CS593 Computer Architecture Final Report - A General Research On Meltdown & Branch Target Buffer

Shau-Shiun Chu
chu264@purdue.edu
Purdue University
West Lafayette, IN, USA

1 INTRODUCTION

This is a report of series of research, implementation and experiment related to cache and security issues in this computer architecture course. I will first run meltdown attack on my Intel coomputer, since all modern CPU produce by Intel seems to affected by Meltdown attack.

Secondly, there is an example of how to measure the ROB size in out-of-order execution CPU in the midterm exam. Today, I will conducted research and manage to test the Branch Target Buffer size with different instructions and block size on my machine, and explain the data I collected.

2 RELATED WORKS

In modern computers, performance is crucial, and computer scientists and engineers are making significant efforts to enhance it. Several methods have been employed. For example, a more sophisticated branch predictor design like the TAGE predictor can achieve a misprediction rate of around 3.5% to 4%, compared to the simpler 2-bit predictor or older bimodal predictors (6.5%), and the gshare predictor (6% to 7%) [1]. Since a mispredicted branch requires discarding all previously issued instructions in the pipeline, CPUs with more complex and deeper pipelines are affected more severely.

On the other hand, we can utilize a cache-like structure to memorize previous instructions and their results, which called Branch Prediction Buffer(BTB). For branch prediction, the result of an instruction in BTB may be the address of another instruction to jump to, we can directly jump to the new instruction to execute without waiting for instruction decoding. Branch predictors and branch target buffers work together to decrease the misprediction rate and penalty.

As computer systems become more complex, they also become more vulnerable to exploitation by malicious actors. In 2017, Moritz Lipp discovered a significant vulnerability, known as Meltdown [2], which exploits the forwarding of transient results in pipelines and out-of-order execution. By carefully crafting covert channels through exception suppression within a program, attackers can bypass privilege checks and gain unauthorized access to kernel space. Similarly, another type of attack called Spectre[3] leverages transient instructions resulting from mispredictions to access other user programs and applications within memory space, without violating the safety checks of the computer system. It is worth noticing that a variant of Spectre attack is utilizing BTB to attack victim program. The programs that run on same CPU will use the same BTB, so the malicious program and first train(or poison) the BTB, then let the victim program to use this BTB, he branch prediction functionality may be misled to jump to a specific address designed by the attacker, thereby executing a program designed by the attacker.

CPU name	Intel i5-7200U	AMD R5-5600H
Published	2016 Q3	2021 Q1
ISA	x86-64	x86-64
Microarchitecture	Kaby Lake	Zen 3
Cores/Threads	2/4	6/12
L1 Cache	I : 64 KB/8-way/WB D : 64 KB/8-way/WB	I : 192KB/8-way/WB D : 192KB/8-way/WB
L2 Cache	512 KB/4-way/WB	3MB/8-way/WB
L3 Cache	3 MB/12-way/WB	16MB/16-way/WB
BTB size	4096 entries	L1 : 1024 entries L2 : 6656 entries

Table1 : The hardware information about machine in experiment

3 EXPERIMENT MACHINE AND OPERATING SYSTEM INFORMATION

I have two computers by my side, one has intel i5-7200U, and the other has AMD R5-5600H. The related spec of these CPU are list in the table above.

For operating system, I have Linux 6.5.0-28-generic on my intel computer, and Linux 5.15.153-WSL2 on my AMD computer. However, I wonder there will be some limitation of using WSL2 instead of pure Linux operating system for experiment, so I will install a Linux virtual machine as experiment backup.

4 MELTDOWN EXPERIMENT

As mentioned in the Meltdown paper, most modern Intel CPUs are affected by the Meltdown attack, whereas AMD CPUs, which have no speculative action in the TLB and no delay on permission check, are less vulnerable to Meltdown. So we are expected to see Meltdown attack success on my Intel CPU.

