Do Not Predict – Recompute! How Value Recomputation Can Truly Boost the Performance of Invisible Speculation

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

Zamshed I. Chowdhury
University of Minnesota
Minneapolis, MN, USA
chowh005@umn.edu

Shayne Wadle
University of Wisconsin
Madison, WI, USA
swadle@cs.wisc.edu

Ismail Akturk
Ozyegin University
Istanbul, Turkey
ismail.akturk@ozyegin.edu.tr

Alberto Ros
University of Murcia
Murcia, Spain
aros@ditec.um.es

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

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

Ulya R. Karpuzcu
University of Minnesota
Minneapolis, MN, USA
ukarpuzc@umn.edu

2021 International Symposium on Secure and Private Execution Environment Design (SEED) | 978-1-6654-2025-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEED51797.2021.00021

***Abstract*—Recent architectural approaches that address speculative side-channel attacks aim to prevent software from exposing the microarchitectural state changes of transient execution. The *Delay-on-Miss* technique is one such approach, which simply delays loads that miss in the L1 cache until they become non-speculative, resulting in no transient changes in the memory hierarchy. However, this costs performance, prompting the use of value prediction (VP) to regain some of the delay.**

However, the problem cannot be solved by simply introducing a new kind of speculation (value prediction). Value-predicted loads have to be validated, which cannot be commenced until the load becomes non-speculative. Thus, value-predicted loads occupy the same amount of precious core resources (e.g., reorder buffer entries) as Delay-on-Miss. The end result is that VP only yields marginal benefits over Delay-on-Miss.

In this paper, our insight is that we can achieve the same goal as VP (increasing performance by providing the value of loads that miss) without incurring its negative side-effect (delaying the release of precious resources), if we can safely, non-speculatively, recompute a value in isolation (without being seen from the outside), so that we do not expose any information by transferring such a value via the memory hierarchy. *Value Recomputation*, which trades computation for data transfer was previously proposed in an entirely different context: to reduce energy-expensive data transfers in the memory hierarchy. In this paper, we demonstrate the potential of value recomputation in relation to the Delay-on-Miss approach of hiding speculation, discuss the trade-offs, and show that we can achieve the same level of security, reaching 93% of the unsecured baseline performance (5% higher than Delay-on-miss), and exceeding (by 3%) what even an oracular (100% accuracy and coverage) value predictor could do.

I. Introduction

With the disclosure of Spectre [21] and Meltdown [26] in early 2018, *speculation*, one of the fundamental techniques for achieving high performance, proved to be a significant security hole, leaving the door wide open for side-channel attacks [6], [16], [27], [48] to "see" protected data [21], [26]. As far as the instruction set architecture (ISA) and the target program are concerned, this type of information leakage through microarchitectural (μ-architectural) state and structures is not illegal because it does not violate the functional behavior of the program. But *speculative* side-channel attacks reveal secret information during *misspeculations*, i.e., discarded execution that is not a part of the normal execution of a program. The stealthy nature of a speculative side-channel attack is based on *microarchitectural* state being changed by misspeculation even when the *architectural* state is not.

First response techniques: delay, hide&replay, or cleanup? A number of techniques have already been proposed to prevent *microarchitectural* state from leaking information during speculation, either by *delaying* such effects [13], [35], [45], [50], *hiding* them and making them re-appear for successful speculation (*hide&replay*) [34], [46] or *cleaning up* the changes when speculation fails [32]. Because these techniques were proposed for different threat models (i.e., responding to a different set of known or unknown threats), provide different protection for parts of the system that can leak secrets (e.g., caches, DRAM, core), and make different assumptions for what other parts of the system are protected (hence carry different costs), a direct comparison of all of them is, as of yet, not feasible. In this paper, without loss of generality, we focus on delay techniques and for convenience we adopt the threat model of the work by Sakalis et al. on *Delay-on-Miss (DoM)* [35].

What problem are we solving? Delay-on-Miss is the simple idea of delaying any speculative load that misses in the L1 cache until the earliest time when it becomes non-speculative. To recover some of the lost performance from delaying critical instructions (loads that miss) Sakalis et al. proposed to use *value prediction* (*VP*) for the delayed misses in hope of performing useful work for the delayed loads and their dependent instructions. In other words, the aim of VP is to increase instruction-level-parallelism (ILP) by executing dependent instructions using load-value prediction.

The conundrum of this approach is the following: VP, as another form of speculation, forces predicted loads to be validated *in-order* in the memory hierarchy, as each load remains speculative until all older loads have been performed non-speculatively. This means that the validation of these loads *cannot have any memory-level-parallelism (MLP)*. Thus, any possible gains in ILP from VP during speculation, could be compromised by the hindrance of MLP at validation [33].

A new perspective: In this paper, we ask the question: Can we create "secret" values, *invisible to an attacker*, for the delayed loads, without having to compromise MLP to validate them afterwards? Our key intuition is that the answer lies in *value re-computation* (*VRC*) also known as *Amnesic Computing* [3]. The idea is that recomputing a value on an L1 miss — a value that otherwise would have been loaded from the memory hierarchy — can replace the need to access the memory hierarchy. This requires having a backward slice of producer instructions on a per (load) value basis, along with the necessary input operands to perform recomputation. By construction, slices do not contain any branch or memory references (be it a store or a load). Most importantly, recomputation is also not speculative by construction, hence prevents nested speculation (and negative side effects thereof).

Our Contributions:

- We propose to apply an unconventional idea, *value recomputation* (previously proposed as a means to evade the cost of moving data in the memory hierarchy) to solve this problem. We devise a μ-architectural framework for security-aware value recomputation, well fitted to the threat model at hand and show the synergy with Delay-on-Miss.
- We evaluate the potential of *value recomputation* in eliminating speculative metadata, which makes classic processors vulnerable to numerous threats, including but not limited to what is known so far.

A summary of our results: This is the first μ-architectural proposal that has the potential of outperforming the (unsecured) baseline in terms of performance and energy-efficiency, reducing the performance overhead of Delay-on-Miss by 42%. In this paper, we provide a quanti-

978-1-6654-2025-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SEED51797.2021.00021

tative discussion on how to unlock this potential. Practically, we cover (known or yet to come) threats posed by speculative memory reads.

II. Background

A. Speculative Shadows

Sakalis et al. introduced the concept of *Speculative Shadows* to reason about the earliest time an instruction becomes non-speculative and is considered safe to execute regardless of its effects on μ-architectural state [34], [35]. Speculative shadows can be of the following types: *E-Shadows* are cast by any instruction that can cause an exception; *C-Shadows* are cast by control instructions, such as branches and jumps, when either the branch condition or the target address are unknown or have been predicted but not yet verified; *D-Shadows* are cast by potential data dependencies through stores with unresolved addresses (read-after-write dependencies); *M-shadows* are cast by speculatively executed memory accesses that may be caught violating the ordering rules of a memory model (e.g., total store order—TSO) and therefore may need to be squashed; and *VP-shadows* are cast by value-predicted loads [35]. To be more specific, shadows demarcate regions of speculative instructions. So far, attacks have been demonstrated under the E- [26], C- [21], and D-Shadows [10] only, but we cannot exclude future attacks using the rest.

B. Delay-on-Miss

The goal of Delay-on-Miss (DoM) is to hide speculative changes in the memory hierarchy (including main memory). To achieve this, Delay-on-Miss delays speculative loads that miss in the L1 cache. Loads that hit in the L1 (and their dependent instructions) are allowed to execute speculatively as their effects (i.e., on the L1 replacement state) can be deferred to when the loads are cleared from any speculative shadow. The miss of a delayed load is allowed to be resolved in the memory hierarchy at the earliest point the load becomes non-speculative. An efficient mechanism to track shadows is proposed by Sakalis et al. [35].

Under Delay-on-Miss, the vast majority of loads are executed speculatively (more than 80% on average [35]), which causes a notable fraction of the loads to be delayed. This takes up precious resources (i.e., entries in the instruction queue, the reorder buffer, and the load/store queue) and eventually stalls instructions from committing. The significant amount of speculation that is performed in modern out-of-order cores, results in each load being covered by several speculative shadows (five on average according to our simulations). This forces the majority of the loads to be executed serially, severely limiting MLP [33], [44]. Furthermore, removing any individual shadow (e.g., the C-Shadow) has a limited effect, as the load can be covered by another overlapping shadow [44].

Speculative interference attacks [5] describe a situation where the execution of younger speculative instructions can interfere with the execution of an older bound-to-commit instruction, such that its execution gets delayed [5]. Such delays of older bound-to-commit instructions can cause a change in the order of executed loads that are not protected by DoM, and thus can leak information. Speculative interference attacks rely on non-pipelined instructions for causing the interference and can be prevented simply by enforcing that non-pipelined instructions (i.e., division, square root, etc.) are executed strictly in order [36].

