

The Australian National University
2600 ACT | Canberra | Australia



Australian
National
University

School of Computing
College of Systems and Society

STTHint: Accelerating Speculative Taint Tracking by Hinting Safe Memory Objects to the Hardware

— 12 pt research project (S1/S2 2026)

A report submitted for the course
COMP3770, Computing Research Project (R&D)

By:
Harold Gao

Supervisor:
Dr. Shoaib Akram

November 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the [Academic Integrity Rule](#);
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the ANU Academic Integrity module/s prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

November (2026), Harold Gao

Abstract

Spectre attacks exploit speculative execution to leak secret information through microarchitectural side-channels. Comprehensive mitigations typically incur significant performance impact or hardware overhead, so none has been widely adopted in commercial CPUs. We find that Speculative Taint Tracking (STT), a leading Spectre mitigation, introduces a high performance cost in long-running loops with access-transmit sequences and slow-to-resolve branches. We find that the resulting stalls are concentrated in a small number of static instructions, which access only a few memory objects. Based on this insight, we propose STTHint, an efficient mechanism to accelerate STT by enabling application programmers and compiler writers to hint safe memory objects to the hardware, recovering smooth speculative execution of instructions that transmit data held in these safe regions. Compared to SSMR, a similar approach, STTHint incurs lower hardware overhead, reducing the number of bits stored by 77.73%. Our experiments show that STTHint reduces the performance degradation of STT from an average of 47.22% down to 2.80% over the GAP benchmark suite.

Table of Contents

1	Introduction	1
2	Background	3
2.1	Speculative Execution	3
2.1.1	Program Execution	3
2.1.2	Speculative Execution	3
2.1.3	Out-of-Order Engine	4
2.1.4	The Cache Hierarchy	4
2.1.5	Memory Objects	5
2.1.6	Performance Metrics	5
2.2	Spectre	5
2.2.1	Accessors and Transmitters	5
2.2.2	Example	6
2.2.3	Visibility Point	6
2.2.4	Transient Execution Attacks	6
2.3	Speculative Taint Tracking	7
2.3.1	Hardware Mechanism	7
2.3.2	Example	7
2.3.3	Safe Memory Objects	8
3	Related Work	9
3.1	Limited Mitigations	9
3.1.1	Targeting Specific Variants	9
3.1.2	Targeting the Cache	10
3.2	Comprehensive Mitigations	10
3.3	Software-Hardware Communication	11
3.3.1	SSMR	11
3.3.2	Protean	12
4	Motivation	13
4.1	Instruction Blame	13
4.1.1	Formulation	13
4.1.2	Blame Stack	14
4.1.3	Blame Distribution	15
4.1.4	Overlapped STT Stalls	15
4.1.5	Implementation	16

Table of Contents

4.1.6	Benchmark Results	16
4.2	Evaluating STT	17
4.2.1	Results	17
4.2.2	Taint-Critical Regions	17
4.2.3	TCR Example	18
5	STTHint	19
5.1	Threat Model	19
5.2	Hints	19
5.2.1	ISA Extension	20
5.3	The Hint Buffer	21
5.4	Hint Prediction	21
5.4.1	Hint Predictor	21
5.4.2	Securing Store-to-Load Forwarding	22
6	Evaluation	23
6.1	Methodology	23
6.1.1	Benchmark Suites	23
6.1.2	Hardware Simulation	24
6.1.3	Hinting Methodology	24
6.2	Empirical Results	24
6.2.1	Prediction Accuracy	26
6.2.2	Hardware Overhead	27
6.2.3	Hint Counts	28
6.2.4	Cache Capacity	28
6.2.5	OoO Engine	29
6.2.6	PARSEC Case Study	29
6.2.7	Hint Insertion Example	30
7	Conclusion and Future Work	33
7.1	Future Work	33
7.1.1	Programmer Responsibility	33
7.1.2	Operating System Considerations	34
7.2	Conclusion	34
	Bibliography	35

Introduction

Spectre [1], discovered in 2018, is an attack against modern CPUs. Modern processors predict control-flow outcomes before they are known, so instructions execute *speculatively* instead of stalling [2]. This is known as *speculative execution* [3], and mispredicted instructions are *transient*. To maintain program correctness, transient instructions are later *squashed* (discarded), and their results are not committed to the architectural state [4]. However, transient instructions can still permanently modify microarchitectural state such as caches, and these traces are observable by an attacker [5, 6]. Such inadvertent information channels are known as *side-channels* [7], enabling attackers to infer secrets that the processor speculatively accessed, even when that information could never have been legitimately read on the correct execution path [1, 8].

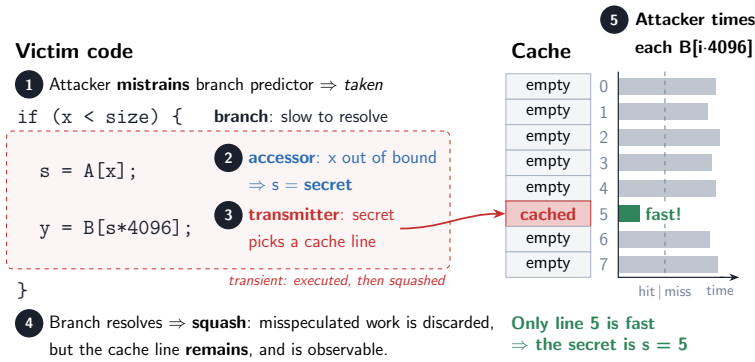


Figure 1.1: Deriving a transiently transmitted secret by timing cache accesses.

Before Spectre, side-channel leakage exposes memory access patterns and a limited subset of memory data [9]. However, a Spectre attack may misdirect speculative execution to access and transmit *arbitrary* memory locations. Figure 1.1 details an example Spectre attack. Therefore, numerous Spectre mitigations have been established [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. Unfortunately, these works introduce significant trade-offs in performance cost [11, 13, 21, 19, 23, 10], hardware overhead [11, 16, 22, 17] or limited coverage of Spectre variants and side-channels [11, 12, 13, 18, 23, 20, 22, 24].

1 Introduction

Therefore, no single, comprehensive defence has been widely adopted for commercial CPUs.

Amongst these, Speculative Taint Tracking (STT) [10] is widely regarded as the state-of-the-art mitigation [16, 14, 17, 26]. STT performs information flow tracking of transient data by propagating a taint bit through dependencies, and STT stalls any tainted instruction capable of transient leakage until its taint source is no longer speculative. Unlike other approaches, STT does not use expensive buffers to track transient state, and STT is a comprehensive defence against the Spectre threat model [10]. STT can be efficient over some workloads: the authors of STT [10] report an average performance degradation of 8.5% for their selected SPEC2006 [27] and PARSEC [28] benchmarks. However, since STT lies only at the hardware level, it lacks semantic knowledge of the program state. There is no way for the hardware to decide whether a load is accessing a legitimate memory object or somewhere out of bounds. Thus, all speculative loads are conservatively treated as secret-accessing, creating a high performance degradation of STT over many workloads [10, 16, 17, 15] (also see Chapter 4).

A deeper analysis into the performance cost of STT is therefore necessary to motivate optimisations. We find that over GAP [29] and PARSEC [28] benchmarks, most of STT’s performance costs are concentrated in a small number of static instructions, which only access a few memory objects (e.g. arrays). This implies that loads to most memory objects do not significantly degrade performance, and optimising them yields limited benefits. Furthermore, we find that these performance-degrading code patterns, which we call *taint-critical regions* (TCRs), meet a common set of criteria (Section 4.2.2). This motivates **STTHint**, a way to accelerate STT by enabling the software stack to communicate safe memory objects frequently accessed in TCRs to the hardware. As a result of our analysis, STTHint incurs lower hardware overhead than SSMR (Speculation-Safe Memory Regions) [16], a similar approach that statically detects safe memory objects and communicates them to the hardware. Our core contributions are as follows:

1. **Profiling STT:** We analytically and experimentally profile STT’s performance cost, which we model using a set of criteria (Chapter 4).
2. **Accelerating STT:** Based on the above insights, we propose **STTHint**, a novel mechanism which accelerates STT over TCRs by hinting safe memory objects to the hardware. To facilitate hinting, we introduce the *hint buffer* and *hint predictor* (Chapter 5).
3. **Evaluating STTHint:** We empirically evaluate STTHint (Chapter 6), and discuss the limitations of STTHint while motivating future work (Chapter 7).

Over GAP workloads, we find that STT incurs a performance degradation of 47.22%. When hints are inserted correctly, STTHint incurs a performance degradation of 2.80%. Furthermore, STTHint falls back to STT when executing unhinted programs, so security-critical dependencies that are not performance-critical do not need to be edited and recompiled to remain secure.

Background

This chapter first explains speculative execution, the underlying vulnerability of Spectre. We then discuss how Spectre attacks can be perpetrated. Finally, we explain Speculative Taint Tracking, a leading defence against Spectre and related attacks, from which this work extends.

2.1 Speculative Execution

This section establishes the vulnerability that makes Spectre attacks possible: speculative execution.

2.1.1 Program Execution

To facilitate program execution, a CPU exposes semantics and encodings of instructions to the software, called the *Instruction Set Architecture* (ISA) [30]. A *load* instruction reads a value from memory into a register, and a *store* instruction writes a register value to memory, and a *branch* redirects control flow by changing the *program counter* (PC), which holds the address of the next instruction to fetch and is incremented by default. Instruction encoded in program memory are known as *static instructions*, the dynamic instances of which, fetched into the CPU are known as *dynamic instructions*. The *instruction PC* of a dynamic instruction is the address of its corresponding static instruction. The hardware hides the complex processing logic, exposing only register values and the memory to the software, collectively known as the *architectural state*.

2.1.2 Speculative Execution

Modern processors use *speculative execution* [3, 2] to maximise performance. When a CPU encounters an unresolved branch instruction, it uses a *branch predictor* to predict whether the branch will be taken, and executes ahead based on this prediction. When a misprediction occurs, the speculatively executed instructions are *squashed*, which means that the speculative results are not committed to the architectural state, ensuring the

2 Background

correctness of the program. Usual CPU workloads exhibit predictable control flow, which means that prediction accuracy is high in branch predictors [31], allowing the processor to make useful progress even when the execution path is unknown.

2.1.3 Out-of-Order Engine

To further improve performance, the out-of-order (OoO) engine overlaps cycles stalled due to unresolved dependencies by executing instructions that are ready [32, 2]. This implies that instructions may execute out-of-order. Instructions fetched from memory are *dispatched* into *reservation stations* [32] until their source operands become available, at which point they are issued for execution. To facilitate this, the CPU frontend renames architectural registers (registers in the architectural state) to *physical registers*, effectively tracking an update history of architectural registers [2]. When instructions retire, the physical registers are committed to the architectural state. In the event of misspeculation, register renaming enables the processor to preserve an accurate version of each architectural register corresponding to the point in which the misspeculation occurs. Furthermore, processors use the *Reorder Buffer* (ROB) [2] to keep track of instructions in program order, which is a circular queue, where *younger* instructions are dispatched in program order to the tail of the ROB, OoO execution then takes place, and *older* instructions commit to the architectural state from the head of the ROB in program order. The ROB facilitates precise exception handling, OoO execution and accurate squashing of misspeculated work.