I found out that another person had conducted an experiment on the i5-7200U in late 2017, which showed that this CPU is vulnerable to the Meltdown attack. However, in my own experiment, it shows that the i5-7200U today is not affected by Meltdown. This means there have been some changes in the Linux kernel that mitigate the effects of the Meltdown attack on Intel CPUs.

We know memory is divided into user space and kernel space for every program, while the kernel space need permission from hardware and software system to access (ex: system call or interrupt). People have long aware of the possible attack from content in

kernel space, thus they introduce the mechanism call kernel space layout randomization (KASLR). However, we find out that attacker can measure the time required by page fault interrupt to identify the kernel location, or use BTB attack exploits the properties of the branch target buffer to recover the lowest 30 bits of the randomized kernel address. The attacker occupies part of the BTB by executing a series of branch instructions, and then obtains virtual address information by executing targeted system calls to break the protection of KASLR.

As the result, the paper[4] introduce the method called KAISER. KAISER employs shadow address space paging structures to separate kernel space and user space. This involves maintaining two address spaces for each process: one with user space mapped and kernel space unmapped (shadow address space), and the other with the kernel space mapped but user space protected. The shadow address space can mitigate the overhead of synchronization, context switch, and to to minimize the number of implicit TLB flushes for improved performance.

Linux kernel released patches for 4.4.0-108-generic and 4.4.0-109-generic in mid 2018 that can solve the problem of meltedown.

The change log of Linux 4.4.0-109 shows that the patches include the mechanism of kernel -page-table isolation (KAISER) to mitigate meltdown. This might explains why my Intel i5-7200U is no longer vulnerable with linux kernel 6.5.0-28 today.

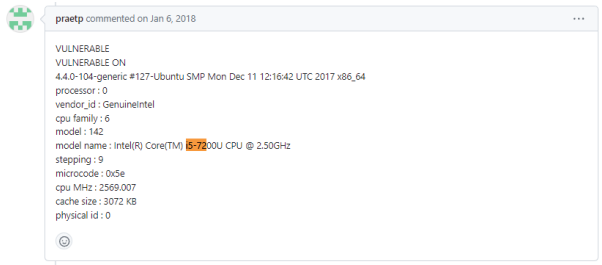

Figure 1: The experiment that had done several years ago shows that i5-7200U is still vulnerable in linux-4.4.0-104

5 BTB EXPERIMENT

I surveyed some blog on the Internet to find out related topic about computer architecture component measurement. Luckily, BTB are often mentioned

1. Branch predictor: How many "if"s are too many? Including x86 and M1 benchmarks!
   - The blog is showing its experiment on AMD EPYC 7642, 7713 and Xeon Gold 6262, and use the experiment result to state that not-taken branch will not cost extra latency cpu cycle when execution.
2. Matt Godbolt's micro-architecture research
   - The author use the framework provide by Agner Fog to test branch predictor and BTB among Arrendale, Ivy Bridge and Haswell architecture, and use the miss rate in different experiment to determine the entries size and number of ways in BTB.s
3. Skylake: Intel's Longest Serving Architecture
   - I discovered that information about Kuby Lake architecture is scarcely found on the Internet. This is origined from the fact that Kuby Lake is essentially a modified version of the popular Sky Lake architecture, but based on manufactured using a 14 nm process. At the architectural level, they remain nearly identical. Therefore, by delving into this blog, we can dive into the design of Intel CPUs. What's even more valuable, the blog provides a a lot of experimental charts detailing various components such as BTB, ROB, fetching bandwidth, cache latency and bandwidth for every memory level, alongside the performance-power curve. While I don't have time to conduct all these experiments by myself today, this serves as a valuable learning experience for me!
4. Measuring Zen 3's Bottlenecks
   - The blog try to illustrate the bottleneck of AMD Zen3 architecture. It find out that compares to frontend(in-order instruction fetching and decode), the operation in backend(out-of-order execution) which are ROB filling, load queue filling, and store queue fill are the primary bottleneck of the system. Mean while, the miss prediction penalty is presents on L2 BTB, so increase L2 BTB size in next generation may be a method to consider.
5. The AMD Branch (Mis)predictor: Just Set it and Forget it!
   - The blog extensively explores mispredictions and their impact on the machine. The author discovers that the clflush instruction reorders speculative fetches, rendering the speculation invalid. AMD CPUs perform poorly compared to Intel CPUs in branch prediction. Through experimentation and speculation, the author identifies that mispredictions are primarily caused by BTB entry evictions, it is caused by the shared BTB in CPU. AMD has shown 3 method to mitigate the issue.
     1. Use LFENCE for serializing so that we preserved valid bounds checking, this will lead to performance issue
     2. "Create a data dependency on the outcome of a compare to avoid speculatively executing instructions in the false path of the branch.
     3. Create a data dependency on the outcome of a compare to mask the array index to keep it within bounds

