BRB: Mitigating Branch Predictor Side-Channels.

Ilias Vougioukas†, Nikos Nikoleris
Andreas Sandberg, Stephan Diestelhorst
Arm Research
firstname.lastname@arm.com

Bashir M. Al-Hashimi, Geoff V. Merrett
University of Southampton†
bmah@ecs.soton.ac.uk, gvm@ecs.soton.ac.uk

***Abstract*—Modern processors use branch prediction as an optimization to improve processor performance. Predictors have become larger and increasingly more sophisticated in order to achieve higher accuracies which are needed in high performance cores. However, branch prediction can also be a source of side channel exploits, as one context can deliberately change the branch predictor state and alter the instruction flow of another context. Current mitigation techniques either sacrifice performance for security, or fail to guarantee isolation when retaining the accuracy. Achieving both has proven to be challenging.**

In this work we address this by, (1) introducing the notions of steady-state and transient branch predictor accuracy, and (2) showing that current predictors increase their misprediction rate by as much as 90% on average when forced to flush branch prediction state to remain secure. To solve this, (3) we introduce the *branch retention buffer*, a novel mechanism that partitions only the most useful branch predictor components to isolate separate contexts. Our mechanism makes thread isolation practical, as it stops the predictor from executing cold with little if any added area and no warm-up overheads. At the same time our results show that, compared to the state-of-the-art, average misprediction rates are reduced by 15-20% without increasing area, leading to a 2% performance increase.

***Keywords*-Branch prediction, branch retention buffer, TAGE, side-channel attacks, Perceptron, microarchitecture, context switching.**

I. Introduction

Branch prediction contributes to high performance in modern processors with deep pipelines by enabling accurate speculation. Since the inception of the idea of speculative execution, the improvement has gradually increased overall processor performance, as *Branch Predictor* (BP) designs have steadily become more sophisticated and more complex.

Initially, accuracy was the dominant factor for predictor design however, as power became a limiting factor, designs had to take into account more constraints. Today accuracy is still the primary goal, but power and area are also critical.

Systems in general have become significantly more complex today, featuring multiple types of cores and accelerators on a single die. Software, taking advantage of the aforementioned improvements, has also changed enabling more applications to be handled simultaneously. Applications today are often multi-threaded and systems *context switch* (CS) frequently between processes.

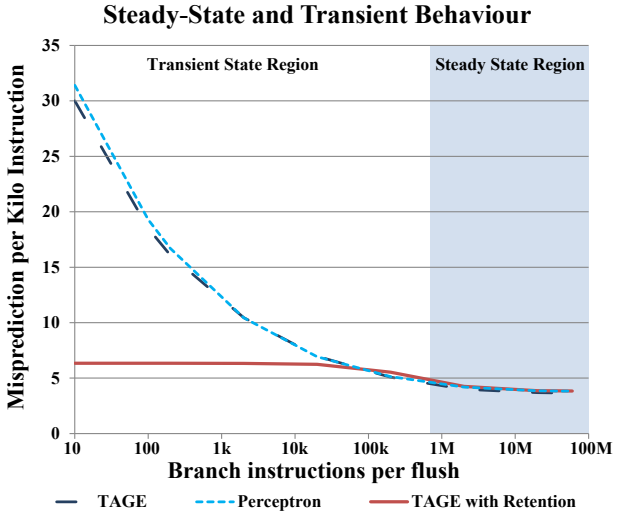


Figure 1: Comparison of current designs with our predictor that preserves a minimal state between flushes. These flushes can be triggered to avoid side-channel attacks that use the branch predictor.

However, recently discovered vulnerabilities in micro-architecture permit side-channel attacks that manipulate the branch predictor [1]. For many of these exploits context switching is a critical enabler, which are hard to overcome.

Using narrow, "hot" fixes for separate security issues that use side-channel attacks does not guarantee that systems are safe from newer variations, whereas scrubbing the entire branch predictor state as a "heavy" approach can have a significant impact on performance. Addressing vulnerabilities in software after discovery is often not possible and usually even more costly in terms of performance.

The above observations motivate us to design future systems that take into account one additional design constraint, that of being able to securely speculate without allowing information to leak. Duplicating or partitioning BP state to satisfy isolation is costly in terms of area. Flushing, on the other hand, inevitably causes frequent operation disruption of the branch predictor and inhibits effective warm-up. This can be described by distinguishing between *steady-state* and *transient* predictor accuracy as shown in Figure 1.

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00058

To prevent the aforementioned branch predictor side-channels our proposal guarantees branch predictor state isolation efficiently, without degrading performance. In detail, our contributions are:

- We introduce the notion of *transient prediction accuracy*, its relevance to designing future side-channel free branch predictors, and show that it differs from the steady-state accuracy, with which designs are evaluated today.
- We show that current state-of-the-art branch predictors perform notably worse in transient state when flushed for security. We perform an in-depth analysis of the TAGE predictor[2], and show how large components do not contribute to the prediction accuracy under these new circumstances.
- To solve this, we propose the *Branch Retention Buffer* (BRB), a novel mechanism that reduces cold-start effects by preserving partial branch predictor state per context. Compared to state-of-the-art, our design achieves the same high accuracy at steady-state, improved transient accuracy and ensures branch predictor state isolation without increasing the overall area.

II. Background

Due to speculative execution, branch predictors are a critical part of modern core design, drastically increasing processor performance. However, mitigation techniques for recently discovered branch predictor side-channels notably degrade predictor accuracy and reduce overall system performance.

A. Speculative side-channels

Recent studies have shown that side-channel attacks leaking sensitive data are possible in most contemporary CPU designs [3, 1, 4, 5]. These exploits take advantage of hardware oversights at design time, leaving the system vulnerable to code that can cause information to leak outside its defined scope. While such attacks target various components like the caches and the DRAM, we focus on those that are based on exploiting vulnerabilities in branch prediction.

1) Branch predictor side-channels: Spectre [1] class attacks target branch predictor components in a variety of ways. We focus on one specific case (variant 2) that exploits conditional branch misprediction that allows malicious code (a gadget) to be executed. That code uses flush and reload type timing attacks on the caches to leak information[6, 7]. For this attack to be plausible, some knowledge of the micro-architectural behavior is needed to influence the predictor to misspeculate and execute the gadget. With that, an attacker can poison the predictor entries and guarantee the necessary misprediction. The attack can also be triggered in cases when the branch mispredicts without altering the branch predictor state, although this is significantly harder to orchestrate.

Similar to how Spectre v2 uses the branch predictor to leak information, BranchScope [4] uses a mechanism similar to how variant 2 of Spectre to target the *Branch Target Buffer* (BTB) to influence the *Pattern History Table* (PHT) of the directional branch predictor. In this attack scheme, the predictor is primed so that it is in a predefined state the attacker can control. From this starting point, the target code is triggered to execute the victim code, which will observably change the PHT state. A simple probe can then calculate the branch flow based on the primed branch predictor conditions. The technique has been shown to successfully leak information from a secure SGX enclave in Intel processors[4].