C. Delay-on-Miss and Value Prediction

The concept behind using value prediction with Delay-on-Miss is to speed up the delayed loads (and their dependent instructions) and regain some of the lost performance. However, value prediction—no mater how good we make it (even under 100% coverage and accuracy)—gives only a limited benefit on top of Delay-on-Miss [33]. Value prediction clearly cannot regain the lost performance because of the following:

Value prediction cannot help much as it simply provides values early; however, the validation is still delayed until all shadows have

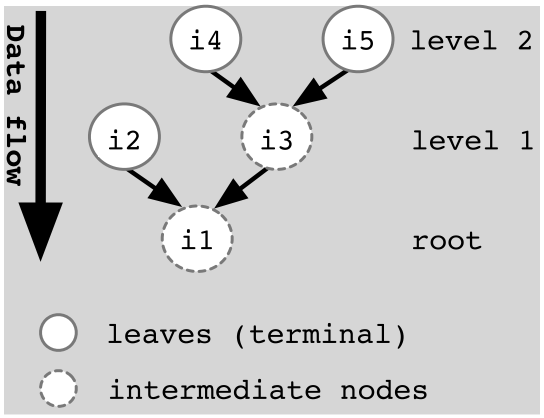

Fig. 1: Backward slice example.

been lifted. Thus, precious core resources are still occupied until the same point in time as simply delaying the load. The only perceptible difference is a faster commit of pre-executed dependent instructions if the validation of a value-predicted load proves to be correct.

Furthermore, value prediction introduces a new speculative shadow, which is referred to as the *VP-Shadow*. This new shadow is only lifted from younger loads when the validation of the value prediction is complete. Thus, preventing younger loads from validating in parallel limits the MLP, which results in value prediction occupying precious resources in the same manner as Delay-on-Miss.

D. Value Recomputation

Due to imbalances in technology scaling, the energy usage (and latency) of data transfers in the memory hierarchy can easily exceed the energy usage (and latency) of value recomputation [15]. Value recomputation (VRC) is proposed as a way to trade off data movement in the memory hierarchy for in-core computation to save energy [2], [3]. The basic idea is to swap slow and energy-hungry loads for recomputation of the respective data values. This is achieved by identifying a slice of producer instructions of the respective data value and executing them when the value is needed. Each such slice forms a backward slice of execution, and strictly contains only arithmetic and logic instructions.

As depicted in Figure 1, each slice represents a data-dependency graph, where nodes correspond to producer instructions to be (re)executed. Data flows from the leaf nodes to the root. The root represents the producer of the store whose value will be recomputed when its corresponding (consumer) load is encountered, i.e., a load accessing the same memory location. Nodes at level 1 are immediate producers of the (input operands of the) root, nodes at level 2 are producers of nodes at level 1, and so on and so forth. The nodes which do not have any producers are terminal instructions whose input operands must be available at the time of recomputation. If these input operands are read-only values to be loaded from memory (such as program inputs) or register values that will be overwritten, then buffering of these values are needed to enable recomputation of the load [3].

Premise: VRC has the potential to render a more energy efficient (and faster) execution than servicing a miss in the memory hierarchy. At the same time, there is no need for MLP, since as opposed to value prediction, *VRC is not speculative* by itself and does not require any costly validation. A recomputed load can be committed as soon as all the shadows are lifted—this is in stark contrast to Delay-on-Miss with value prediction, which require a load/validation to be performed before commit.

E. Threat Model

We target speculative side- or covert-channel attacks that utilize the memory hierarchy (caches, directories, and the main memory) as their side-channel. Non-speculative cache side-channel attacks, as well as attacks that use other side-channels (such as port contention) are not covered by Delay-on-Miss and, although still possible, are outside the scope of this work. We make no assumptions as to where the attacker is located in relation to the victim or if they share the same virtual memory address space or not. As a matter of fact the attacker and the victim can be the same process, as in the Spectre v1 attack [21]. We assume that the attacker can execute arbitrary code or otherwise redirect the execution of running code arbitrarily. How the attacker manages to execute or redirect such code is beyond the scope of this work. Instead of focusing on preventing the attacker from accessing data illegally, we focus on preventing the transmission of such data through a cache or memory side- or covert-channel.

In this work, we use the concept of speculative shadows to determine when a load is safe or not. Speculative shadows determine the earliest point at which an instruction is guaranteed to be committed and retired successfully. Other works, such as InvisiSpec [46] and NDA [45], make different assumptions based on the threat model. For example, InvisiSpec provides two different versions, one based on the initial Spectre attacks where only the equivalent of C-Shadows are considered as part of the threat model, and one based on protecting against all possible future attacks, utilizing all the shadows. Similarly, NDA provides different solutions if only C-Shadows are considered (strict/permissive data propagation), if D-Shadows should also be considered (bypass restriction), or if all shadows should be considered (load restriction). In this work, we follow the strictest approach and assume that all shadows have the potential of being abused, as we cannot reasonably argue that any of them are not exploitable.

III. Invisible Speculative Execution through Recomputation (ISER)

We will next detail the mechanics of our novel approach, *Invisible Speculative Execution through (Value) Recomputation* (ISER). Due to space limitations, we will focus on how value recomputation can help eliminate the targeted threats (Section II-E). For a thorough discussion of value recomputation we refer the reader to previous works [2], [3].

A. Execution Semantics

ISER only resorts to recomputation for regenerating values that otherwise would be read by a speculative load from the memory hierarchy, and only so, if the respective speculative load misses in the L1 cache. Recomputation takes place as long as a slice exists and the input operands to the slice instructions can be made readily available.

While ISER shares basic μ-architectural structures with Amnesiac [3] to facilitate VRC (such as dedicated buffers to prevent corruption of μ-architectural state during recomputation), its execution semantics are quite different when it comes to slice identification and triggering recomputation. These stem from the defining difference in optimization targets: Amnesiac uses VRC to maximize energy efficiency irrespective of security implications. ISER, on the other hand, uses VRC to eliminate (already known or yet to be discovered) threats induced by speculative loads. In a nutshell, differences between Amnesiac and ISER expand along two axes:

- *What to recompute (slice identification):* As opposed to Amnesiac, ISER does not impose any direct constraint to preserve energy efficiency, as we are not after minimizing energy or latency per load. As long as a slice exists, and its inputs can be made readily available at the anticipated time of recomputation, ISER would consider it for recomputation. The only practical limitation on slice length may stem from storage overhead of μ-architectural buffers in this case (Section III-C).
- *When to recompute:* ISER swaps speculative loads that miss in L1 for recomputation (i.e., with producer instructions of the respective value along a slice). Amnesiac on the other hand, triggers recomputation (irrespective of whether the load is speculative or not) only if it is more energy-efficient to do so.

We continue with ISER design specifics, limitations, and side effects including coherence and consistency implications.

B. Slice Formation & Annotation

Similar to Amnesiac, we rely on a compiler pass (backed by profiling) to form and annotate slices, which mainly constitutes dependency analysis to identify the producer instructions for each load.[1] Slice creation is a *best effort* under strict validity guarantees. Not being able to generate a recomputation slice for a load is not a security weakness under a security technique such as Delay-on-Miss, but simply a missed optimization opportunity. Although in this paper the slice formation is conservative, as we will see later, the requirement for strict guarantees of the recomputation validity can be relaxed (potentially increasing the coverage of recomputation, i.e., portion of load values that can be recomputed, and addressing coherence issues) if the appropriate architectural support is available. However, such extensions are outside the scope of this paper and will be fully evaluated in future work.

The slice formation pass builds the slice as a data dependency graph, where the immediate producer of the value to be loaded resides at the root (Figure 1). As opposed to Amnesiac, the restriction to slice length comes from slice inputs or storage requirements (rather than the associated energy cost). If, during the traversal of data dependencies, we encounter other load instructions, we replace them recursively with the respective producer instructions. This recursive growth can continue until a store to the same address is encountered. Loads and stores cannot be present in any slice by definition.

Once construction is complete, each slice gets embedded into the binary. Similar to Amnesiac, the special control flow instruction `RCMP` indicates recomputation opportunities, which semantically corresponds to an atomic bundle of a conditional branch + load (where no prediction is involved for the "branch" portion). The branching condition is resolved during execution of the `RCMP` instruction as follows: if the respective load (while shadowed) misses in L1, `RCMP` acts as a jump to the entry point (starting from the terminal instructions) of the corresponding slice. Otherwise, (i.e., the load is not shadowed, or the shadowed load hits in L1), `RCMP` acts as a conventional load. All operands of the respective load and the starting address of its slice form the operands of the `RCMP`. An `RTN` instruction (similar to a procedure return in nature) demarcates the end of each slice and terminates the execution of the slice and returns the control back to the point where the `RCMP` computation was initiated. Before the return takes place, the recomputed value is provided to the consumers of the respective load, in the same way as if the load was actually performed (i.e., by passing the value in a physical register).

