Reverse-engineering the branch prediction unit in modern processors

Submitted on the 10th of January, 2020
in partial fulfillment of the requirements for

Semester Project

at the **Hexhive laboratory**
of the **School of Computer and Communication Sciences,**
École Polytechnique Fédérale de Lausanne (EPFL)

by
Markus Ding

supervised by
Atri Bhattacharyya
Prof. Mathias Payer

Reverse-engineering the branch prediction unit in modern processors

ABSTRACT

The speculative execution of processors is exploited in several microarchitectural attacks. This performance optimization technique is made possible by a Branch Prediction Unit (BPU) that predicts the outcome of branch instructions. The objective being to reduce the idle time of the processor and avoid stalls. However, this pre-execution of instructions can be exploited through side-channels attacks.

A good understanding of the architecture and organization of such a BPU would allow more reliable attacks and the identification of vulnerabilities. Because these internal details are not made public by the processors' manufactures, a reverse-engineering effort is required to understand how they are implemented, our main focus is the Branch Target Buffer (BTB) which is a cache-like structure and a central element of the BPU.

We run several micro-benchmarks to infer the total size, the number of ways, the set and tag determination as well as the replacement policy of the BTB from two generations of Intel processors: the Broadwell (i7-5600U) and Skylake (i7-6700k) architectures.

1 INTRODUCTION

Modern processors rely heavily on their branch prediction units to achieve the best performances. A wrong prediction causes the pipeline to be flushed. The deeper and wider the pipelines, the higher the penalties[2]. However, as demonstrated recently in the Spectre attacks [7], these speculative executions can be exploited to access arbitrary locations in memory.

The focus of pursuing the best possible performance often comes at the expense of a higher security risks. Due to the nature of these units, hardware implementations, the correction of a design flaw is often difficult and can only be performed on the next microprocessor's generation. Alternatively, if there exist countermeasures in software, their additional cost impacts the performance to a higher degree than hardware fixes. As a result, the security aspect needs to be considered more during the implementation process.

By reverse-engineering the implementation details of the branch prediction process, it is possible to reveal potential security weak points in the design or microarchitectural flaws that need to be addressed.

2 BACKGROUND

In this section, we describe the general structure of a Branch Prediction Unit, the information available on Branch Target Buffers and how we can monitor various events related to the branch prediction process with specific performance registers. The different types of branch instructions are introduced and their interaction with the BTB explained.

2.1 Branch Prediction Unit

This unit tries to predict the outcome of branches, if they are taken or not and what their target is so that the following instructions can be pre-fetched to improve the processors performance in the case of a correct prediction. If the prediction is incorrect, the speculative work must be discarded and the correct next instruction must be fetched.

The prediction mechanism can be static and the outcome of a branch only based on the branch instruction or dynamic and the information used to

make the prediction is not only the current branch instruction but additional information gathered at run-time about the execution history.

A BPU in a modern processor is composed of a main predictor and additional sub-elements that are specialized in a certain task such as predicting loops, storing the branch targets or the prediction of indirect branches.

A good example is the BPU from the Pentium M as can be seen in Figure 1, there are several units that are used to output the prediction.

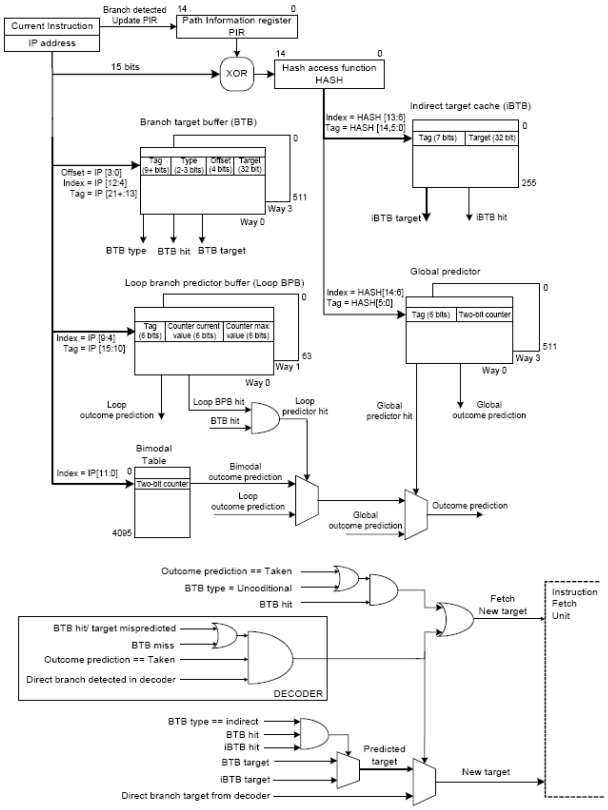

Figure 1: Pentium M Branch Prediction Unit from Vladimir Uzelac's thesis [10]

If the BPU is not able to provide a correct prediction, a front-end resteer is caused meaning that the result of the speculative execution needs to be discarded and the next instruction according to the correct branch outcome is processed.

