SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors

Daniel Townley
Department of Computer Science
Binghamton University
Binghamton, NY, USA
dtownle1@binghamton.edu

Dmitry Ponomarev
Department of Computer Science
Binghamton University
Binghamton, NY, USA
dponomar@binghamton.edu

***Abstract*—Recent advances in side-channel attacks put into question the viability of Simultaneous Multithreading (SMT) architectures from the security standpoint. To address this problem, we propose SMT-COP—a system that eliminates all known side-channels through shared execution logic, including ports and functional units, on SMT processors. At the core of SMT-COP is a small modification to the instruction scheduling logic that allows support for both *spatial* and *temporal* partitioning of the execution units to prevent controllable contention for these resources that causes side-channel attacks. We demonstrate that the security benefits of SMT-COP are achieved with modest performance overhead and hardware complexity, and without relying on software support. Further performance improvements can be achieved when SMT-COP is applied selectively, and we consider three forms of such selective application: a) in response to detecting resource contention; b) in response to detecting manipulations with time measurement infrastructure; and c) in response to explicit requests by applications. This study represents a step towards making SMT processors more secure.**

I. Introduction

Simultaneous Multi-threading (SMT) is an important architectural paradigm that supports concurrent execution of multiple programs on an out-of-order processor core with modest increase in processor complexity, area and power consumption. By allowing instructions from multiple threads to compete for resources on a single core, SMT increases the number of independent instructions available for issue each cycle, allowing more efficient pipeline utilization and thus higher instruction throughput. These advantages have led to widespread adoption by microprocessor vendors.

Unfortunately, a growing body of research shows that the performance benefits of SMT come with significant security risks. In addition to facilitating timing attacks on private caches [1]–[3] and branch predictors [4], SMT creates new opportunities to extract secret information through shared *execution resources*, which include the processor's functional units (FUs) and associated issue ports. Specifically, an attacker can create a timing side-channel by issuing a large number of instructions to a specific execution resource, and by measuring its execution latency. From this measurement, the attacker can detect contention and infer whether a victim executes instructions that use the same execution resource. Variations in the victim's execution time, as measured by the attacker, can leak sensitive data. For example, the well-publicized PortSmash attack [5] uses this approach to extract private keys from an OpenSSL server. As another recent example, SMoTherSpectre attack utilizes port contention on SMT to mount a code-reuse attack to leak data from OpenSSL [6].

Compounded with other vulnerabilities in SMT processors, execution resource side-channels present a serious quandary to chipmakers and their customers. The security challenges of SMT have motivated some developers, such as those contributing to the open BSD operating system, to abandon SMT completely [7], while a recent academic work has suggested re-purposing SMT facilities to accelerate other aspects of execution, such as context switching in virtualized systems [8]. At the same time, SMT represents a decades-long investment of research and development, and continues to enjoy significant commitment from the industry. Intel [9], AMD [10], and IBM [11] continue to make extensive use of SMT in their high-end commercial designs, and recently ARM entered the SMT design space with a new architecture targeting security-critical automotive and aeronautic applications [12]. With the continuing slowdown of Moore's law and the demise of Dennard's scaling, architectural optimizations such as SMT are becoming increasingly vital to sustained processor performance and throughput. The challenge now is to retain these performance benefits in a secure manner.

To this end, we propose SMT-COP (short for SMT COntext Partitioning) — a generalizable, hardware-supported approach that eliminates execution logic side-channels in SMT processors by partitioning functional units and/or associated issue ports between threads running on the same core. This is accomplished using novel, lightweight hardware implemented largely in parallel to the existing dynamic scheduler, which reserves a subset of execution resources for each thread running on the core. Depending on the topology of the underlying architecture, resources can be permanently assigned to a particular thread *(spatial partitioning)* or multiplexed between them *(temporal partitioning)*. When enabled, SMT-COP eliminates FU contention by instructions issued by different threads, and thus closes the execution logic side-channel. Unlike existing approaches that disable SMT acceleration as a trade-off for security, we show that SMT-COP completely eliminates the execution logic side channels with a mean performance degradation of only 8% across the simulated 2-threaded mixes. In addition, we discuss several strategies to apply SMT-COP selectively to further reduce performance impact, while maintaining the same level of security guarantees.

An important benefit of SMT-COP is that all partitioning

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00012

IEEE computer society

policies, including two of the three selective partitioning strategies, can be managed in hardware without reliance on software, including trusted system software. This property is especially important due to the increasing popularity of isolated execution systems such as Intel SGX [13]–[15], whose security models exclude the operating system from the trusted computing base. We show that SMT-COP enforces this security model while allowing the OS to manage the high-level performance impacts of partitioned execution, making our system practical for deployment on shared computation platforms that make use of SGX and other isolated execution technologies.

While SMT side-channel mitigations have been proposed for caches [16]–[20], branch predictors [4], [21], and the TLB logic [22], defenses against execution logic side channels remain in their early stages. To our knowledge, SMT-COP is the first implementation to resolve execution logic side-channels in SMT architectures without requiring costly process migrations or indefinite disabling of SMT capabilities. Moreover, while the partitioning of other SMT resources has been explored in prior, performance-oriented works, SMT-COP is the first to address the unique security challenges of partitioning execution resources. The resources addressed in these works, such as issue queues, issue bandwidth, and decoding bandwidth, are comparatively simple to partition due to their homogeneous structure. In contrast, execution resources in modern SMT processors exhibit complex topologies that can include pipelined and non-pipelined functional units with various execution latencies.

In summary, this paper makes the following contributions:

- We propose SMT-COP — a set of partitioning schemes that eliminate information leakage through shared execution logic in SMT processors, and discuss their application to a realistic SMT architecture.
- We describe the gate-level implementation of the core SMT-COP architecture, and discuss its integration into existing SMT scheduling logic. We show that this integration can be accomplished with at most one extra gate on the critical path of the scheduling logic, and even that can be hidden in typical implementations.
- We evaluate SMT-COP using a cycle-accurate SMT simulator. We demonstrate SMT-COP's security by closing a side channel on a simulated architecture, while incurring only a 8% performance loss compared to baseline SMT when SMT-COP is continuously applied to typical workloads.
- We propose optimizations that selectively apply partitioning to critical program regions, reducing or eliminating partitioning overheads during normal execution, and evaluate the performance benefits of one such optimization.
- We show that the proposed strategies can be implemented even when system software is untrusted, making SMT-COP synergistic with isolated execution systems such as Intel's SGX.

While SMT-COP closes execution logic side-channels and eliminates attacks such as PortSmash and SMoTherSpectre, it is only a step towards completely securing SMT processors. New attacks exploiting other vulnerabilities continue to emerge [23], and defenses from these attacks will be synergistic with SMT-COP. In fact, the mechanisms of SMT-COP can inspire solutions to close other known and hypothetical side-channels to create a fully-secure SMT architecture.

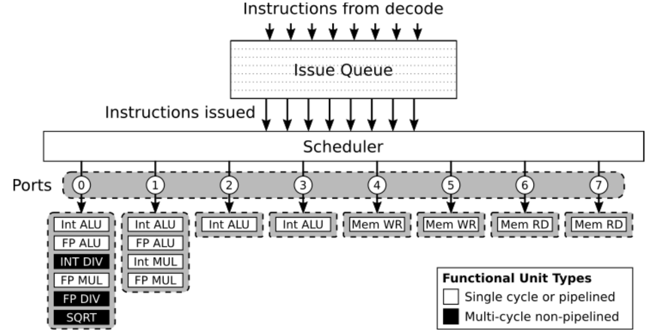

Figure 1: SMT Architecture

II. Background

As a basis for our discussion of SMT-COP, this section describes the baseline SMT architecture used in our system's design and evaluation. Additionally, to motivate the design of our partitioning strategy, this section details the execution logic vulnerability that SMT-COP seeks to eliminate.