The potential scenarios to exploit such attacks are numerous. Any transition from one process to another or a process to/from the kernel can be a potential point of vulnerability, as any context switch or system call can be used to let malicious software "hijack" the branch predictor and leak information.

2) Mitigation techniques: On one hand, Software and firmware mitigation techniques are often hard to implement and induce high overheads. Recent studies [8] measure the actual performance loss for Spectre and Meltdown vulnerabilities and find that it ranges from 15% to 90%. While some mitigations for the known exploits have been deployed, potential unknown exploits can still find ways to exploit shared components such as the caches, the branch prediction logic, and the translation tables as the fixes are topical and do not guarantee security. Furthermore, such vulnerabilities (i.e. Spectre v1, Branchscope) cannot be addressed in software or microcode and require hardware redesign.

On the other hand, hardware isolation is a sufficient measure to ensure no leaks occur, but often comes at the expense of performance. Cache side-channels are extremely difficult to address without significant performance loss or without sacrificing a lot of area. Creating shadow structures to keep cache and TLB speculative state, proposed in [9], isolates the information in the caches and protects against Spectre and Meltdown type attacks. The size of the shadow structures can be very costly, especially in systems with much larger caches and TLBs than the ones evaluated in that study. Similarly, we aim to isolate the branch predictor state to achieve similar security properties for some of the exploits without increasing the area or degrading the performance.

For branch prediction, clearing the branch predictor of any state for each context switch can take care of attacks that manipulate or eavesdrop on control flow to access data stored in the caches. However, flushing the entire state results in a significant accuracy drop with every context switch. Other alternatives such as hard partitioning can negatively impact the steady-state accuracy of the predictor, as the effective size per context is reduced. Tagging the BP entries is also not an acceptable solution, because in the worst case it behaves similar to flushing when most of the entries have been replaced by another context. More importantly though,

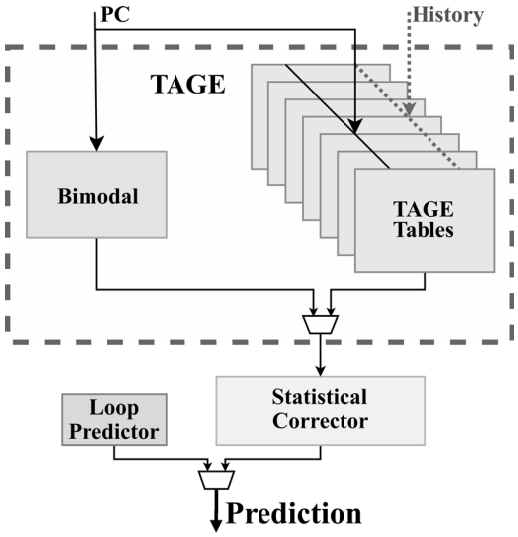


Figure 2: An abstract representation of TAGE.

while using tags eliminates branch predictor entry poisoning, it still allows observation of active entries, which can be flushed to cause deliberate mispredicts and data leaks.

A more secure BP design that uses isolation, can train efficiently, and quickly assume its steady-state performance will be useful in a post Meltdown and Spectre world[5]. Taking into account all of the above, current and future branch predictors need to be able to protect from potential side channels, without significant performance overhead when context switching frequently.

3) Threat model: For this work our threat model assumes a *victim* and an *attacker* application trying to infer without having authority to access victim information directly. In our model we assume:

- Both the victim and the attacker reside in the same core and share the same BP as described in [1, 4].
- Slowdown of the victim execution to be able to detect the behavior of a single branch. This has also been proven to be possible in recent studies [1, 4, 10].
- Ability for the attacker to force the victim code to execute, allowing vulnerable code to be targeted.
- The attacker having the ability to poison BP entries used by the victim application, forcing a misprediction.

B. Branch prediction design

Branch prediction has evolved over the years [11, 12] from small and simple designs to large, complex structures storing long histories of control flow. Here, we focus on the two most common designs used today as the basis of our study: the latest TAGE predictor [13], and the Multiperspective Perceptron predictor [14].

TAGE-based predictors: The TAGE predictor is one of the most accurate designs. It uses tagged geometric history

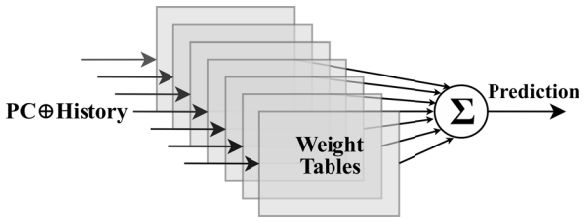


Figure 3: An abstract representation of Multiperspective Perceptron.

lengths that capture correlation from remote branch outcomes and recent history [2]. Internally, TAGE is comprised of tables that store the information for different history lengths. In short, when a prediction is needed, TAGE searches for the match belonging to the table with the longest history. If no match is found it uses its base predictor, a bimodal design, as a fall-back mechanism.

The TAGE design has been improved over the years incorporating other small components in order to further improve its accuracy in cases where the original design was shown to frequently mispredict. For that reason, the latest version published is the TAGE-SC-L predictor that also incorporates a statistical corrector and a loop predictor. In this paper we will use the latest version presented in Figure 2 as described in [13] as our representative example of TAGE-type predictors.

Perceptron type predictors: The other popular design that is widespread today is the Perceptron predictor [15]. Loosely based on neural network theory, Perceptron type predictors achieve high accuracy from efficiently stored state. Similar to the TAGE predictor, we use the latest variant from the *5th championship of branch prediction* (CBP5) [16], achieving higher accuracy but with more complexity [17, 14].

The principle behind Perceptron, as shown in Figure 3, uses a table with sets of weights that are multiplied with the history bits. The output and confidence of the predictor depends on the sign of the sum of all the history$\times$weight products. The confidence can be deduced from the magnitude of the value. Modern versions of Perceptron have reduced the amount of calculations required for each prediction and use multiple hashing tables of weights that are indexed using both the history and the branch address [18]. These tables are commonly referred to as feature tables [14].

III. Predictor Flexibility

Branch predictors have been evaluated based on how accurate their predictions are; given a reasonable amount of history they can learn from. However, given the recent findings described in the previous section, in many real world cases they operate in a time-frame much shorter than the ideal, predicting from only a partially warm or cold state. This happens as in order to fully guarantee isolation of

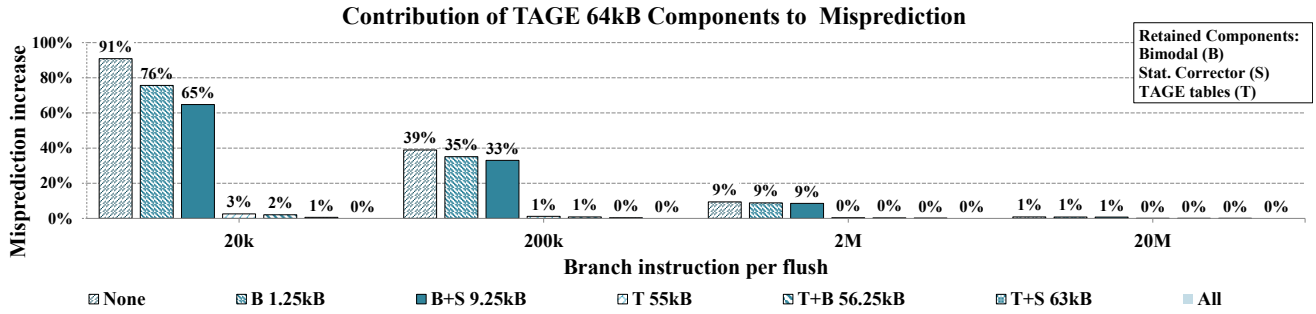


Figure 4: Contribution to MPKI for each TAGE64 component across different flushing periods.

