UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Weird Machines with TLB

Weird Machines mit dem TLB

Masterarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Marcel Pflaeging

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 08. August 2025

Abstract

Caches are an important feature that increases the performance of modern processors. In particular, data caches can be exploited for side-channel attacks, a topic that has already been extensively researched. In recent years, improvements to these attacks have been made through work in the field of weird computation, which uses caches in combination with speculative execution. Microarchitectural weird machines are primarily employed for obfuscation, enabling hidden computations. This demonstrates how different data caches can be exploited.
Since microarchitectural weird machines have not yet been used with many other types of cache, the TLB is used in this work. This cache can be used similarly to data caches, but fewer defense techniques are effective against it. We demonstrate that, while the TLB can be used as an alternative, it has some limitations.

Zusammenfassung

Caches sind ein wichtiges Feature, um die Leistung moderner Prozessoren zu erhöhen. Insbesondere Datencaches können für Seitenkanalangriffe ausgenutzt werden, die bereits umfassend untersucht wurden. In den letzten Jahren wurden diese Angriffe durch Arbeiten aus dem Bereich der Weird Computation verbessert, bei denen die Caches in Kombination mit spekulativer Ausführung genutzt werden. Mikroarchitekturelle Weird Machines werden meist für Obfuscation genutzt, um versteckte Berechnungen durchzuführen. Dabei zeigt sich, wie vielfältig Datencaches verwendet werden können.
Da die Mikroarchitekturelle Weird Machines noch nicht mit vielen anderen Arten von Caches verwendet wurden, nutzen wir in dieser Arbeit den Translation Lookaside Buffer (TLB). Dieser Cache kann ähnlich wie Datencaches verwendet werden, allerdings sind weniger VerteidigungsmaSSnahmen dagegen wirksam. Wir zeigen, dass der TLB als Alternative genutzt werden kann, jedoch einige Einschränkungen aufweist.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 08. August 2025

Acknowledgements

I would like to thank Thore Thiemann and Jonah Heller for their support and assistance throughout my work. I would also like to thank Thomas Eisenbarth for introducing me to this fascinating field of research.

Contents

1 Introduction

Over time, modern processors have become faster due to specialized components in the microarchitecture. But this has also increased the complexity of the processors, leading to vulnerabilities. Many microarchitectural attacks were discovered that can leak sensitive information, such as cache attacks. Caches accelerate subsequent memory accesses and operations by storing them as entries in the cache. The attacks leak sensitive data by observing the cache state and deriving information from this. There are two states for entries in a cache, a hit and a miss, that can have to be distinguished. This is done by a threshold for the timing difference of accesses. Multiple cache attacks target different caches in the CPU, like data caches, instruction caches, and address translation. These attack can lead to leakage of sensitive data within and across processes, if traces of the data are left in the cache.

To protect the CPU from these attacks, some defense techniques have been developed. A few of them restrict the access to high-precision timers, but Katzman et al. [KKC+23] showed a way using weird gates that amplify the cache state to be able to get distinguished by low-resolution timers. Weird gates developed in recent years as part of microarchitectural weird machines often use the data caches. They abuse an additional part of the microarchitecture, the speculative execution, which then enables computations within the microarchitectural state. The speculative execution is forced to misspredict the control flow ahead of time, executing instructions that intentionally influence the caches. The CPU then squashes the wrong branch, but the caches keep their modified state. Therefore, this side effect can be used for computations. By composing weird gates into weird circuits, it is possible to provide turing complete operations. The results are observed by measuring the timing behavior of a cache line, instead of directly reading a value. Wang et al. [WPWB24] built a compiler that allows writing high-level code that is converted into weird machines, allowing for outsourcing computations into the microarchitectural state. The weird machines can be used for program obfuscation, as they hide the intention by side-effect-driven computations. The concept of the weird gates can also improve side-channel attacks.

As it was shown that weird machines are practical on data caches, we investigate whether the address translation cache, i. e. TLB, is also usable for this purpose. We do this by adapting the current state-of-the-art to use the TLB-entries as weird registers.

1.1 Related Work

In this part, we take a closer look at related work regarding microarchitectural weird machines, highlighting the differences from this work.

1.1.1 Translation Leak-Aside Buffer

Gras et al. [GRBG18] showed how to leak fine-grained information with the Translation Lookaside Buffer even when the CPU is protected by cache defense techniques. They reverse-engineered the TLB and developed an attack that overcomes cache defenses.
Their work laid the foundation for efficient TLB-based side-channels, as they discovered the mapping function inside the TLB. In this thesis use the knowledge about the TLB but do not use all TLB properties like the mapping function.

1.1.2 Gates of time

Katzman et al. [KKC+23] created generic weird gates that are turing complete and used them to compute in the microarchitectural state without exposing values to the architectural state. In addition to this, they used the gates for amplifying and improving side channel attacks.
We adapt the weird gates to use another microarchitectural component, which can then be used for further work. We do not directly improve side channel attacks of any kind.

1.1.3 The ghost is the machine

Wang et al. [WBW23] generalized weird gate constructions and built them by using different transient execution primitives. They show that these gates work on different platforms like Intel, AMD, and ARM.
In this thesis, we aim for the TLB component on the Intel platform and therefore only take a look at modern Intel processors. We use the state-of-the-art generic approach for the weird gates and the Return-Stack-Buffer as a transient primitive.

1.1.4 Flexo

Wang et al. [WPWB24] provided a compiler framework for C code to create generic weird gates for different x86 machines. They demonstrated that the weird machines provide performance that can be used in practice.
We build on this work and aim to substitute the microarchitectural component to use the TLB.

1.1.5 Spec-o-Scope

In 2024, Horowitz et al. [HRY24] used weird gates to improve the probing speed and therefore the temporal resolution of the state-of-the-art Prime+Scope side-channel attack. They created a new type of weird gate that uses speculative execution to execute more probes that leave a trace in the cache for a hit.
In this work, we do not create new gates or improve cache attacks. But we want to use weird machines with the TLB and investigate the effectiveness. This would enable one to adapt Spec-o-Scope for TLB probing.

1.2 Contributions

In this work, we provide the following results.
We investigate the TLB on different machines and how to control and read the cache residency of the entries. In this scope, we develop a custom tool to get a threshold for the TLB.
Then we explain the state-of-the-art ţWeird Machine (ţWM) by Wang et al. [WPWB24] that we adapt. We build on their compiler framework Flexo and advance it to use the TLB as the microarchitectural state of the ţWM. Then we evaluate this approach, comparing it to the original Flexo. Finally, we explain the difficulties and side effects when using the TLB, especially regarding the Page-Table-Walk (PTW).

1.3 Overview

We introduce the basic terms and concepts in Chapter 2. Then in Chapter 3, we take a look at the TLB to explain the basic structure and examine how to control its entries. Chapter 4 gives an overview of the state-of-the-art weird machines. In Chapter 5 we adapt them to create weird machines that are based on the TLB. We perform experiments in Chapter 6 that consider the transient window size and compare our TLB-based weird machines with the previous state-of-the-art weird machines. We discuss these experiments in Chapter 7 and take a look at the detection of weird machines. Chapter 8 summarises the results and observations of our work, ending with a description of open problems.

2 Preliminaries

This chapter introduces the basic concepts that are relevant to this thesis. We give an overview of weird machines and the microarchitecture in modern processors.

2.1 Virtual Memory

In modern operating systems, each process has its own virtual address space, which isolates the processes from one another and reduces conflicts and leakage. The virtual and physical address space is divided into pages and page frames of a specific size (e. g. 4 KB). The Memory Management Unit (MMU), a special hardware component in the CPU, is responsible for translating a virtual page to a physical page frame. The mapping for translating virtual pages to physical page frames is stored in multi-level page tables, which are an efficient way to store the translations. The page tables are hierarchically structured with four to five levels and have Page Table Entries (PTEs) that are pointers to a page table in the next level or the requested physical page frame. The translation of a virtual address to a physical address is called a PTW. To reduce the number of PTWs for frequently used translations, they are stored in the TLB.
In Fig. 2.1 it is shown which parts are involved in a memory access. Depending on the TLB cache residency, this translation can be fast, by querying only the TLB, or slow for a full PTW. After resolving the physical address, the data cache is queried next [HP12].
To increase the accuracy of differentiating a cache hit from a cache miss when measuring the time for a memory access for the data cache, ensure that the virtual address translation is in the TLB, as the latencies add up. This can be easily achieved by fetching memory from the same page without triggering the prefetcher to prefetch the target address.
On the other hand, to get precise timing measurements that are necessary to differentiate a TLB hit from a miss, one has to ensure that the data line can be served by the L1 data cache, eliminating additional latencies that are independent of the TLB state.
Since data caches use physical addresses to tag entries, a data cache entry remains unchanged when accessed from different virtual addresses. The translation occurs independently in the MMU beforehand, resulting in the same query to the data caches. This can be used to create multiple virtual pages that map to the same physical page frame. On a memory access to one of those pages, the MMU has to translate its address, but the accessed physical memory stays the same and can be served by the L1 cache. To create

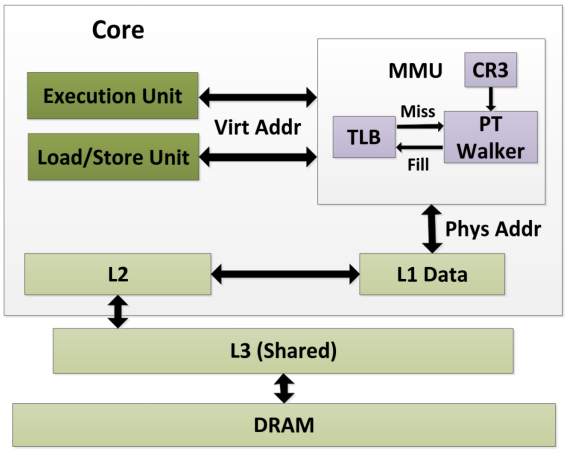


Figure 2.1: Memory organization in a recent Intel processor (adapted from [GRB+17])

multiple virtual pages to one physical page frame, the *mmap*-system-call can be used.

2.2 Caches

One big limiting factor in the execution speed of a process is the access to data in the memory. Such accesses result in the CPU idling for several cycles. To overcome this and improve processing speed, the CPU provides different levels of caches. Their purpose is to provide fast access to recently accessed data. A cache is a small and fast memory that is physically close to the processor core. The data is stored within the cache in blocks of a fixed size that are called *lines*. Typically, a line is 64 bytes wide. Each line is tagged with the physical address of that data in the memory. When accessing memory, the cache is checked for a line tagged with the physical address. A cache hit refers to a line residing in the cache. In this case, the data can be quickly served to the processor. A cache miss occurs when there is no matching line in the cache. This means that the data must be retrieved from the memory or the next cache level. When this happens, the processor stores the retrieved data in the cache for later use. Since caches are fixed in size, they must evict old data when new data is loaded, according to replacement policies. Caches are often organized in multiple levels. Each higher cache level has more latency and entries than the previous. Modern Intel CPUs have an L1, L2, and L3 data cache hierarchy. The L1 cache is generally the smallest and core-local, while the higher levels are larger and often shared between cores or chiplets.

There are separate caches for data and instructions. The data cache stores data fetched from the memory and moved into registers for computation. Instruction caches store instructions from binary or runtime libraries that are likely to be reused, such as loops.