     This blog features rich and profound content that has greatly benefited me and taken me a while to understand.

5.1 Experiment Background

I will briefly explain the BTB testing program here.

Listing 1: BTB test sample code

```
1
2  struct perf perf;
3  memset(&perf, 0, sizeof(struct perf));
4
5  int j;
6  struct blob blob;
```

```
memset(&blob, 0, sizeof(struct blob));
blob_alloc(&blob);
blob_fill_code(&blob, alignment, cycle_count, type);
blob_warm_itlb(&blob);

uint64_t numbers[20][4] = { { 0 } };
for (j = 0; j < 20; j++) {
    blob_warm_itlb(&blob);
    int k;

    uint64_t c0, c1;
    RDTSC_START(c0);
    for (k = 0; k < repeats; k++) {
        blob_exec(&blob);
    }

    RDTSC_STOP(c1);
    uint64_t d = (c1 - c0); // counts full loop latency
    numbers[j][0] = d * RDTSC_UNIT;
}
```

There are several variables and function calls in this program.

- Variables:
  - blob : We use blob to store the instruction test set for experiment.
  - alignment : aligment means the nop we filled between two instructions, we use this to simulate the different size of block in our program.
  - cycle_count : It is related to the total number of test to run in a single program.
  - types : This means the type of instruction we will use for this test.
- Function calls:
  - `blob_alloc(blob)`: This setup the required file descriptor and appropriate protection flags for blob.
  - `blob_fill_code(blob, align, cycle, type)`: Filled the blob with given instruction and environment for testing.
  - `blob_warm_itlb(blob)`
  - `blob_exec(blob)`: execute the code .

I can take a look at the instruction test set for our experiment.

Listing 2: "jmp" test sample with alignment = 4

```
0x00005fffffe0003f: c3  retq
0x00005fffffe00040: eb 02 jmp    0x5fffffe00044
0x00005fffffe00042: 90  nop
0x00005fffffe00043: 90  nop
0x00005fffffe00044: eb 02 jmp    0x5fffffe00048
0x00005fffffe00046: 90  nop
0x00005fffffe00047: 90  nop
0x00005fffffe00048: c3  retq
```

Listing 3: "je always-taken" test sample with aligment = 4

```
    0x00005fffffe0003e: ff c9 dec    %ecx
    0x00005fffffe00040: 74 02 je     0x5fffffe00044
    0x00005fffffe00042: 90  nop
    0x00005fffffe00043: 90  nop
    0x00005fffffe00044: 74 02 je     0x5fffffe00048
    0x00005fffffe00046: 90  nop
    0x00005fffffe00047: 90  nop
    0x00005fffffe00048: c3  retq
```

In this code, the je (Jump if Equal) instruction is used to control the execution of the loop. When the value in the %ecx register is 0, the dec instruction decrements it by 1 and then checks whether %ecx is zero. If %ecx is zero, the condition of the je instruction is met and the execution jumps to the specified address. In this case, the condition of the je instruction will always be true because the value of the %ecx register will be decremented on each loop until it reaches zero. Therefore, the jump instruction is always executed, which is why it is called "always taken"