2.1.4 The Cache Hierarchy

Because access to main memory (DRAM) is slow, a multi-level cache hierarchy is used to keep frequently accessed data close to the CPU [2]. This is usually implemented at three levels: L1, L2, and L3, and each cache entry (known as a cache line) holds a region of contiguous data copied from memory. These caches are implemented using the faster static random-access memory (SRAM). When an access to the cache hierarchy completes, the cache line containing the requested data is brought to each level that missed [33].

Due to the variability in access latencies of cache hits and misses, the cache hierarchy creates an effective timing-based side-channel [9, 34, 7]. For example, by timing access latencies across multiple cache lines, an adversary may infer which address has been recently loaded by the processor. This can be used to derive information which would not otherwise be legitimately available. For example, **Flush+Reload** [9] can be used to derive encryption keys leaked through the L3 cache, and **Meltdown** [8] exploits cache-timing to leak arbitrary kernel memory.

2.1.5 Memory Objects

The hardware accesses a memory location through a *physical address* [30]. However, to facilitate isolation and multi-programming, a process (running instances of a program) accesses memory through the *virtual address* [2], which gets translated to the corresponding physical address by the hardware and the operating system. Furthermore, a process keeps track of its data in memory, and related data is often grouped together (a concept known as *spatial locality*). We define a single chunk of contiguous data held in memory that semantically identifies a single object as a *memory object* (e.g. arrays, structs, or a bitmap). It is therefore possible to identify a memory object by its *bounds*, which are its starting and ending virtual addresses.

2.1.6 Performance Metrics

Computer architects use *cycles per instruction* (CPI) to measure instruction throughput [30]. CPUs incorporate an internal *clock* to synchronise sequential circuits (any hardware that holds state, such as flip-flops [30]). The clock ticks at a fixed frequency, with each tick known as a *clock cycle* (or cycle for short). For example, a clock frequency of 2 GHz means that 2 billion cycles pass per second. A program that commits N instructions in C cycles has a CPI of C/N ; a lower CPI means higher throughput.

2.2 Spectre

Although speculative execution maintains correctness in the architectural state, transient instructions may leave observable side-effects such as loading a specific cache line [1]. Observing these effects enables attackers to derive secret information incorrectly produced by transient instructions through a side-channel. Furthermore, transient instructions may be misdirected to load highly confidential secrets such as cryptographic keys. This class of attacks is known as *Spectre* [1], exploiting speculative execution to leak secrets through microarchitectural side-channels.

2.2.1 Accessors and Transmitters

In a Spectre attack, a speculative *accessor* first loads a secret from memory, and then a younger *transmitter* leaks it through a side-channel [10, 11, 15, 13]. A successful transmission of a transient secret requires the access-transmit instruction sequence to hide behind a mispredicted unresolved branch, and the transmitter resolving before they are squashed. In general, loads are classified as accessors, and many different types of instructions can be classified as a transmitter. Loads are also classified as transmitters, due to the high-observability of cache-based side channels, shown to lead to practical attacks [9, 11, 35]. Depending on the microarchitecture, floating-point instructions [36]

2 Background

and branch instructions [10] can also form side-channels and thus be classified as transmitters.

2.2.2 Example

Figure 1.1 shows an example Spectre attack, bypassing the $x < \text{size}$ bounds check. Note that the program itself is correct, as the correct execution path never loads an out-of-bound value from A. First, assume that the attacker controls the variable x . The attacker then mistrains the branch predictor to predict *taken*, even though an out-of-bound value of x is supplied. This allows any memory location past A to be read into s transiently, which then selects a cache line by influencing the load from B. Finally, the attacker measures memory access timing using techniques like **Flush+Reload** [9] and **Prime+Probe** [34] to find which cache line is hot. This information then allows the attacker to infer the secret s . This variant of Spectre is known as *Bounds Check Bypass*.

2.2.3 Visibility Point

The visibility point [11, 10] is a ROB index where all instructions past this point is no longer speculative, and all other instructions (younger than the visibility point) are speculative. In the Spectre threat model [1], an instruction has passed the visibility point if all older branches are resolved, i.e. no mispredicted branches can squash it. Importantly, it follows that instructions pass the visibility point in program order.

2.2.4 Transient Execution Attacks

The discovery of Spectre led to the development of a wider class of attacks, known as *Transient Execution Attacks* (TEAs). In general, any prediction mechanisms or instructions capable of squashing younger ones pose a security risk. Aside from branch prediction, exception handling and memory disambiguation could also be exploited by an attacker. TEAs encompass misdirecting and exploiting any transient instructions to create a side-channel leakage. In addition to Spectre, a widely discussed TEA is Meltdown [8], which exploits page faults and lazy protection checks to break memory isolation. Other examples include ZombieLoad [37], which leaks stale data in the line-fill buffer of the L1 cache. Although our work targets Spectre, it is easy to generalise it to many other TEAs by redefining the visibility point to consider any instructions capable of squashing younger instructions (such as loads which may raise a page fault, to defend against Meltdown). We leave performance evaluations of such generalisations to future work.

2.3 Speculative Taint Tracking

Speculative Taint Tracking (STT) [10] is a leading hardware-level Spectre defence. It performs information flow tracking of transient data. STT considers a physical register as *tainted* if it holds transient information which should be protected from leakage. The taint tracking mechanism operates as follows.

1. A speculative load taints its output register, as it may have loaded a secret.
2. Taint is then propagated through dependencies. For example, an instruction's output register is tainted, if any of its input registers are tainted.
3. Tainted instructions classified as transmitters are stalled until it is no longer tainted.
4. Finally, registers become untainted when all accessors causing its taint become safe. These accessors are known as the *taint roots*. A taint root becomes safe when it passes the visibility point, as its output is no longer speculative.

Since an instruction may depend on multiple registers, there may naturally be multiple taint roots. Since instructions pass the visibility point in program order, it suffices to track the *Youngest Root of Taint* (YRoT), which holds the index to the ROB entry of the youngest taint root. When the YRoT passes the visibility point, all older taint roots must have also passed the visibility point. This makes STT's hardware especially lightweight, as each instruction only needs to track a single taint root.

2.3.1 Hardware Mechanism

Each physical register is assigned an additional `yrot` and `AccessInstrIdx` field. `AccessInstrIdx` is set to the ROB index of the producing instruction if the instruction is a load (and thus a taint root). `yrot` is set to the YRoT ROB index, when the instruction is tainted through propagation. A register is considered tainted if `yrot` or `AccessInstrIdx` is valid. Taint propagation happens at the register rename stage, and to facilitate untainting, the `yrot` value is carried to reservation stations. When an accessor crosses the visibility point, it broadcasts its ROB index to younger instructions, and any instruction whose YRoT points to an older instruction is untainted and can thus safely execute.

2.3.2 Example

Figure 2.1 is an example of STT's taint tracking logic. At first, `i2` is the oldest unresolved branch, and thus all younger instructions are considered speculative. `i3` is an accessor, which brings a secret into `r2`, causing it to be tainted. Taint then propagates through dependencies, through `i5`, it spreads to `r3`. As a result, the transmitter `i6` is

2 Background

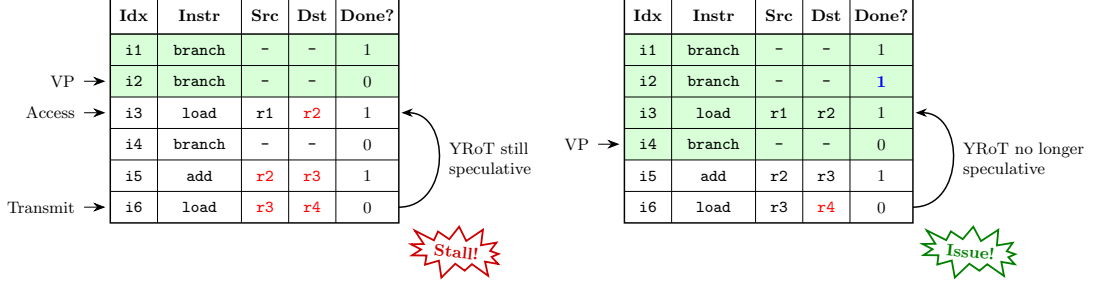


Figure 2.1: An example of taint propagation and stalling logic of STT. The green-shaded instructions are those which passed the visibility point, and is therefore non-speculative. Registers coloured red are tainted.

stalled, despite its input operands being ready. The diagram on the right shows the state of inflight instructions some number of cycles later, where *i2* has resolved, moving the visibility point to *i4*. Since the accessor *i3* has passed the visibility point, *i6* can now safely issue. Note that *r4* remains tainted, since *i6* is still a speculative load despite being untainted.

2.3.3 Safe Memory Objects

A memory object is *safe* if its data are legitimately accessed on the correct execution path. Since Spectre is used to leak data that cannot be legitimately accessed, transmitting safe memory objects constitute to a traditional side-channel attack and is excluded from the Spectre threat model [16, 1, 10]. We also define a load as *safe*, if its resolved address points to a safe memory object. STT conservatively classifies all loads as unsafe, since the whether a load is safe is not known at the hardware-level. Furthermore, we define a *secret* as any data which do not lie inside a safe memory object.

Related Work

This chapter discusses related Spectre and TEA defences. We first discuss mitigations that target specific Spectre variants or side-channels. We then discuss comprehensive mitigations which prevent transient data from spreading to transmitters in the first place. Finally, we discuss related work that communicates semantic information from the software stack to the hardware in order to improve performance.

3.1 Limited Mitigations

Since Spectre, a wide range of variants and side-channels have been discovered. Early mitigations often targeted specific Spectre variants, while later works have proposed generalised approaches by closing the cache-timing side-channel.

3.1.1 Targeting Specific Variants

In Bounds Check Bypass (Spectre-V1) [1], an attacker transiently bypasses a bounds check (e.g. Figure 1.1). A widely discussed defence is to insert an `LFENCE` after the check [19], stalling younger instructions until the fence executes, disabling speculation over protected code at high performance costs [11, 21, 16, 15]. Intel also recommends masking the loaded value with a conditional move (i.e. `CMOV`) before the transmitter, which is safe because current Intel processors do not predict conditional moves [38].

Branch Target Injection (Spectre-V2) [1] mistrains the indirect branch predictor to transiently execute arbitrary code. Mitigations include Intel’s indirect branch controls [39, 40, 41]; SpecCFI [24], which exposes control-flow integrity information [42] to speculative execution; and Retpoline [23], which replaces indirect branches with return instructions. However, Retbleed [43] shows that return prediction can also be exploited under certain conditions. Speculative Store Bypass (Spectre-V4) [44, 45] allows loads to transiently read stale data before an older store’s address resolves, mitigated by preventing loads from executing until older store addresses are known [44]. Finally, the closely related Meltdown attack [8] can be mitigated using KAISER [46], which strengthens the isolation between kernel and user memory.

