTHER RELATED WORK

Speculative execution attacks exfiltrate secret data by exploiting different types of speculation, such as control-flow speculation [10, 12, 22–24, 28, 35], memory dependence speculation [16], and memory consistency speculation [29, 37]. For memory consistency speculation, Ragab et al. [29] and Skarlatos et al. [37] demonstrate how an attacker from a core can repeatedly create squashes due to MCVs in another core. From here, many attacks are possible. For example, if the victim gets a random number, the attacker can force selective squashes and retries and bias the random number generator. Defending against this type of speculation attack is expensive. *Pinned Loads* substantially reduces the cost of such defense.

11 CONCLUSION

To reduce the overhead of defenses against speculative execution attacks, this paper presented *Pinned Loads*, a general technique that helps instructions reach their VPs sooner. Under the Comprehensive threat model, we found that the progress of the VP is mostly impeded by waiting until no MCVs are possible. Hence, *Pinned Loads* tries to make loads invulnerable to MCVs as early as possible—a process we call *pinning* the loads in the ROB. In this paper, we described the several hardware mechanisms needed by *Pinned Loads*, and two possible *Pinned Loads* designs with different tradeoffs. Our evaluation showed that *Pinned Loads* is very effective: extending the fence-insertion, Delay-On-Miss, and STT defense schemes with *Pinned Loads* reduces these schemes' average execution overhead on SPEC17 and on SPLASH2/PARSEC applications by about 50%. Specifically, on SPEC17, the execution overhead of the three defense schemes decreases from 112.6% to 51.3%, from 35.8% to 15.3%, and from 24.8% to 13.2%, respectively; on SPLASH2/PARSEC, the execution overhead decreases from 113.1% to 46.4%, from 15.8% to 7.6%, and from 11.3% to 8.1%, respectively.

ACKNOWLEDGMENTS

This research was funded in part by an Intel Strategic Research Alliance (ISRA) grant and by ISF grant 2005/17.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides a complete gem5 implementation of *Pinned Loads*, along with scripts to evaluate *Pinned Loads*' performance on SPEC17, PARSEC, and SPLASH2X benchmark suites. We also open sourced our gem5 implementation and experiment scripts on GitHub (benchmark applications are not included due to license issues).

A.2 Artifact Checklist (Meta-information)

- **Program:** Gem5
- **Compilation:** We compiled the gem5 simulation infrastructure with gcc-5.4.0.
- **Run-time environment:** Linux with Docker containers.
- **Run-time state:** We use SimPoint methodology to generate up to 10 representative intervals that accurately characterize end-to-end performance for SPEC17 benchmarks. Each interval consists of 50 million instructions. For PARSEC and SPLASH2X, we use *simmedium* input size and run full-system simulation for the region of interest (ROI).
- **Output:** Plots are output by the provided scripts.
- **Experiments:** Please refer to Section A.5.
- **How much disk space required (approximately)?:** 1GB.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1 day.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Workflow framework used?:** HTCondor for job management.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5741384. But we recommend using the latest version from GitHub.

A.3 Description

A.3.1 How to Access. Our complete simulation implementation is available at:
https://github.com/zzrcxb/PinnedLoads (recommended) or
https://doi.org/10.5281/zenodo.5741384.

A.3.2 Hardware Dependencies. Any hardware capable of running the gem5 simulator is sufficient.

A.3.3 Software Dependencies. We use Docker and provide a complete Dockerfile that captures all the software dependencies required to build our simulation infrastructure.

A.3.4 Data Sets. We run SPEC17 with the reference input size. Because of simulation issue with gem5, we exclude 2 applications (omnetpp, imagick) out of 23 from SPEC17. We run PARSEC and SPLASH2X with simmedium input size. We exclude 4 applications (raytrace, dedup, streamcluster, ocean_ncp) out of 27 from PARSEC and SPLASH2X due to simulation issues.

A.4 Installation

Build time: 5 to 10 minutes depends on the machine. More information in Section A.5.4.

Required libraries: All libraries that are required by gem5. The instruction can be found at https://www.gem5.org/documentation/learning_gem5/part1/building/.

A.5 Experiment Workflow

A.5.1 Overview. To reproduce our results, under the `scripts` directory, we provide two scripts, `runner` and `plotter`, to submit jobs and process results.

A.5.2 Clone Pinned Loads. Pinned Loads is publicly available on GitHub. Run

```
git clone https://github.com/zzrcxb/PinnedLoads.git
```

to clone our repository from GitHub. Then enter the cloned project.