BP state the predictor needs to be flushed between context switches causing a notable accuracy drop.

We mitigate these overheads in our BRB design, by identifying the components that contribute the most to accuracy during the warm-up phase and create a mechanism to retain their state per context – effectively replicating them. The rest of the predictor that only contributes to long-term accuracy is instead flushed preventing any state leakage. Our proposal addresses the overall security-performance trade-off, by guaranteeing isolation while also increasing accuracy for frequent switches.

Large predictors that store more information are usually able to deliver better predictions. However, if the state is lost or invalidated before the predictor has time to warm up, then effectively it will not reach peak performance. In this case, a smaller predictor might be able to deliver equivalent accuracy for a fraction of the state.

A. Steady-state and transient accuracy

We therefore distinguish between *steady-state* and *transient* accuracy. The term steady-state accuracy refers to the performance of the branch predictor when it is fully warm and reached its highest accuracy. Conversely, transient accuracy describes the behavior during the warm-up phase.

To quantify transient accuracy, we flush the branch predictor state across different branch instruction periods and track the change in average *mispredictions per kilo-instruction (MPKI)*. As we show in Figure 1 depending on how frequently the state is flushed, the actual (transient) accuracy can be significantly worse than the nominal, steady-state accuracy we normally evaluate against.

Our study examines the TAGE and Perceptron predictors which are heavily modular. Next, we dissect their behavior in situations where their state gets frequently disrupted.

Dissecting TAGE: TAGE uses a bimodal base predictor, 12 tables for the main TAGE predictor, a statistical corrector, and a loop predictor. These components vary in size and how much they affect the overall accuracy. We assess their contribution to the prediction as the predictor warms up for different frequencies of state flushing.

We use TAGE-SC-L to analyse how each of the components contributes to the accuracy when in a transient state. For that, we measure the reduction to the overall MPKI caused by each component, when preserving its state between flushes. In Figure 4 we track the size needed to retain the components in each configuration and the relative MPKI, normalized to the steady-state accuracy of TAGE. The analysis in the figure shows the transient state for a flushing period ranging from 20k to 20M branch instructions. For instance, it shows that, for 20k branches, flushing the entire predictor increases the MPKI by 90%.

Additionally, Figure 4 shows that the majority of the accuracy is delivered by the TAGE tables, which are too large to preserve. We note that for flushing every 20k branches, the bimodal base predictor and the statistical corrector improve the MPKI by 16% and 10% respectively. However, as bimodal needs only 1.25kB of state to be preserved, compared to the 8kB of the statistical corrector, its MPKI improvement per area cost is much higher than that of the statistical corrector.

Dissecting Perceptron: The evaluated Multiperspective Perceptron design uses multiple feature tables to deliver a prediction. Based on the previous studies [14], the features that contribute the most to increased predictor accuracy are identified. The most prominent ones are:

- Global History: The outcome of a branch (taken or not taken). A subset of the entire history is taken for a specific feature.
- Path: A hash of the recent sequence of branch addresses. The entries use truncations of the actual branch addresses to save space.
- Recency: Similar to Path this feature keeps a stack of recently encountered branches (using an LRU policy) and hashes them.

For our experiments we use the above features in various configurations, separately or hashed in combination (i.e. Global History XOR Path). Other features can be used to improve the prediction however, we find that they have limited effect on the overall prediction.

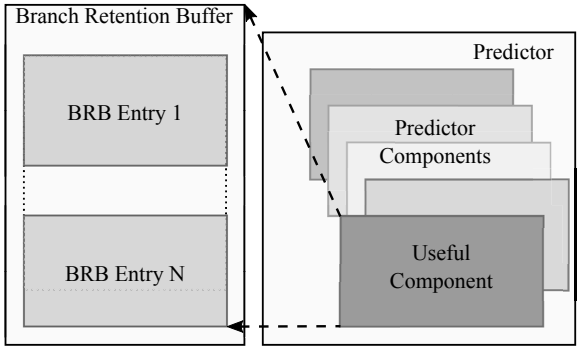


Figure 5: The mechanism which stores the essential state of the predictor. Most of the state is disregarded and only the most useful state is kept to increase transient accuracy.

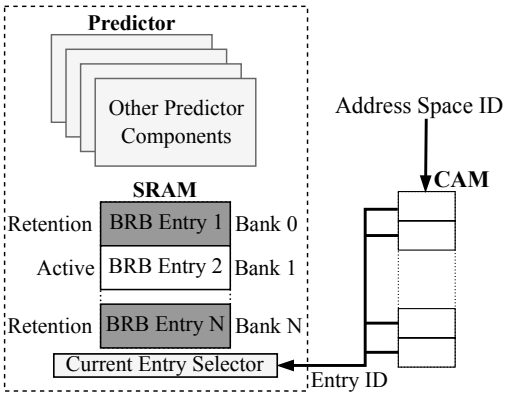


Figure 6: Diagram of the BRB. The retained state is stored in separate SRAM banks which correspond to different contexts.

B. The Branch Retention Buffer

From the predictor analysis, retaining even a small amount of state per context could improve the transient accuracy of the predictor, without hurting the maximum accuracy achieved during long uninterrupted execution. To guarantee isolation and simultaneously improve transient accuracy, we propose a mechanism (Figure 5) where state of the branch predictor (component) is retained per context. The mechanism uses a dedicated component called the *branch retention buffer* (BRB) that replaces part of the predictor.

The branch retention buffer is tightly coupled with the branch predictor and keeps *multiple separate entries*. Ideally, storing the entire state would enable high accuracy without any warm-up time. However, the size of designs today is too large to fully store without incurring notable overheads. To reduce the additional storage costs, we aim to make this component as small as possible. We find from Figure 4 that for TAGE 10kBits (1.25kB) sized entries are ideal, as they match the size of the base predictor for both the 8kB and 64kB variants. As such, the entries are designed to store entire components that can deliver a standalone prediction.

In short, we propose a low overhead mechanism that retains partial state that delivers better accuracy than a large "cold" predictor, with no steady-state accuracy penalty.

We implement the BRB to allow for easy switching between entries without moving or copying data and thus with no added latency during execution. This is done by keeping multiple entries in separate SRAM banks. When a switch occurs, the entry corresponding to the correct context is selected and used instead of swapping out data from the BRB. The selection uses the Address Space ID and triggers only when a context switch occurs. A small CAM is used to map to the correct BRB entry ID, shown in Figure 6. As a context switch occurs, any delay in the rerouting of BRB entries is masked by the larger overhead of storing the context state, which is conducted in parallel.