Lower-level caches often have distinct data and instruction caches, while higher-level caches often combine them.
Entries in modern caches are often split into different sets that are determined by a mapping function applied to the tag. Each set can contain multiple entries, called ways. An eviction is local to each cache set of such a set-associative cache. A cache with only one set is called full-associative. Although these have a better hit rate, they are more complex and difficult to scale up.
On eviction, it is hard to choose the best address to evict. The perfect replacement policy would be to evict the address that is not needed for the longest time. As this is impossible to know, often some type of Least Recently Used policy is implemented. The choice of the replacement policy influences the use of eviction sets.
Knowing the mapping function inside caches can help to create eviction sets. An eviction set is a group of addresses that are mapped to the same cache set. Upon accessing all addresses in the eviction set, an address previously stored in the cache is evicted.
It has been shown [GYCH18, OST05] that one can exploit these caches to get insights about secret information in other processes reliably and efficiently. These attacks abuse the traces left in the caches by computations. Those can be information about the control flow, like taken branches or used functions. They can be read directly, like in Flush+Reload [YF13], by flushing a page and measure the time of the target load, or indirectly like Prime+Probe [TOS10] by using eviction sets to fill the cache and find evictions caused by victims memory load. The CPU caches are a deeply embedded part of the CPU, reflecting side effects as abused by Kocher et al. with the Spectre attack [KGG+18]. This allows many potential attacks and unspecified usage.
Ge et al. [GYCH18] give an overview of attacks and the effectiveness of countermeasurements. It is possible to rewrite the code to leave fewer traces, but it is error-prone and often creates overhead. Simply removing access to precise timers is not sufficient because other ways of measuring timing have been found. The most effective method of preventing cache attacks is cache partitioning, which separates processes' caches, as stated by Gras et al. [GRBG18].

2.3 Translation Lookaside Buffer (TLB)

Performing the PTW includes multiple memory lookups, because the page tables are stored in memory. These memory accesses stall the CPU. The TLB is a cache that solves this problem by storing recently resolved translations, and providing them with minimal latency as part of the MMU. The TLB is a set-associative cache with ways. The TLB hierarchy consists in modern Intel processors out of two levels. The L1 cache consists of separate

caches for data and instruction translations. The data TLB (dTLB) stores translations for data accesses, while the instruction TLB (iTLB) stores translations for instruction fetches. The L2 TLB can be a shared TLB (sTLB) that combines all the translations for instruction and data fetches, or it can also have separate data and instruction translations. Gras et al. [GRBG18] and Tatar et al. [TTGB22] have shown that the dTLB and sTLB are shared between hyperthreads in Intel processors. However, the iTLB is not shared.

Gras et al. [GRBG18] state that, unlike the data caches, the TLB has received less attention and is often not affected by some countermeasures like cache partitioning that prevent cache attacks. But the TLB can leak information about its entries in the same way that data and instruction caches do, enabling equal attacks at the page level, instead of the cache line level.

2.4 Translation Caches

To speed up the PTW, the CPU implements additional caches. These *translation caches* or paging-structure caches are placed within the different levels of the paging-structure. Schaik et al. [vSRG+17] investigated these by reverse engineering, as there is almost no information from the hardware manufacturers. These caches store partial address translations and provide the address of the page table to continue. This additional caching can reduce the timing difference between a TLB Hit and Miss, as the PTW can serve the translation faster, by skipping page-table lookups.

2.5 OoO-Execution

To improve performance and resource utilization, modern processors perform OoO-Execution. Instructions are not executed in the order they are specified; instead, the data dependencies of subsequent instructions are tracked to execute instructions as soon as their dependencies are resolved. It is guaranteed to produce the same result while providing a more efficient order to fill the CPU pipeline. This is since the CPU executes instructions while waiting for other instructions, resulting in a reduction of the time during which the Arithmetic Logic Unit (ALU) is idle.

The specialized hardware component, the Re-order-Buffer (ROB), collects the results of the computations and commits them in-order to the architectural state. This gives the illusion that all instructions were run in-order, but boosts the performance.

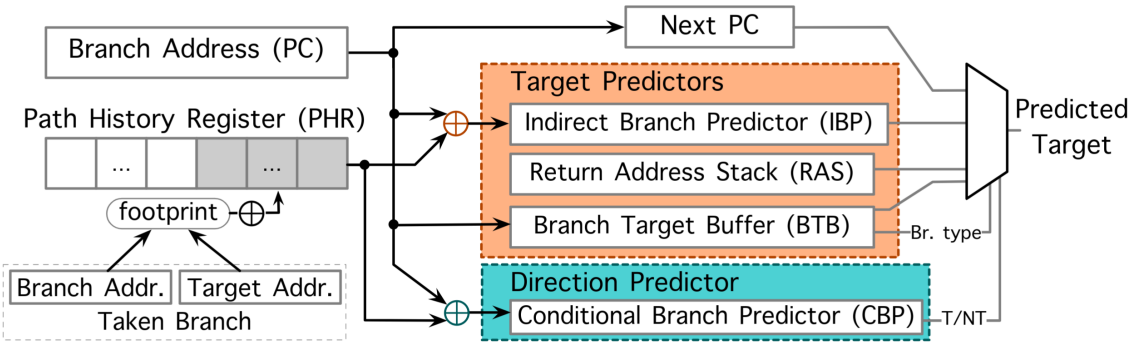


Figure 2.2: Branch Prediction Unit (BPU) in Modern CPUs (adapted from [YAC+24])

2.5.1 Speculative Execution

Speculative execution is an extension to the OoO-Execution. The control-flow of programs generally relies on different instructions. Some of these instructions, such as conditional branch instructions, often depend on previous instructions or memory lookups; the processor guesses the most likely branch and executes these instructions after the branch transiently. If the guess is correct, the CPU saves the time that would have been required to resolve the condition. If the guess is incorrect, the CPU can quickly continue with the correct branch, resulting in minimal time loss. This approach is more efficient because it saves more time than it costs.

This speculative execution is most known for conditional branches, but is used in many more cases, e. g. for returning from a called function or predicting the target of indirect branches. The branch target prediction is provided by the Branch Prediction Unit (BPU), which relies on multiple factors as depicted in Fig. 2.2. We changed the wording from Return-Address-Stack (RAS), as used in the Figure, to Return-Stack-Buffer (RSB). A call instruction pushes the corresponding return address onto the RSB. On return, the address at the top can be popped. This structure is fully independent of the Path History Register (PHR) and can be exploited unlimited times to open a transient window.

The results of the speculative execution are also put into the ROB to retire and therefore can be committed in-order. However, on a misspredicition, the results are not committed; instead, all instructions after the branch are flushed, and the pipeline is squashed. Nevertheless, this does not prevent side effects from occurring. Memory accesses in particular result in data being loaded into the cache. This means that one can detect side effects in code that was speculatively executed, even though it should be undetectable due to the ROB flush.

2.5.2 Transient Window

The instructions that are speculatively executed are called *transient* instructions. The *transient window* is the time frame until the CPU detects the preceding misspredict, executing *transient* instructions that lead to changes in the microarchitectural state.

2.6 ţWeird Machine (ţWM)

The concept of weird machines formalizes exploits as a theoretical model that side effects can be used to change the system's intended behavior to an unintended but observable behavior. This can be programmed with carefully crafted inputs.
A ţWM applies the concept of a weird machine to the side effects of the microarchitecture of the CPU. They use the interactions of microarchitectural components to operate through the normal program with the microarchitectural state to do computations. The state and computations are almost invisible to the CPU, as they use speculative execution that is never run in the normal control flow and interact only by side-effects. Wang et al. [WBW23] show that any transient primitive can be used, which makes them very generic.
The ţWM consists of the microarchitectural state that can be abstracted by weird registers that hold values, and weird gates that interact through side effects with the weird registers to perform logical operations. These weird gates can be composed into weird circuits to implement more complex functions.

2.7 Hybrid architecture

In newer generations, since Intel's Alder Lake (12th Generation), Intel combines two different kinds of cores in a processor, the P-Cores and the E-Cores. The different cores use slightly different architectures that are optimized for different goals. The P-Core aims to provide maximum performance, while the E-Core is designed for multithreaded background tasks and throughput. For this, it works on a lower frequency and achieves less power consumption.
On an Intel Meteorlake processor (13th Generation), the E-Cores are arranged in clusters of four, which requires about as much space as a P-Core. Both Cores differ in their OoO-engine, as the ROB of a P-Core stores 512 entries, and the ROB of an E-Core stores 256 entries. Comparing the caches of them shows that while the P-Cores have access to a private L2 cache, E-Cores share one L2 cache per cluster to be more efficient, but having more latency. The TLB of an E-Core is very large, compared to previous architectures, and larger than the TLB of the P-Cores, to prevent the more power-consuming PTWs.

2.8 LLVM

LLVM [LA04] is a modular framework for compilers. At its core, it provides its own intermediate representation language, LLVM-IR, which can be optimized and modified before being translated into machine code.

3 Translation Lookaside Buffer (TLB)

In this section, we analyze the TLB in detail, to prepare its usage as a binary state for weird machines. For this, it is necessary to understand the factors that influence the TLB timing behaviour and how to set or invalidate entries of the TLB.
We analyze the TLB details on two different architectures, Meteorlake and Kabylake. Meteorlake features Intel's new hybrid architecture with two different core architectures. Table 3.1 displays the basic information we can extract with `cpuid`.
To retrieve information about the TLB, we test two methods. The `cpuid` instruction provides detailed information about the processor's features. Information is arranged in leaves and subleaves, where each one is encoded into 128-bit information. In Intel processors, information about the caches, including the TLB, is provided in leaf 0x2. In recent processors, there is a new leaf 0x16 that supplies more detailed information about the TLB. While `cpuid` gives basic information to the TLB, there are implementation details, as the mapping function and replacement policy, that are not documented. With the tool by Tatar et al. [TTGB22], these details can be inferred via testing the TLB with eviction sets.
Although Tatar et al. [TTGB22] found that the number of ways provided by `cpuid` is incorrect for Kabylake L1 dTLB, using the `cpuid` instruction is very straightforward and provides mostly reliable information about the CPU.

3.1 Controlling the TLB

To abuse the TLB for state management, we need control over its entries that are the translation of a virtual address, its physical address. Inserting an entry into the TLB can be done similarly to data caches by accessing the address. This triggers a PTW and stores the entry, including the translation into both TLB levels. To invalidate entries is not as straightforward as for data caches. For data caches, there is the `clflush` instruction, which can be executed from the user level. It is difficult to use eviction sets when the goal is to invalidate only one address without changing the other entries lying in the same set of the TLB. On the other hand, there are different options to invalidate TLB entries with privileged access, as the Intel Developer Handbook lists [Int25a].
The **`invlpg`** instruction takes a virtual address and invalidates all entries in the TLB for the page number corresponding to the address. In addition to this, all entries in all paging-structure caches are invalidated, regardless of the address they correspond to. A **`MOV`**

Table 3.1: Info for the TLB extracted by `cpuid` and compared to the tool by Tatar et al. [TTGB22]. Running this tool on the E-Core results in a kernel panic, so we only rely on the information by `cpuid`, as investigating the E-Core in depth is out of scope for this thesis.

CPU		Meteorlake		Kabylake
		P-Core	E-Core	
L1 dTLB	Sets	16	48	16
	Associativity	6	full	4
	Mapping Function	linear	-	linear
L2 sTLB	Sets	128	512	128
	Associativity	16	6	12[1]
	Mapping Function	XOR	-	XOR

[1] `cpuid` instruction reports falsely 6

to CR3 also clears all TLB entries, depending on the CR4.PCIDE bit. The CR4.PCID bit enables or disables the PCID for processes for more efficient TLB flushes and is stored in the 12 least significant bits. **`invpcid`** invalidates a single entry, the complete TLB to a given pcid, or the complete TLB regardless of the PCID. Caches in the paging structure can also be flushed. **MOV to CR0** invalidates all TLB entries and all entries in paging-structure-caches. **MOV to CR4** invalidates all entries in paging-structure caches, thus invalidating TLB entries.

As all of these methods can only be used in kernel context, we implement a kernel module that listens to ioctl calls to execute the `invlpg` and `invpcid` instruction to invalidate a single entry or the full TLB.

Another way to invalidate an entry is to use eviction sets that evict the requested address. These eviction sets can be efficiently computed when knowing the mapping function of both TLB levels.