As explained by Akturk and Karpuzcu [3], recomputation is possible, even if the compiler cannot prove that all input operands of terminal instructions correspond to immediate or live register values at the anticipated time of recomputation, by keeping such input operands (e.g., overwritten register values) in a dedicated buffer. For any operand of this sort, a `REC` instruction is inserted directly after the instruction producing the value of the operand. `REC` takes as operands the destination register of the previous instruction and an integer operand, which uniquely identifies the saved value. `REC` practically checkpoints the input operand to a dedicated buffer.

C. ISER Architecture

ISER implements the shadow tracking technique proposed by Sakalis et al. [35]. The shadow tracking consists of a *shadow buffer*

[1]Instead of a compiler, the same job can be performed by *dynamic binary instrumentation* at run time (albeit with probably inferior alias analysis but more dynamic information), rendering recompilation unnecessary in deployments where it is not an option.

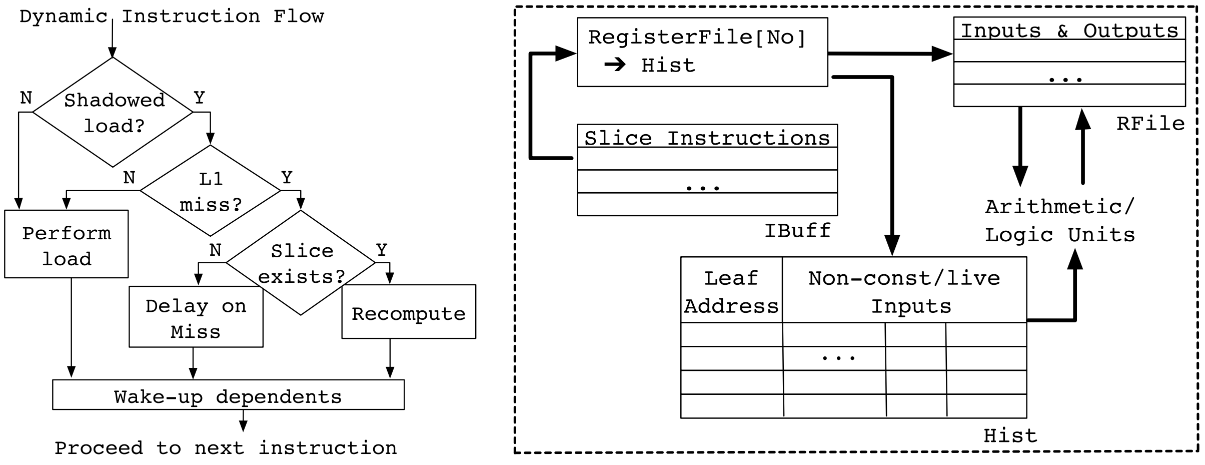

Fig. 2: ISER overview: All μ-architectural buffers have an `invalid` field per entry to manage space (de)allocation.

(SB) that acts as a circular buffer similar to the reorder buffer (ROB). When a shadow casting instruction enters the ROB a new entry is allocated at the tail of the SB. Every load that enters the ROB checks the SB and if not empty, an entry is allocated in a *release queue* that associates the load with the youngest entry in the SB (i.e., its tail). The load remains speculative as long as the head of the SB is marked as unresolved and not equal to the SB entry associated with the load. This mechanism performs a simple comparison between the head of the release queue and the head of the shadow buffer to identify when loads exit all their shadows, thus, avoiding the need for costly content addressable memory (CAM) searches.

On top of this, as depicted in Figure 2, ISER uses a few small buffers that serve two main purposes: (1) Keeping μ-architectural state intact during recomputation; (2) Making slice instructions and operands available at the time of recomputation.

The Instruction Buffer (IBuff) caches slice instructions in order to avoid unnecessary pressure on the instruction cache. Fetch logic fills IBuff while IBuff feeds the decode stage. Each recomputation slice that depends on live data indicates this via its `RCMP` instruction, which causes the rename stage to create a snapshot of the rename tables, similar to any other branch instruction, at the time when the `RCMP` is renamed. If the recomputation is later triggered (i.e., the `RCMP` is speculative and misses in the L1 cache) then the snapshot is restored and used to read live registers and rename any registers written by the instructions in the recomputation slice to new free physical registers. For slices with no live registers the existing rename tables are used as is. This is possible since the already mapped registers are never read by the slice and the rename tables are simply used to write intermediate results during the recomputation to free physical registers. The rename tables are restored to the state before the initiation of the recomputation once all instructions in the slice have been renamed, which is demarcated by `RTN`. `RTN` is not renamed to a new physical register and instead writes the result of the recomputation to the register allocated by the `RCMP` of the slice. All physical registers allocated for slice instructions are freed as soon as `RTN` writes the result to the allocated physical register.

The History Table (Hist) keeps the input operands (such as overwritten register values) for each terminal instruction. `REC` instructions inserted into the orignal program store values to the Hist and `read` instructions in the slices read these values back from the Hist (see Figure 3 for an example).

For ISER, an `RCMP` always translates into `branch on L1 miss` for speculative loads. As shown in Figure 2, for each encountered `RCMP` instruction, ISER first checks whether the corresponding load

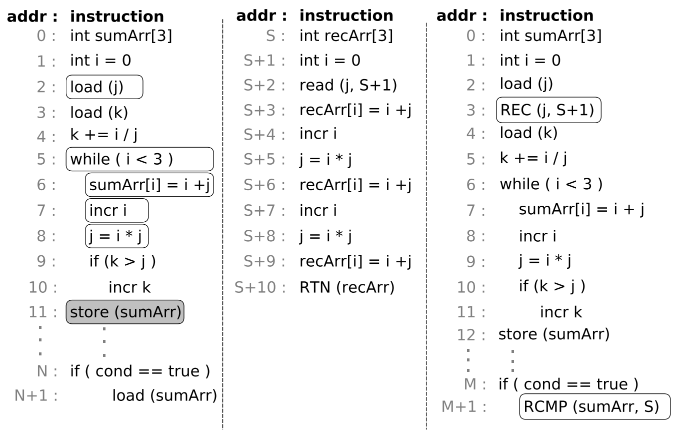
```
addr : instruction
   0 : int sumArr[3]
   1 : int i = 0
   2 : load (j)
   3 : load (k)
   4 : k += i / j
   5 : while ( i < 3 )
   6 :   sumArr[i] = i +j
   7 :   incr i
   8 :   j = i * j
   9 :   if (k > j )
  10 :     incr k
  11 : store (sumArr)
   :       :
   N : if ( cond == true )
 N+1 :     load (sumArr)
```
```
 addr : instruction
    S : int recArr[3]
  S+1 : int i = 0
  S+2 : read (j, S+1)
  S+3 : recArr[i] = i +j
  S+4 : incr i
  S+5 : j = i * j
  S+6 : recArr[i] = i +j
  S+7 : incr i
  S+8 : j = i * j
  S+9 : recArr[i] = i +j
 S+10 : RTN (recArr)
```
```
addr : instruction
   0 : int sumArr[3]
   1 : int i = 0
   2 : load (j)
   3 : REC (j, S+1)
   4 : load (k)
   5 : k += i / j
   6 : while ( i < 3 )
   7 :   sumArr[i] = i + j
   8 :   incr i
   9 :   j = i * j
  10 :   if (k > j )
  11 :     incr k
  12 : store (sumArr)
   :    :
   M : if ( cond == true )
 M+1 :   RCMP (sumArr, S)
```

Fig. 3: Illustration of (a) slice identification; (b) slice generation; and (c) VRC-enabled code.

is speculative, and if so, whether it misses in L1. ISER triggers recomputation for any shadowed load that misses in L1. An `RCMP` instruction will always produce a value (either loaded or recomputed) so for each `RCMP` a physical register is allocated by the conventional renaming mechanism.

On a speculative L1 miss, ISER jumps to the entry point of the corresponding slice and starts fetching instructions. Inputs to a slice instruction can either come from *(i)* live register inputs, *(ii)* live values stored in Hist, or *(iii)* temporary values written to a free physical register. Live registers are read directly from the physical register file using the restored renaming tables. Architectural registers written by slice instructions are mapped to free physical registers using the conventional rename logic. Values stored in Hist are read by specific `read` instructions that use the slice with an offest to identify the value. Instructions are fetched until hitting `RTN`, which stores the produced value to the physical destination register and wakes-up consumers of the recomputed value, and restores the rename tables to the state before the recomputation was initiated. The `RCMP` instruction is then committed as any other instruction without further delays.