A. Reference SMT architecture

The reference SMT architecture for SMT-COP, shown in Figure 1, is similar in general organization to current Intel designs [9]. *The Issue Queue* (IQ) buffers decoded instructions until dependencies are satisfied and issue slots become available. Each cycle, the *scheduler* selects instructions whose dependencies have been satisfied, and issues up to eight ready instructions to appropriate *functional units* (FUs). The FUs are organized into groups, each served by a designated *issue port*. Since each issue port accepts a single instruction per cycle, this organization places an additional constraint on resource availability. For example, because port zero serves both the integer divider and floating point divider, the scheduler cannot issue an `IntDIV` and and `FPDIV` instruction in the same cycle, even if both of the corresponding units are available. Thus, contention on ports, as well as on functional units themselves, can be exploited as a basis for side-channel attacks, as demonstrated in [5].

In the reference architecture, as in typical commercial architectures, most functional units are either pipelined or have a single-cycle execution latency, and are thus able to begin executing a new instruction every cycle. Some units, however, are non-pipelined and have a multi-cycle execution latency, meaning that they are unavailable for multiple cycles while executing an instruction. This is the case, for instance, for division logic in traditional Intel designs [9], as well as the division and square root units in the reference architecture.

B. Execution Logic Side-channels on SMT Cores

The execution logic side-channel arises when instructions from multiple threads compete for FUs or their associated issue ports, which we generically refer to as *execution resources*.

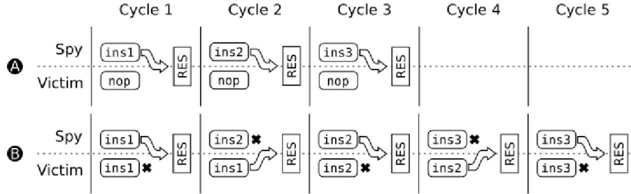

Figure 2: Timing of a spy thread executing (A) with execution logic contention and (B) without execution logic contention.

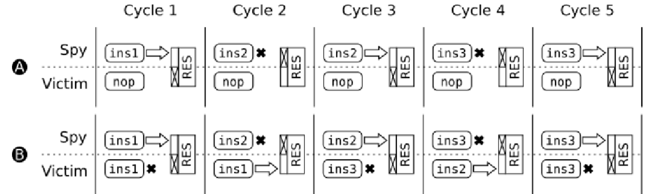

Figure 3: Timing of a spy and victim thread executing under the SMT-COP partitioning scheme. Note that the times observed by the spy for both of the victims execution phases are identical.

Figure 2 shows a generalized side-channel between two threads, a *Spy s* and *Victim v*, which run on the same SMT processor and contend for execution resource *RES*. Thread *v* has two stages of execution, whose ordering and duration depends on sensitive data. In one phase, shown in scenario (A), *v* does not issue any instructions that require RES; in a second phase, shown in (B), *v* uses RES extensively. *s* attacks *v* by attempting to issue multiple instructions to RES. In scenario (A), RES is uncontended by *v*, and RES thus can process an instruction from *s* every cycle. In scenario (B), *v* frequently issues instructions to RES, making the resource unavailable to *s* on alternating cycles. Consequently, the three instructions from *s* take measurably longer to execute. If the alternating phases of *v*'s depend on sensitive data, *s* can recover this data by repeatedly measuring the execution latency. This method can be the basis of a *covert channel*, in which a colluding victim deliberately alternates between contentious and non-contentious behavior in order to exfiltrate sensitive data, or of a *side channel*, in which an oblivious victim leaks information, such as a cryptographic key, through incidental variations in execution behavior.

```
...
loop:
lfence          ; Enforce in order commit
rdtsc           ; Store cycle count in rax
lfence
mov %rax, %rbx  ; Save cycle count in rbx
.rept 100
imul $2, %ecx   ; Force contention on multiply
.endr
lfence
rdtsc           ; Store end cycle count in rax
lfence
shl $32 %rax
or %rax, %rbx   ; move it to ecx with start time
...
(store start/end time in memory and reiterate loop)
```

Listing 1: Spy code exploiting an execution logic side-channel through the integer multiply unit in x86

Listing 1 shows how a spy program similar to the one described in [5] can be implemented in x86-64 assembly language. In this case, the side-channel measured involves the integer multiply unit or its associated port. The *rtdsc* instruction is used to record the value of a continuously incremented cycle counter at the beginning and the end of the chain of contention instructions. The cycle counts stored at the beginning and the end of the spy loop are stored, and the differences can be used to compute the execution latency on the multiplier over the course of execution.

In addition to targeting specific functional units, attacks can be launched against the issue ports that serve groups of functional units. In such attacks, a spy thread issues a large number of instructions whose functional units share a port with those involved in critical phases of a victim's execution. Recent work has shown port-based side channels to be a viable method of extracting cryptographic keys from the openSSL server [5]. Thus, efforts to secure SMT execution logic must include both the functional units themselves, as well as the ports that serve them.

III. Partitioning Strategy

SMT-COP closes execution logic side channels by eliminating contention between threads for execution resources. Rather than allowing threads to compete freely for a shared pool of resources, SMT-COP reserves specific execution resources for each thread according to fixed policies that do not depend on thread behavior. This section discusses two strategies for partitioning execution resources. The first strategy, called *temporal partitioning*, multiplexes a single resource between threads over a period of one or multiple cycles. Temporal partitioning is ideal for unique resources that must be shared among threads. The second strategy, *spatial partitioning*, statically allocates instances of a particular resource class to specific threads. Spatial partitioning can be applied to resources with redundant instances that can be evenly divided among threads. This section demonstrates how each strategy eliminates the vulnerability discussed in section II, and introduces policies that apply partitioning strategically to reclaim most of the performance benefit of SMT without compromising security.

A. Temporal partitioning

1) Pipelined and single-cycle latency resources: Figure 3 demonstrates the elimination of the timing side-channel through temporal partitioning of an execution resource, using the execution scenario introduced in Figure 2. In the execution scenarios shown, SMT-COP multiplexes an execution resource RES between a spy *s* and victim process *v* every cycle. In cycles 2 and 4, RES is allocated to *v* (and denied to *s*) regardless of whether *v* has pending instructions that use RES. The deterministic assignment schedule masks the execution of *v*, and the latency of *s* is unaffected by the behavior of *v* with regards to RES. Consequently, the timing properties of scenarios (A) and (B) are indistinguishable from the perspective of *s*, and information leakage is eliminated.

2) Non-pipelined resources with multi-cycle latency: The baseline temporal partitioning policy shown in Figure 3 applies to resources that can accept an instruction every cycle,

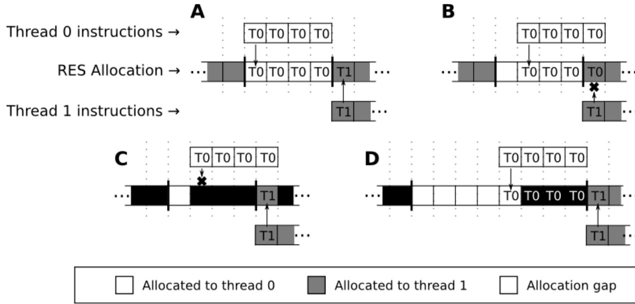

Figure 4: Timing diagram of a non-pipelined, temporally partitioned resource with a multicycle latency, with instructions arriving at various cycles.