When accessing two neighboring page table entries via the page table walk, the second one can be faster, because the relevant page levels remain in the *translation caches*, described in Section 2.4. We consider these caches within this thesis to be out of scope. As we use the `invlpg` instruction to invalidate the TLB, the *translation caches* are also cleared, which reduces the impact on the timing behavior.

3.2 TLB threshold

To find a good threshold that distinguishes a TLB Hit and Miss, we developed a small tool. It does this by using eviction sets to evict an address and measuring the timing differences. While our tool does not find the optimal threshold directly, it gives a good indication for fine-tuning.
The tool measures multiple accesses to pages in different caching states. We want to use the entire TLB for the maximum number of possible entries, including both the L1 and L2 levels. The tool measures the access time for the L1 dTLB Hit, the sTLB Hit, i. e. dTLB Miss, L2 sTLB Miss and a full PTW with flushed *translation caches* of the page tables. The tool creates two disjoint sets of memory, one for the data access and one for individual eviction sets per page. For each TLB level, a target page is loaded, and then evicted until the TLB level to be tested is reached.
The tool then searches for the best threshold within this range based on a score per threshold. The score is created by testing some properties of that threshold.

1. The access time for an uncached page has to be longer than the threshold.
2. When transiently accessing a page within a large transient window, the access time has to be less than the threshold.
3. When transiently accessing a page within a too short window, the access time has to be more than the threshold.
4. When accessing a page and evicting it from L1 dTLB, the access time to it has to be shorter than the threshold.
5. One of two pages is selected at random for eviction from sTLB, while the other one is kept in L1 dTLB. The evicted page must have a longer access time than the threshold, while the page that is kept must have a shorter access time.
6. The middle page of five neighboring pages is used as the target. When accessing the surrounding pages, the target page needs a longer access time than the threshold, while the accessed pages need a shorter access time than the threshold

The timing behavior for an L2 sTLB Miss that is caused by eviction is much faster than that of the full PTW, because the *translation cache* can contain a partial translation after accessing the eviction set, while a full PTW has to go through all the page levels.
We use the range between an L2 sTLB Hit and Miss for candidates of the threshold. To find the best threshold directly is hard, due to the fact that other page accesses can influence the timing behavior of a page. The tool gives the best candidates, but often it is necessary to fine-tune the threshold additionally.

4 Weird Machines

This chapter gives an overview of state-of-the-art ţWMs. ţWMs abuse the microarchitecture in the CPU to hide computations and intermediate values, as only the result is visible to the architecture. For this thesis, we split the weird machine model into two main components, the weird state and the weird computation.

4.1 Weird State

Registers and the memory that are used in the CPU characterize the architectural state, as values are stored directly in the architecture of the CPU. Interacting with these is as straightforward as writing a simple program or assembly code, by using instructions provided by the CPU. On the other hand, the microarchitecture implicitly provides mechanisms to store data, which can be used as weird registers to form the weird state. Weird registers are used for computations in the context of the weird machine, storing either the input or the output.

When initializing a weird machine at first, the architectural state has to be translated to the weird state. After the weird machine has run some computations in the weird state, the values inside the weird state are translated back. This allows to outsource computations into weird machines whose computations are invisible to the architecture.

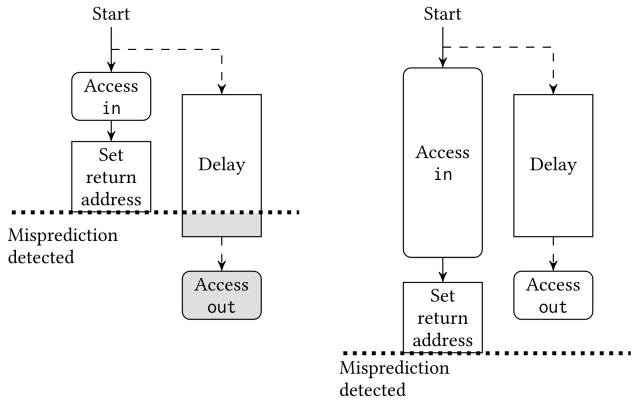
In recent work [KKC+23, WPWB24], it has been shown that the weird state can be efficiently implemented by abusing data caches. With data caches, it is possible to encode two values, true and false, by representing them as a cached and an uncached value. So each entry inside a cache can represent one bit. A weird register is, from the architectural view, just an address that points into the memory.

4.2 Weird Computation

The second part of a ţWM describes the side effects that interact with the weird state. We explain the concept of the gates and give an example of the current type of weird gates based on data caches. Finally, we discuss how weird gates are composed to create weird machines.

4.2.1 Weird Gates

To interact with the values stored in the weird registers, the weird machine uses weird gates, which are the basic blocks for creating weird machines. They work as logic gates to execute an arithmetic operation on input values to get the corresponding output. This is very similar to hardware gates that are the basic blocks within CPUs. The computation mechanism of the gates depends on the chosen weird state, which are data caches in the state-of-the-art.
To perform arithmetic operations, the weird gates abuse speculative execution and its transient window.
There are two types of gates: Inverting gates and Non-inverting gates.
Inverting gates are gates that invert the input value. Examples for this are NOT, NAND, and NOR. The architecture of these gates is based on the length of the transient window that is triggered by accessing the input register directly. So the window size depends on the access time and can be small or large.
Fig. 4.1b shows the code that forms a weird NOT-Gate. The two execution paths for the operation of the NOT gate that depend on the caching state of **in* are shown in Fig. 4.1a. It uses the misprediction of the branch and triggers a transient window. This is done by a call to a helper function that adjusts the return address based on the value of **in* that is known. The processor predicts the return address via the RSB while resolving **in* and speculates the execution after the return. The size of the window depends on the access time and therefore the caching state of **in*. At the end of the transient window, the output register is accessed. But this access is only possible if the transient window is long enough to execute all preceding instructions.
In contrast to this, **non-inverting gates**, e. g. AND, OR, and XOR have a fixed-size transient window size. Again, a call to a helper function adjusts the return address, but not depending on the cache latency of an input weird register, but on a long-running computation. As before the transient window is opened, when the speculative execution reaches the return and uses the value provided by the RSB. In this transient window, the input registers are accessed and used to access the output register. Because the *out* address depends on the inputs, the access to **out* depends on their cache latency. This kind of gate is very reliable and easy to create.
Fig. 4.2b shows example code for a weird AND-Gate that performs as visible in Fig. 4.2a. Because of the constant transient window size, it is possible to combine multiple non-inverting gates within one transient window to make the execution more efficient.
Wang et al. [WPWB24] state that composing inverting and non-inverting gates in one transient window does not work, because setting the window length to a specific value for a gate breaks the other one.

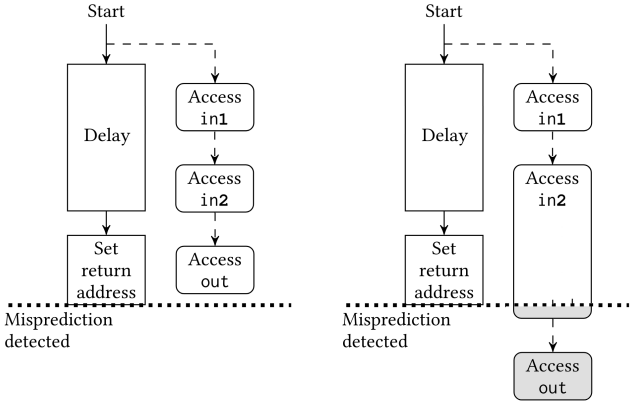


(a) Operation of the NOT gate, left when **in* is cached, right when it is not. Shaded instructions are never executed (not even transiently). Adapted from [HRY24]

(b) Code for the NOT gate

```
1 set_return_address(in[0]);
2 out[delay()];
3 return;
```

Figure 4.1: The NOT gate. **out* is not cached and set_return_addr adjusts the return address within the called function to point to line 3, skipping the access to **out*



(a) Operation of the AND gate. Left when both **in1*, **in2* are cached, right when one of them is not. Shaded instructions are never executed (not even transiently)

(b) Code for the AND gate

```
1 set_return_address(delay());
2 out[in1[in2[0]]];
3 return;
```

Figure 4.2: The AND gate. **out* is not cached and set_return_addr adjusts the return address within the call to point to line 3 and skips the access to **out*

Differential Encoding (DE)

Wang et al. [WPWB24] adopted a Differential Encoding (DE) that provides error detection and the possibility to use the gates more efficiently. This encodes each single bit into two weird registers. One representing the actual state, and one representing the negated state. Thus, an error can be detected if both have the same state. Within the gates, there is now the possibility to work with the real or negated value in the same window, without the need to call another gate for the NOT operation. Because AND and OR gates can be combined in the same transient window, with this Differential Encoding a Disjunktive Normal Form (DNF) can be applied.
An n-to-m gate resembles a gate with n input bits and m output bits. Wang et al. [WPWB24] state that the transient window size restricts the number of inputs that can be used. For the data caches and the tested CPUs, a 4-to-1 XOR gate was the limit per transient window. The n-to-1 XOR is well-suited to measure this limit, because it is the most complex arithmetic operation for the n-to-1 relationship of a gate.
In addition to this, Wang et al. [WPWB24] use an Error Correction that depends on the Differential Encoding and does an advanced majority voting at bit level of the output.

4.2.2 Transient window

Katzman et al. [KKC+23] used classic conditional branch training to trick the speculative execution. In addition to this, they presented an alternative not non-branch-based version, that has slightly less accuracy. To have a very reliable way that tricks the speculative execution, without using branches, the state-of-the-art uses the RSB. The transient window is triggered by making a call to a long-running function that will return, but changes the return address within. At this point, the CPU assumes to continue after the return directly behind the call and takes the return address from the RSB. The OoO-engine runs the code after the call in parallel to the function body. Because the long-running computation changes the return address on the stack within the call, the return finally jumps after the code that was executed ahead of time in parallel. Thus, the speculative execution results in a misprediction and resets after the calls return is reached.

Window size

The transient window size has the most influence on the accuracy of the gates. Wang et al. [WPWB24] state that the transient window size is optimal to be slightly shorter than the cache miss latency. If the window is too short, there is not enough time to reach the load operation on the output register. On the other hand, providing a window that is too large,

all loads can complete within the transient window, and therefore, the output is always true.

4.2.3 Creating WMs

Creating a weird machine is done by combining those weird gates into a weird circuit and wrapping them with translation layers that translate values from architectural state to the weird state and back. Combining gates is very straightforward, as a weird register holding an output value can be directly used as input to another weird gate. It is important to note that reading or using a weird register in a weird gate invalidates the value it holds, because the corresponding cache entry is loaded on the access. Therefore, a weird register can not be used twice. The inputs to a weird circuit are translated into the weird state as needed, instead of all at once. A cache entry is then invalidated or loaded when translating architectural state into the weird state to initialize the corresponding value. This is done just before a gate that uses this input value or constant. So in circuits, there are gates that have cache invalidations before and intermediate gates that only use weird registers, which were computed by previous gates.
We call it the width of a circuit, how many of the weird registers have to be held in parallel within the cache within a weird circuit. This does not necessarily scale with the number of wires, as a circuit can have many wires that lead directly into the next gate and therefore do not need to be held for a while, increasing the number of wires, but having a low width.
Wang et al. [WPWB24] created Flexo, a compiler that integrates C-code functions with weird gates, allowing for the inclusion of weird machines as small parts to which computations can be outsourced. This is done by extracting logical circuits from the C-code functions, and including them as weird circuits into the binary using the LLVM compiler framework, described in Section 2.8. This creates one weird machine for each weird function in the C code.
When running longer computations, e. g. multiple rounds of AES, it is necessary to split the weird function into smaller parts, e. g. the AES round functions. This method exposes some intermediate values to the architectural state, but adds the possibility for much larger functions based on weird machines.