Figure 3 provides an illustrative example. Figure 3(a) shows a pseudo-code excerpt, where we want to create a backward slice for the stored data *sumArr*, which will later be (speculatively) loaded (line

$N+1$). The instructions within boxes in Figure 3(a) are involved in the calculation of *sumArr*, which are identified by the compiler (notice that the store instruction for *sumArr* is not part of the slice but informs us about the memory address of the corresponding value). Figure 3(b) shows the resulting backward slice (i.e., only the instructions involved in generating the value of *sumArr*). In this illustration, we assume that input j to the slice is stored in Hist by a `REC` instruction, as shown in Figure 3(c). Notice that the slice does not contain any control flow instruction, thus the while loop used in Figure 3(a) to generate values of *sumArr* is unrolled in Figure 3(b). Following the semantic explained earlier, `RCMP` instruction at address $M+1$ in Figure 3(c) replaces the ordinary load instruction. Recall that `RCMP` works as an ordinary load instruction if it hits in L1. However, if it misses L1, `RCMP` jumps to entry point of the corresponding slice (which is at address S in Figure 3(b)), and thereby avoids any access to the lower levels of the memory hierarchy. After jumping to the slice entry point, inputs to the slice that were recorded earlier can be read from Hist (by the `read` instruction Figure 3(b)), and the desired output can be recalculated by fetching and executing instructions in the slice. Notice that *recArr* is used as a temporary placeholder for recomputed value, to keep the content of memory address of *sumArr* intact (i.e., recomputation has no side effect/change in existing architectural state of the ongoing computation). Finally, the intended value of *sumArr* is recomputed and returned by the `RTN` instruction (by copying the *recArr* to the destination of the RCMP instruction). Then, the control flow jumps back to the next instruction following `RCMP` in Figure 3(c). The instructions contained in boxes in Figure 3(c) are extra instructions to be added into the binary to facilitate VRC.

D. Limitations & Side Effects

Overhead: Latency or energy per recomputing instruction in a slice is not any different than the non-recomputing, conventional counterparts. The only difference is that ISER executes these instructions using a dedicated instruction supply rather than the instruction cache.

Coverage: We cannot guarantee that all speculative loads missing in L1 have a corresponding slice. This may be due to complex producer-consumer chains, which cannot be expressed by a chain of arithmetic/logic instructions only, and/or slice inputs that cannot be guaranteed to be available during recomputation. Furthermore, some values are not produced by the application and are impossible to recompute, such as I/O.

Locality: Any speculative load that misses in L1 and gets replaced with recomputation would never reach the memory hierarchy. As a result, subsequent memory requests to the same cache block become more likely to miss in the cache hierarchy, as well. This adverse effect can easily degrade performance, but recomputation targeting such new misses may be able to recover some of the lost performance. We will discuss this effect further in the evaluation (Section V).

Exception Handling (during Recomputation): Exception handling during recomputation should be rare as it simply re-executes a previously seen slice of instructions with equivalent inputs. However, in case an exception would be raised we revert back to the Delay-on-Miss alternative and simply wait until all the shadows have been lifted (no longer speculative) and execute the load as normal.

Pipeline Integration: The only negative impact may be due to potential increase in the pressure on execution units, as execution units are shared with the rest of the instructions. However, recomputing instructions along a slice (which form a dependency chain) are executed sequentially, one at a time. The impact would, therefore, be one additional instruction competing for the respective functional unit at a time. This can also be regarded as an opportunity to utilize the cycles (and functional units) that could be wasted otherwise due to stalled instructions waiting on delayed loads.

E. Architectural Support for Slice Coherence

ISER is based on the premise that we do not have to validate recomputed values: *VRC is not a speculation* (i.e., it is not a prediction). This is certainly the case for *immutable* values that we can safely recompute instead of fetching them from the memory hierarchy. As long as the compiler guarantees via alias analysis that recomputed loads access *immutable* values (from the time they were written by the corresponding producer), the approach is compatible with any consistency model and coherence protocol, simply because neither is needed to ensure correctness. We evaluate this case which, however, restricts VRC coverage and limits the potential gains.

Here, we sketch one approach on how to increase coverage by relaxing the restrictions on slice formation but the actual mechanisms are beyond the scope of this paper. Our aim is to show that there is significant untapped potential in this direction. In the evaluation we show the upper bound for such a potential approach with an oracle model.

The central question is what happens if it is not possible to statically ascertain the immutability of a load's value. In other words, what happens for recomputed values that are considered as immutable but there is a possibility, however small, that they can change by some unknown store. We refer to such values as *mostly-immutable*.[2]

For mostly-immutable values, we still want to maintain the essential property for our purposes, that VRC is not a prediction that needs to be validated. Instead, what we want is to be able to make a simple binary decision: to recompute (if the value has not changed) or not (if the value has changed). In other words, we never validate VRC, but we expect that a store would *prevent* future recomputation of loads that access the same address. This implies that we must *track* any possible change of the data that could be accessed by recomputed loads.

For single-threaded applications, handling the recomputation of mostly-immutable values, implies a mechanism to match the thread's *own* stores to the recomputed loads and invalidate the corresponding VRC slices when such matches are found.[3] To enable such a mechanism, the target address of the producer instruction is saved as a tag for the corresponding slice in the ISER structures. This tag can be matched by future stores on the same address, to invalidate the slice (and cancel recomputation) by invalidating, selectively or in bulk, ISER structures. Since we expect this to be a rare occurrence (for what we choose to recompute), we can optimize for the case when it does not happen: Producer tags (store target addresses) can be encoded in signatures (Bloom filters) and if a future store hits in a signature, ISER structures and signatures are reset in bulk and need to be repopulated anew.

For multithreaded-applications, this matching and invalidation of recomputation slices should be expanded to include stores from other threads besides the thread's own stores. This requires an additional "coherence" mechanism to detect remote writes even when there is no copy of the relevant cacheline in the local cache. A solution can be based on an approach that serves a similar purpose: detecting remote writes in the absence of cached copies.

Specifically, the *Callback* concept, introduced by Ros and Kaxiras [30] can serve as the substrate on which to build a solution. A callback simply says "notify me if someone writes on this address" and it does not need cached copies that invite invalidations. Callback was introduced for synchronization, as an explicit *request* for an invalidation in the absence of coherence invalidations (or more broadly absence of sharing). Callback can be generalized to perform a similar role in our situation with regards to detecting changes on what we would otherwise consider immutable values. Similarly to the single-threaded case, tracked addresses can be encoded in signatures for efficient matching. Security implications of using callbacks (such as perhaps new side-channels enabled by the callback directories [31]) must also be addressed in the same way as in the work of Yan et al., SecDir [47].

[2]Naturally, we are not targeting *mutable* values as successful VRC would likely be much *less* prevalent.

[3]We assume, for the single-threaded case, that we would not recompute loads that touch I/O space that can be changed by a device without seeing any of our own stores modifying that space.

To conclude, we argue that VRC slices can be made *coherent* by explicitly detecting changes to what we would consider immutable values. Techniques for explicitly detecting writes without invalidations have been proposed in prior work [30], [31] and their adaptation to our purposes is feasible.

F. Impact on Consistency

While the coherence approaches sketched above enable us to *explicitly detect* changes in mostly-immutable values and *invalidate* the corresponding VRC slice, here, we discuss the order that this would need to happen in relation to the consistency model of the baseline architecture. We use total store order (TSO) and release consistency (RC) as our prime examples but our reasoning can be generalized to other consistency models. We use the term *callback invalidation* to distinguish from the normal coherence invalidation, which may not be available when we have no cached copy of the corresponding data. The question here is, once a change is detected to a value that we are capable of recomputing, when exactly is VRC canceled?

If VRC occurs well in advance of the callback invalidation it is safe in any consistency model such as TSO or RC. By "well in advance" we mean that the recomputed load is retired from the reorder buffer. In this case, it is as if the corresponding load has seen the *old* value, well in advance of the change in the value. Once the callback invalidation reaches the core, there will be no further VRC of that load. Thus, we only need to clarify what happens when a callback invalidation and the corresponding VRC occur in a critical window when consistency rules could be violated.

In RC, VRC is safe between memory fences. (RC, allows both loads and stores to be reordered, unless otherwise enforced by memory fences.) Callback invalidations received before an acquire memory fence must take hold and cancel VRC before crossing the fence.

In TSO, load–load reordering is not allowed to be observed. A recomputed load is considered *performed* as we consider it equivalent to accessing the actual data. In a *speculative* implementation of TSO, a recomputed load would be speculative with respect to an older load that is not performed. In other words, a recomputed load can be in the M-Shadow of one or more older loads. A callback invalidation reaching the core while a recomputed load is still under an M-Shadow (e.g., one or more older loads are still not-performed) should squash the recomputed load (and its dependents) and cancel further VRC.

To conclude, we argue that VRC is compatible with both TSO and RC by observing a correct ordering between callback invalidations and VRC.