3 Related Work

Variant-specific mitigations are limited in their generality and sometimes have high performance costs (such as using `LFENCE`). This motivates mitigations secure against all attack variants, by addressing the cache-timing side-channel itself as the root cause.

3.1.2 Targeting the Cache

Several proposed mitigations have targeted the cache hierarchy [11, 22, 18, 25, 12], eliminating cache-based side-channels. SafeSpec [22] and InvisiSpec [11] stores transient data separately from the cache, committed to the cache hierarchy only when it becomes non-speculative. Unfortunately, these shadow structures introduce significant hardware overhead, and synchronising them to cache coherence protocols [47] incurs high performance costs [11]. In contrast, DAWG [18] isolates cache hits and misses across protection domains, and HidFix [25] rolls back transient data leaks.

Unfortunately, cache-timing is not the only side-channel that can be exploited. For example, some floating-point instructions have data-dependent execution latencies, which may be exploited by an attacker [36]. NetSpectre [48] allows an adversary to conduct Spectre attacks over a network by exploiting channels based on the Advanced Vector Extension (AVX). Thus, although mitigations that target the cache generalise well over Spectre variants, they are still not comprehensive Spectre defences.

3.2 Comprehensive Mitigations

The poor generality of limited mitigations motivated the need for mitigations to be comprehensive over all Spectre variants and side-channels. The idea is to stall an accessor or transmitter until these instructions are known to be safe, by containing or tracking the spread transient information through dependencies. These approaches have the following advantages:

1. It generalises well over any type of instructions, the computer architect may classify an instruction as an accessor or transmitter as they wish.
2. It introduces low hardware overhead, since tracking transient information largely involves combinational logic, without the need for expensive shadow buffers.
3. It creates an abstraction for orthogonal work at the software-level to further improve performance.

DelayExecute [10] is the simplest approach, which prevents accessors from executing until they pass the visibility point, so secrets never enter the transient state in the first place. However, *all* loads must then wait for all older branches to resolve, which causes a significant performance impact [10]. A more efficient approach, used by SpecShield [21] and NDA [13], let loads execute speculatively, but delays the wake up of their dependents

until the load becomes non-speculative. We refer to this as **DelayWakeup**. It introduces very little hardware overhead and lower performance degradation. It is also a natural fit for software-hardware communication based mitigations like SSMR [16], since whether a load is treated as an accessor can be decided at address resolution time, whereas STT must make this decision at rename, before the address is available. However, stalling all dependents is still conservative, since it also blocks instructions that could never transmit a secret. STT [10] addresses this with taint tracking, stalling only transmitters that are connected to an accessor through a transient dependency chain. This costs more hardware than DelayWakeup, but still remains low.

3.3 Software-Hardware Communication

The lack of semantic information about the program state in the hardware has led to the development of comprehensive mitigations that allows the software stack to communicate the secrecy or safety of data held in memory. These approaches significantly reduce performance costs, but unfortunately, they often introduce other trade-offs such as hardware overhead that are difficult to circumvent.

3.3.1 SSMR

The work most similar to STTHint, and its main inspiration, is SSMR [16]. SSMR provides a compiler pass which inserts `ssmr` instructions that communicate speculation-safe memory regions to a hardware buffer. When a load's virtual address resolves to a region held in the buffer, the loaded value is forwarded to speculative dependents as normal. Any other load is treated as an accessor under a DelayWakeup policy, so its dependents do not wake up until it becomes non-speculative. STTHint shares this core idea of communicating to the hardware which memory regions are safe to transmit, but differs from SSMR in three key ways.

1. SSMR uses DelayWakeup as its security mechanism. STTHint instead uses STT, a more efficient mechanism which only stalls transmitters down the dependency chain.
2. The SSMR compiler pass communicates a large number of memory regions, even those that do not cause a significant performance cost. Storing this information creates a higher hardware overhead. Instead, STTHint exploits the fact that many stalls are concentrated in a small number of code regions, which means fewer memory objects need to be communicated to the hardware to effectively reduce performance cost (see Chapter 4).
3. Unlike STTHint, SSMR cannot communicate dynamically sized regions.

3 Related Work

We hypothesise that SSMR is bottlenecked by its inability to hint dynamically sized memory regions, since many performance-critical applications are dominated by loads from dynamically sized data structures. For example, STT incurs a very high performance overhead on the GAP workload `cc_sv`, and we find that almost all problematic accessors are loads from a dynamically sized array. Because SSMR’s evaluation over GAP workloads does not include `cc_sv` (and many other GAP benchmarks), it is unclear whether the reported performance improvements would hold for this benchmark. The authors also integrated SSMR with STT, but the methodology in doing so is not reported at all. This is non-trivial because STT must classify a load as an accessor at rename, before its address is known, a problem we address for STTHint.

3.3.2 Protean

Protean [17] communicates the secrecy of memory to the hardware. Protean’s ISA extension allows its compiler to automatically specify at each program point exactly which registers and memory bytes require protection, giving it finer control than SSMR. Like STTHint, Protean extends STT’s taint tracking with a predictor structure similar to ours. We note that our hint predictor (Section 5.4) was developed independently, before we became aware of Protean. However, Protean has several limitations. First, Protean’s ISA extension marks which data is *protected* rather than which data is *safe*. As a result, secret-accessing code not compiled with Protean’s custom compiler remains vulnerable even with Protean’s hardware defences enabled. Every security-critical library, dependency and program must thus be recompiled. In contrast, STTHint falls back to full STT protection for any unhinted legacy code. Second, Protean tags every byte in the L1 cache with a protection bit, which requires a parallel bit array whose size scales with the cache, regardless of how many regions actually need protection. STTHint instead uses a small, fixed-size hint buffer, exploiting the observation that a few contiguous memory regions account for most STT stalls in many workloads. Finally, because protection bits live only in the L1, Protean conservatively treats every L1 miss as a load of protected data. It is unclear how this affects memory-bound workloads with high miss rates.

Motivation

The core motivation behind STTHint is that for many workloads, STT stalls come from static instructions concentrated in a few code regions. We observe that these instructions largely belong to a class of code patterns we call *taint-critical regions* (TCRs). Effective hinting should therefore target the memory regions accessed by TCRs, while hinting other regions brings limited benefit.

4.1 Instruction Blame

To proportionally attribute STT overhead to individual static instructions, we introduce the *instruction blame* metric, which counts how often a static instruction is responsible for an STT-stalled instruction. *Accessor blame* is attributed to accessors and *transmitter blame* to transmitters, estimating each instruction’s share of responsibility for STT stalls. Note that a transmitter may be tainted by multiple accessors, but STT tracks only the *youngest* taint root, so the latency of older roots reaching the visibility point is hidden behind that of the youngest. Useful accessor blame must therefore account for *all* taint roots of a stalling transmitter.

4.1.1 Formulation

For a given workload, we first compute the following.

1. $\mathbf{T}$, total number of dynamic transmitters stalled for at least one cycle due to STT.
2. T_i for each static instruction i : portion of $\mathbf{T}$ caused by dynamic instances of i .
3. $\mathbf{A}$, the total number of dynamic accessors that taint at least one transmitter that stalls for at least one cycle while the accessor is still speculative. A dynamic accessor is counted once for each stalled transmitter it taints.
4. A_i for each static instruction i : portion of $\mathbf{A}$ caused by dynamic instances of i .

4 Motivation

Let S be the set of all static instructions. Since every dynamic instruction comes from exactly one static instruction, the following statements hold.

$$\sum_{i \in S} T_i = \mathbf{T}, \quad \sum_{i \in S} A_i = \mathbf{A}.$$

We define the *transmitter blame portion* of a static instruction i as $T_i/\mathbf{T}$, and its *accessor blame portion* as $A_i/\mathbf{A}$. Both blame portions estimate static instruction responsibility for STT's performance cost. We blame accessors on the first cycle a transmitter stalls, as this captures all guilty taint roots, whose number decreases as they pass the visibility point. Note that each problematic dynamic instruction contributes equally to the blame, regardless of how many cycles it stalls. Thus, this metric only approximates the contribution of a static instruction to the overall CPI degradation of STT.

4.1.2 Blame Stack

Instruction blame can be visualised using a stack. The height of each component on the stack represents a single blame portion. To find the most problematic instructions, we sort the blame stack by component height. Note that the colours of stack components have no significance, used only to differentiate component heights.

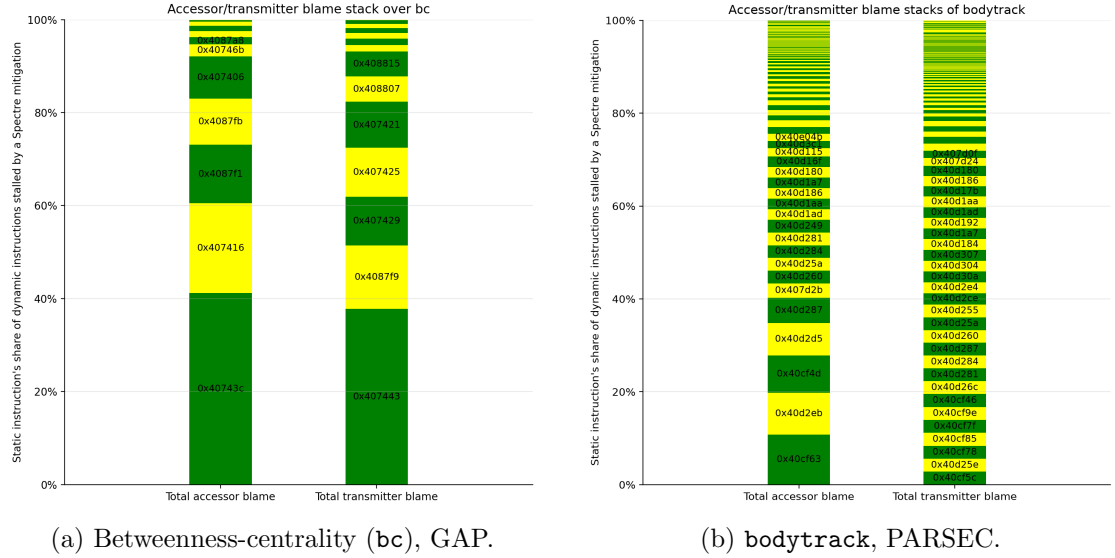


Figure 4.1: Example blame stacks. In each subfigure, the stack on the left is the accessor blame stack and the stack on the right is the transmitter blame stack.

Figure 4.1a shows an example blame stack for the GAP benchmark bc, where more than 40% of the responsible accessors come from a single static instruction address, and just 4 static instructions make up more than 80% of the accessor stack. Figure 4.1b is a

blame stack for the PARSEC app **bodytrack**. In contrast, the components in this stack are smaller, and there are more critical instructions in the program binary than there are in the GAP benchmark. This is expected, as PARSEC apps are larger than GAP workloads (GAP workloads are kernels).