2.2 Branch Target Buffer

One of the central units in a BPU is the Branch Target Buffer that, as the name indicates, stores the targets of the branches. It is a cache-like structure [8], usually set-associative which uses the branch's address to determine the tag and set.

The detailed architecture is unknown, we will try to find its different properties with the help of our micro-benchmarks, more precisely the following elements:

- Total size
- Number of ways
- Set determination
- Tag field
- Replacement policy

If the destination of a branch is not present in the BTB or if it is incorrect, its value will be updated.

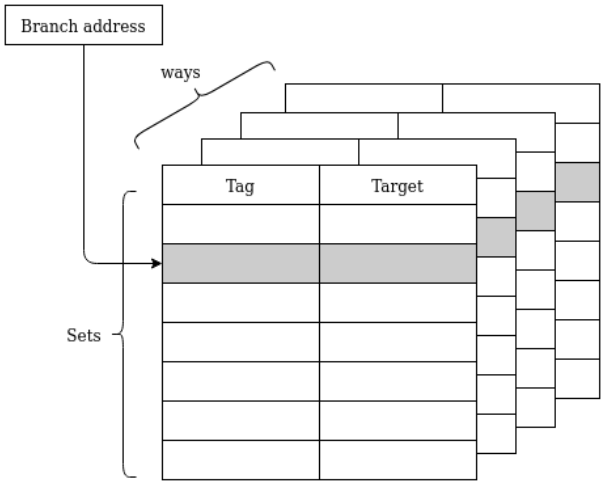

Figure 2: A generic set-associative BTB

2.3 Intel Performance registers

On Intel CPUs, there exist special registers that are incremented on specific events. They allow the monitoring of a program's performance by counting the number of clock cycles, the amount of cache misses and many more events. They are all listed in the Intel manual [5, 6].

There are 7 registers in total, 3 which have a fixed function and can not be changed and 4 programmable ones that can be set to monitor the events we are interested in. We will use these performance registers for branch prediction specific events. The list of which can be found in Table 1.

The documentation if a specific event is operable and its specific description is incomplete. Some events can be monitored on newer generation of chips despite not being mentioned in the documentation. Before making the decision which registers to use and designing the micro-benchmarks, we had to write some basic testing scenarios to verify if said register in operable or not. The results of these tests are shown in Table 1 as well.

2.4 Type of branches

The different branch instructions can be separated into the following categories:

2.4.1 Direct branches: The destination of the branch is known at compilation.

2.4.2 Indirect branches: The destination of the branch is defined through the value of a register during the execution of the instruction and is not known at compile time.

2.4.3 Conditional branches: Can be taken or not taken depending on the condition on which they depend.

2.4.4 Unconditional branches: They are always taken.

Not taken branches do not enter the BTB as their target is irrelevant. Depending on our different micro-benchmarks, we use different type of branches.

2.5 Tools used

2.5.1 Agner tools. We use a variation of Agner's tools[1] which a series of small test programs for testing small pieces of x86 and x86-64 code. They read the performance registers and provide us with the values of the specified counters. A maximum of 4 counters can be monitored at the same time. Due to some inaccuracies of the counters, a single test is repeated 100 times and we take the mean value of all runs.

We can insert our own assembly in the NASM syntax to create custom micro-benchmarks and create our own experiments. All tests are run on a single thread on a 64 bit machine.

2.5.2 Radare2 toolchain. We use this toolchain to obtain the addresses of the branches from our benchmarks so that we can analyze in more detail which bits are used for the assignment to a specific set in the BTB.

3 METHODOLOGY

3.1 Microbenchmarks

We design several benchmarks to determine how the BTB is organized. Each one of them is written in assembly and is repeated 100 times to account for some inaccuracies of the performance registers (we take the mean of all runs) and to train the BTB according to the specific test scenario. On the first run of a code snippet, the BTB state is unknown and its content can cause additional evictions that distort the counter values.