5 Weird Machines on TLB

This chapter builds on top of the state-of-the-art weird machines and explains the concept for substituting the data cache-based concept with a TLB based one.

5.1 Weird State

We adopt the data cache-based weird state model of Wang et al., explained in the Chapter 4, to use the TLB. Since the TLB is a cache similar to the previously used data caches, it can be used accordingly.
We encode one-bit values into the TLB. Similar to the data cache, this yields a weird register. A cached entry is a 'true' value, uncached entries denote 'false'. As described in Section 2.3, the TLB entries are addresses of pages, so each weird register needs to point to a different virtual page. To change the value of a weird register to 'true' (i. e. 'cached') we can simply load the address. On a load, the virtual address is translated and put into the TLB.
To retrieve the value of a weird register, we only need to measure the access time to the address of the weird register that corresponds to the TLB entry. This access time depends strongly on the caching state.
For invalidating an TLB entry, we use a kernel module executing the `invlpg` instruction. This could be improved by using other invalidating techniques that do not require root access.
We set up the virtual pages for the TLB as described in Section 2.1 to remove the additional overhead. This ensures that the measured access time only depends on the TLB residency by reducing the impact of the data caches.

5.2 Weird Gates

To test this approach in an early version of the TLB-based weird gates, we adopt the code of Katzman et al. [KKC+23]. When implementing the NOT-Gate as shown in Fig. 4.1b, the transient window is too short, regardless of the input value, to reach the access on **out*. To enable the functionality, it is necessary to increase the duration of the TLB latency additionally. This can be done by a chain of arithmetic operations in addition to the load operation on **in* that forms another delay. This is shown in Listing 5.1.

Listing 5.1: The NOT gate. **out* is uncached and set_return_addr adjusts the return address within the call to point to line 3 and skips the access to **out*. The access on **in* is also delayed to increase the timing effect

```
1 set_return_address(another_delay(in[0]));
2 out[delay()];
3 return;
```

5.2.1 Transient Window

Prior work using data caches used DIV instructions to open the transient window. DIV instructions are one of the instructions that use the most cycles for completion. Because the TLB has much faster access times, a shorter window is needed. DIV instructions are too coarse-grained, as only very few are required (e. g. 2), leading to a window size that is not optimal. Creating the sequence via some alternative instructions leads to more fine-grained control over the window size and can increase the accuracy of the gates. We tested MUL instructions in this context.
We built a program to examine the relationship between different instructions on the transient window size. The program measures the window size via load operations on uncached addresses that have been completed within the transient window. This enables a fine-grained evaluation of the window length. We use an instruction chain of dependent instructions of a specific kind to trigger the transient window. Using this method, one can exactly find the window sizes corresponding to a specific timing behaviour, e g. exactly one load finished. As the TLB-weird state is very time sensitive, it is important to find an optimal window size to increase the accuracy of weird gates.

5.2.2 Differential Encoding

Using the differential encoding with the TLB-based approach, we find the maximal gate size is limited to a 3-1 XOR, which allows for all 3-1 gates to be built. This reduces the maximal inputs for gates to 3, leading to more gates per circuit than with the data cache-based approach.

5.3 Flexing TLB

Several adjustments are needed to support the TLB-based weird state while preserving the functionality of the easy-to-use Flexo compiler. We refer to the original Flexo compiler that uses data caches as Flexo using data caches (Flexo-DCache). We call our modification Flexo using the TLB (Flexo-TLB).

Weird Registers

Originally, Flexo-DCache uses data cache lines with 64-byte granularity. To prevent interaction with the prefetcher, an offset is between each cache line of the weird registers. The TLB stores entries at page-size granularity and requires only one physical page-frame to be mapped into memory for all registers, one virtual page each. Since the offset between each register is now the page size (i. e. 4 KB), an additional offset to prevent prefetching is unnecessary. This is because the prefetcher assumes the usage of local data lines and therefore prefetches only the next lines into the cache.
Using the Flexo-TLB we can save space in memory, compared to Flexo-DCache. Flexo-DCache needs only 64 bytes per Weird Register, which is the smallest unit to access the memory. But in addition to this, they need to allocate space between those weird registers to trick the prefetcher. This padding is mostly about 500 bytes for small circuits and about 1000 bytes for larger ones.
In contrast, Flexo-TLB only needs one full page for all circuit sizes, regardless of the number of weird registers. The physical page frame exists only once in the physical memory, though it is referenced by many virtual pages, limiting the memory usage to the page size, e. g. 4 KB. The number of virtual pages, each for one weird register, increases the number of PTEs that are needed, possibly resulting in more page tables.
Flexo-DCache offers two memory allocation methods. The first one is to allocate memory directly on the stack, which is very fast but space-limited. The second method uses *mmap* to allocate space on the heap. As the heap is much larger than the stack, this method is less limited in size.
To reduce the overhead of the memory allocation, the allocation is only done once at the startup of the program, such that memory does not have to be allocated multiple times throughout the program.

TLB Hit-Threshold

The threshold to distinguish between Hits and Misses has to be more precise for working with the TLB as the timing differences are smaller than for data caches. The tool we built to detect a good threshold may also be included directly into the compiler toolchain.

Timing Function

When using a cache as a weird state, it is necessary to measure the access time for a given address to check if the load is above a threshold or below. Flexo-DCache provides a timing function, shown in Listing 5.2, that takes an address and returns the latency of that access in cycles as measured by the time stamp counter in the CPU.

```
1  rdtscp              ; wait for previous instructions + loads
2  shl $32, %rdx
3  mov %rdx, %rsi
4  or %eax, %esi       ; store timestamp in rsi
5
6  mov (addr), %al     ; access the given address
7
8  rdtscp              ; wait for previous instructions + loads
9  shl $32, %rdx
10 or %eax, %edx
11 sub %rsi, %rdx      ; get the timing difference
12 mov %rdx, %rax
```

Listing 5.2: Original timer function of Flexo-DCache

While Flexo-DCache uses a straightforward approach using **rdtscp** to measure access times, the same measuring routine does not yield precise results when measuring the TLB caching state. This could not be fixed by only adjusting the threshold accordingly. We investigated the timing function to get a full picture of the timing behavior.
Listing 5.2 shows the assembly of the timing function in Flexo-DCache.
Lines 1-4 use **rdtscp** to get the timestamp and store it in %rsi. Then in line 6 the MOV instruction to access the data at *addr*, leading to the latency based on its cache residency. Lines 8-12 get the timestamp after the load operation and compute the difference between the two timestamps.

It is important to use fences that prevent the OoO-execution. This is done with the `rdtscp` instruction. In the Intel manual the `rdtscp` instruction is described as follows.
`RDTSCP`:

> Reads the current value of the processors time-stamp counter (a 64-bit MSR) into the EDX:EAX registers.[...]
> The `RDTSCP` instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible. But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. [Int25b]

This means that the instructions are fenced. Nevertheless, the OoO-execution can impact the result. By using the `rdtscp` instruction, the OoO-execution waits for previous loads and subsequent instructions, but can start line 6 directly before the read of the timestamp has finished, as the lines are fully independent. But this behavior can lead to lower values and less accurate results. The `rdtscp` in line 8 then waits for all preceding instructions

```
1  rdtscp                          ; wait for previous instructions + loads
2  shl $32, %rdx
3  mov %rdx, %rsi
4  or %eax, %esi                   ; store timestamp in rsi
5
6  cmp    $0xbaaaaad, %rsi         ; Create dependency of addr on rdtscp result
7  sete   %al
8  movzbl %al, %eax
9  or     %rax, addr
10
11 mov (addr), %al                 ; access the given address
12
13 rdtscp                          ; wait for previous instructions + loads
14 shl $32, %rdx
15 or %eax, %edx
16 sub %rsi, %rdx                  ; get the timing difference
17 mov %rdx, %rax
```

Listing 5.3: Timer function with dependency, used in Flexo-TLB

and has no side effects on the result.

To fix this behavior, we create a slightly modified version that uses dependencies to serialize the code. Listing 5.3 is identical to Listing 5.2, except for lines 6-9.

In lines 6-9, the timestamp value of `rdtscp` is used as a dependency for the load of *addr*. This forces the OoO-execution to wait for the read to complete before the load instruction in line 11 is executed.

We tested Flexo-DCache with both variants, the original measuring routine and the optimized variant, having nearly no difference in their accuracy. But for Flexo-TLB the modified measuring routine increases the accuracy by $> 30\%$. This is because the TLB leads to much smaller timing differences between a hit and a miss, while the difference for data cache accesses (Cache hit) and a memory access (Cache miss) is very high. Thus, Flexo-TLB requires an accurate timing function such as our modified version.

6 Experiments

In this chapter, we show the results of our experiments. All experiments in this thesis are performed on a modern Intel CPU with Meteorlake micro-architecture and on an older Kabylake CPU, which is based on the Skylake architecture, that was used in previous research by Wang et al. [WPWB24].

6.1 Processor TLB

We collect the TLB threshold for distinguishing between Hit and Miss by using our custom tool and fine-tuning it. The results are shown in Table 6.1. These values work on the testing machines; however, they are not necessarily universally applicable.
We use this threshold for all subsequent experiments.

6.2 Transient Window

When working with weird gates, each gate triggers a transient window. The accuracy of a weird gate strongly depends on the size of the transient window, because a too short window results in all outputs of weird registers being uncached, and a too long window results in all of them being cached. The TLB is more time sensitive than the data caches, leading to the question of whether a shorter instruction than used before can help to find the best window size and which property determines the best transient window size.
We investigate the transient window size in the context of TLB loads that finished within the transient window. In these experiments, we trigger the speculative execution the same way, as described in Section 4.2.2. The instructions in the instruction chain that determine the transient window size are all either DIV or MUL instructions. Within the transient window, we execute instructions that access randomly selected pages from an allocated block of pages that have invalidated TLB-entries, each triggering a PTW. Within one transient window, up to 14 loads can be started. After the transient window is squashed, it is measured how many of the PTWs have completed within the transient window. This is done by checking the caching state of those addresses afterwards.
The experiment is conducted with a growing length of the instruction chain.
We run this experiment with different instructions on the Meteorlake P-Core to see if there is an advantage in using the MUL instruction. In the following, we evaluate Kabylake and

Table 6.1: Best thresholds and the duration of a PTW in cycles for the different CPUs

CPU	Turbo	Threshold	PTW
Kabylake	✗	60	90
	✓	40	58
Meteorlake E-Core	✗	260	410
	✓	-	96
Meteorlake P-Core	✗	121	184
	✓	40	58

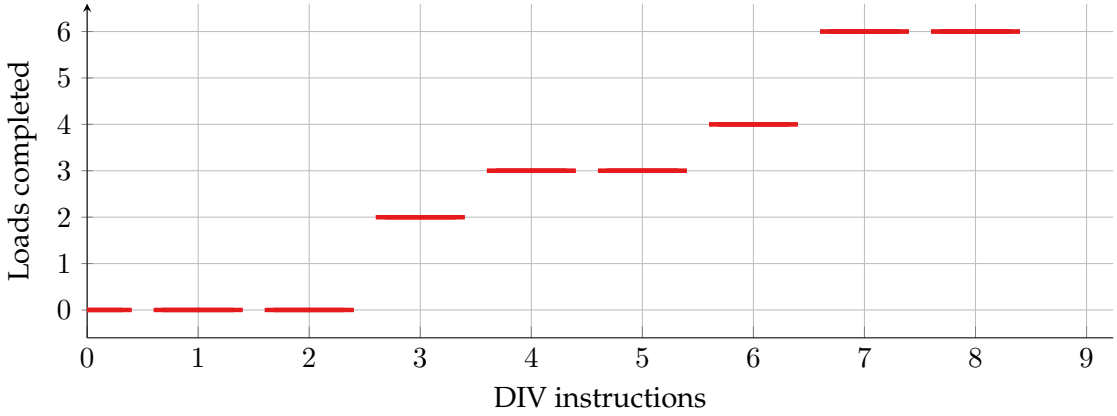


Figure 6.1: Boxplots that show the number of loads completed within the transient window that was triggered by the instruction chain for a given number of DIVs on Meteorlake P-Core

Meteorlake E-Core in detail and compare them against each other.

