Pinned Loads: Taming Speculative Loads in Secure Processors

Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, Josep Torrellas

University of Illinois     Tel Aviv University

ASPLOS’22
Speculative Execution Attacks

Modern microprocessors are threatened by speculative execution side-channel attacks

```c
if (x < array1_size) {
    uint8 secret = array1[x]; // access
    ...
    uint8 y = array2[secret * 4096]; // transmit
}
```

- **Malicious input:** $x > array1\_size$
- **Predicts taken**
- **Speculative access (out-of-bound)**
- **Speculative leakage**
Visibility Point (VP)

**Hardware Defenses:** protect the execution of vulnerable instructions until they reach their *visibility point (VP)* --- i.e., they are no longer vulnerable to pipeline squashes that are relevant to the threat model considered

Reach VP sooner ⇒ Better performance

```java
if (x < array1_size) {
    // b1
    secret = array1[x]; // l1
    ...
    load secret; // l2
}
```

Assume: x is in-bound, b1 resolves to taken

Fence
(delay speculative loads until VP)

![ROB diagram](image)

Pre-VP

Post-VP

l1 and l2 are allowed to execute

Resolved to taken

Unresolved
VP Under the Comprehensive Model

Consider: vulnerable instruction=> Load \( (L) \)

Comprehensive Model

- All control-flow instructions older than \( L \) are resolved
- \( L \) plus any older instruction cannot suffer exceptions
- \( L \) plus any older load cannot alias with other loads/stores
- \( L \) plus any older load cannot cause a memory consistency violation (MCV)

\[ \text{ROB Head} \]

\( L \) needs to reach the ROB head or become the oldest load to be invulnerable to MCV
Focus on Memory Consistency Violations

Execution overhead breakdown of Fence
(by each reason that delays reaching the VP)

Conditions of Reaching VP
(Comprehensive Model)

- No MCVs
- No exceptions
- No aliasing
- No control-flow mispredictions

Ensuring no MCVs causes the most overhead
Pinned Loads

**Goal:** a **general** hardware mechanism that makes loads invulnerable to MCVs as early as possible

**Intuition:** When certain conditions are satisfied, *Pinned Loads* will **pin** a load $L$ and **guarantee:**

- No squashes of $L$ due to invalidations
- No squashes of $L$ due to cache evictions

$\Rightarrow$ after pinning, $L$ is **invulnerable to MCVs**

$L$ reaches VP sooner (before reaching the ROB head) $\Rightarrow$ higher performance
Potential Performance Gain

Conventional Unsafe

All independent loads in parallel

Issue to mem

Flopped

Fence

Serialized!

Quickly “pass” the VP downstream, issue all independent loads in parallel

Fence + Pinned Loads

Quickly “pass” the VP downstream, issue all independent loads in parallel

All independent loads in parallel

Issue to mem
Threat Model

Assume the Comprehensive threat model:
- No control-flow mispredictions
- No aliasing
- No exceptions
- No MCVs

Preserve the speculative execution security properties of the baseline defense schemes:

A load $L$ can be pinned only if $L$ has met all the conditions to reach VP, except for guaranteeing $L$ itself will not cause an MCV

<table>
<thead>
<tr>
<th>L</th>
<th>br1</th>
<th>ld2</th>
<th>ld1</th>
</tr>
</thead>
</table>

ROB Head

Implication: Loads are pinned in the program order

No exceptions, aliasing, and mispredictions
Pinned Loads Overview

Intuition: defer invalidations and prevent evictions to lines that are read by a pinned load

Conceptually, line x is “pinned” (no actual pinned bit)

Line x is automatically “un-pinned” when ldx retires
Pinned Loads Overview

**Intuition:** defer invalidations and prevent evictions to lines that are read by a pinned load

Conceptually, line x is “pinned” (no actual pinned bit)

Line x is automatically “un-pinned” when ldx retires

**Design Overview:**

- **Defer Invalidations:** introduce new coherence messages to defer remote writes
  - Safety guarantees:
  - + No starvation
  - + No deadlock (detailed in the paper)

- **Prevent Cache Evictions:** guarantee space in cache 
  & directory:
  - + Late Pinning (LP)
  - + Early Pinning (EP)
Defer Invalidations to Pinned Lines