Listing 4: "jne never-taken" test sample with aligment = 4

```
0x00005fffffe00039: b9 01 00 00 00  mov    $0x1,%ecx
0x00005fffffe0003e: ff c9 dec    %ecx
0x00005fffffe00040: 75 fe jne    0x5fffffe00040
0x00005fffffe00042: 90  nop
0x00005fffffe00043: 90  nop
0x00005fffffe00046: 75 fe jne    0x5fffffe00046
0x00005fffffe00048: 90  nop
0x00005fffffe00049: 90  nop
0x00005fffffe0004c: c3  retq
```

In this code, the je (Jump if Equal) instruction is used to control the execution of the loop. When the value in the %ecx register is 0, the dec instruction decrements it by 1 and then checks whether %ecx is zero. If %ecx is zero, then the condition of the je instruction is true and a jump to the specified address will occur. In this case, the condition of the je instruction will always be true because the value of the %ecx register will be decremented every loop until it reaches zero. Therefore, the jump instruction will always be executed

Listing 5: callq/retq function call instructions

```
0x00005fffffe0003b: e9 0c 00 00 00   jmpq   0x5fffffe00047
0x00005fffffe00040: c3                retq
0x00005fffffe00044: c3                retq
0x00005fffffe00047: e8 ef ff ff ff   callq  0x5fffffe00040
0x00005fffffe0004b: e8 ef ff ff ff   callq  0x5fffffe00044
0x00005fffffe0004f: c3                retq
```

First of all, we will run the experiment with three different condition, jump without any condition, condition with always taken, condition with never-taken, and check what will happen if the number of instruction exceed the capacity of BTB. Secondly, we will test the BTB with callq/retq, just like the regular function routine, and different instruction block size (we use nop to align and simulate the block size). Since BTB is indexed by the instruction pointer address. Its value and alignment can impact its placement within the BTB, potentially revealing its layout. The increase of block size may exhaust the instruction cache or BTB with fewer instruction.

5.2 Experiment Result

Let's start by examining Figures 2 and 3, which illustrate the relationship between different types of instructions and their execution cycle for Kuby Lake and Zen3 architectures.

We notice that for Kuby Lake, the latency begins to rise once the number of instructions exceeds 4000, aligning with the design specification of 4096 entries in its BTB. Similarly, the latency for Zen3 CPUs initially increases by 2 cycles after 1000 instructions, likely due to the 1024 capacity of its L1 BTB, before experiencing a significant surge around 5000 instructions. However, it's noteworthy that this increase seems unexpected given Zen3's L2 BTB capacity

of 6566. This anomaly might be attributed to the replacement policy, where previously targeted entries may have been replaced by others even before reaching the BTB's maximum capacity in our experiment.

Moreover, we can see there is a 6 cycles latency gap between unconditional jmp and conditional branch in AMD, it means that for conditional prediction, the front-end(instruction fetch and decode) pipeline bubble of AMD Zen3 is around 6 cycles. While in Intel Kuby Lake, we do not see this latency gap, which may indicate a different design of front-end architecture.

Another interesting observation is that the latency cycle remains unaffected by not-taken branches, mirroring the findings presented in the 'half & half' paper. In conclusion, the not-taken branch does not occupy BTB capacity.

There is one more thing that we have to concern, the cpu latency cycle is not only related to branch target buffer, but also related with the instruction cache size, If the storage size of the instruction use for testing is larger than the cache, then the latency will increase - even the number of instruction haven't reach the capacity of branch target buffer. The L1 instruction listed on the table in first page shows that the L1 I-cache size for Intel and AMD are 64 & 192KB respectively; however, we know that the L1 cache is not shared among cores - every core has their own L1 cache. Thus, we must divide the total size of L1 cache with the number of cores it has - since the program run on single core. By doing this, we can find out the size I-cache of Intel and AMD are 32KB.