G. Recomputation Security

ISER is based on slice formation, replacement of corresponding loads with `RCMP` instructions, and checkpointing of input operands with `REC` instructions. The question here is what happens if any part or all of the ISER infrastructure can be abused by an adversary. This is of course equivalent to hijacking the compiler, or dynamic instrumentation (or even the binary of an application where the same security risks would apply). However, even under such assumptions, *ISER still cannot leak information speculatively*, which is the main goal of our work.

To see this, assume that the compilers are compromised. Attackers can make them do anything they want. We are still safe with respect to leaking information via speculative side-channel attacks because of the following reasons:

1) VRC itself cannot be used to construct a speculative side-channel in the memory hierarchy because it does not perform any memory accesses at all.
2) VRC is only used if the load is already under a speculative shadow. Even if VRC recomputes a secret value, all future loads will be restricted under Delay-on-Miss.

To expand on (2), VRC only starts if the `RCMP` is under a speculative shadow. While VRC has access to input operands that may hold secrets, the recomputation slice cannot perform any memory accesses to leak those secrets and the only way would be to pass the secret value to another (younger) load, which *will also be speculative.* Speculative interference attacks [5] that use older bound-to-commit instructions to leak information are prevented in our system by enforcing that non-piplined instructions are executed strictly in order [36]. Delay-on-Miss guarantees that the younger load cannot have any visible side-effects, preventing any information leakage. Essentially, VRC maintains the Delay-on-Miss invariant that only non-speculative loads are allowed to cause side-effects in the memory hierarchy. Therefore, we conclude that VRC is *safe* from *cache and memory speculative side-channel attacks*, no matter how compromised the compiler, dynamic instrumentation, or the binary is.

In addition, the VRC structures are local to the core and cannot be observed by another core. While under speculation, the only changes allowed are ones that cannot be observed from the outside, such as writes to free physical registers that can only be read by instructions belonging to the recompuation slice. Any other changes (e.g., to the IBuff and Hist) are buffered or squashed, i.e., they are only updated once the instruction causing the change is no longer speculative. Furthermore, if SMT is present, the VRC structures can be partitioned where necessary, to avoid contention attacks between SMT threads. It should be mentioned that if SMT is present, since the slices use the functional units (FUs) of the core, it is possible to perform an FU-contention attack, such as SmotherSpectre [7]. Such attacks are outside the scope of Delay-on-Miss and this work but, more importantly, they are only possible under VRC if they are already possible without it, as VRC's slices only consist of instructions already found in the application. Thus, *VRC does not open up any new attack opportunities under our current threat model.* Note also that, disabling SMT has been recommended by vendors (e.g., Microsoft [42]) as a measure against several attacks.

TABLE I: The simulated system parameters.

Parameter	Value
Technology node	22 nm
Processor type	out-of-order x86 CPU
Processor frequency	3.4 GHz
Issue / Execute / Commit width	8
Cache line size	64 bytes
L1 private cache size	32 KiB, 8-way
L1 access latency	2 cycles
L2 shared cache size	1 MiB, 16-way
L2 access latency	20 cycles
Value predictor	VTAGE
Value predictor size	13 comp.s $\times$ 128 entries

IV. Evaluation Setup

We use a Pin-based tool [28] to identify and annotate recomputation slices. For practical reasons, we limit the maximum slice size during construction to 100 instructions (which represents a loose upper bound in practice). The annotated slices, together with the original binary, are fed to the gem5 [8] simulator where the shadows, Delay-on-Miss, and VP have been implemented as described in the Delay-on-Miss work by Sakalis et al. [35]. In gem5, we begin with fast-forwarding through the first one billion instructions of the application and then simulate in detail for another billion. We use McPAT [24] with CACTI [25], as well as the dynamic DRAM energy provided by gem5, to calculate the energy breakdown of the system. The configuration used for simulations are shown in Table I. We evaluate the following versions:

Baseline: An unsecured out-of-order CPU.
DoM: Delay-on-Miss without any value prediction or recomputation. This is considered as the *secure* baseline.
VP: DoM with an added VTAGE value predictor.
VRC: DoM with the added value recomputation. This is the solution we are proposing. This does not include callbacks, only immutable values are recomputed.

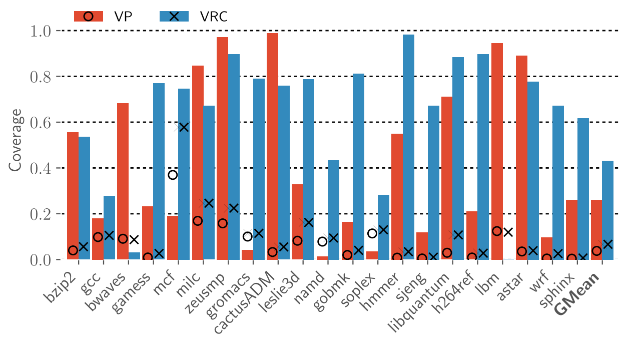

Fig. 4: The coverage of VP and VRC, i.e., the ratio of shadowed L1 misses that can be predicted or recomputed instead of being delayed (bars). Also depicted on the same plot is the L1 miss ratio for both versions (circles/crosses).

VRC (2 cycles): Same as the VRC version but we have *artificially* limited the latency of every slice to at most two cycles. We have also limited the number of instructions needed for the recomputation accordingly. As all VP versions take at most 2-cycles per prediction in our implementation, this VRC version enables iso-performance comparison with VP variants.

Oracle VP: Same as the VP version but with an oracle predictor capable of predicting correctly 100% of all speculative L1 misses. Even though the predictor is perfect, its results are still being validated once the loads have been unshadowed.

Oracle VRC: Same as the VRC (2 cycles) version but with an oracle compiler capable of recomputing 100% of all speculative L1 misses. Note that this is the Oracle in regards to VRC coverage, not performance. We discuss the implications of re-computing all speculative L1 misses in the evaluation, section V.

For the sake of brevity, the last three versions are only shown in the performance (IPC) results and are excluded from the rest of the figures.

We evaluate all these versions using the SPEC2006 benchmark suite [41], with the reference inputs, as in previous work [35]. For one of the benchmarks, `GemsFDTD`, none of the techniques we tried produced any improvement. `GemsFDTD` is a floating point benchmark that is dominated by overlapping C-Shadows. It achieves only about 20% of the baseline performance with DoM (also corroborated by Sakalis et al. [35]). In our work, we were unable to achieve any improvement with either VP or VRC because of near-zero coverage. In contrast, it shows an impressive 3.5× (350%) improvement with an oracle VRC (100% coverage)—however, this may be impractical to attain. Energy results follow the same pattern, either showing high energy consumption (3× of the baseline) with all the techniques we tried or 56% lower than the baseline with the VRC oracle. We surmise that `GemsFDTD` performs badly, in general, under any "delay" technique (including NDA [45] and STT [50], but it is unfortunately not included in these works to allow for comparisons). Because `GemsFDTD` represents such a special case for delay techniques we believe that further work is required to specifically address its shortcomings. For these reasons, we point out its idiosyncrasy here, instead of discussing it with the rest of the benchmarks.

V. Evaluation

A. Recomputation Coverage

The coverage for the VRC can be seen in Figure 4, together with the value prediction (VP) coverage. We can immediately observe that, on average, VRC has higher coverage than VP, at 43% of all speculative L1 misses vs. 26% with the VP. A notable example is `mcf`, which is one of the worst performing benchmarks with DoM (Section V-B). On the other hand, `lbm` is a counter-example, where we have almost zero VRC coverage. This, however, does not affect the performance negatively, as `lbm` does not suffer from any performance penalties even with the plain DoM.

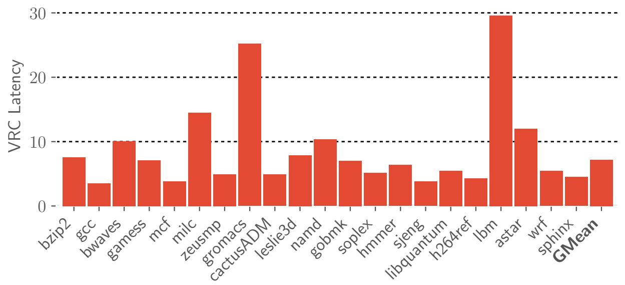

Fig. 5: The mean latency for recomputing a shadowed L1 miss.

In the same figure, we have also superimposed the cache miss ratio for both versions. We only predict or recompute L1 misses, so the miss ratio is needed in conjunction with the coverage to infer the percentage of loads in the application that are being predicted or recomputed. More detailed L1D miss data can be found in Figure 6. Note how, as discussed in Section III-D, VRC increases the miss ratio.

With VP, all loads that can be predicted are predicted in the same amount of time (two cycles in our setup), but the same is not true for the VRC, where the latency depends on the slice length and the instructions it contains. In Figure 5 we can see the mean recomputation latency for each benchmark, as well as the overall mean. In all cases, VRC requires more cycles than VP to recompute a value, with a mean of seven cycles per slice. However, as we will see in Section V-B, this does not impact the performance significantly.