6.2.1 Experiments on Meteorlake P-Core

Fig. 6.1 shows the transient window size by completed loads when using DIV instructions using boxplots. As the boxplots are only visible as lines, the measured values are all exactly the same. The Graph shows that by increasing the number of DIV instructions, the number of loads either stays the same or rises. This increase is usually by one, but sometimes by two. This implies that, since some values are skipped, the resolution of the DIV instruction is too coarse-grained, and some values cannot be reached.

Fig. 6.2 shows the number of completed loads per number of DIVs while running in turbo mode. When compared to Fig. 6.1, we see that enabling the turbo mode does lead to a very similar course, with a faster increasing number of loads. But the number of loads also sometimes increases for the first loads by two.

Fig. 6.3 shows the transient window size of completed loads when using MUL instruc-

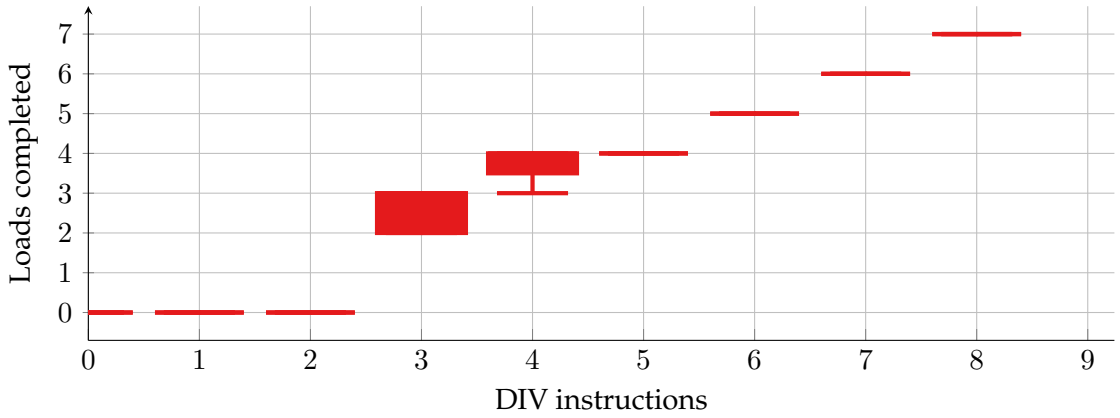


Figure 6.2: Loads completed within transient window triggered by DIV instruction chain on Meteorlake P-Core with turbo enabled

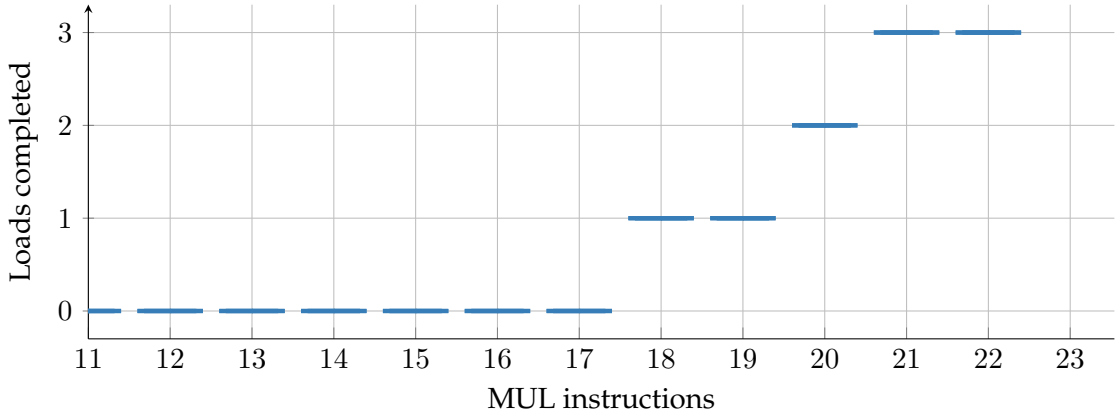


Figure 6.3: Loads completed within transient window triggered by MUL instruction chain on Meteorlake P-Core

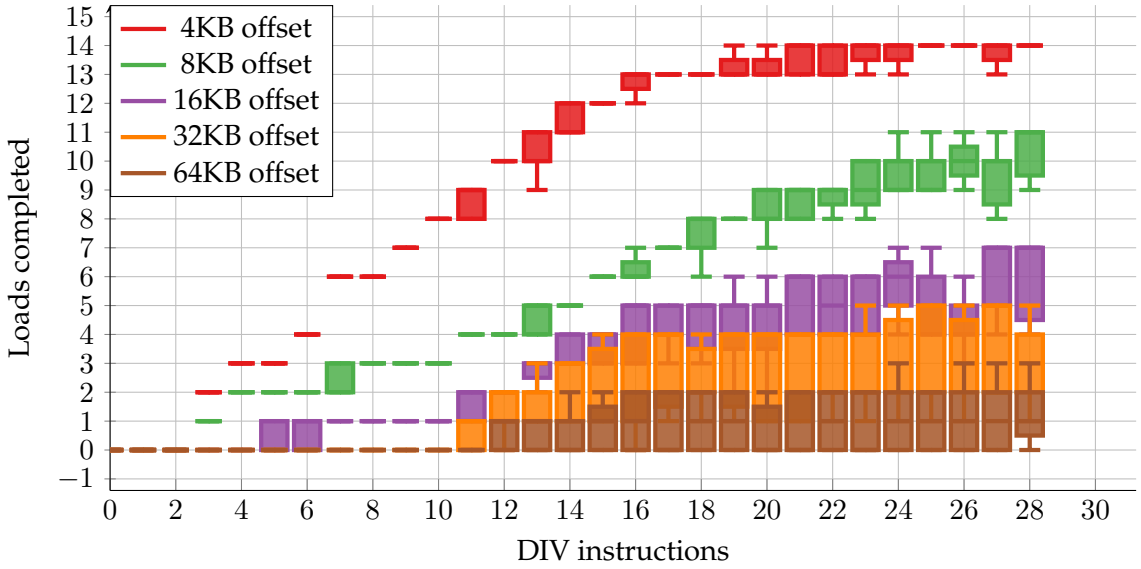


Figure 6.4: Loads completed within transient window triggered by DIV instruction chain on Meteorlake P-Core for different offsets between allocated pages.

tions. When comparing it to Fig. 6.1, we see that the MUL instructions have less impact on the window size. In particular, the number of loads is no longer increased by two because the MUL instruction requires fewer cycles, resulting in smaller steps. This demonstrates that the MUL instruction can be used to reach all numbers of loads, resulting in a more fine-grained instruction chain.

Subsequent Page-Table-Walk (PTW)

We note that in Fig. 6.3 the number of loads increases linearly, but only after several MUL instructions for which the number of loads remains zero. We assume that this is due to the translation caches of the paging structure. The first PTW requires a significant amount of time, and places partial translations into the translation caches of the page tables, resulting in faster subsequent PTWs. We support this hypothesis by testing the progression over DIV instruction for the pages that are not allocated in one big block, but each page has an offset to the other pages, leading to less caching in the translation caches.
Fig. 6.4 shows the number of loads for this allocation with offsets. We see that with increasing offset, the number of completed loads shrinks. In particular, the number of instructions until one load happens increases, due to the page tables are not cached in the data caches.
We measure the accuracy for different transient window sizes for both instruction types. These results are shown in Fig. 6.5. We see that the usage of both instructions can lead

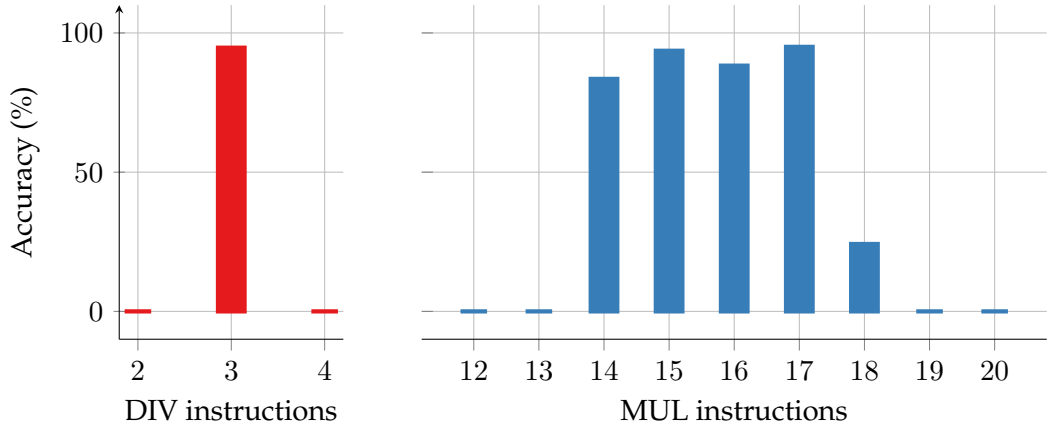


Figure 6.5: Accuracy for Flexo-TLB executing the 4-bit ALU for different transient window sizes, defined by a DIV/MUL instruction chain on Meteorlake P-Core

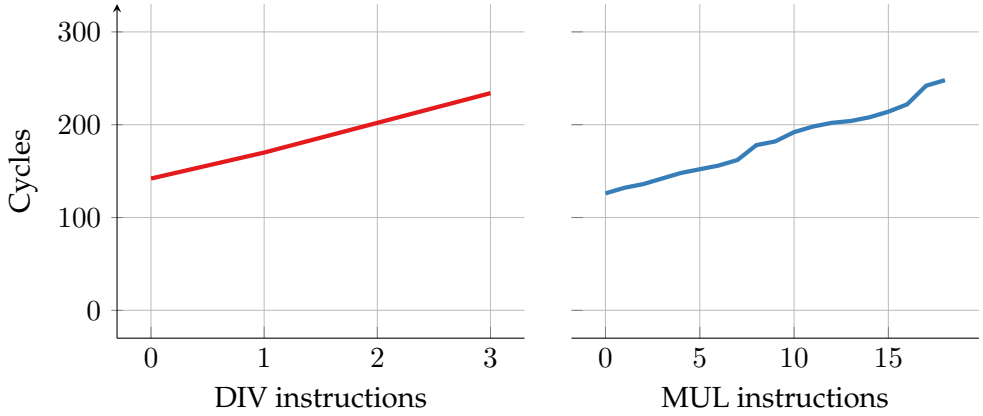


Figure 6.6: Loads completed within transient window triggered by DIV/MUL instruction chain on Meteorlake P-Core

to similar accuracies, but the MUL-based instruction chain has multiple different window lengths other than zero, while the DIV instruction only has one peak. This supports the assumption that using MUL instructions enables more precise finetuning for the transient window size. The best accuracy is directly before the step from zero to one completed load, corresponding to a bit less than a Miss latency as stated by Wang et al. [WPWB24].