3.1.1 Total capacity. We want to fit a maximum number of branches into the BTB so that we can determine its size, to do so we fill it with a large amount of branches and observe when we start to observe mispredictions.

Hypothesis: We expect to see a low misprediction rate when the number of branches is equal to or lower than the number of entries of the BTB as they all fit in. Above that number, we expect to see an increasing number of mispredictions as the cache needs to evict some previous content in order to make space for the new branch, the increase of the rate will vary depending on the replacement policy.

We use a similar approach as specified in Uzelac's thesis [10] where we are varying two factors: the number of branches (N) and their distance

Event Name	Event:Mask	Description	Broadwell	Skylake
BR_MISSP_EXEC	89:00	Mispredicted branch instructions executed	(✗)	(✗)
BR_BAC_MISSP_EXEC	8a:00	Branch instructions mispredicted at decoding	(✗)	(✗)
BR_CND_EXEC	8b:00	Conditional branch instructions executed	(✗)	(✗)
BR_INST_RETIRED.ALL_BRANCHES	c4:00	Branch instructions at retirement	✓	✓
BR_INST_RETIRED.CONDITIONAL	c4:01	Counts the number of conditional branch instructions retired	✓	✓
BR_INST_RETIRED.NOT_TAKEN	c4:10	Counts the number of not taken branch instructions retired	✓	✓
BR_INST_RETIRED.NEAR_TAKEN	c4:20	Number of near taken branches retired	✓	✓
BR_MISP_RETIRED.ALL_BRANCHES	c5:00	Mispredicted branch instructions at retirement [comment: Seems to only be conditional?]	✓	✓
BR_MISP_RETIRED.CONDITIONAL	c5:01	Mispredicted conditional branch instructions retired	✓	✓
BR_MISP_RETIRED.ALL_BRANCHES	c5:04	Mispredicted macro branch instructions retired [comment: seems to be all branches not just macro]	✓	✓
BR_INST_EXEC.NONTAKEN	88:40	Qualify non-taken near branches executed. Applicable to umask 01H only.	✓	(✓)
BR_INST_EXEC.TAKEN	88:80	Qualify taken near branches executed. Must combine with 01H, 02H, 04H, 08H, 10H, 20H	✓	(✓)
BTB_Misses	e2:00	Number of branches the BTB did not produce a prediction	(✗)	(✗)
BOGUS_BR	e4:00	Number of bogus branches.	(✗)	(✗)
BACLEAR.ANY	e6:01	Number of BAClears asserted.	(✗)	(✗)
BACLEARS.ANY	e6:1f	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	(✓)	(✓)

Table 1: Some of the performance registers as specified in Intel's manual and if they are functional: ✓for a working counter and otherwise marked with a ✗. The parenthesis are added for the registers that are not officially documented for that specific processor generation.

from each other (D). We need our branches to fill the different sets completely in order to find the total size. This relies on the assumption that some part of the branch address is used in the determination of the set index and the tag so that the different branches all fit into the BTB.

We are using direct unconditional branches to avoid any fluctuations due to a taken / not taken prediction as our unique goal is to stress test the BTB's capacity. To monitor the misprediction rate, we count the number of front-end resteers (*BACLEARS.ANY* event) as this event happens if the BPU can not provide a correct prediction. This would happen if the amount of branches is too high for the BTB to handle. The resteer rate depends on the replacement policy but we want to see how many branches can fit into the BTB without any resteers at all.

The corresponding assembly code can be seen in Listing 1

```
jmp j1
nop
nop
...
nop
nop
j1: jmp j2
nop
nop
...
nop
nop
j2: jmp j3
...
jn: END
```

Listing 1: N equidistant branches with an alignment D created by padding with nop instructions

3.1.2 Number of ways and set determination. If the higher order bits of the branch address are used as the tag and the lower order bits for the set as follows

branch address: | tag | set index | offset |

We can expect the following behavior:

- If the branches have an alignment that is large enough, their set index would be identical and only the tag would be different because the only bits in the tag are changed. This would map all branches into the same set and therefore we would start seeing mispredictions once the set is full, ie when the number of branches exceeds the number of ways.
  Benchmark parameters: $N \geq$ NB of ways + 1 and $D \geq 2^{MSB(index)}$
- If the spacing between the branches is even larger, so that the tag is also the same, we would start seeing mispredictions with 2 or more branches as they would map to the same BTB entry. This would allow us to determine if all the bits from the address are used or not.
  Benchmark parameters: $N = 2$ and $D \geq 2^{MSB(tag)}$