B. Performance

Figure 7 contains the number of committed instructions per cycle, normalized to the unsecured baseline processor. Delay-on-Miss without VP or VRC, which is our *secure* baseline, performs at 88% of the unsecured baseline, similar to the results reported by Sakalis et al. [35]. The benchmarks that incur the biggest hit in performance are `mcf` (at 44% of the baseline), followed by `milc` (68%), `cactusADM` (74%) and `libquantum` (76%). Out of these benchmarks, three (`mcf`, `milc`, and `libquantum`) have high LLC MPKI, but that in itself is not the only factor, as other benchmarks (e.g., `lbm`) also have a high MPKI. Instead, the cost of Delay-on-Miss also depends on the amount of MLP that the benchmarks exhibit; the more MLP that is taken advantage of in the baseline, the higher the performance loss.

If VP is introduced, then the performance is similar, at 89% of the unsecured baseline. This result contradicts the results given by Sakalis et al. [35], where the VP gives a significant performance advantage[4]. The reason that VP does not offer a significant advantage is because VP itself is speculative: When a value is predicted it still needs to be validated at a later point. By predicting the value, a small amount of parallelism (ILP) can be exploited during execution, but the slow L1 misses still need to be satisfied for the validation. Due to the high number of speculative shadows, validations become serialized and are not able to take advantage of any MLP that might be found in the application. In essence, the VP pushes the cost of delaying speculative loads from the execution stage to the validation stage, but it does not eliminate it. This can be seen in the Oracle VP results, where even 100% prediction rate (i.e., all shadowed L1 misses are successfully predicted) only leads to a marginal performance improvement of one percentage point.

The same is not true for VRC, as once a value has been recomputed, it does not need to be validated, meaning that the cost for delaying a long latency miss is eliminated and no serialization is enforced. While VRC does not increase the amount of MLP that can be taken advantage of, it does eliminate some of the need for it. Overall, VRC performs

[4]We contacted the authors and verified that our results are indeed valid.

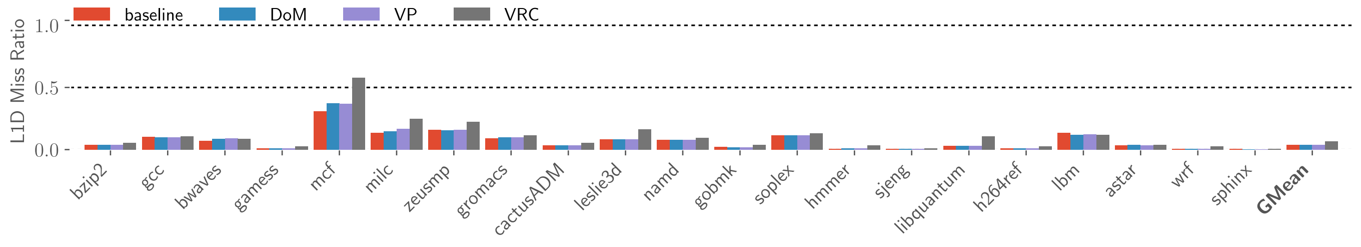

Fig. 6: L1D miss ratio for Delay-on-Miss with VR and VRC.

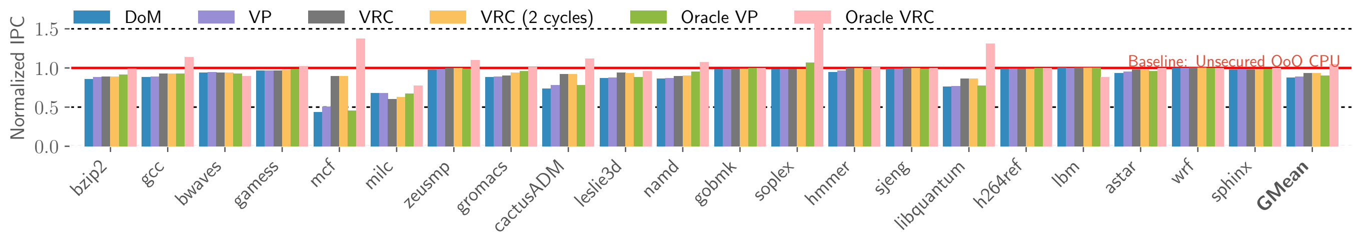

Fig. 7: Performance (IPC – higher is better) normalized to an unsecured OoO baseline.

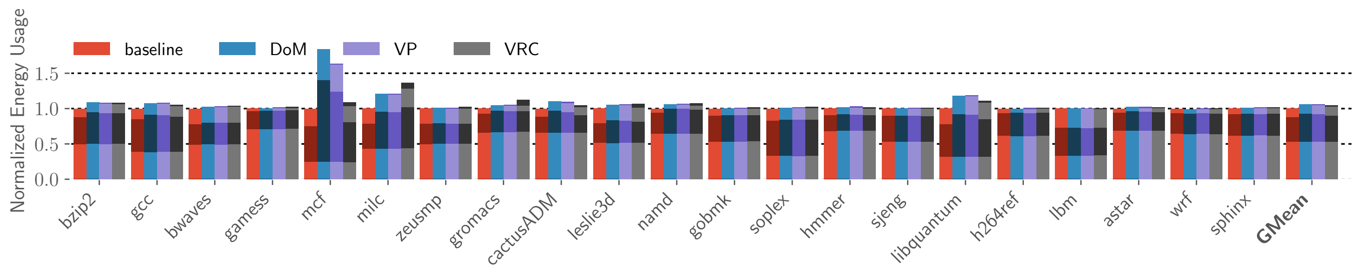

Fig. 8: Energy usage, where each bar consists of four parts (from bottom up): The bottom, light colored part is the dynamic energy of the CPU, the middle, dark colored one is the static energy of the CPU, the middle light part is the DRAM energy, including refresh and power-down energy, and the top dark part is the overhead of VP and VRC, both static and dynamic.

at 93% of the unsecured baseline, decreasing the performance cost of Delay-on-Miss by more than one third (specifically, by 42%). The benchmark with the most dramatic performance increase is `mcf`, which is the worst performing benchmark for Delay-on-Miss. VRC improves the performance from 44% to 90%, reducing the performance cost to one fifth of that of Delay-on-Miss.

We have also evaluated an artificial version of VRC where we keep the same slice coverage but reduce the cost of the slices to at most two cycles. This version exhibits almost identical performance to the real VRC, with a mean performance difference of half a percentage point. This strongly indicates that instead of trying to keep the cost of the slices low, it is more important to increase the coverage, even if large slices are required. This is further corroborated by the results from the Oracle version, discussed below. However, large slices increase the energy usage, as we will see in Section V-C, so a balance still needs to be kept.

If we introduce an Oracle VRC that can recompute all shadowed L1 misses, the difference between the VP and the VRC approaches becomes even more apparent. Both Oracle versions have 100% coverage and the same latency, the only difference is that with VP the loads need to be validated when they are unshadowed, while with VRC they are completed as soon as the value has been recomputed. While, as we have seen, the VP Oracle can only achieve marginal improvements over the non-Oracle version, the VRC Oracle is able to outperform even the baseline, including benchmarks such as `mcf`, `cactusADM`, and `libquantum`. Of course, such an Oracle is unrealistic, but it does support our argument that the limiting factor for VP is the cost of validation.

However, it is worth noting here that a 100%-coverage VRC does not necessarily guarantee that the performance will exceed that of the baseline. In fact, there are four benchmarks where the Oracle VRC is slower than the baseline: `bwaves`, `milc`, `leslie3d`, and `lbm`. Out of these, the `bwaves` and `lbm` VRC Oracle is also slower than DoM. There are various factors that contribute to this result: In `bwaves` and `leslie3d`, the L1 and the L2 miss ratio (not shown) are increased significantly with the Oracle; in `milc` the Oracle increases the number of write misses in the L1 (not shown), as well as the average write miss latency (not shown); finally, in `lbm` a combination of many factors contribute to worse cache performance. The problem is that, even with 100% coverage, not every single memory access is recomputed: Stores, non-speculative loads, and speculative L1 misses that hit in the MSHRs, are still served by the memory hierarchy. By recomputing the rest of the loads, which account for the majority of the L1 misses, the Oracle VRC disrupts the normal operation of the cache and the prefetcher, resulting in performance losses. Essentially, there is a trade-off between the benefits of eliminating long-latency L1 misses and the cost of disrupting the normal cache operation. For the majority of the benchmarks, this trade-off leans towards the benefits, but this is not true for all of the benchmarks. Future work aiming to increase VRC coverage must account for these factors to achieve optimal performance.