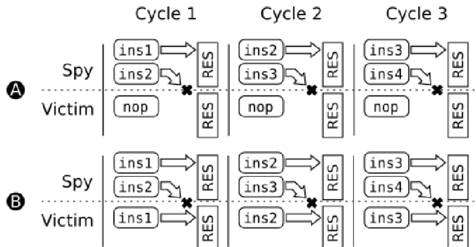

Figure 5: Spatial partitioning of duplicated resources.

including fully pipelined resources and those with a single-cycle execution latency. While such resources are prevalent in modern architectures, some functional units, such as the radix-16 dividers in historical Intel designs, are non-pipelined and require multiple cycles to operate on an instruction. A naive extension of spatial partitioning would involve extending the allocation period for a non-pipelined resource from one cycle to match the maximum execution latency of the resource. However, as shown in Figure 4, this simple approach is not sufficient to eliminate contention. In Figure 4-A, Thread 0 issues an instruction to RES on cycle 1. Since RES is currently assigned to A, the instruction is issued successfully and completes within the assignment period, avoiding contention with Thread 1. However, if an instruction arrives in the second cycle of the allocation period, as shown in Figure 4-B, the instruction will continue to occupy the functional unit into the allocation period for Thread 1, creating the possibility for contention. To avoid such conflicts, the partitioning logic must enforce an *issue gap*, shown in Figure 4-C, to ensure that *no* instructions issue in the last L-1 cycles of the allocation period, where L is the maximum latency of RES. Instructions that become ready during the issue gap for Thread 0 are delayed until the beginning of the thread's next allocation period. To provide greater flexibility in issuing instructions, the allocation period can be extended beyond the maximum latency of the protected resource, as shown in Figure 4-C, to create additional cycles during which instructions from a given thread can be committed.

B. Spatial partitioning

Figure 5 demonstrates the use of spatial partitioning to close a similar side channel when multiple instances of a functional unit class are available. In the absence of partitioning, *s* could construct a side channel by issuing enough instructions each cycle to contend for multiple instances of the same resource class. Under spatial partitioning, however, *s* and *v* are each assigned a functional unit to which their instructions have uncontested access, regardless of their respective instruction mix. Because *s* is confined to a designated resource instance, it cannot take advantage of the idle resource assigned to *v* during the program phase shown in (A), and thus cannot observe a timing difference with respect to *v*'s execution. Similarly, in phase (B), the static resource allocation policy masks *v*'s use of the resource. In this way, spatial partitioning of resources also eliminates the side-channel.

C. Partitioning example

Though conceptually simple, the implementation of partitioning must account for the complexities of contemporary execution logic. In addition to using a combination of temporal and spatial schemes to partition various functional units, SMT-COP must also manage the partitioning of issue ports. Figure 6 applies the policies developed in this section to a toy version of the Intel-style architecture introduced in section II. In this example, two threads issue instructions to the execution logic, which has three issue ports. Port 0 can be used to issue an instruction to either an ALU, a multiply unit (MUL) or a divider (DIV). DIV is a non-pipelined unit with an execution latency of 2 cycles and an allocation period of 3 cycles, meaning that the partitioning policy allows instructions to be issued to this unit during the first two cycles of each allocation period. All other resources have a single-cycle latency, and are temporally partitioned with an allocation period of one cycle. Port 0 is itself vulnerable to side channel attacks, and is thus temporally partitioned on the same schedule as the associated single-cycle resources. Ports 1 and 2 each serve a single ALU. Both ports and their associated resources are spatially partitioned between the two threads.

Threads 1 and 2 execute the code fragments shown in Figure 6-B, while their execution over four cycles is shown Figure 6-A. In cycle 1, thread 1 has three ready instructions: a *div* instructions, and two *add* instructions. Under the temporal partitioning policy, port 0 and its associated resources are initially assigned to Thread 1, allowing the `DIV` to be issued to the corresponding functional unit over port 0. The first add can be dispatched over the spatially partitioned ALU attached to port 1. While a second ALU is allocated to thread 1 on port 0, the second `ADD` cannot be issued because the scheduler uses port 0 in this cycle to issue the `DIV`. Similarly, spatial partitioning prevents this add from being issued to the ALU on port 1, which is assigned to thread 2. This allows thread 2 to schedule an add on port 1, although a second `add` from thread 2 stalls because ports 0 and 1 are assigned to thread 1.

In cycle 2, port zero and its single-cycle resources are allocated to thread 2. This prevents thread 1 from issuing the outstanding add to the unit on this port, causing it to use its spatially assigned ALU instead. The `add` executed by thread 2 in the previous cycle provides the dependencies for a `multiply` instruction, which is free to use the multiply

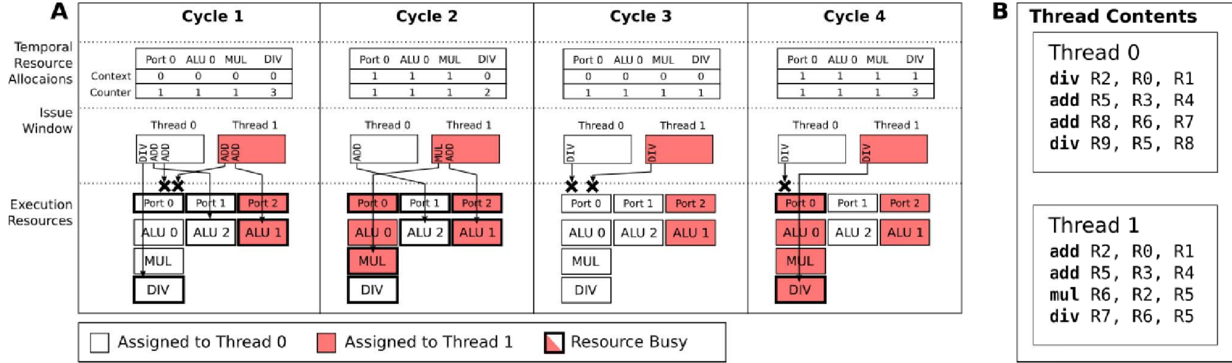

Figure 6: Timing diagram of contention-free execution in a simplified SMT architecture.

unit now allocated to thread 2. The `add` instruction that was previously stalled is also able to execute on port 2.

In cycle 3, previous instruction satisfies dependencies for a `DIV` in each thread. Additionally, the first DIV instruction issued by context 0 in the first cycle has completed execution, freeing the divider on port 0. However, neither `div` instruction can execute in this cycle. The `DIV` instruction from thread 2 is prevented from executing because the divide unit is still assigned to thread 1, as is the associated port. Even though thread 1 has access to the divider, it is prevented from using it by the issue gap policy, which prevents its execution latency from creating contention when the divider is reassigned to thread 2 in the following cycle.

Finally, in cycle 4, port 0 and the divider become available to thread 2, which is free to issue its final `DIV` instruction. The pending `DIV` from thread 1 will be stalled until cycle 7, when these resources are re-allocated to thread 1.