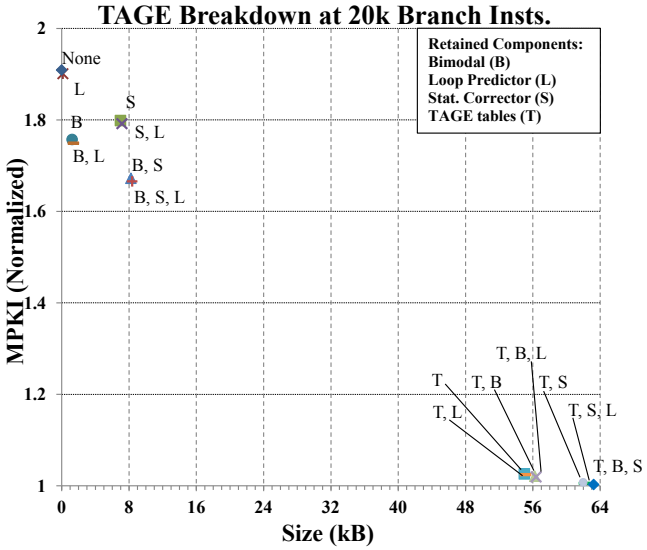


Figure 7: Contribution to MPKI for each TAGE64 component that is retained at 20k branch instructions. The bimodal reduces the MPKI by 16% while the statistical corrector by 10% despite it being 8x larger.

After the newly selected entry is activated, the others are put into retention to save energy. While operating under the same context, the active BRB entry is directly accessed for predictions and does not go through the CAM. When the BRB is full and a new context requests an entry, the least recently used entry is evicted. Previous studies show that 3 entries are enough to handle 2 communicating processes and operating system implications [19] without negatively impacting system performance. If more contexts are active and are consistently being switched out, more entries can be introduced. The added entries increase the area of the BRB component linearly, while timing remains the same irrespective of entries, for realistic sizes.

Retaining the reduced state makes isolation of processes efficient and, as a consequence, improves security. Keeping that reduced state separate per process (or between untrusted parts of the same process, such as browser and script engine), while emptying the rest of the structures, ensures that no state is ever shared between mutually untrusted processes and the kernel. The performance hit is softened in this case, as the preserved state increases the transient state accuracy.

The obvious question, in this case, is what method is used to reduce the data in the most impactful way. This question is not trivial, as it is directly tied to the BP implementation and the amount of state that can be stored efficiently when taking into account the overhead constraints.

One way to preserve state is to select certain components that provide a good balance of the amount of data stored and accuracy achieved; and discard the state of the remaining components. This "vertical cut" method can be used in the case of TAGE as it is comprised of various components that can provide accurate standalone predictions (Figure 7).

For instance, the base predictor can be isolated from the rest of the components and still provide reasonable accuracy. Similarly, separate TAGE tables can be preserved instead of the entire design, to target certain history lengths.

Other components, however, provide only complementary benefit to the predictions [14]. Therefore, it does not make sense to consider preserving the state of them by themselves. The loop predictor, for instance, is a relatively small component that identifies regular loops with a fixed number of iterations and needs few instructions to warm up. Its overall effect of on the accuracy of TAGE-SC-L is measured to be around 0.3% improvement [13].

In Perceptron, the vertical "cut" can be applied by only retaining certain features. In contrast to TAGE, this is significantly more complicated, as the empty features do not properly add up to the sum of all the weights and therefore skew the prediction outcome. To address this a special mechanism is needed to adjust all the preserved weights to deliver valid predictions. We consider this to be an impractical extra step needed every time a flush occurs.

Another way to preserve state is to store partial state for all the predictor components. For TAGE this can be done by naively saving a portion of each of the TAGE tables. This "horizontal" approach captures information across all of the history, albeit with less accuracy than storing the entire state.

As mentioned, in Perceptron the tables are combined to provide accurate predictions. A "horizontal cut" can be done by merging multiple neighbouring entries, which also reduces the accuracy.

C. Perceptron amplified / reinforced TAGE

As a specific showcase of our BRB design methodology, we propose a hybrid approach to preserving the state we call *ParTAGE* – Perceptron amplified / reinforced TAGE, shown in Figure 8. The branch retention buffer in this case stores

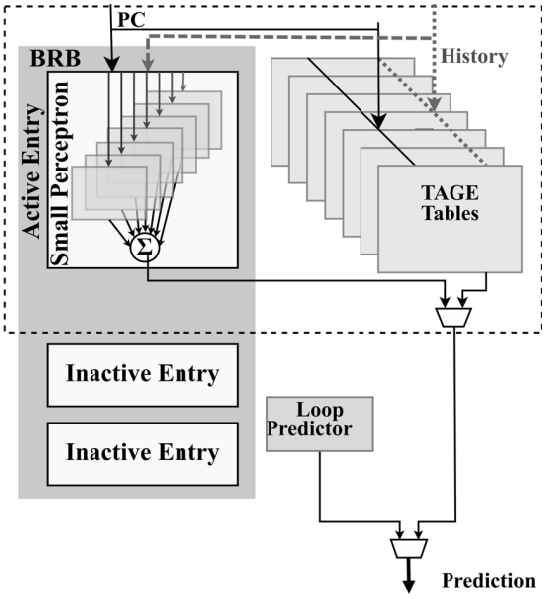


Figure 8: The ParTAGE predictor combines benefits from both perceptron and TAGE predictors. It also allows partial state to be preserved to improve transient accuracy.

a Multiperspective Perceptron predictor using smaller and fewer feature tables than the Perceptron from the literature as the base predictor of the TAGE design, replacing the bimodal predictor.

Each entry in the BRB stores an an independent small perceptron predictor enabling contexts to keep some reduced state separate. The rest of the branch predictor design can be cleared to eliminate the possibility of side-channel leaks targeting the branch predictor.

To reduce the size of the Perceptron within the range of the allocated budget, we use a similar configuration to the 8kB Multiperspective Perceptron [14] with 8 smaller feature tables instead of 16. The limited size of the Perceptron in ParTAGE enables its state to be preserved when context switching. In the intermediate warm-up state, ParTAGE has to select between transient prediction of the TAGE tables and the steady-state prediction of the preserved Perceptron. We design two versions of ParTAGE; one that hardwires the Perceptron predictor to be always chosen during the transient operationof the predictor (based on the context switching frequency), and one that assesses the confidence of the prediction of the base perceptron predictor before selecting the outcome.

IV. Experimental Setup

The experiments conducted use the CBP5 framework with the traces from 2016 [16] as a base. The framework uses 268 traces, ranging from 100 million to 1 billion instructions for both mobile and server workloads.