4.1.3 Blame Distribution

An alternative visualisation is the *blame distribution*, which shows the blame portion of each static instruction address in the program binary ordered by address along the horizontal axis. The blame distribution gives us insight into whether blame portions are evenly spread or highly concentrated. Note that instructions outside the program source, such as dynamically linked library code, are not considered.

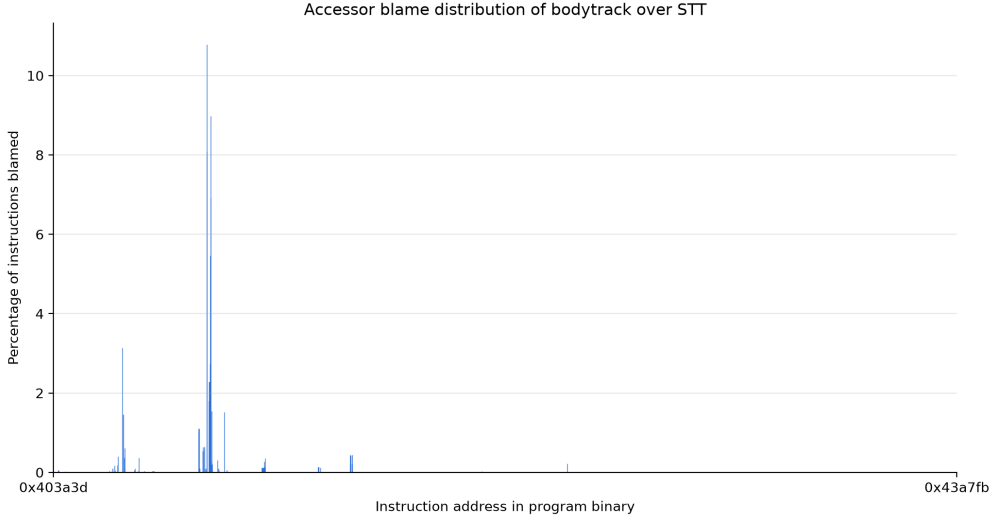


Figure 4.2: Accessor blame distribution of **bodytrack**

Figure 4.2 shows the accessor blame distribution of **bodytrack**. The two major spikes in the plot show that STT’s performance cost comes from static instructions concentrated in relatively small code regions.

4.1.4 Overlapped STT Stalls

Although each blame portion is considered individually, in reality, due to high instruction-level parallelism (ILP), STT stalls from one instruction may be well-hidden by one on the critical path [49] of the OoO engine. This is further complicated by the YRoT hiding transmitter stalls caused by older taint roots. We intentionally do not capture the overlap of STT stalls in the instruction blame metric to expose all responsible instructions, enabling a more comprehensive set of targets when accelerating STT.

4.1.5 Implementation

Alongside STT, we implement our metric in Sniper using hash tables that record the blame for each instruction PC, along with **A** and **T**. Our target benchmarks are compiled with position-independent executable (PIE) disabled and with debug symbols. We use the Linux tool `addr2line` to convert instruction PCs to source line numbers in order to analyse problematic source-level code patterns.

4.1.6 Benchmark Results

We experimentally derive the instruction blames for GAP benchmarks and PARSEC applications. Figure 4.3 shows the minimum number of accessors and transmitters needed to make up a majority of the blame stack. For GAP benchmarks, only 1–3 instructions

(a) GAP					(b) PARSEC				
Benchmark	A		T		Benchmark	A		T	
	Min	Total	Min	Total		Min	Total	Min	Total
bc	2	246	2	294	blackscholes	2	62	2	91
bfs	3	240	2	304	bodytrack	9	438	19	711
cc	2	195	3	230	facesim	6	396	14	681
cc_sv	1	193	1	238	ferret	18	1760	35	2409
pr	1	188	1	232	fluidanimate	10	324	9	318
pr_spmv	1	195	1	240	freqmine	12	207	16	229
sssp	2	325	2	463	raytrace	25	626	50	864
tc	2	211	2	231	swaptions	7	265	17	357
					vips	26	768	33	1081
					x264	23	1457	20	1525

Figure 4.3: Minimum number of accessors (A) and transmitters (T) needed to make up a majority (50%) of the blame stack (Min), and the total number of accessor/transmitter instructions with at least one blame (Total).

account for the majority of the blame, suggesting that recovering baseline speculation for these few instructions could substantially reduce STT overhead. PARSEC is more varied, as blame is concentrated in a few instructions for workloads such as **blackscholes**, but more spread out for workloads such as **raytrace** and **ferret**. However, **raytrace** also has far more instructions with non-zero blame (626, versus 62 for **blackscholes**), and likewise, **ferret** has a total of 1760. Relative to this total, both are similarly concentrated: in **blackscholes** and **raytrace**, only approximately 3–4% of blamed accessors account for half of the accessor blame, while for **ferret**, only 1% account for half of the accessor blame.

4.2 Evaluating STT

To further motivate STTHint, we evaluate STT on the GAP and PARSEC benchmarks and analyse the results. All experiments follow the methodology described in Chapter 6.

4.2.1 Results

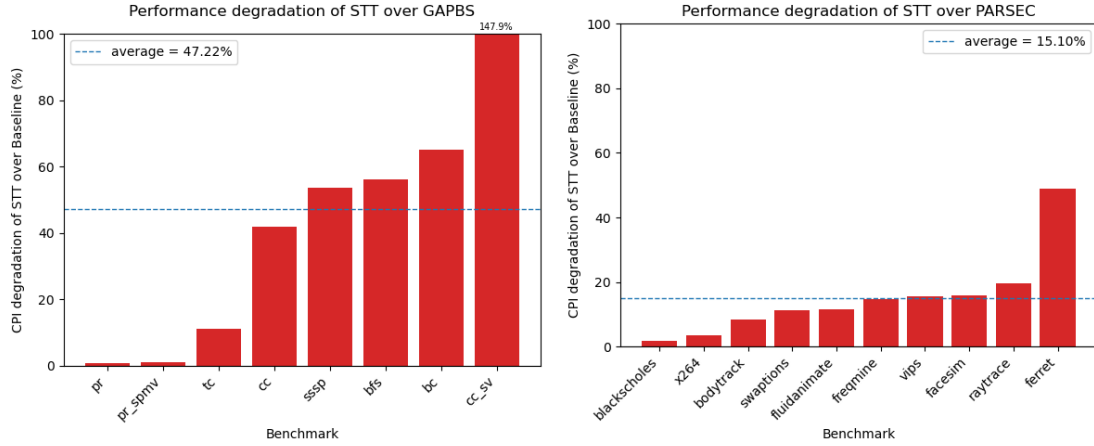


Figure 4.4: STT CPI overhead over the insecure baseline for GAP and PARSEC workloads.

Figure 4.4 shows the CPI overhead of STT relative to an insecure baseline. STT degrades performance by 15.10% on average for PARSEC, and an average of 47.22% for GAP. Its impact is also highly variable: depending on the workload, CPI ranges from close to baseline to double the baseline.

This difference arises because GAP benchmarks are graph kernels dominated by pointer chasing over array-based graph structures and, more generally, by *taint-critical regions* (Section 4.2.2). Because these kernels spend much of their runtime inside these regions, they suffer far more than our PARSEC benchmarks, which are general-purpose applications with more varied code patterns.

4.2.2 Taint-Critical Regions

For a transmitter t to stall on its YRoT a , both a and t must resolve before a reaches the visibility point; otherwise, a is already safe and t is untainted. Let A and T be the PCs of a and t . For A and T to cause a *large* number of STT stalls, they also need *multiplicity*: many dynamic instances of both must be dispatched into the ROB.

4 Motivation

Due to instruction locality, this almost always arises from *loops*. We therefore define a *taint-critical region* (TCR) as a local region of static instructions that incurs very high STT overhead. A TCR meets three criteria:

1. *Access-transmit sequence*: a transmitter t has its YRoT set to an accessor a in the ROB.
2. *Critical branches*: slow-to-resolve branches let the operands of t become ready before a reaches the visibility point, typically causing STT stalls.
3. *Multiplicity*: a long-running loop, possibly with deep, nested speculation, dispatches many instances of the critical branches and access-transmit sequences into the ROB.

Because a TCR requires several specific conditions to hold, TCRs are scarce and concentrated in a few regions of the program. As a result, only a small number of static instructions need to be handled, which is further highlighted by our aforementioned instruction blame results. Moreover, a static load usually maps to the same memory object on each dynamic instance. This does not always hold (for example, in linked-list traversals), but it holds for memory objects with good spatial locality (like arrays). A few critical static instructions therefore map to a few memory objects, so only a small number of hints are needed, keeping the hardware overhead low while substantially reducing STT's performance cost (see Chapter 6).

4.2.3 TCR Example

```
for (NodeID v : g.in_neigh(u)) {           // accessor
    if (front.get_bit(v)) {                 // transmitter
        parent[u] = v;
        awake_count++;
        next.set_bit(u);
        break;
    }
}
```

Figure 4.5: An example TCR the GAP bfs benchmark.

Figure 4.5 shows an example TCR from GAP's bfs benchmark, whose accessor and transmitter account for 34.3% and 37.3% of their respective blame stacks. The accessor loads a neighbour of a graph node into v , which supplies the address operand of the transmitter, the load in `front.get_bit(v)`. The `if` statement on its result forms the critical branch, and the `for` loop provides multiplicity.

STTHint

This chapter describes our threat model and STTHint. We first establish the programmer interface (ISA extension), then the *hint buffer* used to store hints. Finally, we address the challenges of classifying loads as safe or unsafe at the rename stage.

5.1 Threat Model

We extend the threat model of SSMR [16]. The adversary interacts with the victim program as a user by supplying inputs. The adversary may observe any side-channels including cache-timing [9], data-dependent arithmetic timing [36], and branch-driven implicit channels [10]. The attacker has full knowledge of the hardware and can arbitrarily mistrain the branch predictors. We only consider transient execution caused by branch misprediction, so an instruction reaches the visibility point once all older branches resolve. Traditional side-channels that transmit data accessed on correct execution paths which lies outside the Spectre threat model are also outside our scope, and many prior works already address these attacks [7]. Like prior work [16, 17], we trust the programmer communicate information *correctly* to the hardware (see Section 5.2). Furthermore, we consider a single victim process on a single hardware thread, and leave the handling of hints across context switches and simultaneous multithreading (SMT) to future work (see Chapter 7).

5.2 Hints

A *hint* H is defined as a 48-bit virtual start address H_s and a 48-bit virtual end address H_e of a memory object. We use 48 bits instead of 64 bits to save space, as modern operating systems only utilise the lowest 48 bits in a virtual address [15, 50, 51]. A hint H can be set *active*, in which case any data addressed by v such that $H_s \leq v < H_e$ is considered safe, and can be transmitted. We refer the process of setting a hint active as *hinting* a memory object. Preferably, loads to hinted data should not be considered a root of taint.