C. Energy

Energy, in our case, is affected by three main factors: The execution time/performance, the number of accesses in the memory hierarchy (especially the DRAM), and the cost of predicting (VP) or recomputing (VRC) a value. Figure 8 shows, starting from the bottom, the dynamic (bottom, light color) and static (middle, dark color) energy of the CPU, the total DRAM energy (middle, light color), and, finally, the overhead (if any) for VP and VRC (top, dark color). Overall Delay-on-Miss and VP increase the mean energy usage over the unsecured baseline by 6%, while VRC increases it by 5%. The dynamic energy of the CPU (excluding the overheads) remains mostly the same across all versions, instead it is the static, DRAM, and overhead energy that changes.

Static energy is affected because the execution time is affected. This is most obvious in `mcf`, the application with the worst DoM performance, followed by `milc`. None of the evaluated solutions affect the LLC MPKI significantly (not shown), so the increase in the DRAM energy is not due to an increase in the number of accesses but due to other operations such as refresh and power-down states. These operations do depend on the access patterns, but they also depend on the execution time, similar to the static energy usage of the system.

On the other hand, the overheads introduced by the VP and the VRC are affected both by the execution time (static energy) and by the operations performed. This is particularly visible in the case of the VRC, where the majority of the overhead is due to the instructions of the slices. As we have discussed in Section V-B, smaller slices do not lead to better performance, but the same is not true for the energy costs. Instead, a balance between coverage (which increases the performance) and slice length (which increases the energy usage) needs to be achieved.

Out of all the benchmarks, the ones with the highest (relative to the baseline) energy usage are `milc` (at 37% over the baseline), `gromacs` (13%) and `libquantum` (12%). The rest of the benchmarks have energy overheads of less than 10% over the baseline. `milc` is the benchmark with the worse performance, so part of the energy increase is due to static and DRAM energy. It also has a high VRC coverage and also some of the third most expensive (in cycles, on average) slices among all the benchmarks, which increases the VRC overhead energy. On the other hand, `gromacs`'s performance comes very close to the baseline, but it has the second most expensive slices, while also having high coverage. Finally, `libquantum` also sees an increase in execution time and by extension, energy usage. The next benchmark with the higher energy increase over the baseline is `mcf` (9%), but this is far better than DoM, with or without VP, which are at 63% and 84% respectively.

D. Hardware/Software Overhead

Thus far, related security proposals exert a toll on performance and/or increase cost/complexity. In ISER, as well, microarchitectural support for VRC increases hardware complexity, but only slightly: Slices differ in length, but here we conservatively assume that all would be as long as the maximum-length slice we observe across all benchmarks. In this case, 22 KiB suffices to accommodate all "live" slices for the complete execution of the largest benchmark. In practice, Hist would only have to store a fraction of this as the slices would be much shorter on average and all values for a program do not have to be stored at the same time. Furthermore, static loads that need to be recomputed at runtime are few, so the overhead in the binary is small; $\leq$3% across all applications. Finally, as we pointed out throughout the evaluation, since our conservative VRC implementation leaves many optimization opportunities untapped, it still has potential for even further improvement.

VI. Related Work

The architecture community promptly proposed a number of techniques (starting with *InvisiSpec* [46]) to prevent *disclosure gadgets* from revealing secrets. The techniques fall in one of the following three broad categories shown below, but each individual proposal has different assumptions as to the threat model (type of speculative shadows covered) and prevention of information leakage (disclosure gadgets). It is obvious that at this point no direct comparison is possible, but we make an effort to compare the solutions qualitatively.

Hide&Replay: Perform speculative memory accesses in a manner that does not perturb any μ–architectural state in the memory system; subsequently, perform a replay of the access (when it becomes non-speculative) to affect the correct changes in the μ-architectural state [1], [18], [23], [34], [46]. *Invisispec* (Yan et al.) [46] and *Ghost loads* (Sakalis et al.) [34] were the first such proposals. Hide&Replay techniques, as the first to be proposed, showed a significant cost in performance (and a moderate implementation cost). They only protect against information leaks via the memory hierarchy (and not even all of it, as DRAM leaks are possible [29]). On the other hand, both of these techniques were designed to protect against attacks on any possible speculation primitive, i.e., cover all the speculative shadows mentioned above. A recent work, InvarSpec [51] relies on compile time analysis to identify instructions that may become non-speculative during execution (i.e., speculation invariant). The protection scheme used for these speculative instructions can be lifted at runtime, thus, reducing the performance overhead associated with speculation-related protection mechanisms in hardware. Reported performance improvement from such HW-SW co-design, however, cannot reach the negative overhead of ISER. Instead, as it takes an orthogonal approach, InvarSpec can be used in conjunction with ISER to further improve the performance while also reducing the size of the structures needed for recomputation, by reducing the number of loads that trigger recomputation.

Delay: Delaying speculative changes in μ-architectural state until execution is non-speculative. Sakalis et al. proposed to delay loads that miss in the L1 (*Delay-on-Miss*) until they are non-speculative [33], [35]. This delays any μ-state change in the memory hierarchy. A different form of delay (such as *NDA*, proposed by Weisse et al. [45]), is to prevent speculative data propagation by delaying *dependent instructions* from executing with speculative inputs [4], [13], [37], [43], [45], [50]. Delay-on-Miss protects against *all* speculative shadows (i.e., any possible "Speculation Primitive") but delays only changes in the memory hierarchy (including DRAM). Subsequent work, that delays speculative propagation of data [45], achieves good performance by protecting against any μ-state changes (i.e., a much larger gamut of "disclosure gadgets" than just the memory hierarchy) but responding only to C-Shadows, i.e., control speculation primitives. Another similar alternative, STT [50], also protects against other shadows (referred to as the "Futuristic" model) but at a higher performance cost. In a recent publication, STT has been extended to utilize speculation as well, referred to as "speculative data-oblivious speculation–SDO" [49], in order to replace the potentially leaky speculative paths with secure, data-independent paths. This approach is similar to the approach that ISER takes, only ISER is non-speculative and does not require any verification or squashing, further reducing the runtime overhead. Tran et al., propose a SW-HW extension that can reduce the time in which loads are shadowed (i.e., loads are speculative) and thereby can increase the MLP [44]. Their proposal includes instruction reordering to prioritize calculations that minimize the speculation window, such as target address computation of memory accesses, and resolution of branch conditions. Much like InvarSpec, their approach may reduce the performance overhead of delay-based security solutions by reducing the number of speculative loads or time spent in speculations, and it is orthogonal to our proposal. Both approaches can be combined together to offer better security coverage with minimum performance overhead. SPECCFI [22] uses the Control-Flow Integrity (CFI) to prevent Spectre-type attacks that abuse illegal control flow during speculative execution. Not all possible speculative side-channel attacks are covered by this technique but, much like the other compiler-based techniques we have discussed, it can be used in conjunction with our technique to limit the cases where recomputation is needed.

Cleanup: Perform a speculative change in μ-architectural state but then *undo* if speculation is squashed. In the first such proposal, *CleanupSpec*, by Saileshwar et al. [32], the undo is expensive so its application is restricted to the L1 cache. The rest of the memory hierarchy (L2, LLC, and coherence directory) is assumed to be protected in other ways, including randomization and delaying of coherence state changes, but DRAM row buffers still remain a security hole. Cleanup techniques only protect the L1, assuming—at a cost—that the rest of the hierarchy (excluding DRAM) is protected otherwise [32].

(Generic) Recomputation: Amnesiac [3] introduces a μ-architecture for recomputation that differs from ISER in the way slices are generated and their usage. The goal of Amnesiac is to replace as many energy-hungry loads as possible with recomputations of the respective data value. In contrast, ISER recomputes slices selectively, such that recomputation is triggered only for shadowed loads that miss in L1.

Kandemir et al. proposed a recomputation-based approach to reduce off-chip memory space in embedded processors [17]. Koc et al. investigated how recomputation of data residing in memory banks in low-power states can reduce the energy consumption [20], and devised compiler optimizations for scratchpads [19] that are limited to array variables. The dual of recomputation, *memoization* [14], [39] replaces computation with table look-ups for pre-computed values (for the ones that are frequent and expensive to recompute). Memoization can mitigate the communication overhead – as long as table look-ups are cheaper than long-distance data retrieval, but is only effective if the respective computations exhibit significant value locality. Therefore, memoization and recomputation can complement each other in boosting energy efficiency. *Idempotent Processors* [11] execute programs as a sequence of compiler-constructed idempotent (i.e., re-executable without any side effects) code regions. As the name suggests, idempotent regions regenerate the same output regardless of how many times they are executed with the given program state. Generally, idempotent regions are larger, and therefore tend to incur higher overhead for recomputation, while slices for VRC employ fine-grain data recomputation, where each slice contains only the necessary instructions to generate a value. Accordingly, slices for VRC may provide more flexibility than idempotent regions.