D. *Selective Application of SMT-COP*

While the evaluation of SMT-COP in Section V shows modest performance impacts even under continuous partitioning, various selective invocation strategies can be deployed to limit the impact of partitioning even further. These approaches involve enabling and disabling SMT-COP protections based on the security needs of a thread, thus limiting partitioning overheads to periods of execution where information leakage poses a meaningful threat. This section discusses three strategies for selective partitioning: *Contention-Driven Partitioning (CDP)*, *Measurement-Driven Partitioning (MDP)* and *Application-Driven Partitioning (CDP)*

1) Contention-Driven Partitioning: The CDP approach is an optimization, where partitioning for each functional unit type is enabled only when active contention from the executing threads for this type of functional units exists. As an example, consider a dual-threaded execution, where the two threads share a non-pipelined DIV functional unit. Two counters are initialized to zero, one for each thread. When a DIV instruction from a given thread is decoded, the corresponding counter is set to its maximum predetermined value (lets say, 10K cycles). If in a given cycle no DIV instructions are decoded, the corresponding thread counter is decremented by one, or remains at zero if it is already zero. Partitioning of the DIV functional unit is enabled when both counters are at non-zero values. If one of the counters is at zero, this implies that the corresponding thread did not encounter any DIV instructions for a long period time, which is deemed sufficient to indicate the absence of the attack.

CDP exploits the observation that side-channel attacks must typically measure contention at a certain granularity in order to extract meaningful information. Although an attacker could detect the initial contention event that triggered the CDP policy, additional contentions occurring during the CDP invocation period would be masked, severely limiting the granularity of the information that can be extracted. In this sense, CDP relaxes the principled security guarantees of continual partitioning, but can nonetheless make side-channel attacks impractical if the invocation period is long enough to mask meaningful variations in the victims resource utilization. CDP is best used to protect specialized functional units, such as dividers, which experience limited contention during normal operation, and can be applied selectively to the functional units that are determined to have the largest negative impact on performance due to partitioning.

2) Measurement-driven Partitioning: The MDP approach triggers partitioning when a thread performs a timing measurement, for instance by using an `rdtsc` instruction. A successful side-channel attack requires that multiple such measurements be performed in a relatively short interval of time. When the first measurement is detected from any thread, MDP enables partitioning and keeps it active for a period of time long enough to make the attack based on subsequent timing readings ineffective, essentially making all critical accesses that are made under attack in fully-partitioned mode, thus masking timing information and preventing information leakage. After the predetermined number of cycles, the processor returns to normal, non-partitioned operation. The use of `rdtsc` instructions, which is the most common and reliable measurement instruction used in timing attacks, can be detected with minimal modification to the decode logic, and other mechanisms could be included to detect other measurement strategies. For example, [24] observed that software-based timers could be detected using the method described in [25].

Of the selective partitioning schemes we discuss, MDP has the strongest potential for performance savings. Indeed, because only a small number of benign threads make regular use of

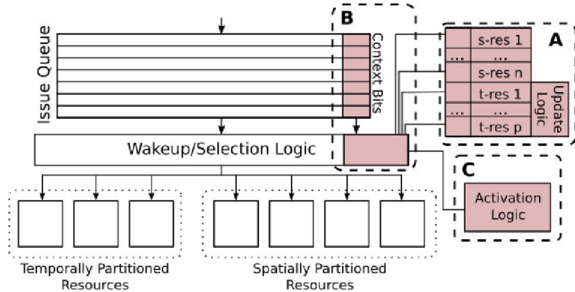

Figure 7: SMT-COP components

rdtsc instructions or other timing operations, the majority of execution would be able to proceed with full SMT sharing. Thus, the overhead relative to baseline of SMT-COP running in MDP mode is likely to approach zero for most workloads.

3) Application-driven partitioning: Finally, ADP uses explicit requests from the applications to request partitioned resources. The duration of partitioning can be managed either by allowing the application to set a timeout counter, or by providing an additional instruction that allows an application to explicitly disable partitioning. This method is ideal for applications that combine sensitive sections of code with other instructions that do not require protection. For instance, a thread that occasionally performs cryptographic operations can request ADP protections only while those operations are in progress, and return to baseline SMT while executing code that does not depend on sensitive data.

4) Managing selective partitioning: The enabling and disabling of partitioning in SMT-COP balances the needs of OS-based scheduling with the needs of sensitive threads for a secure execution environment. While the high-level partitioning policy for each processor, and the corresponding performance penalties, can be managed by the untrusted OS, secure processes can verify that the necessary guarantees are provided and decline to execute if they are not. This model borrows from the concept of SGX, which makes the OS responsible for managing secure execution environments, but uses trusted hardware to guarantee that environment is configured correctly. In the case of SMT-COP, this eliminates the requirement of trusting the OS, while preventing programs claiming sensitive status from abusing SMT-COP and incurring excessive slowdowns. Section IV-B describes the hardware support for this mechanism.

IV. SMT-COP Implementation

Although the partitioning policies themselves are conceptually simple, their implementation at the level of hardware requires careful evaluation. First, SMT-COP components must be integrated without significantly disrupting the timing and organization of the underlying architecture. Second, the mechanisms for selectively invoking SMT-COP partitioning must balance applications' security requirements with the system software's ability to manage performance at a high level. Specifically, an application must be able to establish partitioning without relying on untrusted system software, but should not be able to abuse SMT-COP partitioning in ways that effect system performance unnecessarily. Respectively, these criteria inform the design of the partitioning scoreboard and scheduler integration in sections IV-A and IV-C, and the design of the activation logic in section IV-B.

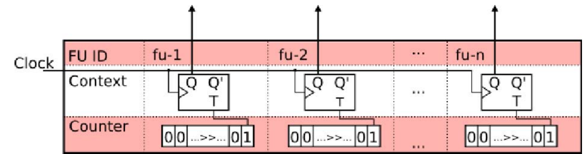

Figure 8: SMT-COP partition table

Figure 7 shows the high-level organization of the hardware needed to implement SMT-COP in a typical SMT core. The *partition scoreboard* (A) maintains the context assignments of the execution resources, as well as the update logic to multiplex temporally partitioned resources between threads. This scoreboard maintains an interface (B) with the wake-up/scheduling logic of the baseline architecture, which controls the issue of instructions to various execution resources. In some scheduler designs, a thread-id must be included in each IQ entry to facilitate partitioning. Finally, to allow secure activation and deactivation of the partitioning scheme, SMT-COP adds a dedicated set of *activation registers* (C) with associated ISA instructions to securely enable and disable partitioning according to the current security requirements. This mechanism supports the three selective partitioning policies discussed in Section III-D. The following sections detail the design and operation of these components.

A. Partition scoreboard

To enforce partitioning, SMT-COP maintains a scoreboard to track the availability of each resource relative to each thread context. For a 2-way SMT (which is the norm in most of today's designs), a single bit can represent the context assignment of a given resource. However, the mechanisms that we discuss can be easily extended to accommodate more aggressive architectures with larger number of contexts.

While spatial allocations are maintained throughout execution, and can simply be hardwired at design time, temporal allocations vary continuously and must be updated after each assignment period. Figure 8 shows the logic to maintain a set of temporal resource allocations. The entry for each FU is a state machine that switches between contexts after a predefined number of cycles. The context ID of the current assignment is stored as a single bit in a T flip-flop. The state of the T flip-flop, and thus the assigned context, is switched every N cycles by a simple ring counter — essentially an N-bit shift register containing a single bit set to ”1”, which drives the output high and inverts the bit in the T flip-flop whenever the asserted bit shifts into a designated position.

B. Selective Invocation of SMT-COP

The hardware shown in Figure 9 implements the selective partitioning policies described in section III-D. The core selective partitioning support consists of three registers and a flag. The *partition flag* pf enables and disables partitioning. The *partition request register* preq stores the number of

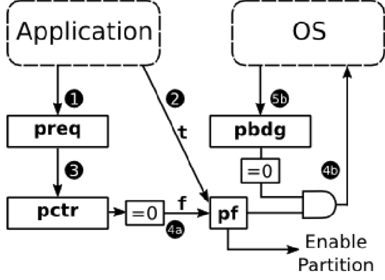

Figure 9: Selective partitioning logic in SMT-COP

cycles in each selective partitioning period, which is loaded into *partition period counter* `pctr` whenever partitioning is invoked. Finally, a *partition budget counter* `pbdg` allows the OS to manage the availability of partitioning.