Semantically, hinting an object declares that its bytes are legitimately accessed while the hint is active; *unhinting* it or *deactivating* the hint declares the memory location no longer public. Within a TCR, frequent accesses to a hinted region transmit only data already exposed through conventional side channels, so hinting removes STT's stalls without a new leakage. Data outside hinted regions keeps STT's full protection, so with correct hinting STTHint transiently leaks only what STT already permits. Correct hinting requires that (1) an object is hinted only if it is legitimately accessed before being unhinted, and (2) it is unhinted before being freed or overwritten with unsafe data.

5.2.1 ISA Extension

To facilitate hinting, we extend the x86-64 instruction set with two new instructions.

1. **shint**, which takes two register operands holding H_s and H_e and hints H .
2. **sunhint**, which takes a single register operand holding an address, deactivating all hints containing that address.

These instructions only executes non-speculatively (when they pass the visibility point), as otherwise transiently executed hint instructions may be misdirected by an attacker, forming a new TEA. Furthermore, OoO execution may allow a younger load to bypass an older **sunhint**, incorrectly enabling transient leakage of data in a memory region that should have been unhinted. Thus, we make **sunhint** serialise all younger loads: younger loads cannot execute until the **sunhint** itself executes. Although these restrictions introduce a performance cost, such cost is negligible, as usually, a TCR itself takes much longer execute compared to the constant number of **shint** and **sunhint** instructions surrounding it. Furthermore, keeping multiple **sunhint** instructions close together allows the latency of serialising younger loads to overlap with each other.

```
shint(array, array_len * sizeof(int));
for(int i = 0; i < N; i++){
    if(y % 10 == 0) y += 1;
    y += array[array[i]];
}
sunhint(array);
```

Figure 5.1: High-level example usage of **shint** and **sunhint** in C.

Figure 5.1 shows an example of using **shint** and **sunhint**. The instructions are wrapped by higher-level C functions, which are called around a TCR frequently accessing **array**. For ease of use, the C function **shint** takes the starting address and the size of the memory object, instead of the ending address.

5.3 The Hint Buffer

The hardware must keep track of active hints. We introduce the *hint buffer*, which holds 16 entries, each entry holds a single hint: two virtual addresses, and a valid bit. The hint buffer is fully-associative; each entry needs the mechanism to check if a given virtual address v satisfies $H_s \leq v < H_e$. Thus, each entry is paired with two comparators. We say a load *hits* the hint buffer if its target address v lies within an active hint. A load that does not hit the hint buffer is said to *miss* the hint buffer.

When an **shint** instruction is issued with H , the buffer first checks if it is adjacent to an existing hint by checking whether there exists an active hint H' such that $(H_s = H'_e)$ or $(H_e = H'_s)$. If this condition holds, H'_s or H'_e will be extended to accommodate H . This mechanism can reuse the equality comparator for H'_s , but requires an additional equality comparator for H'_e . This optimisation is motivated by the fact that often times, we may hint several adjacent memory objects at once, such as hinting statically allocated variables on a function stack frame. If no existing hint can be extended, we will instead allocate a free entry to hold H . If the HB is full, then the **shint** instruction is ignored. When a **sunhint** instruction is issued, with a virtual address v , the buffer checks invalidates all entries such that the hint it holds contains v .

5.4 Hint Prediction

Ideally, a load that hits the hint buffer should not be classified as an accessor (and thus its output register should not be tainted). However, STT's taint algorithm happens in the rename stage, and the target memory address of a load is unknown at this stage, which means we cannot check the hint buffer prior to tainting. Attempting to untaint on a later hint buffer hit is challenging, as it violates the property that accessors untaint their transmitters in program order, which breaks the invariant that if an accessor becomes safe, all older accessors are safe.

5.4.1 Hint Predictor

To solve this issue, we instead predict the hint buffer outcome of a load at register rename stage. Often times, a static instruction maps to a single memory object. Thus, if a load hits the hint buffer, the same load will likely hit it again. For example, in Figure 5.1, the load instruction corresponding to `array[i]` almost always accesses `array`. This locality allows us to introduce a *hint predictor* which uses the instruction PC to predict whether a load will hit the hint buffer, in which case it predicts **safe** (S), or otherwise it predicts **unsafe** (US). A load instruction is only considered an accessor if it predicts S. In the situation where the instruction is predicted as **safe**, but actually misses the hint buffer, it is too late to propagate taint. Instead, we set a **DelayWakeup** bit next

to the instruction’s load queue entry, and conservatively delay waking up dependant instructions until the accessor reaches the visibility point. Since DelayWakeup is a more conservative strategy than taint tracking, it incurs a higher performance overhead. However, temporal locality creates a very high hint predictor accuracy in GAP and PARSEC benchmarks (see Table 6.2), making fall back to DelayWakeup rare in most programs.

Taking inspiration from early branch predictors [52] and a similar predictor in PROTEAN [17], we use a simple *1-bit bimodal* predictor. We introduce a *hint predictor table* of 1024 0-initialised 1-bit entries, each set if the most recent prediction was S. Each entry is indexed by the least significant 10 bits of the instruction PC. In the rename stage, whether a load taints its output register depends on the outcome of the predictor table. Furthermore, a load instruction will update the table, when the hint buffer is accessed.

Figure 5.2 demonstrates an example of hinting and unhinting a memory object. A load passes through the hint predictor, which gets classified as either safe or unsafe.

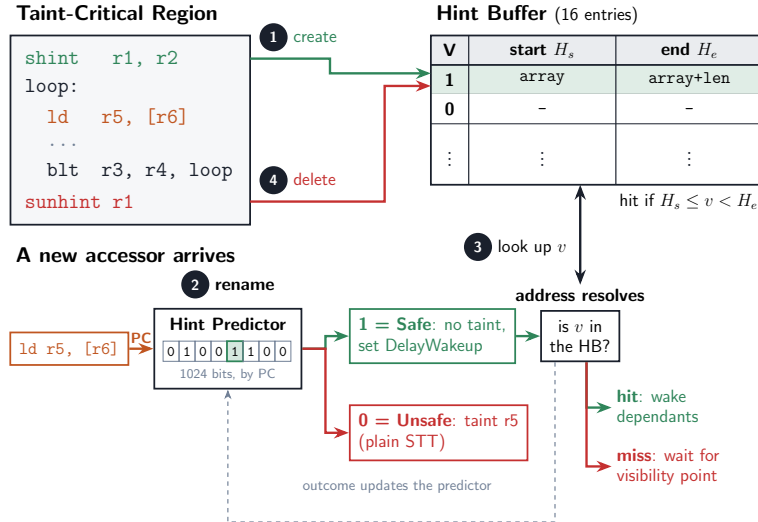


Figure 5.2: STTHint example including hint creation, deletion, and a load passing through the hint predictor.

5.4.2 Securing Store-to-Load Forwarding

STT does not propagate taint through the store queue, which is safe when all loads are treated as accessors. But for STTHint, a speculative store may write a tainted value to a hinted region, which violates the invariant that all data in hinted regions are not tainted. We patch this vulnerability by enforcing that loads to a hinted region which aliased with a resolved older store fall back to DelayWakeup.

Evaluation

In this chapter, we first describe our evaluation methodology, then we present the findings from our experiments, while discussing notable insights derived from our results.

6.1 Methodology

To evaluate hardware performance, we use The Sniper Multi-Core Simulator [53], for its efficiency in simulation time, and accuracy in capturing the altered instruction scheduling algorithm of STT and STTHint.

6.1.1 Benchmark Suites

GAP benchmarks [29] expose the performance issues of STT, since GAP kernels are graph algorithms with TCRs involving complex, transient pointer-chasing and dependencies, which we correctly suspected to cause significant performance issues over STT. Furthermore, GAP benchmarks are relatively small, allowing us to narrow down the sources of STT stalls more easily. The authors of SSMR have also reported performance impacts of STT over a subset of GAP workloads. However, we present results covering the full benchmark suite, along with a deeper analysis in Chapter 4.

We also run our experiments against PARSEC *applications*, which are a set of larger, general-purpose programs written in C and C++. We present STT performance costs for every PARSEC application, and STTHint results for the **ferret** workload, since **ferret** incurs the highest CPI overhead. Hints are currently inserted manually, which is time-consuming for larger codebases such as PARSEC ones. Since this work aims to show that hints can be effective rather than to automate their insertion, we leave automatic hint insertion to future work (see Chapter 7).

6.1.2 Hardware Simulation

We use Sniper’s Gainestown configuration, which models Intel’s Xeon x5500-series processors [54]. We implement STT and STTHint in Sniper’s ROB performance model, simulating the instruction scheduling algorithm. We focus our simulation on a single-core setting, as STT does not alter cache coherence or inter-core communication. For STT, we classify loads, branches, floating-point multiplication, division and square root instructions as transmitters. Doing so closes cache-based side-channels [1], floating-point timing side-channels [36] and branch-related implicit channels [10]. For STT, we classify all load instructions as accessors and define the visibility point according to the Spectre threat model (Section 5.1). Table 6.1 shows our simulation parameters.

Table 6.1: Simulation parameters.

Parameter	Value
Architecture	1 Core at 2.66 GHz
ROB size	128 entries
Dispatch width	4 instructions
Issue width	5 instructions
Branch Predictor	Pentium M predictor
L1 Cache	32 KB in both I-cache and D-cache
L2 Cache	256 KB
L3 Cache	8192 KB

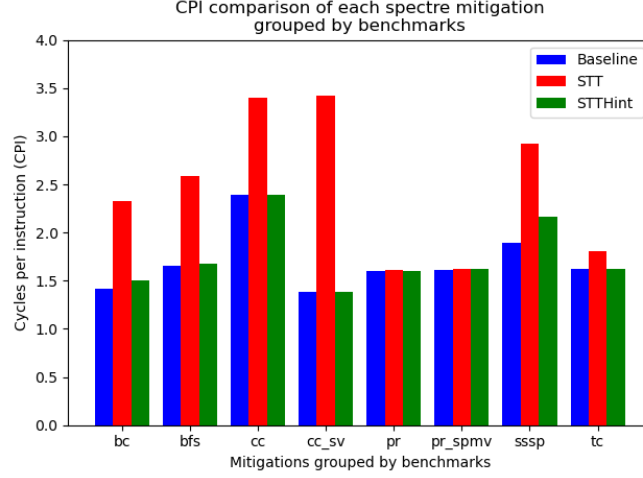
GAP experiments use randomly generated graphs of size 2^{18} . We exclude graph generation time from our region of interest, and workloads are run to completion. For PARSEC benchmarks, we use the `simmedium` input size. For each PARSEC experiment, we perform a cache warm-up of 3 billion instructions followed by a detailed simulation of another 3 billion instructions.