In Figure 4, we can see that the result of block size = 64 first raises at the number around 600, 64 bytes * 600 / 1024 ≈ 32KB, which is exactly the size of L1 I-cache on a core. Respectively, we can see that the first pop up of AMD test is happened on block size = 64, around 600 in Figure 5 , which lead to the same result with Intel as they have the same L1 I-cahce per core. The miss penalty of L1 cache in Intel Sky Lake is around 4 cycle, which confirmed the latency over here is caused by L1 I-cache miss.

Moving on to Figures 6 and 7, when analyzing the combined use of call and return instructions (typical in function calls), we notice a significant reduction in the BTB's capacity to handle function calls. In Intel CPUs, BTB caching fails when the number of function calls exceeds 2000 (half of the 4096 capacity), while for AMD CPUs, latency starts to rise when the number of function calls equals half the size of our previous experiment.

Finally, we can find out that as the size of a block increase, the number of instruction it can handle suddenly reduce by half, as shown in the block size = 64 test in Figure 6 and block size = 256 test in Figure 7. This is because of the associativity of BTB structure. Associativity means that the target in same set are using the same index as reference. Increase the associativity with a fix size of storage can bring flexibility to the replacement policy, but reduce the utility of the entire cache system. Thus, this reduced storage number tells us that BTB is not directly mapping but set-associated.

6 CONCLUSION

By setting up the experiment, we can find out related information and design about Branch Target Buffer for different CPU. We use jmp instruction to identify the size of different BTB size in memory hierarchy, and the front-end pipeline bubble it has resulted from

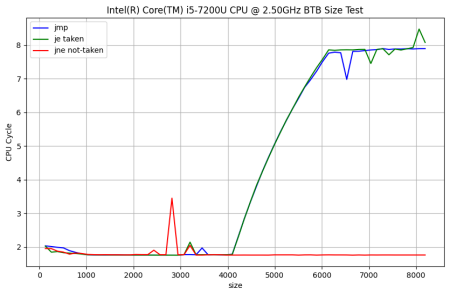

Figure 2: The cpu cycle latency of "jmp", "always-taken", "never-taken" with block size = 8 in Intel i5-7200U(Kuby Lake)

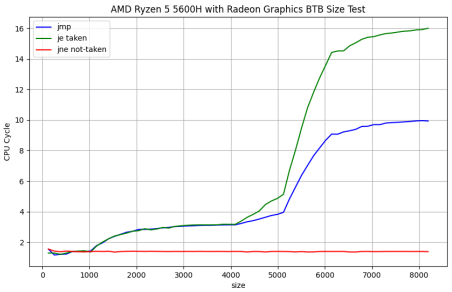

Figure 3: The cpu cycle latency of"jmp", "always-taken", "never-taken" with block size = 8 in AMD R5-5600H(Zen3)

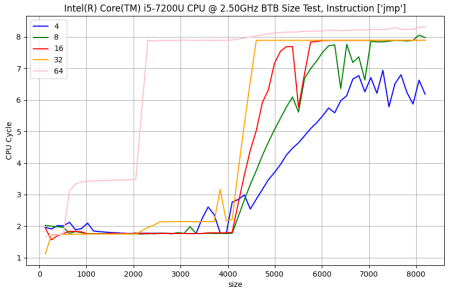

Figure 4: The cpu cycle latency of different block size with jmp on Intel i5-7200U

the miss prediction. The experiment result also tells us that a not-taken miss prediction will not affect BTB and runtime latency. The experiment also shows that the latency is not only contributed by the BTB itself, the large block size of instruction may also exhaust the capacity of I-Cache, which result in a hit miss in L1 and cause extra cycle to fetch from lower level of memory. The decrease of BTB utility when the block size is getting larger also indicate that branch target buffer, like general cache memory system, has associativity with its entries.