Figure 9 shows the operation of these components. A program with requirements for secure execution first configures the selective partitioning logic by populating `preq` with the number of partitioned cycles required by its security policy (1). Next, `pf` is asserted to invoke partitioning (2). This operation can be performed either by hardware in response to a security event (in CDP and MDP), or by the application itself (in ADP). The assertion of `pf` triggers the filling of `pctr` with the value in `preq`, (3) and enables the decrementing of both `preq` and `pbdg` every cycle.

In general, partitioning will be turned off after the requested period indicated in `pctr` has elapsed (4a). However, if the cycle budget provided by the OS in `pbdg` is depleted before `pctr`, a hardware exception will occur, transferring control to the OS (4b). The OS then has the option to allocate additional cycles in `pbdg`, allowing partitioned execution to continue (5c). Alternatively, the OS can reschedule or terminate the process to prevent abuse of partitioning in accordance with established policies.

Under ADP, `preq` and `pctr` could be ignored, allowing the application to use a special `cpart` instruction to directly disable partitioning after a sensitive set of operations is completed. In this configuration, `pbdg` would still enforce high-level restrictions on the use of partitioning facilities.

Table I shows the instructions needed to implement the logic in Figure 9, and the privilege level needed to use the instruction. The *enclave* privilege level is borrowed from Intel's SGX, and applies to operations that can be used by attested user applications but not by the OS. Critically, the instructions that establish partitioning requirements are available only under the enclave privilege level, and are inaccessible to the OS. While the OS can manage the performance effects of partitioning by setting the value of `pbdg`, a compromised OS cannot force sensitive processes to execute without the requested level of partitioning: if `pbdg` is set to zero, the logic described in Figure 9 will prevent enclaves with an asserted partition flag from executing until the OS can provide the requested partitioning. During context switches, SMT-COP can use existing SGX-style mechanisms to store the value of `pf` in OS-inaccessible memory, and restore it when the process re-scheduled. In this way, practical selective partitioning can be deployed without

Name	Action	Use	Level
`lpreq op0`	`op0`→`preq`	Set partitioning request for `op0` cycles	Enclave
`spf`	`1`→`pf`	Invoke partitioning directly in ADP mode	Enclave
`upf`	`0`→`fpart`, `0`→`flpart`	Disable partitioning directly in ADP mode	Enclave
`spbdg op0`	`op0`→`pbdg`	Allow partitioning for `op0` cycles upon request	OS

Table I: Selective partitioning interface in SMT-COP

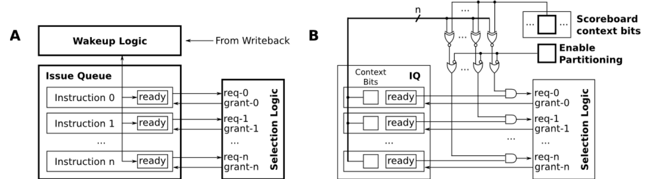

Figure 10: Generic SMT scheduling logic (A) with SMT-COP modifications (B)

assuming the OS to be trusted.

C. Integration

Ideally, the integration of the partitioning logic must be done in such a way as to minimize disruption to existing architectural structures. Given the diversity of SMT architectures, the details and optimization of SMT-COP integration require consideration in light of existing design choices. While it is beyond the scope of the present paper to describe every possible integration scenario, we describe integration at the abstract level using a generalization of SMT hardware, then explore a specific implementation in which the parallelism of the underlying architecture can be exploited to remove the SMT-COP logic from the critical path.

1) Integration with generic wakeup logic: In contemporary pipelines, the logical place to install SMT-COP is at the scheduling stage of the pipeline. This pipeline stage is responsible for issuing SMT instructions to the processor's execution resources, and can be leveraged to provide fine-grained control. Although integration at an earlier stage, such as instruction dispatch, may be possible, it would be more complex to implement in a precise manner. Despite the variety of existing schedulers, most can be abstracted as shown in Figure 10. Instructions fetched from memory are dispatched to an issue queue, which stores the instructions until their dependencies are resolved. The *wakeup logic* receives information on completed instructions, and sets a ready bit for instructions whose dependencies have been computed. Each instruction with satisfied dependencies asserts a *ready signal* to the *selection logic*, which determines which ready instructions can be issued to the execution logic, and asserts a *grant signal* for each selected instruction. Instructions for which grant signals are asserted are then issued to execution resources.

In effect, SMT-COP adds an additional layer of scheduling to deny instructions the use of resources that are not currently assigned to the same context. The abstract logic defined thus far presents a simple opportunity for integration by intercepting ready signals as shown in Figure 10. The gates shown compare

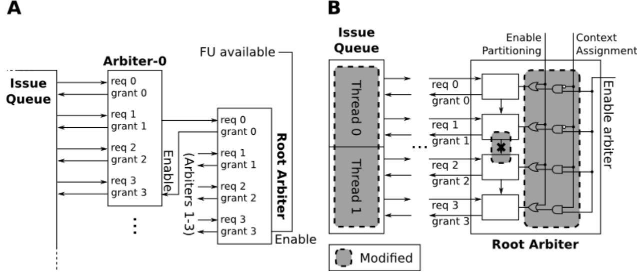

Figure 11: Integration of SMT-COP into the classical scheduler described in [26], eliminating additional gate depth.

the thread ID of the instruction, assumed to be present in the IQ entry, with the allocation of a functional unit in the partition table. The OR gate is used to allow disabling/enabling partitioning based on the state of the selective invocation logic. The resulting signal is then combined with the original ready signal to generate a new ready signal that respects partitioning. While this mechanism involves four gates, only the final AND gate is in series with the baseline scheduling logic; the gate delays for comparison and selective invocation can be masked by the complex wake-up logic.

2) Example of optimized integration into a specific scheduler: While the preceding example illustrates a generic integration framework that can apply broadly to many classes of scheduling logic, integration into specific schedulers presents opportunities to eliminate even the single-gate increase in logic depth required by this solution. To illustrate this point, we use the classic scheduler presented in [26], [27].

The scheme described in [26], shown in Figure 11A, uses a tree of arbiter cells to generate grant signals based on both instruction readiness and functional unit availability. Each arbiter receives a fixed number of request signals from entries in the issue logic, and asserts the corresponding grant signals based on a priority encoding scheme. The arbiter also sets an *anyreq* signal when any of the request lines is asserted, and accepts a *enable* signal, which can be set low to suppress any grant signals that would otherwise be generated. The *anyreq* and *enable* signals are used to compose a tree of arbiters as shown in 11A, which can be scaled to cover a large issue window. The enable signal for an arbiter represents the availability of the corresponding functional unit.

The root arbiter and issue window can be modified to adapt this scheme to SMT-COP without incurring additional overhead in terms of gate depth. Note that in the tree scheme shown in Figure11, *grant-0* and *grant-1* on the root arbiter enable all grant signals for one half of the issue window, while *grant-2* and *grant-3* enable signals of the other half. SMT-COP can thus be incorporated by partitioning the issue window between the two threads, and ensuring that the grant signals received by each are modified by the functional unit's availability with regards to that thread. As shown in Figure 11B, this is accomplished by modifying the root arbiter logic to condition the enable signals on the functional unit's context assignment bit in the partition table. Specifically, new gates are integrated with the arbiter's grant generation modules to ensure that grants signals are only asserted for the portion of the issue queue containing instructions authorized to use the functional unit.