6.1.3 Hinting Methodology

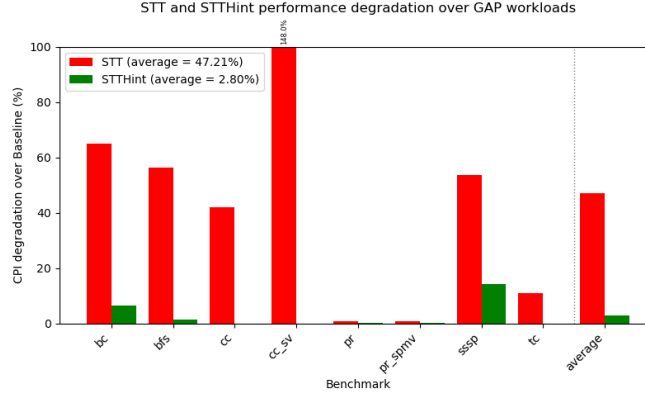
Guided by blame stacks, we surround TCRs with `shint` and `sunhint` instructions. We only consider regions that satisfy the TCR criteria described in Section 4.2.2, while ignoring code regions from linked libraries. Furthermore, hints are inserted securely according to our threat model (see Section 5.1).

6.2 Empirical Results

Figure 6.1a compares the CPI of STT and STTHint for each GAP workload, and Figure 6.1b compares the performance degradation of STT and STTHint for each GAP



(a) CPI of GAP benchmarks on the (insecure) Baseline, STT and STTHint processors.



(b) Performance degradation of STT and STTHint relative to baseline.

Figure 6.1: Performance results of our GAP experiments.

workload. Across all GAP experiments, we found that 16 entries in the hint buffer is sufficient to hold hints needed to significantly reduce STT performance costs.

Our experiments shows that with good hint placement, STTHint effectively accelerates STT by reducing its performance degradation from an average of 47.21% down to just 2.80%. Across `cc`, `cc_sv` and `tc`, STTHint effectively recovers baseline performance, while STT is already efficient over `pr` and `pr_spmv`. `bc` still incurs a performance degradation of 6.42%, while `sssp` incurs a performance degradation of 14.14%.

Our reductions of STT performance degradation supports our claim that STT stalls are concentrated in small numbers of static instructions, and hinting memory objects accessed by these instructions alleviates STT stalls. Figure 6.2 compares the blame stack before and after hints are inserted. The components in the STT blame stack are large,

6 Evaluation

and attributes a large number of stalled instructions to a few accessor/transmitters. After hinting, the only remaining components on the stack are various instructions that contribute little to the remaining STT stalls, such as those from library code. Although STTHint reduces CPI close to the baseline for some GAP benchmarks, for workloads like **sssp**, there is still a nontrivial performance degradation of 14.14%. By observing the remaining instructions on the STTHint blame stack, we find that the remaining performance costs come from accesses to compiler-generated memory objects, either invisible to the application programmer, or difficult to hint correctly. This exposes a limitation of hinting at the source level, which further motivates the need for automated hinting as a late-stage compiler pass (Section 7).

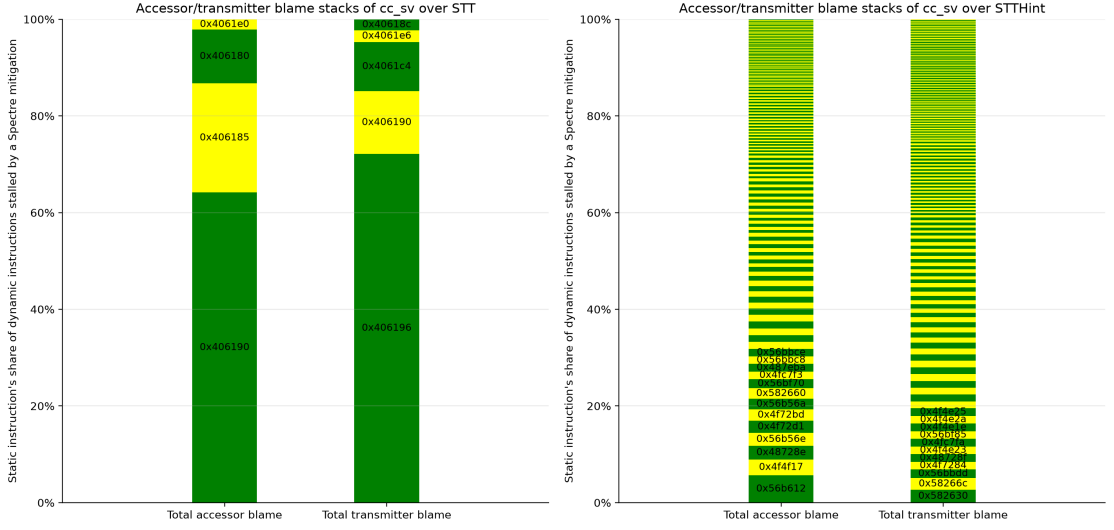


Figure 6.2: Comparing blame stacks of `cc_sv` over (left) STT and (right) STTHint.

6.2.1 Prediction Accuracy

Since we introduce a hint prediction mechanism, it is crucial to assess its *prediction accuracy*. If a load is predicted safe, the prediction is considered *correct* if it hits the hint buffer when its address resolves, and the prediction is considered *incorrect* (mis-prediction), if otherwise. If a load is predicted unsafe, then the prediction is considered correct if it does not hit the hint buffer, and considered incorrect otherwise. We only count an instruction toward the predictor accuracy, if at issue time, the hint buffer is non-empty, as otherwise, an unhinted program running on an STTHint processor would incur a deceptive prediction accuracy of 100%.

We observe a very high accuracy for hint prediction in our experiments, with an average misprediction rate of just 0.2077%. One of our experiments, **sssp** is an outlier with a higher misprediction rate of 1.6596%. The average misprediction rate excluding **sssp** is

0.0003%. Table 6.2 presents the number of correctly and incorrectly predicted loads for each GAP benchmark, along with the miss rate.

Table 6.2: Hint predictor accuracy across GAP kernels.

Benchmark	Correct Predictions	Incorrect Predictions	Misprediction rate (%)
<code>bc</code>	42,457,875	87	0.0002
<code>bfs</code>	6,898,134	83	0.0012
<code>cc</code>	8,239,101	50	0.0006
<code>cc_sv</code>	60,835,855	32	0.0001
<code>pr</code>	76,806,373	40	0.0001
<code>pr_spmv</code>	114,815,029	28	0.0000
<code>sssp</code>	57,910,865	977,319	1.6596
<code>tc</code>	155,941,635	33	0.0000

The high prediction accuracy comes from the fact that if a static load instruction accesses a memory object in a loop, there is a high probability that the load instruction will access the same memory object again. The reason why `sssp` exhibits a relatively high misprediction rate is due to the thrashing of a few instructions in the predictor table. This is caused by a *nested* TCR in the program, where a TCR frequently calls another TCR. We were unable to hint the whole region as a single TCR, since memory is dynamically allocated within the loop itself, and thus, the inner TCR requires its own `shint` and `sunhint` instructions executed for each iteration of the outer TCR loop. Since each hinted region surrounded by `shint` and `sunhint` incurs an initial, compulsory misprediction for each accessor in the region that hits the hint buffer, this compulsory misprediction is repeated for each iteration in the outer TCR loop, causing a relatively higher misprediction rate.

6.2.2 Hardware Overhead

The main hardware overhead of STTHint and SSMR comes from storing hints. SSMR incorporates a Speculation-Safe Region Buffer (SRB), which takes 45 entries [16], each of which contains four 64-bit virtual addresses, and a valid bit [55]. This introduces 11,565 flip-flops (a flip-flop is a digital circuit storing a single bit [30]). Since an SRB look-up conducts a data address bounds check and also a PC bounds check, we suspect that the comparator overhead is also greater in SSMR. In contrast, our hint buffer has 16 entries, each holding just two 48-bit virtual addresses and a valid bit. We use 48-bit virtual addresses to save space, since modern operating systems leaves the upper 16 bits unused [15]. Thus, STTHint only takes 1,552 flip-flops; also counting the hint predictor table (1024 flip-flops), we introduce only 2,576 flip-flops of storage overhead. This approximately reduces the storage overhead of SSMR by 77.73%.

6.2.3 Hint Counts

The core insight from Chapter 4 is that only a small number of hints are needed to accelerate STT. To show that this insight holds, we present the number of dynamic `shint` and `sunhint` instructions executed over each GAP benchmark in Table 6.3.

Table 6.3: Number of `shint` and `sunhint` instructions per GAP benchmark.

	bc	bfs	cc	cc_sv	pr	pr_spmv	sssp	tc
<code>shint</code>	10	7	5	5	3	3	1,048,584	3
<code>sunhint</code>	15	17	7	7	5	5	1,048,583	5

Most workloads have just a few `shint` and `sunhint` instructions, except `sssp`. This is caused by the nested TCR described in Section 6.2.1. However, given that our `sssp` experiments executed 174,828,698 instructions in total, the performance impact of approximately 1 million `sunhint` instructions is low.

6.2.4 Cache Capacity

We also experiment with various L1 and L2 cache capacities; the results are shown in Figure 6.4. We find that the performance costs of STT and STTHint remains similar regardless of cache capacities, and we expect the reduction in STT performance costs to persist across many configurations of the memory subsystem.

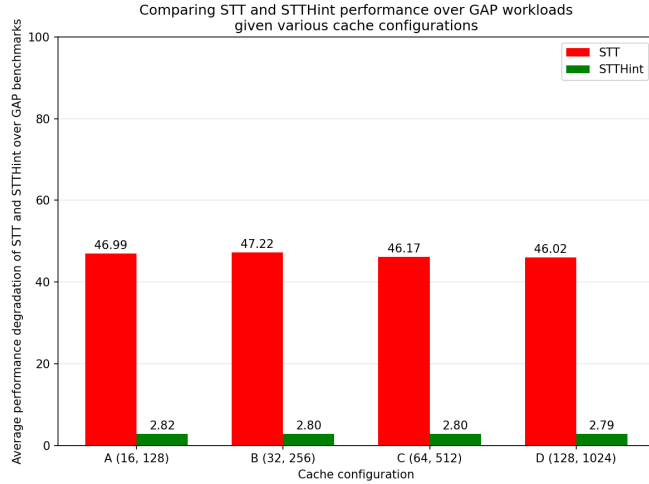


Figure 6.3: Average performance cost of STT and STTHint over GAP benchmarks and various cache configurations. Each pair (L_1, L_2) denotes L1, L2 cache capacities in kilobytes.

6.2.5 OoO Engine

We also vary the ROB size, dispatch and issue width (maximum number of instructions that can be dispatched/issued in one cycle), which are parameters more relevant to the mechanism of STT and STTHint. We find that the performance degradation of both STT and STTHint increases as the ROB size and the dispatch/issue width increase. These results are shown in Figure 6.4.