In Fig. 6.6, one can see that the cycles needed for the execution of the instruction chain that opens the transient window grow linearly with the number of instructions. When comparing the duration on the Meteorlake P-Core for DIV instructions with the duration for MUL instructions, we see that both figures are very similar in their progression, but for each DIV, around 5 MUL instructions are needed for a similar number of cycles. We see that for the best window sizes e. g. 3 DIV or 17 MUL instructions have an equal timing

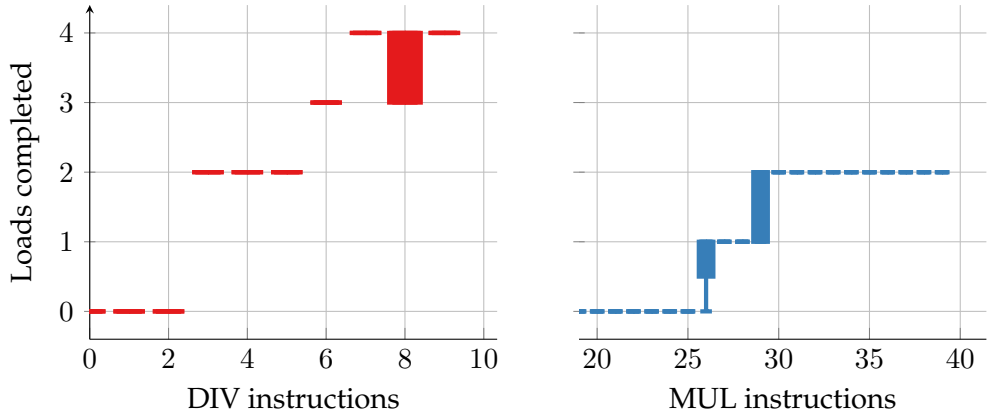


Figure 6.7: Loads completed within transient window triggered by DIV/MUL instruction chain on Kabylake

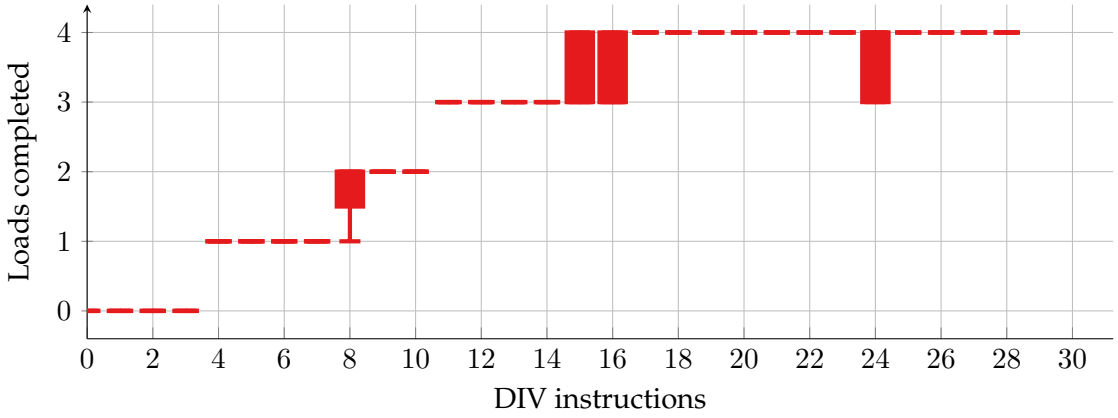


Figure 6.8: Loads completed within transient window triggered by DIV instruction chain on Meteorlake E-Core

behavior. When comparing the number of cycles with the thresholds given in Table 6.1, we see that the transient window size is much larger than the thresholds and even larger than a full PTW. This can be explained by the fact that there is some overhead involved in executing the instructions transiently.

6.2.2 Experiments on Kabylake and E-Core

When we compare the transient window size of Kabylake, shown in Fig. 6.7, with that of Meteorlakes P-Core in Fig. 6.1, we see that Kabylake is very similar in the progression for the DIV instructions and also increasing by two. But we can see that, as for the Meteorlake P-Core, this is solved when using MUL instructions.
When measuring the transient window also for the Meteorlake E-Core, shown in Fig. 6.8,

we see that this does not have the problem of increasing by two loads. But the faster MUL instructions also lead to smaller steps on the Meteorlake E-Core.
We note that both Kabylake and Meteorlake E-Core provide much more fine-grained control when using MUL instructions, like on the Meteorlake P-Core.

6.2.3 Maximum Transient Window Size

Table 6.2: Maximum TLB-loads per CPU

CPU	Maximum Loads DIV	Maximum Loads MUL
Kabylake	4	4
Meteorlake E-Core	4	2
Meteorlake P-Core	$\geq$14	$\geq$14

To investigate the maximum transient window size with respect to the number of PTWs completing within, we execute the same experiment as above (Section 6.2) until the number of PTWs does not increase further. The results are shown in Table 6.2. The Kabylake and Meteorlake E-Core reach a maximum number of loads, while the Meteorlake P-Core exceeds the measurement limit. This demonstrates the significant architectural differences between the older Kabylake architecture and the power-aware E-Core and the modern Meteorlake P-Core architecture. Testing the MUL instruction on the e-core shows that the limit is much lower than that of the DIV instruction chain. This is because MUL instructions require fewer cycles to execute and fill the reorder buffer to its limit without consuming enough time to enable more loads within the transient window. Unlike the Kabylake, the Meteorlake E-Core reaches this limit for the MUL instruction because it needs fewer cycles for MUL instructions than Kabylake. The performance difference of Kabylake and Alderlakes E-Core, which is very similar to Meteorlakes E-Core, can be seen in the uops.info tool by Abel et al. [AR19].

6.3 Flexing TLB

To compare Flexo-TLB with Flexo-DCache, we measure the performance of Flexo-DCache on Meteorlake using the reproduce scripts and example circuits in the Flexo repository by Wang et al. [WPWB24].
For simple gates, the results of Flexo-TLB are very promising on the Meteorlake P-Core, as shown in Table 6.3. The strongly reduced accuracy for the 4-1 XOR indicates that the maximum number of inputs per single gate is limited to three. This will lead to a higher

Table 6.3: Results for the gates of Flexo-TLB using MUL instructions with turbo disabled on Meteorlake P-Core.

Gate	Accuracy	Runtime
AND	99.90%	2.62
OR	99.10%	2.59
NOT	99.80%	2.12
NAND	99.95%	2.62
XOR	99.20%	2.62
3-1 XOR	98.20%	3.26
4-1 XOR	38.30%	3.86

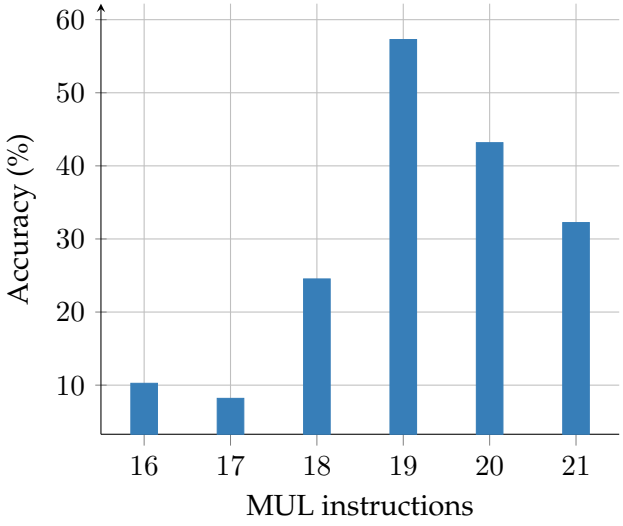


Figure 6.9: 4-1 XOR Accuracy for MUL instruction chain on Meteorlake P-Core

number of wires and gates for composed circuits, as more gates are needed to map the same logical function.

Fig. 6.9 shows the accuracy of the XOR4 gate for larger window sizes, so the accuracy could be increased with a larger window. The 4-1 XOR needs a larger transient window, because each constraint needs 4 Hit thresholds. These cumulate to a wider size than one Miss. But when adjusting the transient window size above a single Miss threshold, e. g. 18 MUL instructions, it is too large for the other gates and causes them to collapse, producing an invalid result.

Table 6.4: Results for the Flexo-DCache using DIV instructions with turbo disabled on Meteorlake P-Core

		4-Bit ALU		**SHA-1 (1 round)**		**AES (1 round)**		**Simon (1 round)**	
		Wires 62	Gates 32	Wires 1061	Gates 544	Wires 4990	Gates 2524	Wires 8288	Gates 4322
Configuration	Error Correction (EC)	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime
Original	✗	97.70%	39.52	0.60%	870.97	11.75%	3664.12	32.20%	4255.63
	✓	100.00%	40.96	100.00%	6389.07	100.00%	15965.13	99.90%	11127.69
New timer	✗	97.60%	38.33	0.60%	850.37	10.10%	3919.29	13.10%	4109.43
	✓	100.00%	40.19	100.00%	6293.92	100.00%	17273.57	99.90%	11307.40
Preallocation + ioctl	✗	97.30%	76.91	0.50%	1454.75	20.65%	6786.32	33.45%	10447.00
	✓	100.00%	79.89	100.00%	12219.35	100.00%	20799.14	99.90%	28128.67
Preallocation + ioctl + max 3 inputs	✗	97.50%	95.18	1.20%	1594.89	25.80%	8024.75	25.80%	9814.36
	✓	100.00%	95.81	100.00%	11491.51	100.00%	24669.22	99.90%	31139.50

Table 6.5: Results for the Flexo-TLB using MUL instructions with turbo disabled on Meteorlake P-Core

		4-Bit ALU		**SHA-1 (1 round)**		**AES (1 round)**		**Simon (1 round)**	
		Wires 76	Gates 41	Wires 1172	Gates 653	Wires 6114	Gates 3440	Wires 7490	Gates 4240
Configuration	EC	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime
Flexo-DCache timer	✗	66.50%	84.17	-	-	-	-	-	-
	✓	99.70%	130.76	-	-	-	-	-	-
New timer	✗	96.50%	84.35	0.00%	1244.23	0.00%	6787.69	0.00%	8377.67
	✓	99.90%	87.59	0.00%[1]	280104984.88[1]	-[2]	-[2]	-[2]	-[2]

[1] Due to its long runtime, we run the experiments with fewer iterations
[2] Due to its long runtime, the experiment was aborted

6.3.1 Comparing circuits

For the Flexo-DCache we run different configurations, as shown in Table 6.4. The results show that the original Flexo-DCache gives good values comparable to those measured by Wang et al. [WPWB24]. When using the new timing function, we see, there is a negligible difference to the original Flexo-DCache.
In Table 6.5 for results of Flexo-TLB, we see that all circuits larger than the 4-bit ALU give no results.
For the timing function of Flexo-DCache, the threshold was adjusted for best accuracy. When we compare the results for the 4-bit ALU, we see that the new timing function has a significant impact on the results. This is due to the very short timing difference for TLB hits and misses, so the inaccuracy of the timing function has a significant influence on determining the cache residency, in contrast to the data caches used Flexo-DCache.
Comparing this to values for Flexo-TLB with CPU in turbo mode, shown in Table 6.6, we see that the impact of the timing function is increased much more.

Table 6.6: Results for the Flexo-TLB using MUL instructions with turbo enabled on Meteorlake P-Core. SHA1, AES and Simon are not included, as they have zero accuracy

		4-Bit ALU	
		Wires 76	Gates 41
Configuration	EC	Accuracy	Runtime
Flexo-DCache timer	✗	54.75%	26.28
	✓	100.00%	47.38
New timer	✗	97.00%	26.28
	✓	99.90%	31.59

When running Flexo with the given example code, many complex circuits need more weird registers than the TLB has entries. Even though not all of them are used at the same time, this reduces the accuracy to 0%, as too large a proportion of the TLB would need to be protected from eviction. The larger SHA1 circuit produces 0% accuracy with Error Correction, but has a very long runtime. The other large circuits perform similarly with 0% accuracy, but the experiments have too few iterations to prove the results statistically. Comparing only the smaller circuits, we can see a similar performance for Flexo-TLB. and Flexo-DCache.

In addition to this, we created a bigger version of the 4-bit ALU that is tested for different Bit sizes. The ALU supports the following operations: ADD, SUB, AND, OR, XOR, shift left, shift right, NOT. The ALU is tested evenly on all operations with random inputs.

The results are shown in Table 6.7. The accuracy shrinks with larger circuit size, even with Error Correction enabled. But even for larger weird circuits, it provides results above 50%. This shows that the accuracy does not depend on the number of wires, but on the number of weird registers that are used in parallel.