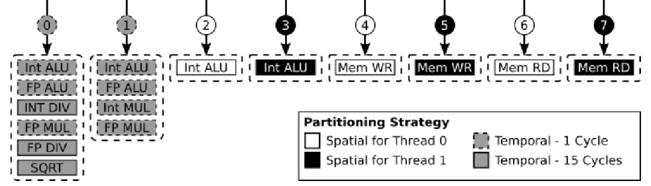

Figure 12: Partitioning strategy for the evaluated SMT-COP implementation.

Because the additional gates are inserted at the root of the tree, their impact is masked by the comparatively long delay required for request signals to propagate up the tree. This design advances on the generic logic introduced in Figure reffig:generic by providing stronger assurances that the SMT-COP logic will not effect the critical path.

V. Evaluation

A. Experimental Setup

To evaluate the security and performance properties of SMT-COP, we implemented SMT-COP using M-Sim [28], a cycle-accurate simulator for SMT architectures. While M-Sim is based on the DEC-Alpha RISC instruction set, the results are indicative of performance for other instruction sets, such as the ARM instruction set or RISC-like x64 μops. Our model supports two thread contexts, similar to the majority of commercial designs.

We modified M-Sim by adding issue ports and a SMT-COP partitioning scoreboard, and implemented the execution logic shown in Figure 1. Figure 12 shows the partitioning rules applied to the ports and functional units. Ports 0-1 were temporally partitioned between the two supported thread contexts. The Integer and Floating Point ALU units on each port, as well as the pipelined multipliers on port 1, were temporally partitioned in concert with their respective ports. The floating point divider, integer divider, and square root units, were assumed to be non-pipelined with a latency of 14 cycles, and were temporally partitioned with an allocation period of 15 cycles. The additional cycle in the allocation period is needed to prevent the first cycle of the allocation period, during which instructions can be issued, from becoming synchronized with the temporal partitioning policy of the port, thus permanently denying access to one of the threads. The remaining functional units and ports were spatially partitioned between threads.

Initially, we experimented with various longer allocation periods for the non-pipelined units, but found that these changes had little impact on performance, with a slight tendency toward degradation.

The remaining simulation parameters are shown in table II. Our issue queue capacity was partitioned equally between the two thread contexts.

B. Security Evaluation

To validate our implementation of SMT-COP, we constructed a classical SMT covert channel similar to the one described

total IQ size	64	**RF size**	128
decode width	8	**issue width**	8
l1 data KiB	32	**l1 ins KiB**	32
l2 KiB	256	**l3 MiB**	2

Table II: Configuration of a Simulated Processor

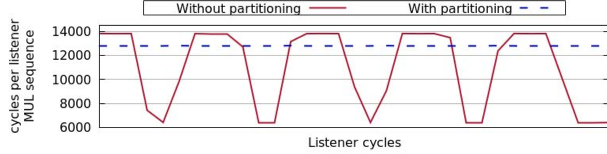

Figure 13: Signal through multiplier covert channel. The signal ”1010101010” is received by the spy.

in [1] and showed the elimination of information leakage under partitioning. Because a covert channel involves collusion between the attacker and its victim, such attacks permit significantly more reliable leakage of information than side-channels that exploit incidental aspects of the victim's execution. Thus, the elimination of the covert channel can be extended to the analogous side channel without loss of generality. The attack involves a spy process equivalent to the one described in listing 1, and a *Trojan* process which runs on the same SMT core and discloses information through contention on the multiply unit. The Trojan signals a ”1” by issuing a series of multiply instructions in rapid succession, and a ”0” by issuing `nop` instructions that leave the multiply unit free. The Spy measures the relative timings of its own multiply operations to recover the message.

Figure 13 shows the signal received by the spy, with partitioning enabled and disabled, in terms of the latency to execute a series of MUL instructions over the course of the spy's execution. The solid, red line shows the signal received through an un-partitioned multiply unit. This signal shows clear variations that disclose the simple message (`1010101010`) transmitted by the Trojan process. The blue, dashed line shows the signal received by the same spy when partitioning is enabled. The latter signal shows no significant variation over time; indeed, the only variation observed was limited to ± 1 cycle. By experimenting with simulator parameters, we verified that this variation is related to contention over commit bandwidth, and not the execution logic. These results confirm that SMT-COP partitioning eliminates the execution logic covert channel, and by extension the execution logic side-channel.

C. Performance Evaluation

To evaluate the performance of SMT-COP, we randomly created workloads from combinations of the SPEC CPU 2006 benchmarks compiled for the Alpha architecture. Although it was infeasible to simulate all possible benchmark combinations, we followed the methodology of previous SMT performance evaluations [29] by selecting combinations to cover many possible execution scenarios. Harmonic means are used unless otherwise noted.

int	fp	mix
bwaves-gamess	astar-gcc	astar-sphinx
cactusadm-namd	astar-omnetpp	cactusADM-omnetpp
GemsFDTD-zeusmp	gcc-bzip2	gobmk-dealII
milc-gromacs	gobmk-hmmer	h264ref
namd-povray	h264ref-omnetpp	hmmer-zeusmp
namd-povray	hmmer-sjeng	leslie3d-gobmk
povray-calculix	sjeng-libquantum	kibquantum-milc
sphinx3-zeusmp	bzip2-h264ref	sjeng-povray

Table III: SMT workloads

These workloads, shown in table III, fall into three categories. Workloads in the *int* category contain two SPEC Integer benchmarks, while those in the *fp* category contain two SPEC floating point benchmarks. Workloads in a third category, *mix*, pair one floating point and one integer-intensive benchmark. The *int* and *fp* workloads are expected to use different sets of functional units, and their differential performances provide insights into the performance characteristics of partitioned architectures.

In our first experiment, we evaluated the performance impact (in terms of combined IPC throughput) of an always-on SMT-COP architecture compared to non-partitioned SMT. The first 5 billion instructions of each thread were fast-forwarded, and the workloads were then simulated until either thread committed one Billion instructions.

1) IPC Comparison by Workload Class: Figure 14 shows the results of these evaluations. Under always-on partitioning, SMT-COP achieved a mean IPC throughput of 1.52, in comparison to a mean IPC of 1.64 for non-partitioned SMT, thus encountering only a 8% performance loss.

Partitioning affected the *int*, *fp* and *mix* workloads in different ways. The *int* workloads showed a mean IPC throughput of 1.38 under partitioning, compared to 1.43 for the baseline. Integer-heavy workloads are thus markedly favorable for SMT-COP partitioning, showing only 4% performance loss on average. In contrast, *fp* workloads incurred a mean IPC of 1.69, relative to a baseline IPC of 1.87, resulting in 10% loss of performance. The *mix* workloads had a mean IPC of 1.52 compared to 1.68 for baseline, resulting in 10% performance degradation.

We ascribe the low overheads of *int* workloads to two primary factors. First, the comparatively large number of integer ALUs available to each thread during any cycle clearly has favorable implications for throughput under partitioning. Second, the functional units required by integer workloads are almost entirely pipelined, reducing the potentially disruptive effects of multi-cycle allocation periods. We note that while the integer divider has such an allocation period, it was practically never used by the benchmarks in our evaluation set. Conversely, the relatively high overheads associated with *fp* and *mix* workloads arise from their use of resource types with few instances, and the multi-latency temporal partition that must be applied to some of these resources.