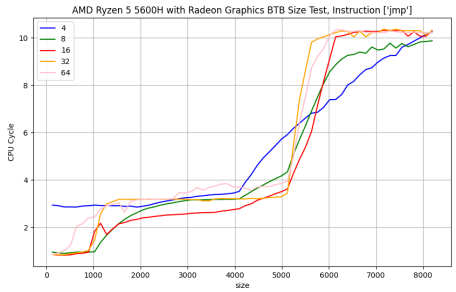

Figure 5: The cpu cycle latency of different block size with jmp on AMD R5-5600H

Figure 6: The cpu cycle latency of different block size with function call on Intel i5-7200U

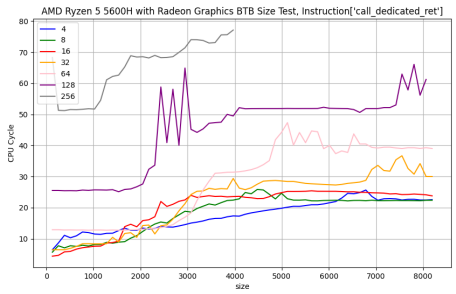

Figure 7: The cpu cycle latency of different block size with function call on AMD R5-5600H

7 FUTURE WORKS

We can see that there are something that hard for me to explain in BTB test:

1 The early raising of latency in AMD Zen3 CPU. I hypothesis that the replacement policy in BTB lead to this result. However, this only happen in L2 BTB on Zen3, while I did not see the same thing happened on Intel CPU, or on AMD L1 BTB. On one hand, I cannot find the replacement policy of BTB from Intel and AMD, perhaps it is the confidential secret for manufactures, On the other hand, there are research that discussed the replacement policy on BTB[5]. The research pointed out a few policy like the good-old LRU, Random Choose, Static Re-reference Interval Prediction and Sampling Dead Block Prediction. The paper also propose it own idea - Global History Reuse Prediction (GHRP). This replacement technique uses the history of past instruction addresses and their reuse behaviors to predict dead blocks in the I- cache and dead entries in the BTB .The experiment shows that under 4K-entry BTB, GHRP lowers MPKI by an average of 30% over LRU, 23% over SRRIP, and 29% over SDBP.
Later on, I decide to send a letter to the blog author to see if me can give me some advice, which shows as Figure 6.

2 The data I collect has some noise and fluctuation, which limited my ability to identify or inference the real cause of increasing latency in a specific value.

3 I cannot directly find how many ways-associated with the BTB for AMD Zen3 and Intel Sky Lake, However, there is a paper[6] discussing the reduce cost of BTB storage, it has mentioned serveral methods, including storing target offset instead of complete address of target, set different ways to store different range of offsets on BTB to reduce the front-end bottleneck and enhance to storage efficiency.

REFERENCES

[1] PIERRE MICHAUD An Alternative TAGE-like Conditional Branch Predictor 2018.
[2] Moritz Lipp1, Michael Schwarz1, Daniel Gruss Meltdown: Reading Kernel Memory from User Space 2017.
[3] Paul Kocher, Jann Horn, Anders Fogh Spectre Attacks: Exploiting Speculative Execution 2017.
[4] Daniel Gruss, Moritz Lipp, Michael Schwarz KASLR is Dead: Long Live KASLR
[5] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer 2018
[6] Truls Asheim NTNU, Norway Boris Grot University of Edinburgh UK A Storage-Effective BTB Organization for Servers 2023

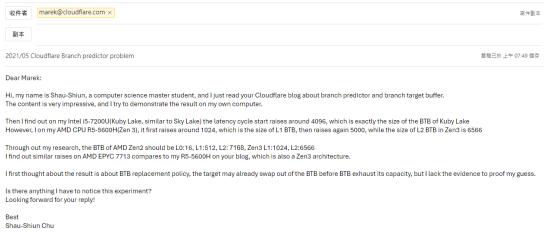

Figure 8: The letter I send to the author