Name	Size	Details
BiM	64kB	2-bit counter
BiMH	1.25kB	8kbits counter
		2kbits hysterisis
	625B	4kbits counter
		1kbits hysteresis
Perceptron	64kB	37 feature tables
	8kB	16 feature tables
TAGE	64kB / 8kB	Loop Predictor, Statistical Corrector, BiMH1 base predictor
ParTAGE	64kB / 8kB	Loop Predictor, Statistical Corrector, 1.25KB 8 table perceptron

Table I: The evaluated branch predictors.

OoO Core	Value	Memory	Value
Pipeline Width	3-wide	L1D Size	32kB
Pipeline Depth	15	L1I Size	32kB
Clk. Frequency	1 GHz	L2 Size	1MB
		L2 Prefetcher	Stride

Table II: The specifications of the system used in our gem5 experiments [20].

We modify the CBP framework so that branch predictors can perform full or partial flushes on their state. This enables temporal studies of the behaviour of branch predictors, revealing the effects of full or partial loss of state.

For our experiments, we use a variety of predictors that are commonly used today. As a baseline design, we implement a set of bimodal type predictors, with and without hysteresis. To compare more contemporary predictors, we use the submitted TAGE-SC-L [13] and Multiperspective Perceptron without TAGE [14] from CBP5. We lightly modify both designs so that we can flush their designs partially or completely when needed; carefully retaining their their exact steady-state behaviour.

Furthermore, we implement two variants of ParTAGE that express different policies for the selection of the best transient prediction. ParTAGE-S overrides the prediction of the TAGE tables below a period threshold which we have set to be 200k branch instructions. ParTAGE uses an integrated confidence value that is assessed by the rest of the TAGE design in order to indicate the most accurate prediction.

We focus on 8kB and 64kB predictors, similar to the ones that are evaluated at CBP. A detailed list of all the evaluated predictors is shown in Table I. We assess the transient accuracy of the evaluated predictors for the cases outlined in Section II. To achieve this, we cover a range of flushing periods from 10 to 60M branch instructions per flush; extracting the optimal design for each use case.

We also perform a limit study, that identifies the upper limit of core performance drop when flushing the predictor periodically, ranging from 1k to 10M total instructions. We use an Arm OoO model (Table II) in a gem5 [20] full system simulation running MiBench [21]. We compare two systems, one that periodically flushes the state and one that does not.

Application	Time(s)	Cxt. Switches (CS)	CS/s
adobereader	73	459,699	6,279
gmail	23	228,924	9,833
googleslides	108	343,788	3,185
youtube	36	418,254	11,487
yt_playback	21	224,631	10,698
angrybirds_rio	81	406,711	5,044
camera@60fps	60	1,958,557	32,643
geekbench	60	129,533	2,159

Table III: The frequency of context switches on a Google Pixel phone.

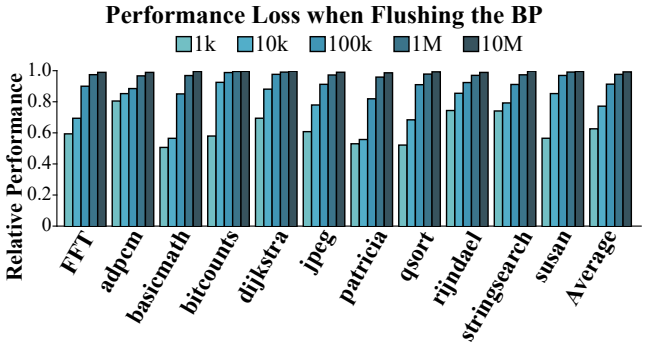


Figure 9: Impact of periodical BP flushing (1k to 10M instructions) on core performance normalized to a system without flushing. Results extracted from gem5 using full system simulation [20] of an Arm OoO model running MiBench[21].

V. Results

We perform three different types of comparisons focusing on simple bimodal predictors, current TAGE and Perceptron designs, and our ParTAGE proposal. We measure the frequency of context switches on a modern mobile device (Table III) and find that switches can happen on average as often as *every 12k branch instructions.* This leads to significant core performance loss, as much as 15% based on our limit study (Figure 9).

A. Quantifying transient accuracy

1) Bimodal accuracy results: We use Bimodal as a simple first experiment, consisting of a single table of counter bits. In Table IV we show how the MPKI of different bimodal designs improves as the state retention period increases.

	MPKI									
	Branch instruction flushing period (instructions)									
Name	**10**	**100**	**200**	**2k**	**20k**	**200k**	**2M**	**20M**	**40M**	**60M**
BIM 64kB	36.77	22.22	19.52	14.12	12.13	11.40	11.16	11.14	11.15	11.15
BIM 1kB	36.78	22.26	19.58	14.40	12.87	12.59	12.52	12.51	12.51	12.51
BIMH 40kB	30.97	19.33	17.23	13.40	12.04	11.56	11.41	11.39	11.39	11.39
BIMH 10kB	30.96	19.34	17.24	13.43	12.10	11.63	11.49	11.47	11.47	11.47
BIMH 1.25kB	30.97	19.39	17.33	13.68	12.53	12.17	12.08	12.07	12.07	12.07
BIMH 625B	30.99	19.45	17.42	13.93	12.91	12.62	12.55	12.54	12.54	12.54

Table IV: Mispredction rates of a wide range of bimodal predictors across different state flushing periods measured in branch instructions.

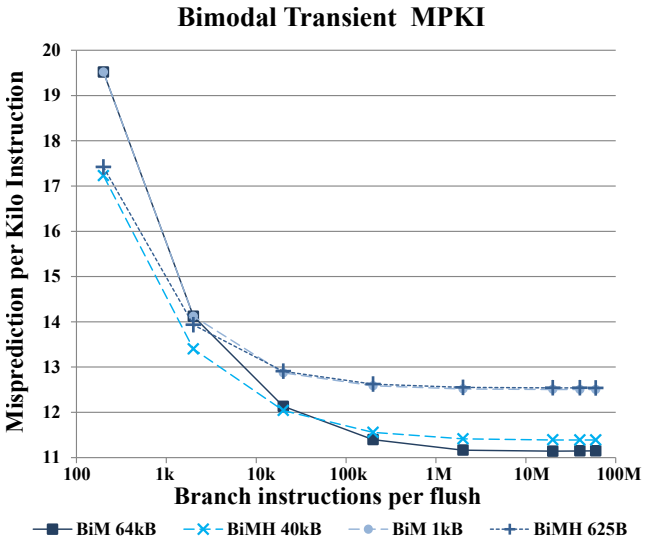


Figure 10: Comparison of sizes and types of bimodal predictors. Size contributes to steady-state accuracy, hysteresis to transient accuracy. Note the Y-axis begins at 11 MPKI offset.