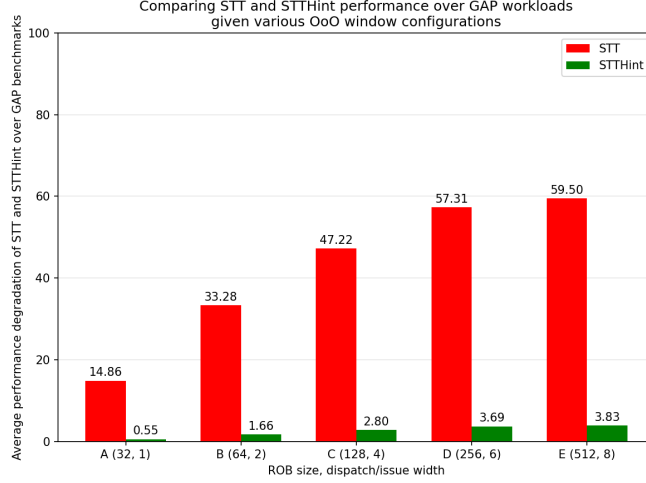


Figure 6.4: Average performance costs of STT and STTHint over GAP benchmarks, and various ROB sizes and dispatch/issue bandwidths. Each pair (r, d) denotes a configuration with ROB size of r entries, and a dispatch width and issue width both set to d . Configuration C is our original configuration in Table 6.1.

This trend is expected. A more powerful OoO engine implies deeper speculative execution, and thus incurs more STT stalls. Our configuration E is closer to OoO engines used in more powerful, contemporary Intel designs such as the Golden Cove [56]. Although this trend also applies to STTHint, STTHint’s reduction in STT performance degradation remains high at 93 – 97% across all configurations. Thus, we find that the relevance to accelerate STT through software-hinting becomes more significant as modern processors use more powerful OoO engines.

6.2.6 PARSEC Case Study

For STT, figure 4.4 shows that the most performance-degrading PARSEC workload is **ferret**, with a performance degradation of 48.84%. We insert hints into the **ferret** source code, and run it over STTHint. We find that STTHint incurs a performance degradation of 15.07%, a reduction of STT performance costs by 69.14%. Out of the

6 Evaluation

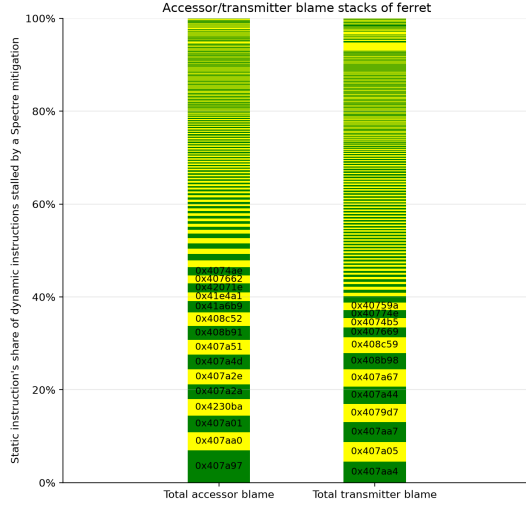
3 billion instructions executed, 934,344 of them were `shint` instructions and 821,306 were `sunhint` instructions. The hint predictor has made approximately 646.66 million predictions, out of which only approximately 3.46 million instructions were mispredicted. Thus, the misprediction rate is approximately 0.54%.

Figure 6.5 compares the `ferret` blame stacks and distributions under STT and STTHint. The stacks show that hinting is effective over many instructions on the stack, but a few high-contributing instructions still remain on the STTHint stack. This is further illustrated by the blame distributions: costly TCRs (those with lower instruction addresses) have been effectively neutralised, while TCRs in higher addresses remain performance-degrading. Inspecting the disassembly shows that the remaining instructions are compiler-generated `pop` and `leave` instructions used to restore the caller function's base pointer. Since the memory objects holding these base pointer values are compiler-generated, they are invisible to the high-level C++ programmer. Therefore, the `ferret` case study further motivates future work to extend STTHint by allowing hints to be inserted at a lower level (see Chapter 7), such as within the compiler backend.

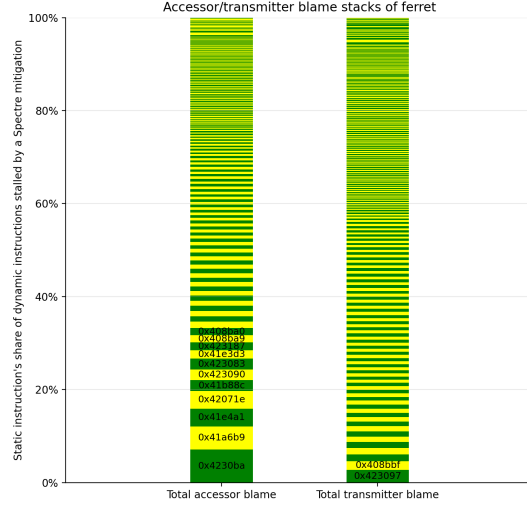
6.2.7 Hint Insertion Example

Figure 6.6 shows a modified function from the `cc_sv` source code, with hints inserted around a costly TCR to reduce STT performance degradation. The problematic memory object is `comp`, which is frequently accessed within the TCR. The object `comp` is a list implemented internally as a dynamically sized array. The `shint_pvector` helper function hints the internal array and re-hints whenever the array is resized. The `shint_object` function hints the C++ object `comp` itself. At the end of the TCR, analogous `sunhint_pvector` and `sunhint_object` are called to unhint `comp`.

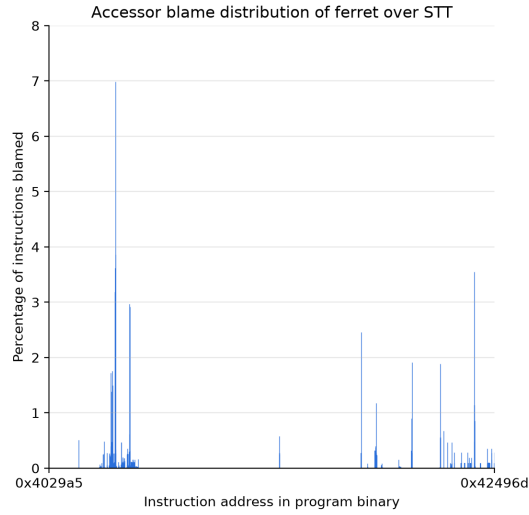
6.2 Empirical Results



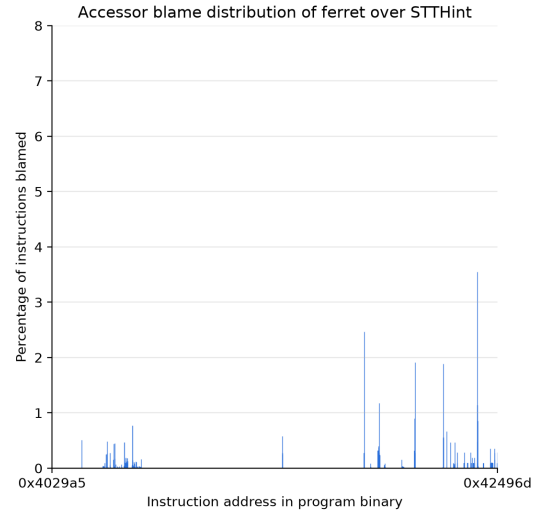
(a) **ferret** blame stack over STT.



(b) **ferret** blame stack over STTHint.



(c) Accessor blame distribution of **ferret** over STT.



(d) Accessor blame distribution of **ferret** over STTHint.

Figure 6.5: Blame stacks (top) and accessor blame distributions (bottom) of **ferret** over STT and STTHint. The X-axis of the blame distributions has been truncated to focus on TCRs.

```

pvector<NodeID> ShiloachVishkin(const Graph &g) {
    pvector<NodeID> comp(g.num_nodes());

    // Hinting comp, a problematic memory object:
    shint_object(comp);
    comp.shint_pvector();

    #pragma omp parallel for
    for (NodeID n=0; n < g.num_nodes(); n++)
        comp[n] = n;
    bool change = true;
    int num_iter = 0;
    while (change) {
        change = false;
        num_iter++;
        #pragma omp parallel for
        for (NodeID u=0; u < g.num_nodes(); u++) {
            for (NodeID v : g.out_neigh(u)) {
                NodeID comp_u = comp[u];
                NodeID comp_v = comp[v];
                if (comp_u == comp_v) continue;
                NodeID high_comp = comp_u > comp_v ? comp_u : comp_v;
                NodeID low_comp = comp_u + (comp_v - high_comp);
                if (high_comp == comp[high_comp]) {
                    change = true;
                    comp[high_comp] = low_comp;
                }
            }
        }
        #pragma omp parallel for
        for (NodeID n=0; n < g.num_nodes(); n++) {
            while (comp[n] != comp[comp[n]]) {
                comp[n] = comp[comp[n]];
            }
        }
    }

    // Unhinting comp after the TCR.
    comp.sunhint_pvector();
    sunhint_object(comp);

    cout << "Shiloach-Vishkin took " << num_iter << " iterations" << endl;
    return comp;
}

```

Figure 6.6: Hint insertion in the `cc_sv` source code to accelerate STT over a TCR.

Conclusion and Future Work

This section discusses some limitations of our work, and further considerations to make STTHint practical. We then conclude this report with a summary of our main contributions.

7.1 Future Work

This section discusses limitations to STTHint, which motivates future works to address for practical adoption of STTHint.

7.1.1 Programmer Responsibility

Our main contributions are our analysis into STT (Chapter 4) and the STTHint mechanism itself. Our goal is to show that a lightweight hardware mechanism with a simple software interface enables effective hinting. Thus, we do not provide a compiler to automate this process, as its design lies outside of our goal.

Currently, STTHint requires manual, source-level insertion of custom instructions, where existing approaches [16, 17] automate this process using compiler passes. Many memory objects and their bounds are largely unavailable at the source level. For example, the adjacency of local variables and compiler-generated memory objects are only visible in low-level compiler passes. Given that most major applications are written in high-level languages like C, this limits STTHint. Thus, we would like to motivate future work to automate hint placements, driven by the insights discussed in Chapter 4. Detecting TCRs may involve a static analysis into the dependency chains between instructions to detect access-transmit sequences, identifying loop structures, and estimating branch criticality. `shint` and `sunhint` instructions may then be inserted around the detected TCRs.

7.1.2 Operating System Considerations

Modern operating systems (OS) virtualise compute and memory resources. Doing so exposes important problems to address. Firstly, the hint buffer holds virtual addresses. Keeping hints in the buffer across a context switch is erroneous, as stale hints remaining in the buffer unprotects parts of the address space of the new process. An easy approach is to invalidate all entries in the hint buffer on a context switch, as proposed in SSMR [16]. However, this implies that an expensive process executing a time-consuming TCR will only enjoy the benefits of STTHint over a single timeslice, since the `shint` instructions are not re-executed when the OS switches back. Alternatively, the kernel may keep track of hints each process owns. On a context switch, the kernel will then save the hints to memory, and load in the new hints. Doing so introduces higher memory use and worsens context switch overhead. However, this approach might be feasible, as our Hint Buffer is small, and context switches are already expensive. A bolder approach is to store our hints in kernel memory, and make the Hint Buffer a cache. This may require `shint` and `sunhint` to become system calls as a compromise. Doing so has the advantage of security checks on hinting *shared pages*, where for instance, an attacker may exploit the fact that kernel memory is mapped to the user space, and a malicious programmer can hint the entire kernel region to unprotect it from Spectre attacks.