Elnawawy et al. demonstrated the applicability of recomputation to loop-based code [12] to reduce checkpointing overheads. In their proposal, a whole loop is (re)executed during recovery, where only the initial state of the loop is required to be checkpointed. The loops may contain extra computations that are not relevant to the production of the value to be recovered. Compared to such a coarse-grain recomputation, slice-based recomputation does not contain any irrelevant instructions. Also, slices used for VRC do not contain load instructions, as opposed to [12]; and recomputation applies outside of loops, providing wider applicability. To summarize, although value recomputation has been explored in different contexts before, to the best of our knowledge, none of the prior works has evaluated recomputation in the context of security.

Slice Generation: Automatic creation of VRC slices in hardware is complicated because we are not after the slice of the load to be replaced (which could be created by existing techniques like IBDA [9]) but the slice of the corresponding store creating the value. This would require tracking of all stores and their slices, and somehow matching these with a (speculative) load missing in the L1 cache. Srinivasan et al. [40] generate "forward" slices for loads that miss in LLC. This is easier in hardware since the dependency tracking starts with the producer (i.e., load that misses in LLC) and the consumers (following use-def chains) are executed after the producer. However, in our case, we have to identify "backward" slices – i.e., the producers, not consumers, of a value that will be loaded – where all the producers were executed before the load itself. Such backward dependency tracking would likely require expensive bookkeeping in hardware.

VII. Conclusion

Delay techniques aim to hide the effects of transient execution by simply delaying instructions until they become non-speculative. Whether delaying loads that miss in the L1, as Delay-on-Miss does, or delaying the propagation of speculative data to dependent instructions, as NDA and STT do, delay techniques extract a heavy toll in performance, in direct relation to the set of speculative shadows they protect against. Delay techniques would be at an impasse with respect to improvement if we could not regain some of this lost performance in some other way. To this end, value prediction, invisible from the outside, was initially proposed as a solution.

However, value prediction (VP) is not the right abstraction for recovering lost performance in Delay-on-miss. This is not because of coverage or accuracy but because value prediction is just another form of speculation that needs to be validated. Validation limits the potential benefits to the point where even an oracle VP (100% coverage and accuracy) does not do any better than a practical VP. In our evaluation we found that, no matter how good, VP is limited to just one percentage point improvement over Delay-on-miss.

Instead, we propose another, *non-speculative*, abstraction to regain performance for delay techniques, and in particular for Delay-on-miss. We propose to use recomputation that yields correct values—not predictions—as the key to overcome Delay-on-miss performance limitations. We describe the architecture, we evaluate it using a practical approach to generate recomputation slices albeit with modest coverage, and we exceed the performance of Oracle VP (90% vs. 93%) with lower energy usage. Finally, we discuss the potential for increasing the coverage of recomputation with future architectural support. Because, as we show, oracle recomputation easily exceeds even the performance of the unmodified (unsecured) baseline, this direction provides tangible motivation for researching techniques for a future secure processor.

To regain the performance cost of securing the memory hierarchy, we need to identify methods that improve the MLP. This paper demonstrates, for the first time, value recomputation's unique ability in overcoming the MLP restriction that is inherent in VP when applied on the Delay-on-Miss technique. To the best of our knowledge, *no previous study on recomputation considered any security impact*. Finally, these findings should be considered in the context of our representative threat model (Section II-E). In the end, no threat model can cover all possible security vulnerabilities. But, as explained in Section III-G, ISER does not introduce any new attack opportunities under the provided threat model.

That said, as any technique that affects the control flow timing – including value prediction or even Delay-on-Miss to name a few – recomputation may give rise to timing channels, where information to be leaked gets encoded in timing differences between various microarchitectural events. Even if a given value is recomputed multiple times throughout execution, as resource contention and speculation can easily change timing of microarchitectural events non-deterministically, there is also a very good chance that recomputation rather *obfuscates* control flow timing. Specifically, provided that a slice is executed only upon an associated L1 miss, which by itself constitutes a variable latency depending on where in the memory hierarchy the data resides. This time may cover part of or the whole recomputation time, making it indistinguishable with the timing of the memory operation.

To conclude, potential timing channels, if at all, would not necessarily be straight-forward to exploit. In fact, recomputation is more likely to result in control flow obfuscation. We leave the exploration of such effects to future work, confining the analysis in this paper to only memory side-channels because they are easier to exploit and can be exploited across cores. This does not imply that side-channels such as functional unit contention based ones are not possible, they are just outside the scope of our threat model.

Acknowledgments

This project was funded by the Swedish Research Council grants 2015-05159 and 2018-05254, by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53, and by MRL grant 2021-020, and by NSF CAREER CCF-1553042. The simulations were performed on the IDUN cluster [38], provided by NTNU.

References

[1] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[2] I. Akturk and U. R. Karpuzcu, "Trading computation for communication: A taxonomy of data recomputation techniques," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[3] I. Akturk and U. R. Karpuzcu, "AMNESIAC: Amnesic Automatic Computer - Trading Computation for Communication for Energy Efficiency," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[4] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding speculative data from microarchitectural covert channels," in *Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 151–164.

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

[6] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, p. 785–800.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[9] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[10] N. V. Database, "CVE-2018-3693." Available from MITRE, CVE-ID CVE-2018-3693., Dec. 28 2017. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3693

[11] M. de Kruijf and K. Sankaralingam, "Idempotent Processor Architecture," in *International Symposium on Microarchitecture (MICRO)*, December 2011.

[12] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2017.

[13] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," in *The Design Automation Conference (DAC)*, 2019, pp. 1–6.

[14] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[15] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," *Keynote at International Conference on Solid State Circuits (ISSCC)*, April 2014.

[16] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ASIA Conference on Computer and Communications Security (ASIACCS)*, 2016, pp. 353–364.

[17] M. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk, "Studying Storage-Recomputation Tradeoffs in Memory-Constrained Embedded Processing," in *Design, Automation and Test in Europe (DATE)*, 2005.

[18] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *ACM/IEEE Design Automation Conference (DAC)*, Jun. 2019, pp. 1–6.

[19] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk, "Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-processors," in *The Design Automation Conference (DAC)*, 2007.

[20] H. Koc, O. Ozturk, M. Kandemir, and E. Ercanli, "Minimizing Energy Consumption of Banked Memories Using Data Recomputation," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2006.

[21] 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," in *IEEE Symposium on Security and Privacy (SP)*, May 2019.

[22] E. M. Koruyeh, S. Haji Amin Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SpecCFI: Mitigating spectre attacks using CFI informed speculation," in *IEEE Symposium on Security and Privacy (SSP)*, 2020, pp. 39–53.

[23] 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 *International Symposium High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 264–276.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[25] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *International Conference On Computer Aided Design (ICCAD)*, 2011, pp. 694–701.

[26] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," Jan. 2018. [Online]. Available: http://arxiv.org/abs/1801.01207

[27] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (SP)*, May 2015, pp. 605–622.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[29] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security Symposium*, 2016, pp. 565–581.

[30] A. Ros and S. Kaxiras, "Callback: Efficient synchronization without invalidation with a directory just for spin-waiting," in *International Symposium on Computer Architecture (ISCA)*, 2015, pp. 427–438.

[31] A. Ros and S. Kaxiras, "Racer: TSO consistency via race detection," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016, p. 33.

[32] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An undo approach to safe speculation," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 73–86.

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

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

[35] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *International Symposium on Computer Architecture (ISCA)*, 2019, pp. 723–735.

[36] C. Sakalis, M. Själander, and S. Kaxiras, "Preventing priority inversion in instruction scheduling to disrupt speculative interference," in *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[37] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "ConTExT: Leakage-Free Transient Execution," *arXiv:1905.09100 [cs]*, May 2019, arXiv: 1905.09100. [Online]. Available: http://arxiv.org/abs/1905.09100

[38] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure," *arXiv:1912.05848 [cs.DC]*, 2020. [Online]. Available: https://arxiv.org/abs/1912.05848

[39] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *International Symposium on Computer Architecture (ISCA)*, 1997.

[40] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines: achieving resource-efficient latency tolerance," *IEEE Micro*, vol. 24, no. 6, 2004.

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

[42] M. Support, "Windows guidance to protect against speculative execution side-channel vulnerabilities," Nov.12 2019. [Online]. Available: https://support.microsoft.com/en-us/help/4457951/windows-guidance-to-protect-against-speculative-execution-side-channel

[43] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 395–410.

[44] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, "Clearing the shadows: Recovering lost performance for invisible speculative execution through hw/sw co-design," in *Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020, p. 241–254.

[45] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 572–586.

[46] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 428–441.

[47] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2019, pp. 332–345.

[48] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*. USENIX Association, 2014, pp. 719–732.

[49] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[50] 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 *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 954–968.

[51] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas, "Speculation invariance (invarspec): Faster safe execution through program analysis," in *International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1138–1152.