Conventional Protocol

Core 1

L1

S

② Inv

② Data

Directory

① GetX

① Write

Core 2

L1

Data

Sharer Bits

1

0
Defer Invalidations to Pinned Lines

**Conventional Protocol**

- Core 1
  - L1
  - \( S \rightarrow I \)
  - ① GetX
  - ② Data
  - ③ Ack
  - Directory
  - 1 0 Data
  - Sharer Bits

- Core 2
  - L1
  - ① Write
  - ④ Unblock

**Pinned Loads**

- Core 1
  - L1
  - \( S \)
  - ① GetX
  - ② Data
  - Directory
  - 1 0 Data
  - Sharer Bits

- Core 2
  - L1
  - ① Write
  - ② Inv
Defer Invalidations to Pinned Lines

Conventional Protocol

- Core 1 writes to L1
- Core 2 reads from L1
- Core 2 sends a GetX
- Core 1 sends a Data
- Core 1 sends an Ack
- Core 2 sends an Unblock

Pinned Loads

- Core 1 writes to L1
- Core 2 reads from L1
- Core 2 sends a GetX
- Core 1 sends a Defer
- Core 2 sends an Abort

Directory state is unchanged after Abort

Core 2 will retry the write

Changes from the Conventional: new coherence messages (and the logic of handling them)
Prevent Store Starvation

**Cannot-Pin Table (CPT):** a per-core hardware structure that records the addresses of lines that the core is not allowed to pin *at the moment*

New *load* $x$ cannot be pinned

Failed Retry

- **Core 1**: Update CPT
- **Core 2**: Write $x$
- **Directory**: Data
- **Sharer Bits**

Line $x$ is now unpinned

Successful Retry

- **Core 2**: Write $x$
- **Core 1**: GetX*
- **Directory**: Data
- **Sharer Bits**
Prevent Store Starvation

Cannot-Pin Table (CPT): a per-core hardware structure that records the addresses of lines that the core is not allowed to pin at the moment.

New load x cannot be pinned

1. GetX*
2. Inv*
3. Data
4. Abort

Update CPT

Core 1

CPT

x

Core 2

① Write x
③ Defer
④ Unblock

Line x is now unpinned

1. GetX*
2. Inv*
3. Ack
4. Unblock

Core 2

CPT

x

Core 1

① Write x
③ Ack
④ Unblock

Directory

Sharer Bits

Failed Retry

Successful Retry
Prevent Store Starvation

**Cannot-Pin Table (CPT):** a per-core hardware structure that records the addresses of lines that the core is not allowed to pin *at the moment*

*New load x cannot be pinned*

**Core 2**
- ① Write x
- ② Data
- ③ Defer
- ④ Abort
- ⑤ Clear
- ⑥ Remove x

**Core 1**
- ① GetX*
- ② Inv*
- ③ Update CPT
- ④ Ack

**Directory**
- Sharer Bits

**Failed Retry**

**Successful Retry**
Prevent Evictions of Pinned Lines

**Intuition:** Pinned Loads denies evictions to pinned lines

1: ld x // L1 Hit
2: ld y // L1 Miss

Line x and y are mapped to the same L1 set

2-way L1

```
     a
   x
```

1. Issue
2. Evict
3. Deny
4. Update LRU for line x
Prevent Evictions of Pinned Lines

**Intuition**: Pinned Loads denies evictions to pinned lines

1: `ld x` // L1 Hit
2: `ld y` // L1 Miss

Line x and y are mapped to the same L1 set

4. Update LRU for line x
5. Retry
Guarantee Space in Cache & Directory

**Insight:** a core cannot pin more lines than a set can hold, otherwise, deadlocks may occur

Line x, y, and z are mapped to the same L1 set

Ld x, y, and z are pinned before issuing

Two possible designs to avoid deadlock
Design 1: Late Pinning (LP)

**Intuition:** receive the data first (meaning it can find space in cache and directory sets), then pin the load

1: `ld x` // L1 Miss
2: `ld y`
3: `ld z`

Line x, y, and z are mapped to the same L1 set.

![Diagram of Core 1 showing a ROB head with a miss for ld x, and ld z mapped to the same L1 set.](image)
Design 1: Late Pinning (LP)

**Intuition:** receive the data first (meaning it can find space in cache and directory sets), then pin the load

Line x, y, and z are mapped to the same L1 set

```
1: ld x // L1 Miss
2: ld y
3: ld z
```
Design 1: Late Pinning (LP)

**Intuition:** receive the data first (meaning it can find space in cache and directory sets), then pin the load

1: `ld x` // L1 Miss  
2: `ld y`  
3: `ld z`  

Line x, y, and z are mapped to the same L1 set

Ld z will stall until ld x retires and unpins x  
(slow but safe from deadlock)

Core 1

Serialized memory access, but it issues loads much earlier than it would in Fence

+ Simple hardware  
- Low performance for programs with high L1 miss rates
Design 2: Early Pinning (EP)

Intuition: add a small local hardware table called **Cache Shadow Table (CST)** in each core. CST tracks, for each set in L1 and LLC/Dir, how many lines are pinned by *in-flight loads*.