A.5.3 Environment Setup. Set environment variables

```
export GEM5_ROOT=<path to gem5 root>
export M5_PATH=<path to full-system images and disks>
export WORKLOADS_ROOT=<path to benchmark suites root>
```

Note that the `WORKLOADS_ROOT` directory must be structured properly to include required benchmarks before using any of the scripts. Please refer to instructions in `$GEM5_ROOT/scripts/README.md` [1] for more details.

A.5.4 Compile gem5. Due to a gem5 bug[2], it must be compiled on Ubuntu 16.04 to avoid crashing on some benchmarks. To address this issue, we provide a Docker image for compilation. Under `$GEM5_ROOT`, run command

```
./cgem.sh
```

will invoke Docker and start the compilation process.

For the instructions of building gem5 manually, please refer to `$GEM5_ROOT/README.md`.

A.5.5 Submit Jobs. Enter `$GEM5_ROOT/scripts/` and run command:

```
./runner submit SPEC17 &&
./runner submit PARSEC &&
./runner submit SPLASH2X
```

to submit all the required jobs to HTCondor. It takes about 10 minutes to finish job submission.

A.5.6 Check Status. To check job status via condor, run command:

```
condor_q
```

which prints the total number of running jobs and remaining jobs.

To check job status for each configuration, under `$GEM5_ROOT/scripts/`, run command:

```
./runner status
```

to print detailed job status information for each configuration.

Depending on the performance of your servers, it can take several days to finish all the jobs.

A.5.7 Collect Results. After all the jobs are finished, under `$GEM5_ROOT/scripts/`, you can collect the results by executing:

```
./runner collect
```

which should generate a file named `data.csv` that contains execution overhead for each benchmark and configuration, normalized to an unsafe baseline.

Then, execute:

```
./plotter perf && ./plotter breakdown
```

to generate Figure 7 (`perf-spec.pdf`), Figure 8 (`perf-sp.pdf`), and Figure 9 (`brkd.pdf`), respectively.

A.6 Evaluation and Expected Result

Because the benchmark checkpoints can be different from the ones in our evaluation, the collected plots may not exactly match their corresponding figures in our paper, but they should be similar.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

[1] https://github.com/zzrcxb/PinnedLoads/blob/main/scripts/README.md#Structure-of-Workload-Directory
[2] https://gem5.atlassian.net/browse/GEM5-631

REFERENCES

[1] Sam Ainsworth and Timothy Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *International Symposium on Computer Architecture (ISCA)*.

[2] ARM. 2018. Cache Speculation Side-channels. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper.

[3] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[4] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[5] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* (2011).

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 769–784. https://doi.org/10.1145/3319535.3363219

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.

[11] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[13] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP'91*.

[14] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[15] John L Hennessy and David A Patterson. 2017. *Computer Architecture: a Quantitative Approach* (6th ed.). Morgan Kaufmann.

[16] Jann Horn. 2018. Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[17] Intel. 2018. Speculative Execution Side Channel Mitigations. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[18] Intel. 2020. Refined Speculative Execution Terminology. https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology.

[19] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 151–162.

[20] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[21] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[22] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv e-prints* (Jul 2018). arXiv:1807.03757 [cs.CR]

[23] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[24] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! Speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[25] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating spectre attacks using CFI informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–53.

[26] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 264–276.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[28] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2109–2122.

[29] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 1451–1468.

[30] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security & Privacy*.

[31] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 187–200. https://doi.org/10.1145/3079856.3080220

[32] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An Undo Approach to Safe Speculation. In *International Symposium on Microarchitecture (MICRO)*.

[33] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 723–735.

[34] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[35] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 279–299.

[36] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors. *CACM* 7 (July 2010), 89–97.

[37] Dimitrios Skarlatos, Zirui Neil Zhao, Riccardo Paccagnella, Christopher W. Fletcher, and Josep Torrellas. 2021. Jamais Vu: Thwarting Microarchitectural Replay Attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1061–1076. https://doi.org/10.1145/3445814.3446716

[38] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 395–410.

[39] Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. 2020. Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 241–254.

[40] Paul Turner. 2018. Retpoline: a Software Construct for Preventing Branch-target-injection. https://support.google.com/faqs/answer/7625886.

[41] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[42] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[43] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[44] David L Weaver and Tom Germond. 1994. *The SPARC architecture manual, Version 9*. Prentice-Hall.

[45] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *International Symposium on Microarchitecture (MICRO)*.

[46] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[47] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.

[48] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[49] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360.

[50] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[51] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 707–720.

[52] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[53] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1138–1152.