Fig. 6.10 shows the best window sizes in MUL instructions for the different ALU bit-sizes. We note that they increase with circuit size, which can be explained by the increasing number of weird registers that lead to different timing due to more different addresses, as explained in Section 6.2.1.

Table 6.8 shows that Flexo-DCache provides not perfect, but good results for such large circuits.

When comparing the time between Flexo-DCache and the Flexo-TLB, as shown in Table 6.4 and Table 6.5, Flexo-TLB Version is much slower. We assume that is due to the ioctl call to the kernel module that flushes the TLB entries.

Table 6.7: Results for the Flexo-TLB using MUL instructions for ALUs of different bit-sizes with turbo disabled on Meteorlake P-Core

	8-Bit ALU		**9-Bit ALU**		**10-Bit ALU**		**12-Bit ALU**	
	Wires 507	Gates 316	Wires 602	Gates 381	Wires 675	Gates 427	Wires 817	Gates 517
EC	Acc.	Runtime	Acc.	Runtime	Acc.	Runtime	Acc.	Runtime
✗	47.70%	642.31	5.52%	761.18	0.10%	870.42	0.00%	1050.43
✓	99.90%	1243.31	99.33%	2898.14	52.24%	8979.19	96.10%	6041.59
	14-Bit ALU		**16-Bit ALU**		**24-Bit ALU**		**32-Bit ALU**	
	Wires 976	Gates 615	Wires 1111	Gates 701	Wires 1760	Gates 1120	Wires 2404	Gates 1521
EC	Acc.	Runtime	Acc.	Runtime	Acc.	Runtime	Acc.	Runtime
✗	0.00%	1267.35	0.00%	1426.46	0.00%	2288.14	0.00%	3123.32
✓	97.05%[1]	9275.53[1]	77.53%[1]	28711.68[1]	76.53%[1]	90139.56[1]	58.19%[1]	246514.56[1]

[1] The runtime of this circuit is very long, leading to fewer iterations for this experiment

Flexo-DCache uses the stack to allocate memory and does this on every run of a weird machine within the program. For comparison, we adjusted Flexo-DCache to pre-allocate the weird memory using mmap like Flexo-TLB does. In addition to this, Flexo-DCache is adjusted to do the `clflush` instruction in combination with an `ioctl` call to deliver comparable timing behavior for the flush of an entry. In Table 6.4 we see that the version with preallocation and ioctl is again faster than Flexo-TLB.
To test the performance difference for the same circuit sizes, we adjust Flexo-DCache to also use only 3 inputs per gate, like Flexo-TLB does. In this comparison, Flexo-TLB is faster, which is due to a lower penalty on TLB-Miss compared to the data caches miss. The performance advantage is not as large as possible, because of the more complex `invlpg` instruction that needs more time than the `clflush` instruction.

6.3.2 Performance on Kabylake and E-Core

When investigating the results for the basic gates on Kabylake, shown in Table 6.9, we see that the gates themselves have low accuracies. But we note that there is a large difference between the results for MUL and DIV instructions. We see that using the DIV instructions does not give the possibility to fine-tune the transient window fine-grained enough to give accuracies as good as the faster MUL instruction, as predicted in Section 6.2.1.
Table 6.10 shows the results for Flexo-TLB run on Kabylake. The simple ALU shows low

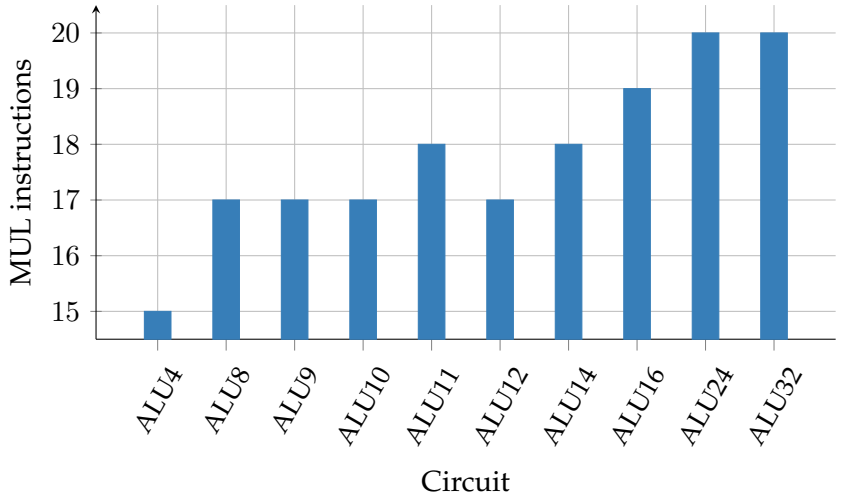


Figure 6.10: The best number of MUL instructions to trigger the transient window for different ALU circuits on Meteorlake P-Core

Table 6.8: Results for the Flexo-TLB using MUL instructions for ALUs of different bit-sizes with turbo disabled on Meteorlake P-Core

	24-Bit ALU		**32-Bit ALU**	
	Wires	Gates	Wires	
	1462	861	2027	1194
EC	Acc.	Runtime	Acc.	Runtime
✗	30.95%	1428.76	38.76%	1993.11
✓	88.76%	3508.95	77.24%	405.88

accuracy even with EC in-place, due to composing the gates with their low accuracy results in even lower accuracy.

Table 6.11 shows the results of testing Flexo-TLB on the Meteorlake E-Core with a small transient window, because it gives the best accuracy and a large transient window. Even though the accuracies look promising for the small transient window, they reflect the accuracy for always guessing zero, showing that the window is too small. Using the best larger window does lead to almost zero accuracy. We note that Flexo-TLB does not work on the Meteorlake E-Core.

Table 6.9: Results for the gates of Flexo-TLB using DIV or MUL instructions with turbo disabled on Kabylake, having Differential Encoding (DE) enabled or disabled

		AND		**OR**		**NOT**		**NAND**		**XOR**	
Instr.	DE	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime
DIV	✗	6.60%	6.11	7.00%	6.11	99.60%	4.12	6.50%	6.11	0.30%	6.11
	✓	7.10%	4.14	6.90%	4.16	99.60%	3.14	6.00%	4.14	0.30%	4.14
MUL	✗	48.60%	6.12	61.70%	6.12	99.60%	4.12	35.70%	6.12	60.20%	6.12
	✓	48.60%	4.15	61.60%	4.15	99.70%	3.14	36.10%	4.15	60.70%	4.16

Table 6.10: Results for the Flexo-TLB using MUL instructions with turbo disabled on Kabylake

	4-Bit ALU		**4-Bit ALU 2-1 gates**	
	Wires	Gates	Wires	Gates
	76	41	104	68
EC	Accuracy	Runtime	Accuracy	Runtime
✗	0%	153.21	0%	212.36
✓	2.10%	50124.69	2.50%	3363.44

Table 6.11: Results for the gates of Flexo-TLB using MUL instructions with turbo disabled on Meteorlake E-Core. Using a small and a large transient window. Differential Encoding (DE) enabled, but having equal performance as disabled.

	Small Window		**Large Window**	
Gate	Accuracy	Runtime	Accuracy	Runtime
AND	73.00%	4.73	2.67%	4.72
OR	26.20%	6.83	2.67%	4.71
NOT	48.80%	4.69	2.57%	3.62
NAND	24.40%	6.82	2.57%	4.72
XOR	48.60%	6.87	2.67%	4.71

7 Discussion

In this chapter, we examine the results of the experiments of Chapter 6 to evaluate the TLB based approach. First, we discuss the timing behavior of PTWs in the transient window. Then we go over the advantages and disadvantages of the new TLB-based approach, closing with some words about the detection of weird machines.

7.1 Transient Window

Section 6.2.1 shows that to trigger a transient window in addition to the DIV instruction, the MUL instruction can be used. We note that the best transient window size can be found by testing the number of completed loads in the transient window, selecting the highest number of instructions that does not enable a full PTW. The tests show that the performance on meteorlake P-Core is mostly equal for both instructions, MUL and DIV. As the MUL instruction is faster, it provides smaller steps that can be used for finetuning, if the DIV instruction provides less accurate results on different processors. The DIV instruction just provides one result that is indeed comparable to the MUL results. But this could be very different on other CPUs, as we see for Kabylake there is an advantage when using the MUL instruction. We use the MUL instructions in Flexo-TLB and get promising results that are best for a specific number of MUL instructions, e. g. 17, which is indicated by Fig. 6.3.
When comparing Fig. 6.6 and Table 6.1 we see, that the number of instructions for a full PTW within the transient window does not map directly to the duration of a PTW and can not be directly calculated for a known instruction length, as there is some overhead involved before executing the transient window.
Our experiments in Section 6.2.1 show that allocating pages with an offset instead of continuously, changes the timing behavior for subsequent PTWs and influences the number of loads that complete within the transient window. The number of loads that complete is lower when allocating pages with offsets between them. This is due to the locality of the addresses of the pages that have been allocated. Translating the virtual address results in partial translations being placed into the translation caches and faster subsequent translations for neighboring pages, leading to more completed loads. Allocating the pages with a larger offset results in differences in the page tables that resolve higher levels of the addresses, resulting in more translation time, as a full PTW is necessary. The observed effect

of fewer finished loads could also be strengthened by the fact that other page tables now have to be loaded, that are not necessarily in a fast data cache level as before.

We also see in Fig. 6.10 that with a larger circuit size, the best window size increases for Flexo-TLB. We explain this with the fact that the allocated pages, because of the higher number of weird register, span a larger address range. Because of this they differ more in their addresses, leading to longer timing behavior for the PTW, due to the same explanation as before.

We use the `invlpg` instruction within the weird machines, which flushes the translation caches, leading to a full PTW afterwards. We note that the runtime for a full PTW increases due to more page tables that are in higher-level data caches. On the other hand, the runtime for a partial PTW also increases due to the less matching partial translations in the translation caches. We see this effects in the increase of the best transient window size. We assume that the timing behavior of a full and partial PTW increases unevenly, resulting in a difference for the best transient window size for subsequent gates with and without an `invlpg` instruction before. This timing difference within the PTW itself could decrease the overall accuracy.

This raises the question of whether the accuracy could be increased by using a wider allocation with offsets between pages and not flushing the translation caches, to stabilize the timing behavior of the PTW, as this leads to fewer cached partial translations. Using a wider allocation with offsets could also increase the time for the PTW and therefore make it easier to differentiate between a TLB Hit and a Miss.

The problem with this approach could be that the different page tables also have to be loaded from the data cache. When many different page tables are used to prevent caching of partial translations, the page table itself is not necessarily in the data caches anymore, resulting in an increased overhead for loading the page table. This also has to be controlled to provide almost constant timing behavior for the PTWs.

Kabylake and the Meteorlake E-Cores both yield similar results in Section 6.2.2. When testing the transient window size, the Meteorlake E-Core has a limit of completed loads for MUL instructions that is lower than for the DIV instructions, shown in Table 6.2. This can be explained by the size of the ROB that can only hold a specific number of instructions, e. g. 256, and the duration of the MUL instruction. As the speculative execution can only progress the instruction within the transient window in parallel to the other instructions in the ROB, the maximum duration of the transient window is limited to the ROB size. A chain of the maximum number of MUL instructions has a fixed duration, which is less for the Meteorlake E-Core than for Kabylake, due to its faster performance on MUL instructions.

7.2 Weird TLB

Substitution of the data caches with the TLB for a weird state, as described in Chapter 5, comes with some disadvantages and limitations. The experiments in Section 6.3 show that only modern CPUs and P-Cores have the performance to run it with comparable accuracy to the data caches. In Section 6.3.2 Kabylake and the E-Cores both yield no favorable results, indicating that they are not usable for this approach. As the PTW does need more time and subsequent memory lookups than using data caches, this could be due to the OoO-engine, which is smaller than the OoO-engine of the Meteorlake P-Core. Another possibility is that Kabylake and the Meteorlake E-Core have no fourth-level translation cache in the page tables, leading to slower PTWs as explained in Section 6.2.1. The Meteorlake E-Core is very similar to the E-Core of Alderlake. For both Alderlake and Kabylake, Ertmer et al. [EDY25] have shown that there is no fourth-level translation cache.