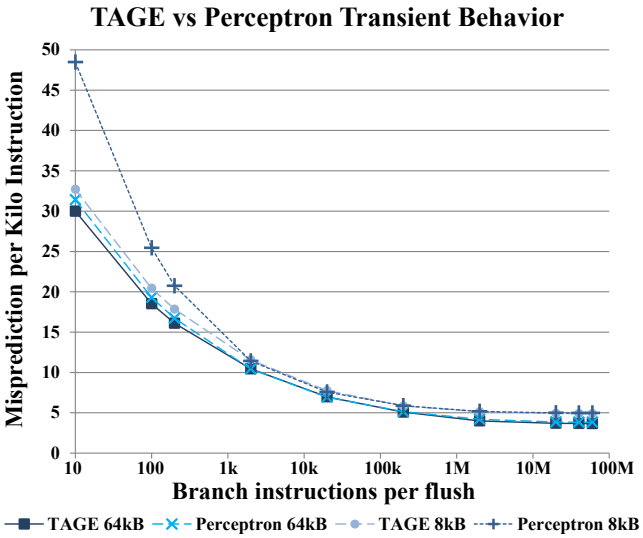


Figure 11: A comparison between TAGE and perceptron. While both exhibit similar steady-state accuracy TAGE shows better transient behavior.

Despite a 100x difference in size, the steady-state MPKI increase is only 12.28%. The results reveal that while size contributes to the steady-state accuracy, transient accuracy is not affected by the size of a predictor design.

Instead, a variation in the design such as adding hysteresis improves transient accuracy, even for smaller predictors. This happens as the hysteresis bits also affect neighbouring branches and ultimately warm-up the design faster. This is clearly visible at smaller flushing periods where the misprediction is on average 18% lower for the bimodal designs with hysteresis. Figure 10 shows the difference in accuracy for different bimodal sizes and designs.

2) Transcient accuracy: TAGE vs Perceptron: Our second set of results focuses on comparing the transient behaviour of the two most prominent designs in modern systems; TAGE and Perceptron. Figure 11 shows the different transient behaviour between TAGE and Perceptron designs. We notice that the 8kB variant of Perceptron has the worst cold start, however it manages to rapidly improve and assume similar steady-state accuracy.

The transient MPKI for a flushing period of 20k branch instructions is 7.75 and 6.98 for the 8kB and 64kB TAGE designs, and 7.57 and 6.93 respectively for Perceptron. Switching every 20k branch instructions is within a realistic range for applications like the ones presented in Table III. This result shows that TAGE can deliver better steady-state accuracy. However, for applications that perform frequent context switching (20k branches), Perceptron is marginally more accurate.

Another observation can be extracted when comparing the 64kB variants with the 8kB ones at smaller windows of uninterrupted execution. Considering for instance, flushing every 20k or 200k branch instructions, the 8kB predictors

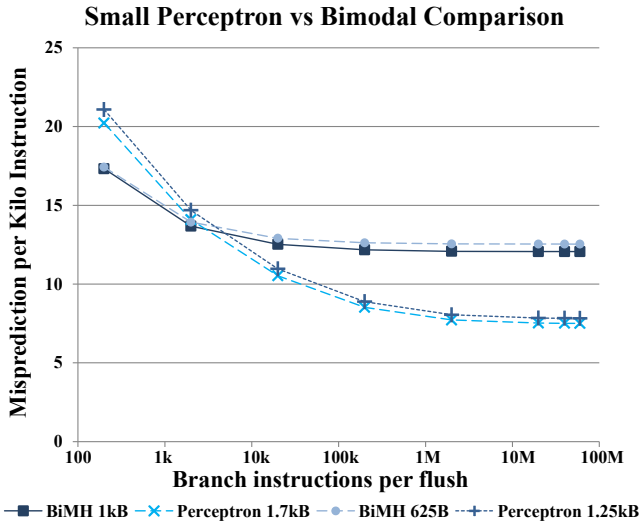
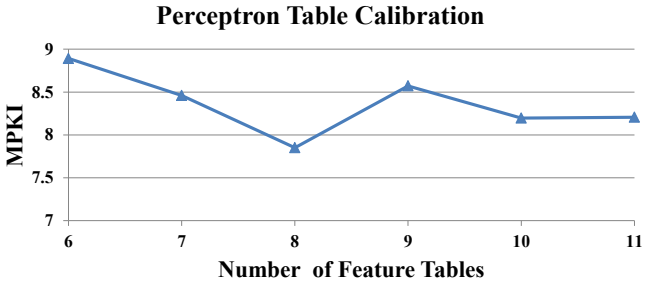
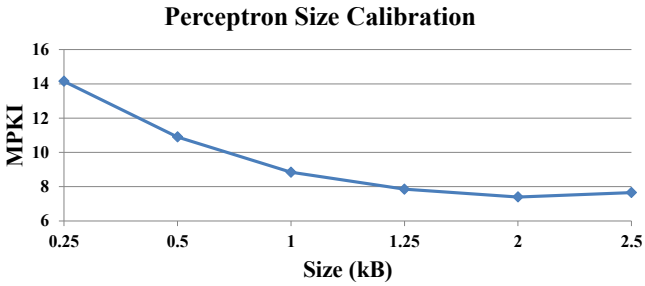


Figure 12: Comparing bimodal and Perceptron designs below 1.25KB as base predictors. Results show even when reduced this size Perceptrons greatly outperforms competition.



(a) Calibrating the number feature of tables for 1.25kB size.



(b) Different sizes of perceptron with a constant of 8 feature tables.

Figure 13: Calibrating the small perceptron

perform on average 10% and 15% worse than the 64kB designs. However, the accuracy gap increases to 33% when observing the same designs at steady-state.

Using the steady-state accuracy as a baseline and comparing the transient accuracy, across all granularities as fine as 20k branch instructions, we calculate how much worse the accuracy can be in context-switch-heavy workloads. From Figure 11, the MPKI increase is as much as 90% and 80% for TAGE and Perceptron respectively. This reinforces our belief that predictors today are evaluated without taking into account disruptions that can occur during realistic execution.

B. ParTAGE

To improve on transient accuracy of branch prediction, we present the results from our proposal, ParTAGE a design that is influenced from both Perceptron and TAGE. Next, we motivate its design choices and proceed to analyse its accuracy compared to the competitive designs today.

Finding the right, "small" predictor: The TAGE breakdown analysis leads to the underlying idea for the hybrid predictor: replace components which use a significant amount of area with other, retainable components that can deliver a higher transient accuracy.

From Figure 11 we observed that Perceptron predictors rapidly improve their MPKI over time before reaching steady-state accuracy. This reveals an interesting insight about Multiperspective Perceptron predictors: compared to bimodal predictors, the hashed and common weight values in modern Perceptron designs are a more effective design, being able to aggressively train and and store information in a denser format. We focus on the latter attribute, that of dense branch information storage, to design the base predictor as we intend to store and restore it for each context, thus never letting it return to its transient accuracy. Furthermore, in Figure 7 the statistical corrector is shown to add little accuracy when the TAGE tables are cold, despite its large size (8kB). We use this to experiment with larger perceptrons but maintaining a balance in terms of size.

In Figure 12 we compare bimodal and Perceptron designs that will fit roughly within 1.25kB of budget. Results show that while Perceptron has higher MPKI at granularities as fine as 200 branch instructions, its steady-state accuracy is significantly better than bimodal.