Checks before pinning a load:
1) Hardware determines the L1 set and the LLC/Dir set where the line maps
2) Access CST sets and check if such sets can hold the *additional* pinned line
3) Pin the load if find space in each cache level and directory

Line x, y, and z are mapped to the same L1 set (and the *same CST set*).
Design 2: Early Pinning (EP)

**Intuition:** add a small local hardware table called **Cache Shadow Table (CST)** in each core. CST tracks, for **each set** in L1 and LLC/Dir, how many lines are pinned by **in-flight loads**

Checks before pinning a load:
1) Hardware determines the L1 set and the LLC/Dir set where the line maps
2) Access CST sets and check if such sets can hold the **additional** pinned line
3) Pin the load if find space in each cache level and directory

Line x, y, and z are mapped to the same L1 set (and the same CST set)
Design 2: Early Pinning (EP)

Intuition: add a small local hardware table called **Cache Shadow Table (CST)** in each core. CST tracks, for each set in L1 and LLC/Dir, how many lines are pinned by in-flight loads.

Core 1

1: ld x
2: ld y
3: ld z

Line x, y, and z are mapped to the same L1 set (and the same CST set)

The L1-CST set is full, ldz will remain unpinned (slow but safe from deadlock)

CST for a shared cache requires small changes in its geometry (detailed in the paper)

2-way L1

2-way L1-CST (LLC/Dir CST omitted)
Design 2: Early Pinning (EP)

Intuition: add a small local hardware table called **Cache Shadow Table (CST)** in each core. CST tracks, for each set in L1 and LLC/Dir, how many lines are pinned by **in-flight loads**

![Diagram of Design 2: Early Pinning (EP)](image)

- **Core 1**
  - Line x, y, and z are mapped to the same L1 set (and the **same CST set**)
  - All independent loads in parallel (assume enough space)
  - + Parallelized access
  - - Require CSTs

<table>
<thead>
<tr>
<th>1: ld x</th>
<th>2: ld y</th>
<th>3: ld z</th>
</tr>
</thead>
</table>

- Parallelized access
- Require CSTs

(2-way L1, LLC/Dir CST omitted)
**Pinned Loads Summary**

**Mechanism:** defer invalidations and prevent evictions to lines that are read by a pinned load

For a load \( L \), it can be pinned if:

- **Security**
  - \( L \) has met all the conditions to reach the VP except for guaranteeing \( L \) itself will not cause an MCV

- **Avoid Starvation**
  - The line that \( L \) tries to pin is not in **Cannot-Pin Table (CPT)**
    - Write buffer has enough entries for all the yet-to-complete stores older than \( L \) (detailed in the paper)

- **Avoid Deadlock**
  - Guaranteeing space in cache & directory
    - \( L \) has received the data, or \( \Rightarrow \) **Late Pinning (LP)**
    - CSTs report enough space in L1 cache and LLC/Dir \( \Rightarrow \) **Early Pinning (EP)**
Performance Evaluation

Workloads: single-threaded (SPEC17) and parallel (SPLASH2 + PARSEC)  
Defenses: Fence, DOM, and STT

Geo. Mean Execution Overhead over a Conventional Unsafe Core (SPEC17)

≈50% overhead reduction (with EP)
Conclusions

- Under the Comprehensive model, most execution overhead is caused by ensuring no memory consistency violations (MCVs)

- *Pinned Loads* is a general technique to reduce the execution overhead of speculative-execution defense schemes by making loads invulnerable to MCVs as early as possible

- *Pinned Loads* can substantially reduce execution overhead of many existing defense schemes by $\approx 50\%$

Open Source: [https://github.com/zzrcxb/PinnedLoads](https://github.com/zzrcxb/PinnedLoads)
Pinned Loads: Taming Speculative Loads in Secure Processors

Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, Josep Torrellas

University of Illinois       Tel Aviv University

ziruiz6@illinois.edu

ASPLOS’22 – Session 3B