These results indicate that even with comprehensive partitioning, SMT-COP conserves much of the benefit of baseline SMT. For reference, we evaluated the mean IPC throughput for single-threaded runs for the individual integer and floating point benchmarks used in our experiments, and compared

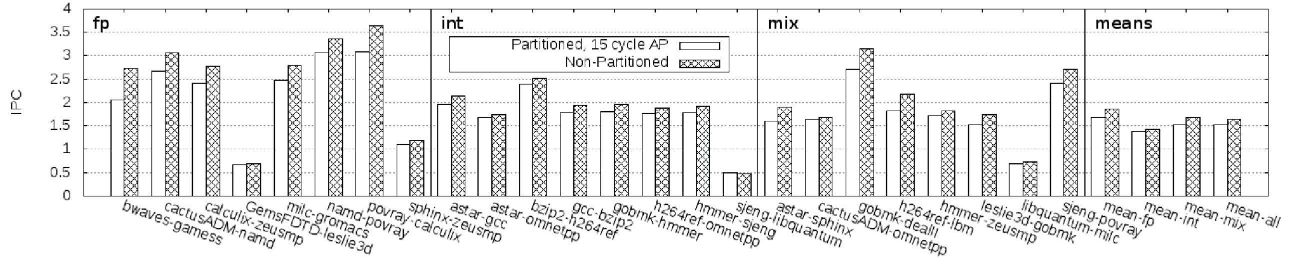

Figure 14: Performance Overheads of SMT-COP for SPEC 2006 Benchmarks

these values to the mean IPC throughputs for baseline and partitioned SMT reported above. In this evaluation, comprehensive SMT-COP retained 69% and 62% of the performance speedup provided by baseline SMT, for *fp* and *int* workloads, respectively. Retention improroves significantly for selective partitioning strategies, as described below.

In addition to evaluating IPC throughput, we measured the fairness of SMT-COP using the methodology presented in [30]. The results of this evaluation were closely aligned with those for IPC. The mean degradation in fairness under SMT-COP for the complete set of workloads was 8%, while degradations of 11%, 5%, and 6% occurred for *fp*, *int*, and *mix* workloads respectively.

To gain more detailed insights into the source of SMT-COP overheads, we instrumented the simulator to measure the number of issue stalls arising from the partitioning of each resource. Figure 15 shows the results of this evaluation. Because some workloads experienced zero stalls for certain resources, arithmetic means are used to summarize the results for each workload class. The results of this evaluation confirm our inferences regarding the sources of higher overhead in the *fp* workloads.

Although the baseline performance results demonstrate SMT-COP's ability to retain much of the acceleration provided by SMT, the performances of different workloads suggest ways in which a system could be optimized to further enhance SMT-COP. Competition among floating point intensive applications for limited resources was a significant source of slowdown in SMT-COP. This effect could be mitigated by implementing policies that avoid co-scheduling multiple floating point intensive threads on the same SMT core in cases where *int* and *mix* workloads would provide better throughput. Symbiotic scheduling schemes such as those explored in [31] could be used to implement this policy, and could be applied by the OS without compromising SMT-COP's security guarantees.

2) CDP Performance Evaluation: In addition to evaluating the baseline configuration of SMT-COP, we implemented CDP to reduce the overheads associated with floating point workloads. Specifically, we applied CDP to the fp-multiply and fp-divide units in our baseline architecture, which are the primary source of the increased overheads of the floating point workloads under SMT-COP. Initially, we experimented with various partitioning periods ranging from 1000 to 40k cycles. However, as with the allocation periods for temporally partitioned functional units, these experiments did not produce significant differences in performance. We thus report results for a representative 10K cycle partitioning period.

Figure 16 shows the results of this evaluation for floating point workloads. The mean IPC performance for the CDP implementation is 1.77, which increases the floating point performance of SMT-COP to about 94% of baseline IPC. This represents the retention of 82% of the performance gains of baseline SMT, relative to mean single-threaded floating point IPC. These results show that CDP can largely close the gap between floating point and integer performance in SMT-COP.

VI. Related Works

Previous proposals have examined the effects of SMT resource partitioning on throughput and/or fairness. The work of [32] examined the impact of statically partitioning multiple datapath resources among SMT contexts, including the issue queue, issue bandwidth, reorder buffer, and commit bandwidth. This study is only concerned with performance aspects of partitioning, without considering security implications. The partitioning schemes of [32] do not stop attacks on execution units addressed in this paper. Various policies have also been proposed to partition fetch and issue bandwidth based on metrics such as memory accesses [29], [30], branch behavior, [33], or the utilization of other pipeline resources [34]. Similar to [32], these works addressed performance and fairness, rather than security: since bandwidth allocations depend on program execution features, they do not eliminate possible side-channels, including those through the execution logic. Previous work also investigated performance-driven optimizations to SMT scheduling logic [35], [36]. These solutions can be combined with SMT-COP to add security. The general approach of temporally partitioning a resource to eliminate timing side-channels was previously explored in [37]. This work applied temporal partitioning to a shared memory controller, rather than SMT execution logic.

Prior work demonstrating information leakage through SMT execution logic [1], [5], [38] motivated the development of SMT-COP. The work of [1] identified contention on functional units as a potential source of information leakage, and demonstrated a covert channel using contention on the multiply unit. In addition, [1] briefly proposed two mitigations to execution side-channels, but did not pursue implementation or provide performance evaluations. The first proposal selectively disables SMT on processors running sensitive workloads, while the second modifies existing fairness-enforcement hardware

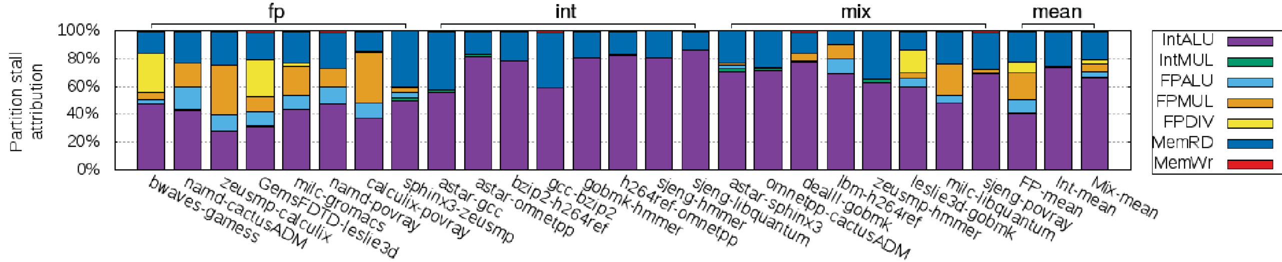

Figure 15: Fraction of stalls attributed to each type of functional unit, with arithmetic means

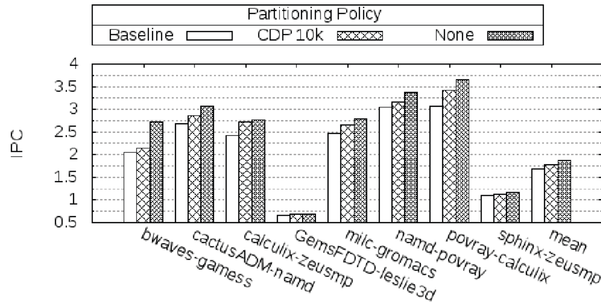

Figure 16: Comparison of IPC throughput for baseline SMT-COP partitioning, CDP with 10k cycle timeout, and unpartitioned SMT.

on Intel architectures to partition scheduler queues between executing threads. Though effective in terms of security, the first method involves expensive rescheduling operations that rely on potentially insecure systems software. The second approach is presented as a mitigation rather than a hard defense, and leaves open the possibility of information leakage, as per-thread caps on queue occupancy would still allow execution delays due to the presence of instructions previously enqueued by other threads.