Optimizing the base predictor: We reduce the size of each BRB entry to 1.25kB and calibrate the amount of feature tables and size of the Perceptron predictor. We fix the size of the predictor initially to 1.25kB (Figure 13a). We find that eight feature tables deliver the most accuracy. We repeat the process for 3kB perceptron entries, which increase the size of the predictor by 10% . For this reason, we also evaluate a design without the statistical corrector maintaining equal area to the original TAGE design.

Figure 13b shows the best configuration for each size. Note that changing the feature sizes affects the prediction more than the altering the size does. We find that for completeness, a genetic algorithm as proposed in past studies [14] can deliver even better results. We leave these optimisations for future studies.

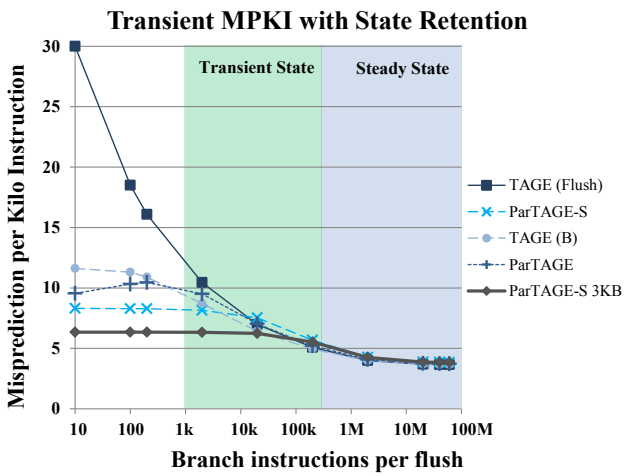


Figure 14: Decrease of MPKI when retaining 1.25KB and 3kB of state. Note that TAGE clears its entire state, while all other designs keep their base predictor state intact.

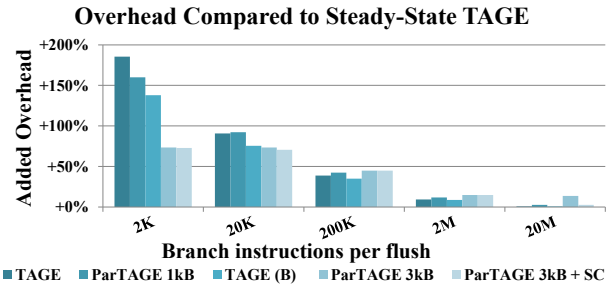


Figure 15: MPKI increase compared to steady-state when retaining 1.25kB. In coarse granularity the effect is negligible, but for frequent flushes the improvement is significant.

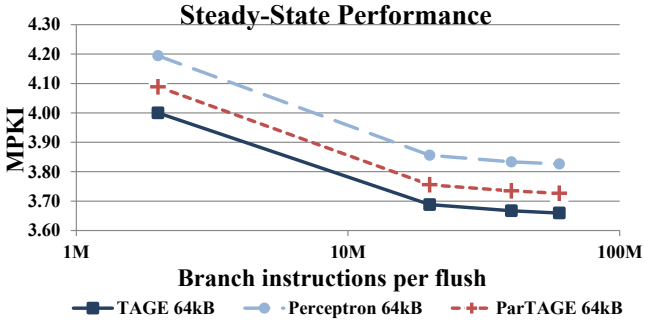


Figure 16: The steady-state reveals that ParTAGE can perform competitively and even outperform existing high accuracy designs.

C. ParTAGE results

We create the ParTAGE predictor based on our observations for small predictors replacing the 1.25kB bimodal with a perceptron.

Comparing ParTAGE variants: The results in Figure 14 compare the designs featuring the BRB, all variants of ParTAGE and TAGE (B) (that retains the bimodal), to TAG without the BRB. For ParTAGE, evaluate both variants that retain both 1.25kB and 3kB BRB entries. ParTAGE-S statically overrides the TAGE tables for a brief period after a flush is performed, forcing the Perceptron prediction to be selected, while ParTAGE selection prediction based on each component confidence. The 1kB entry designs are overall 3% larger in area, while the 3kB variant removes the statistical corrector, maintaining the same area budget.

When considering current system upper limit context switching, we see that the most accuracy is delivered by our proposed BRB extension with the 3kB ParTAGE variant which is iso-area compared to TAGE-SC-L . Figure 15 shows the improvement is roughly 20% and 15% for 3kB ParTAGE and TAGE(B) respectively. Overall, preserving a minimal state can have a significant improvement (15-20% less MPKI) when the state is frequently flushed but has a small effect at the steady-state (maximum 5% MPKI increase). To maintain the same steady-state accuracy the statistical corrector is included (ParTAGE 3kB + SC in Figure 15) increasing the overall area by 10% but delivering better accuracy throughout.

Comparing ParTAGE to TAGE and Perceptron: Comparing the different implementations of ParTAGE, we notice that while ParTAGE-S works well for the fine grain switches, the transient accuracy suffers at larger flushing periods, as the TAGE tables have not been trained adequately. In contrast, ParTAGE, which simply feeds the confidence into TAGE, does not achieve the same transient accuracy. This happens when TAGE tables are completely cold, but their prediction is prioritized over the more accurate one from the perceptron component. This is also why ParTAGE does not improve the transient accuracy at 20k branch instructions. We propose to solve this with better tuning of the selection policy in the future.

Improving the steady-state accuracy of the base predictor and retaining its state effectively enables efficient operation at finer granularities. This can be done by either increasing the size of the branch retention buffer to fit more state, or develop predictors ranging between 1kB to 3kB with better steady-state accuracy. For instance, approaching the accuracy of an 8kB perceptron can further reduce the misprediction, shown in 11.

While the primary focus of this study is the improvement of the transient accuracy of branch prediction, it is equally important to maintain a competitive steady-state accuracy for our proposed design. We perform a direct comparison between TAGE-SC-L, both Multiperspective Perceptron versions submitted to CBP5 [16] and the best version of ParTAGE. Figure 16 provides a detailed look at the steady-

state MPKI compared to TAGE and the best version of the Multiperspective Perceptron [22, 14]. We observe that ParTAGE delivers 3.726 MPKI, competitive to TAGE (3.660 MPKI) and even outperforming perceptron (3.826 MPKI).

VI. Related Work

The effects of context switches on predictor accuracy and overall performance has been studied in the past with varied results. Studies like [23] show that in the past when systems did not switch often (above 400k instructions) and branch predictors could fully train in 128k instruction periods, the effects of context switching to the BP accuracy were negligible.

When switching at a faster rate, [11, 24, 25] show that the branch predictor accuracy drop is in the range of 7% - 20%. We show that context switches on current hardware can happen more frequently today than in the past (III), in cases as often as every 64k instructions. Furthermore, compared to past studies current branch predictor designs like TAGE and multiperspective perceptron take much longer to warm up than 128k instructions. Our results show that branch mispredicts can increase by as much as 90% when applications today are switching aggressively and flushing.