7.2 Conclusion

In this work, we provide an analytical and experimental assessment of STT. We find that STT incurs a high performance degradation, averaging 47.22% over GAP workloads. We find that most STT stalls concentrate in a small number of TCRs, which are loops dispatching many access-transmit sequences and slow-to-resolve branches into the ROB. Our analysis motivates STTHint, our novel mechanism to accelerate STT by hinting safe memory objects to the hardware, enabling smooth speculative execution for a large number of instructions in TCRs. With correct hinting, STTHint reduces STT's performance degradation over GAP workloads from 47.22% to 2.80%, and over `ferret` from 48.84% to 15.07%. STTHint also remains as secure and efficient as STT when executing legacy code or dynamically linked libraries without hints. Our lightweight hint predictor and hint buffer require 77.73% less storage than SSMR's Speculation-Safe Region Buffer. Finally, we hope that our insights could motivate future research on automatic hint placements and integration into modern operating systems, paving the way for a practical adoption of STTHint.

Bibliography

- [1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020. [Cited on pages 1, 5, 6, 8, 9, and 24.]
- [2] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. [Cited on pages 1, 3, 4, and 5.]
- [3] F. W. Burton, “Speculative computation, parallelism, and functional programming,” *IEEE Transactions on Computers*, vol. C-34, no. 12, pp. 1190–1193, 1985. [Cited on pages 1 and 3.]
- [4] W. M. Johnson, “Super-Scalar Processor Design,” Stanford University, Tech. Rep. CSL-TR-89-383, 1989. [Online]. Available: https://vlsiweb.stanford.edu/people/alum/pdf/8906_MikeJohnson_SuperScalar_Processor_Design.pdf [Cited on page 1.]
- [5] Intel Corporation, “Intel Analysis of Speculative Execution Side Channels,” Intel Corporation, Tech. Rep. 336983-004, May 2018. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/671163/intel-analysis-of-speculative-execution-side-channels.html> [Cited on page 1.]
- [6] MITRE Corporation, “CVE-2017-5753,” 2018. [Online]. Available: <https://www.cve.org/CVERecord?id=CVE-2017-5753> [Cited on page 1.]
- [7] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, “A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021. [Cited on pages 1, 4, and 19.]
- [8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading Kernel Memory from User Space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018. [Cited on pages 1, 4, 6, and 9.]
- [9] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom> [Cited on pages 1, 4, 5, 6, and 19.]

Bibliography

- [10] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968. [Cited on pages 1, 2, 5, 6, 7, 8, 10, 11, 19, and 24.]
- [11] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441. [Cited on pages 1, 5, 6, 9, and 10.]
- [12] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, “Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks,” in *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 264–276. [Cited on pages 1 and 10.]
- [13] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “NDA: Preventing Speculative Execution Attacks at Their Source,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586. [Cited on pages 1, 5, and 10.]
- [14] T. H. Kim, D. Rudo, K. Zhao, Z. N. Zhao, and D. Skarlatos, “Perspective: A Principled Framework for Pliable and Secure Speculation in Operating Systems,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 739–755. [Cited on pages 1 and 2.]
- [15] S. Ganjisaffar, E. M. Koruyeh, J. Zellmer, H. A. Esfeden, C. Song, and N. Abughazaleh, “SpecASan: Mitigating Transient Execution Attacks Using Speculative Address Sanitization,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 2032–2045. [Cited on pages 1, 2, 5, 9, 19, and 27.]
- [16] A.-T. Ishimwe, S. McDiarmid-Sterling, Z. McKevitt, and T. S. Lehman, “SSMR: Statically Detecting Speculation Safe Memory Regions to Mitigate Transient Execution Attacks,” in *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction*, 2026, pp. 67–80. [Cited on pages 1, 2, 8, 9, 11, 19, 27, 33, and 34.]
- [17] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, “Protean: A Programmable Spectre Defense,” in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2026, pp. 1–20. [Cited on pages 1, 2, 12, 19, 22, and 33.]
- [18] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, “DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987. [Cited on pages 1 and 10.]

- [19] Intel Corporation, “Speculative Execution Side Channel Mitigations,” Intel Corporation, Tech. Rep. 336996-003, May 2018. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/671221/speculative-execution-side-channel-mitigations.html> [Cited on pages 1 and 9.]
- [20] M. Schwarz, M. Lipp, C. A. Canella, R. Schilling, F. Kargl, and D. Gruss, “ConTExT: A Generic Approach for Mitigating Spectre,” in *Network and Distributed System Security Symposium 2020*, 2020. [Cited on page 1.]
- [21] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, “SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels,” in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164. [Cited on pages 1, 9, and 10.]
- [22] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtvyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6. [Cited on pages 1 and 10.]
- [23] Intel Corporation, “Retpoline: A Branch Target Injection Mitigation,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html>, 2022, doc ID 758357. Accessed: 2026-10-27. [Cited on pages 1 and 9.]
- [24] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 39–53. [Cited on pages 1 and 9.]
- [25] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, “HidFix: Efficient Mitigation of Cache-Based Spectre Attacks Through Hidden Rollbacks,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9. [Cited on pages 1 and 10.]
- [26] B. Chen, R. Choudhary, K. Khulbe, A. Lee, A. Morrison, and C. W. Fletcher, “μSTT: Microarchitecture Design for Speculative Taint Tracking,” in *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, 2025, pp. 289–297. [Cited on page 2.]
- [27] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006. [Cited on page 2.]
- [28] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81. [Cited on page 2.]

Bibliography

- [29] S. Beamer, K. Asanović, and D. Patterson, “The GAP Benchmark Suite,” *arXiv preprint arXiv:1508.03619*, 2015. [Cited on pages 2 and 23.]
- [30] D. Haris and S. Harris, *Digital Design and Computer Architecture*. Oxford, UK: Denise E. M. Penrose, 2007. [Cited on pages 3, 5, and 27.]
- [31] J. E. Smith, “A study of branch prediction strategies,” in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215. [Cited on page 4.]
- [32] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967. [Cited on page 4.]
- [33] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. 4180 IL Route 83, Suite 101, Long Grove, IL, 60047-9580: Waveland Press, Inc, 2013. [Cited on page 4.]
- [34] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient Cache Attacks on AES, and Countermeasures,” *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010. [Cited on pages 4 and 6.]
- [35] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-Level Cache Side-Channel Attacks are Practical,” in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622. [Cited on page 5.]
- [36] M. Andryscio, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On Subnormal Floating Point and Abnormal Timing,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 623–639. [Cited on pages 5, 10, 19, and 24.]
- [37] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-Privilege-Boundary Data Sampling,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768. [Cited on page 6.]
- [38] Intel Corporation, “Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html>, 2018, doc ID 660214. Accessed: 27 October 2026. [Cited on page 9.]
- [39] —, “Indirect Branch Restricted Speculation,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html>, 2018, doc ID 758390. Accessed: 2026-09-15. [Cited on page 9.]

- [40] —, “Single Thread Indirect Branch Predictors,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html>, 2018, doc ID 758387. Accessed: 2026-09-15. [Cited on page 9.]
- [41] —, “Indirect Branch Predictor Barrier,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html>, 2018, doc ID 758388. Accessed: 2026-09-15. [Cited on page 9.]
- [42] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009. [Cited on page 9.]
- [43] J. Wikner and K. Razavi, “RETBLEED: Arbitrary Speculative Code Execution with Return Instructions,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/wikner> [Cited on page 9.]
- [44] Intel Corporation, “Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115,” <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html>, 2018, doc ID 660230. Accessed: 27 October 2026. [Cited on page 9.]
- [45] AMD, “Security Analysis of AMD Predictive Store Forwarding,” AMD, Tech. Rep., 2023. [Online]. Available: <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/security-analysis-of-amd-predictive-store-forwarding.pdf> [Cited on page 9.]
- [46] J. Corbet, “The current state of kernel page-table isolation,” LWN.net, <https://lwn.net/Articles/741878/>, Dec. 2017, published 20 December 2017. Accessed: 2026-10-21. [Cited on page 9.]
- [47] H. Shukur, S. Zeebaree, R. Zebari, O. Ahmed, L. Haji, and D. Abdulqader, “Cache coherence protocols in distributed systems,” *Journal of Applied Science and Technology Trends*, vol. 1, no. 2, pp. 92–97, 2020. [Cited on page 10.]
- [48] Schwarz, Michael and Schwarzl, Martin and Lipp, Moritz and Masters, Jon and Gruss, Daniel, “NetSpectre: Read Arbitrary Memory over Network,” in *Computer Security – ESORICS 2019*, Sako, Kazue and Schneider, Steve and Ryan, Peter Y. A., Ed. Cham: Springer International Publishing, 2019, pp. 279–299. [Cited on page 10.]
- [49] E. S. Tune, D. M. Tullsen, and B. Calder, “Quantifying instruction criticality,” in *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2002, pp. 104–113. [Cited on page 15.]

Bibliography

- [50] Intel Corporation, “What Does the Number of Bits in Physical Address Extensions (PAE) of Memory Specifications on Intel® Xeon® Processors Mean?” <https://www.intel.com/content/www/us/en/support/articles/000029063.html>, 2025, intel Support Knowledge Base, Article ID 000029063. Accessed 26 October 2026. [Cited on page 19.]
- [51] The Linux Kernel Documentation, “Memory Management — Complete virtual memory map with 4-level page tables,” https://docs.kernel.org/6.0/_sources/x86/x86_64/mm.rst.txt, accessed 26 October 2026. [Cited on page 19.]
- [52] Lee and Smith, “Branch Prediction Strategies and Branch Target Buffer Design,” in *Proceedings. 2015 IEEE International Symposium on Multimedia (ISM)*. IEEE, 1984, pp. 6–22. [Cited on page 22.]
- [53] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12. [Cited on page 23.]
- [54] Intel Corporation, “Intel Xeon Processor X5550,” <http://ark.intel.com/products/37106/>, 2017, product specification page. Archived 13 September 2017 at <https://web.archive.org/web/20170913233847/http://ark.intel.com/products/37106/>. Accessed 1 October 2026. [Cited on page 24.]
- [55] A.-T. Ishimwe Scriptor, “Defending Against Transient Execution Attacks Efficiently,” Ph.D. dissertation, 2026. [Online]. Available: <https://www.proquest.com/dissertations-theses/defending-against-transient-execution-attacks/docview/3340730241/se-2> [Cited on page 27.]
- [56] Intel, “Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1,” Intel, Tech. Rep. 248966-048, 2023. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html> [Cited on page 29.]