The work of [38] demonstrated the feasibility of using a functional unit side-channel in an attack, specifically targeting the OpenSSL implementation of RSA key generation. [5] evaluated a related vulnerability involving execution ports, which is used to construct both a covert-channel, and in a side-channel attack to extract a private key from OpenSSL. As another recent example, SMoTherSpectre attack utilizes port contention on SMT to mount a code-reuse attack to leak data from OpenSSL [6]. In briefly discussing mitigations, [5] recommends simply disabling SMT for sensitive processes. SMT-COP protects from both of these attacks without disabling SMT.

Prior research also demonstrated information leakage through other SMT components, including private caches [39], [40], branch predictors [38], and TLB logic [41]. Vulnerabilities through shared caches and branch predictors have also been extensively studied in the context of single-threaded cores, both in multi-core and single-core settings. For example, [42] uses collisions in the branch predictor tables to extract ASLR offset data, which can be used to circumvent address randomization. Along similar lines, the Spectre attack [43] combines branch predictor and cache vulnerabilities to force the execution of malicious code that extracts sensitive data. Cache side-channels [18], [39], [44]–[48] are well established as a generalized vector for information leakage, and have also been shown to be significantly enhanced by SMT architectures [1], [2]. Recent speculative attacks also demonstrated possible information leakage in SMT processors through shared fill buffers [23].

Researchers have also investigated defense mechanisms [4], [16]–[21] against attacks on these resources, and these types of attacks and defences are generally well understood. These proposals are orthogonal and complimentary to SMT-COP, and could be combined with our system to create a comprehensively leakage-free SMT architecture.

The timestamp fuzzing approach proposed in [24] attempts to hinder efforts to measure contention latencies by adding random noise to `rtdsc` measurements. Consequently, this solution mitigates existing timing side-channels, including those through execution logic. However, this policy is potentially challenging to deploy, as a small number of applications have legitimate uses for `rtdsc` instructions and must be granted permission to use them in a secure manner. More significantly, subsequent work has shown attacks that circumvent protections based solely on timestamp fuzzing [49].

VII. Concluding Remarks

Side-channel attacks exploiting shared processor resources have recently emerged as a serious security threat. SMT processors are particularly vulnerable to such attacks due to the degree of sharing and their ability to execute several programs simultaneously, one of which can be a victim and one an attacker. In this paper we proposed SMT-COP — a mechanism to spatially and temporally partition the execution ports and associated functional units in SMT processors to close execution side-channels and stop recently emerged SMT attacks. We demonstrated that the protection can be achieved with modest performance impact and minimal impact on the design complexity and timing of the scheduling logic. We also proposed several optimizations to enable execution logic partitioning selectively, achieving further improvements in performance.

VIII. Acknowledgements

The work in this paper is partially supported by National Science Foundation grants CNS-1422401 and CNS-1617915.

REFERENCES

[1] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," *Proceedings - Annual Computer Security Applications Conference, ACSAC*, pp. 473–482, 2006.

[2] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," 2017.

[3] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, p. 494, 2007.

[4] D. Evtyushkin, D. Ponomarev, and N. Abu-ghazaleh, "Understanding and Mitigating Covert Channels Through Branch Predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–24, 2016.

[5] A. Cabrera Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," *Cryptology ePrint Archive: Report 2018/1060*, 2018.

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[7] M. Kettenis. https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html.

[8] L. Vilanova, A. Nadav, and Y. Etsion, "Using SMT to Accelerate Nested Virtualization," in *Proceedings of the International Symposium on Computer Architecture*, 2019.

[9] 2018.

[10] "The zen core architecture." https://www.amd.com/en/technologies/zen-core. Accessed: 2019-04-14.

[11] "Simultaneous multi-threading (smt)." https://www.ibm.com/support/knowledgecenter/en/SSZJY4_3.1.0/liabp/liabpsmt.htm. Accessed: 2019-04-14.

[12] ARM, "ARM Cortex-A9 Technical Reference Manual revision r4p1."

[13] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution.," in *HASP@ ISCA*, p. 10, 2013.

[15] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions.," in *HASP@ ISCA*, p. 11, 2013.

[16] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. V. Rozas, G. Heiser, and R. B. Lee, "CATalyst:Defeating Last-Level Cache Side Channel Attacks in Cloud Computing.," no. Vm, pp. 1–27, 2016.

[17] Z. Wang and R. B. Lee, "A Novel Cache Architecture with Enhanced Performance and Security," vol. 93, pp. 88–93, 2008.

[18] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 605–622, 2015.

[19] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks," *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 347–360, 2017.

[20] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, IEEE, 2018.

[21] I. Vougioukas, A. Sandberg, N. Nikoleris, S. Diestelhorst, B. Al-Hashimi, and G. Merrett, "Brb: mitigating branch predictor side-channels," 2019.

[22] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the International Symposium on Computer Architecture*, ISCA, June 2019.

[23] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," *arXiv:1905.05726*, 2019.

[24] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp : Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks," *Proceedings of the 39th Annual International Symposium on Computer Architecture*, vol. 00, no. c, pp. 118–129, 2012.

[25] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, "Demand-driven software race detection using hardware performance counters," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 165–176, ACM, 2011.

[26] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the Complexity of Superscalar Processors," tech. rep.

[27] S. Palacharla, N. F. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, 1997.

[28] J. Sharkey, "M-Sim: A Flexible, Multithreaded Architectural Simulation Environment," *Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton*, 2005.

[29] F. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically Controlled Resource Allocation in SMT Processors," *37th International Symposium on Microarchitecture (MICRO-37'04)*, pp. 171–182, 2004.

[30] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 164—-171, 2001.

[31] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234—-244, 2000.

[32] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, vol. 2003-Janua, pp. 15–25, 2003.

[33] H. Ando, "Performance Improvement by Prioritizing the Issue of the Instructions in Unconfident Branch Slices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 82–94, IEEE, 2018.

[34] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, L. Stamm, and J. L. Lo, "Exploiting Choice: Instruction Simultaneous," *Proceedings of the 23rd annual international symposium on Computer architecture*, pp. 191–202, 1996.

[35] J. J. Sharkey and D. V. Ponomarev, "Efficient instruction schedulers for smt processors," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pp. 288–298, IEEE, 2006.

[36] F. M. Sleiman and T. F. Wenisch, "Efficiently scaling out-of-order cores for simultaneous multithreading," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 431–443, IEEE, 2016.

[37] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 225–236, IEEE, 2014.

[38] O. Aciiçmez and J. P. Seifert, "Cheap hardware parallelism implies cheap security," *Proceedings Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2007*, pp. 80–91, 2007.

[39] C. Percival, "Cache missing for fun and profit," *BSDCan 2005*, pp. 1–13, 2005.

[40] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 110–124, Springer, 2010.

[41] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 955–972, 2018.

[42] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 40:1–40:13, IEEE Press, 2016.

[43] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[44] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - Bringing access-based cache attacks on AES to practice," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 490–505, 2011.

[45] Y. Yarom, K. Falkner, and K. Falkner, "F lush + R eload : A High Resolution , Low Noise , L3 Cache Side-Channel Attack," 2014.

[46] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing - And its application to AES," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 591–604, 2015.

[47] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 72, ACM, 2016.

[48] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 35, 2012.

[49] S. Bhattacharya and C. Rebeiro, "Unraveling Timewarp : What all the Fuzz is About ?," *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.