Some studies [26, 27] focus on the effect that context switches have on actual performance. They find that when switching is done too frequently, cache misses overshadow all other overheads and greatly degrade performance. Other work, namely [28] shows that even when applications have all of their data in the caches, the performance hit caused by the branch predictor can be as much as 20% for 100k instruction switching frequencies for Out-of-Order cores while In-Order cores show little degradation. Our results from Figure 9 agree with such findings. Out-of-Order execution successfully masks cache misses, but relies heavily on speculation and branch predictor accuracy to deliver performance. Flushing for security reasons perfectly matches these simulations and we expect dimilar performance drop.

Studies break down the contribution to accuracy of the components in Perceptron predictors [14], but a similar study of the breakdown of the accuracy into components has not been done so far for TAGE-type designs.

While many studies track the effects of accuracy across different sizes of predictors and switching frequency [29, 30, 2, 23] the notion of accuracy under frequent flushes we believe to be a novel insight, especially when taking necessary security precautions for side-channel attacks. Studies have been conducted [19] that focus on isolating the kernel from the applications in the branch predictor for performance. This can potentially increase security but will not completely eliminate BP side channels. Furthermore, the design proposes a hard partition of the entire BP state which would be costly in terms of area for current large predictor designs, unless steady-state accuracy is sacrificed.

Security studies [1, 5] identify practical threats that can compromise the system using the branch predictor, in their work they mention branch predictor flushing as a mitigation technique for the security aspects however they do not provide an estimate of the performance loss of such an approach.

Our results contribute to quantifying the performance loss in such scenarios. The branch retention mechanism we describe, to the best of our knowledge, has not been preciously proposed and provides a viable solution when the performance degradation is significant.

VII. Concluding Remarks

In this work, we have focused on creating hardware security mitigations for side-channels in the branch predictor which can be disruptive to performance when dealt in software. We highlight realistic scenarios where this can occur, such as frequent context switches and system calls, and show that current mitigation techniques for side-channel attacks targeting the speculation engine are both expensive and impractical. We show that these disruptions create a disconnect between the reported nominal accuracy of branch predictors and the actual one in a real world applications. To distinguish between the two, we introduce the notions of steady-state and transient branch predictor accuracy.

We propose a novel mechanism, *the Branch Rentention Buffer* (BRB), that keeps a minimal, isolated state per context to reduce the high number of mispredictions. We propose two designs that store essential context state in the BRB; first, an extension to TAGE, named TAGE (B), that keeps the state of its bimodal predictor. Second, a novel hybrid branch predictor design, ParTAGE, that replaces the bimodal in TAGE with a Perceptron. We evaluate these variants with a new methodology, which modifies the Championship Branch Prediction framework so we can clear predictor state across different frequencies and components.

We show that branch predictors can have as much as 90% more mispredicts than what is evaluated today at steady-state, under certain realistic conditions. Using the BRB we manage to guarantee BP state isolation and consequently increase security when context switching, while achieving reductions on MPKI of 15% and 20% for TAGE and our hybrid predictor design respectively.

Improving the steady-state was not the goal of this study as we focused on isolation while simultaneously mitigating the negative effects in the transient state. However, we believe that with future optimisations, our designs can improve the steady-state as well. We also aim to enhance the transient accuracy by focusing on improving policies that select between retained and “cold” state components, in order to fully exploit all the benefits from the base predictor. Finally, we aim to focus on more accurate “small” designs in the 1kB - 3kB range.

ACKNOWLEDGEMENTS

This work was supported in part by the Engineering and Physical Research Council (EPSRC) under grant number EP/K034448/1 PRiME: Power-efficient, Reliable, Many-core Embedded systems (www.prime-project.org). Experimental data used available at https://doi.org/10.5258/SOTON/D0739.

This research was supported by the people of Arm Research. We thank our colleagues for their valuable feedback that greatly assisted us in delivering a better version of this work.

REFERENCES

[1] 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 e-prints*, 2018.

[2] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Jilp*, vol. 8, pp. 1 – 23, 2006.

[3] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *International Symposium on Computer Architecture*, pp. 361–372, 2014.

[4] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope," in *Architectural Support for Programming Languages and Operating Systems*, pp. 693–707, 2018.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, 2018.

[6] Y. Yarom and K. Falkner, "Flush + Reload : a High Resolution, Low Noise, L3 Cache Side-Channel Attack," *USENIX Security 2014*, pp. 1–14, 2014.

[7] 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*, pp. 605–622, 2015.

[8] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, C. Yee, A. Reuther, and J. Kepner, "Measuring the Impact of Spectre and Meltdown," *ArXiv*, pp. 1–5, 2018.

[9] 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," *ArXiv*, 2018.

[10] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Annual Conference on Computer Security Applications*, pp. 422–435, 2016.

[11] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in *International Symposium on Computer Architecture*, vol. 24, pp. 3–11, 1996.

[12] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *International Symposium on Microarchitecture*, pp. 51–61, 1991.

[13] A. Seznec, "TAGE-SC-L branch predictors," *JWAC-4: Championship Branch Prediction*, 2014.

[14] D. A. Jiménez, "Multiperspective Perceptron Predictor," *JWAC-4: Championship Branch Prediction*, 2014.

[15] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *International Symposium on High-Performance Computer Architecture*, pp. 197–206, 2001.

[16] "5th Championship Branch Prediction," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5)*, 2016.

[17] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *International Symposium on Computer Architecture*, pp. 394–405, 2005.

[18] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 3, pp. 280–300, 2005.

[19] J. Rubio and N. Vijaykrishnan, "OS-Aware Branch Prediction: Improving Microprocessor Control Flow Prediction for Operating Systems," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 2–17, 2007.

[20] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, 8 2011.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, pp. 3–14, 2001.

[22] D. A. Jiménez, "Multiperspective Perceptron Predictor with TAGE," *JWAC-4: Championship Branch Prediction*, 2014.

[23] M. Co and K. Skadron, "The effects of context switching on branch predictor performance," in *International Symposium on Performance Analysis of Systems and Software*, pp. 77–84, 2001.

[24] S. Pasricha and A. Veidenbaum, "Improving branch prediction accuracy in embedded processors in the presence of context switches," in *International Conference on Computer Design*, 2003.

[25] A. S. Dhodapkar and J. E. Smith, "Saving and restoring implementation contexts with co-designed virtual machines," 2001.

[26] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," in *Architectural Support for Programming Languages and Operating Systems*, vol. 19, pp. 75–84, 1991.

[27] F. Liu and Y. Solihin, "Understanding the behavior and implications of context switch misses," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 4, pp. 1–28, 2010.

[28] I. Vougioukas, A. Sandberg, S. Diestelhorst, B. Al-Hashimi, and G. Merrett, "Nucleus: Finding the sharing limit of heterogeneous cores," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5, 2017.

[29] M. Das, A. Banerjee, and B. Sardar, "An empirical study on performance of branch predictors with varying storage budgets," in *International Symposium on Embedded Computing and System Design*, pp. 1–5, 12 2017.

[30] C. Zhou, L. Huang, Z. Li, T. Zhang, and Q. Dou, "Design Space Exploration of TAGE Branch Predictor with Ultra-Small RAM," in *Great Lakes Symposium on VLSI*, pp. 281–286, 2017.