Even the modern P-Core in Meteorlake has a limit for the number of inputs to a gate (e. g. 3). This limit is below the limit for data caches (e. g. 4), which leads to larger circuits for the same function. As almost each wire in a circuit corresponds to a weird register, the number of weird registers grows too. These weird registers are used in the more weird gates of the circuit, leading to more noise and evictions in the TLB. This intensifies the problem that the TLB has many fewer entries that can be used compared to the data caches.

As listed in Table 3.1, the TLB has only up to about 3000 entries, while the Meteorlake P-Core L2 cache has about 20,000 entries. Additionally, Flexo-DCache can also use the L3 cache with about 500,000 entries. The accuracy of a weird circuit strongly depends on its width, the number of weird registers in parallel.

As seen in Table 6.7, even circuits with many wires can give results with high accuracy for Flexo-TLB when having a low width. Large circuits, as SHA1 and AES, also have a larger width, as they need many weird registers that hold some values, while using other weird registers and doing computations. The TLB is much more sensitive to such a case, due to its small size, leading to many more evictions, because of the usage of the other weird registers. Testing different sizes of an ALU reveals that accuracy decreases with increasing complexity. Using the Error Correction with the ALU gives back the high accuracy, while increasing the runtime.

We see that the number of registers used in parallel is the main problem that arises with the complexity of the circuits. To overcome this, and by using larger weird machines with Flexo-TLB, circuits can be split into more reasonably sized circuits and that are combined afterwards. Using this method, more intermediate values are exposed, as the smaller parts are weird machines that translate the results back in architectural state to do the Error Correction.

As described in Chapter 3, the TLB is harder to control, because there is no user-level equivalent to the **clflush** instruction. A kernel module makes this easier, but gives a much larger timing penalty for flushing an entry, due to the ioctl call and context switch. Using eviction sets is possible, but it brings two problems. The main problem is that with eviction sets for just evicting a single address can affect other weird registers that are in the same set of the TLB cache. The second problem comes with the large timing penalty that comes from accessing multiple addresses that are not cached by the TLB, as each of them triggers a PTW. Finding another method to flush the complete TLB at once and then set the initial value to the weird registers by accessing them can help speed up the flushing and overcome the kernel module. But with this, weird registers can only be used once, as there is no method to reinitialize after reading the value, setting it to 'true'. This is problematic, as input registers, which get their value directly from architectural values, are used multiple times by resetting and rewriting them before usage. Especially in a larger, weird machine, an input is likely read much more often. This problem can be solved by using more registers, but the required number of registers can increase very quickly. In particular, this does load more different pages into the TLB and, as there is no invalidation, leads to more evictions and can have an effect on other weird registers, which hold intermediate values over time.

However, using the TLB also has advantages. In theory, the TLB has a faster access time, which was slower in practice due to the kernel module and the larger circuits, but this could be improved to create a comparable alternative.

In addition to this, it has been shown that some defense techniques against cache side channels do not harden the TLB. Using the TLB is a working, promising fallback technique if a system's defense technique prevents a weird state with data caches, increasing the resilience and interoperability of a program using weird machines.

7.3 Detection

Weird machines can be used to obfuscate parts of the program. They make it harder to analyze the program, as weird machines often behave differently when being debugged and hide the operations by expressing them as circuits.

In dynamic analysis with a debugger, breakpoints are used to step through the program. When analysing a weird circuit via breakpoints, they interrupt the weird gates and prevent speculative execution, resulting in either no execution or only partial execution of the transient window, which produces no correct results for that gate. This makes it hard to understand the operation of a gate, and in particular, the understanding of the whole circuit. This protects the intended operation executed within weird gates by obscurity.

Using breakpoints to analyse the program, but skipping over weird gates and observing only the results of them, works as usual. The executed operations within the weird circuits or the weird circuits themselves are only a black-box for an analyst. This means that an analyst may not be able to understand directly what the weird machine does, but can try to classify it and everything else in the binary.

Because the code that executes the weird machines ships with the binary, it can also be analyzed. The structure of the weird gates is very unambiguously when knowing weird machines, so one could write a decompiler and detector that reduces the complexity for the program analysis. Changing the structure or functionality of the gates can prevent this, but only until the decompiler is updated. This leads to the question of whether weird machines are a strong defense against static analysis. We conclude that the weird machines are a little bit more secure than other obfuscation techniques, but could also be removed automatically.

The main application area for these machines is to hide operations. Either an attacker may attempt to write a malicious tool that hides some of its operations to prevent detection, or someone may be running software on a foreign system and does not trust the architectural state of it. With weird machines, one can reduce the leakage of computations to the architectural state.

In the context of an attacker, it is desirable to detect such programs.

It is possible to detect TLB-based weird machines and data cache-based weird machines similarly at runtime. Both approaches change the state in the caches, and result in different behaviour than for a normal program. This difference can be detected by using performance counters, as they reflect the interaction with the cache that leads to more evictions and invalidations. The performance counter then gives higher values and peaks while weird machines are running. To protect the computer from weird machines, one could flush its caches using `invpcid` for the complete TLB or using `WBINVD` for data caches, if a weird machine is detected using performance counters. This can result in unusable values within the weird machines, but may also lead to false positives, slowing down the application.

The specific structure of weird gates could also be detected by automatically scanning the binary, as described above.

To protect weird machines from detection by scanning the binary, one could change the schema of the circuits or pack the whole program, which is an often-used obfuscation technique. This does not affect the runtime detection. To hide peaks of cache misses, one could also flush the cache in the non-weird machine parts, to have a more constant high course. As this does not reduce the high number of cache misses, a detection by looking for abnormally high numbers is more likely. Because of this, running computations on

weird machines could be less obvious when running smaller parts with longer breaks between. A change between the types of weird machines is possible, e. g. alternating data cache- and TLB-based weird machines, to have fewer cache misses of one kind.

8 Conclusions

In this chapter we summarise this thesis and give some ideas for future work.

8.1 Summary

The TLB is a cache that stores translations of the virtual memory, for which the CPU provides instructions to invalidate entries similar to data caches. However, these instructions are only accessible from kernel space. This makes it less accessible for user-level programs, but other tricks like eviction sets can be used. It is much smaller than data caches, with about only 3000 entries. On a TLB Miss, a PTW is executed to translate virtual addresses into physical addresses. Within this PTW, the CPU has to load page tables that are stored in memory, leading to additional memory loads. This increases the latency, especially when the page table is uncached. The PTW does store partial translation in the translation caches, leading to faster subsequent PTWs. To distinguish between a PTW and a TLB Hit a threshold is necessary. This threshold can be found by using our custom tool we provide, as this threshold not only depends on the timing for a PTW, but also on the caching state of the translation caches.
State-of-the-art Weird machines are based on data caches that provide good results and can be used by directly writing C code. They use an approach of differential encoding that enables more fine-grained Error Correction. Substituting the weird state from data caches to the TLB is nearly straightforward, since both are caches. When differentiating between Hits and Misses, we adjust the timing function to be more precise, as the TLB has a smaller timing difference. Using many TLB-based weird registers requires less memory than using data caches, because we only abuse the translation that can always map to the same physical memory. However, this also reduces the number of inputs to a weird gate to only support 3-1 gates, which leads to larger circuits.
We have seen that the transient window can be opened with MUL or DIV instructions in the same way, allowing to choose the instruction for the the best granularity for fine-tuning the window length. But when having an allocation with offsets between the addresses of the pages, the time for a PTW increases. This is due to the translation that can use fewer cached partial translations and has to load more different page tables. We conclude that this is also visible in the increasing window size for larger circuits, leading to the question of whether offsets between pages can be used to increase the accuracy.

We have investigated the TLB as a weird state in weird machines. It has been shown that although it is possible, there are some limitations compared to the data caches. Because of the more complex PTW, only modern processors like the Meteorlake P-Core are able to produce sufficient results for weird machines. The TLB is due to its small size very sensitive to weird circuit width, i. e. the number of parallel used weird registers, leading to a decrease in reliability for larger circuits. In addition, the PTW does not have a constant timing behavior, but depends on the range of the weird register addresses, as this influences the caching within the paging structure.
We conclude that Flexo-TLB does work, but also has potential to get improved to bring the results closer to Flexo-DCache.
It is usable as a fallback solution for data cache-based weird machines, if they can not be used.

8.2 Discussion and open problems

As explained above, to increase the usability of this approach, it is important to omit the kernel module and find another technique for invalidating entries. Naive approaches, as eviction sets, seem as if they cannot solve this problem entirely, due to it evicts not only a single target address. But resetting all weird registers at the start is maybe feasible. This could be done by using eviction sets or maybe the *fork*-systemcall, leading to a different processor internal PCID. This results in the TLB ignoring all previous entries, as the entries are tagged with the PCID.
To address the small number of TLB entries that lead to evictions within weird machines and lower accuracy, it would be great to choose them in a way and order that they have as few evictions as possible when using them. This can be based on work like [TTGB22] to use replacement policies and the mapping function to simulate the emerging evictions.
The order of weird gates also has an impact on the accuracy of the weird circuits, as the TLB has a problem holding a value of a weird register for a long time, due to its size. The Flexo compiler by Wang et al. [WPWB24] could be improved to order the gates in a way that leads to short residence times for a weird register or less parallel usage of weird registers. This would result in fewer evictions of weird registers holding intermediate values. Reordering the gates to an optimal order would improve both Flexo-TLB and Flexo-DCache.
Another interesting research would be to reduce the impact of the translation caches within the page-tables on the timing behavior of the TLB. This could be done by choosing the addresses for the pages that are used as weird registers in a way that their addresses do not interact with each other in subsequent PTWs.

References

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[EDY25] Philipp Ertmer, Robbie Dumitru, and Yuval Yarom. Reverse-engineering the address translation caches. In Manuel Egele, Veelasha Moonsamy, Daniel Gruss, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 22nd International Conference, DIMVA 2025, Graz, Austria, July 9-11, 2025, Proceedings, Part I*, volume 15747 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2025.

[GRB+17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 955–972. USENIX Association, 2018.

[GYCH18] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[HRY24] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-scope: Cache probing at cache speed. *IACR Cryptol. ePrint Arch.*, page 775, 2024.

[Int25a] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual*, volume 3. Intel, Jun 2025.

[Int25b] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual*, volume 2. Intel, Jun 2025.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *meltdownattack.com*, 2018.

[KKC+23] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The Gates of Time: Improving Cache Attacks with Transient Execution. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1955–1972. USENIX Association, 2023.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[OST05] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. *IACR Cryptol. ePrint Arch.*, page 271, 2005.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010.

[TTGB22] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. Tlb;dr: Enhancing tlb-based attacks with TLB desynchronized reverse engineering. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 989–1007. USENIX Association, 2022.

[vSRG+17] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Revanc: A framework for reverse engineering hardware page table caches. In Cristiano Giuffrida and Angelos Stavrou, editors, *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, pages 3:1–3:6. ACM, 2017.

[WBW23] Ping-Lun Wang, Fraser Brown, and Riad S. Wahby. The ghost is the machine: Weird machines in transient execution. In *2023 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, May 25, 2023*, pages 264–272. IEEE, 2023.

[WPWB24] Ping-Lun Wang, Riccardo Paccagnella, Riad S. Wahby, and Fraser Brown. Bending microarchitectural weird machines towards practicality. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[YAC+24] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean M. Tullsen. Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 770–784. ACM, 2024.

[YF13] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. *IACR Cryptol. ePrint Arch.*, page 448, 2013.