The first behavior is observable on both of our CPUs: with a large enough alignment (2^{14} or above) we see a misprediction rate of 0 with few branches and that goes up after reaching a certain value that we assume to be the number of ways + 1. However, the second behavior does not occur, the misprediction rate remains at 0 independently of the value of the branch alignment. This means that our initial assumptions that a subset of the branch addresses are used for the set and tag determination is incorrect!

Therefore, we need to find how the index and tag are computed, we assume that it is a hash-like function of the bits of the address. It is not simply applying a mask to keep a subset of the branch address but does a more complex operation, one possibility is a function using XORs, we explore this possibility in Section 3.1.4. We also can not consider the number of ways that we found as correct because we have no certainty that the branches in our experiment were indeed mapping to the same set.

3.1.3 Eviction sets. Instead of filling the entire BTB, we can try to fill a single set and then try to understand how the set index and tag are computed from the addresses of the branches that fit into that set. To do so, we need at least nb_ways+1 branches that map to the same set so that at least one eviction is necessary and causes a misprediction. We can then further reduce the number of branches in order to find a minimal eviction set [11] that can be used to determine the replacement policy of the BTB.

As a starting point, we use the results from our naive approach to the set determination from section 3.1.2 indeed for the parameters N = 6 and D = 2^{14}, we have a resteer rate of approximately 16.6% which corresponds to $\frac{1}{6}$ meaning that one of our 6 branches caused a resteer. With the set-associativity being 4 as explained in sections 4.1.2 (for Broadwell) and 4.2.2 (for Skylake), we only need to reduce the number of branches by one in order to have a minimal eviction set.

We transform the 6 unconditional branches into conditional ones with additional instructions in order to be able to manually toggle the branch to taken or not taken:

Listing 2: Not taken

```
mov ecx, 1
cmp ecx, 0
je next
nop
nop
...
```

Listing 3: Taken

```
mov ecx, 1
cmp ecx, 1
je next
nop
nop
...
```

We are then able to run the microbenchmark repeatably and setting each branch to not taken which will result in its absence from the BTB. If the branch is part not part of our eviction set, the resteer rate will remain the same however if it is part of our eviction set, we will no longer have an eviction from the BTB.

Finally, we can run decompile the benchmark to collect the address of each branch and try to find which hash function is used for the set determination as all 5 branches map to the same set.

3.1.4 Index and tag. As we have 1024 sets (see Sections 4.1.3 and 4.2.3), the length of the index is $log_2(1024) = 10$ bits. We therefore try to find a hash function that would generate those 10 bits.

Our first approach is to check if an XOR based hash function similar to the *XOR-8* function from the Broadwell TLB [4] is used. This function XORs 8 consecutive virtual address bits, in our case we would XOR 10 bits as can be seen in Figure 3

After finding a potential hash function that maps all 5 branches of our eviction set to the same index, we need to repeat the same process with other eviction sets in order to see if the found hash function also maps these other branches to a single set. If it is not the case, we can discard the hash function and try to find a new one (and repeat the full process again).

$$Hash = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

Figure 3: The index bits would be computed by the matrix multiplication $Hash \cdot$ VirtualAddress$[20:0]$ or any other 20 bits of the VA.

Further, we assume the tag is determined by a subset of the bits of the branch address. Most probably the higher order bits that are not used to determine the set index.

3.1.5 Replacement policy. We start from our previously built eviction set of 5 unconditional direct branches. We add additional backwards jumps in order to be able to repeat some of the branches from the eviction set to test the replacement policy of the BTB.

These backward jumps need to be conditional so that we can monitor the different scenarios appropriately. We want to observe the following events and use the appropriate performance registers:

- Number of taken conditional branches

- Number of mispredicted conditional branches
- Number of total branches
- Number of resteers

This way, if a resteer occurs, we are able to determine if it was caused by one of our control flow branches (the conditional backward ones) as it would result in a mispredicted conditional branch. This way we can isolate the impact of our eviction set and determine which replacement policy is used or at least dismiss some of them.

4 RESULTS

4.1 Broadwell

4.1.1 Total capacity. As can been observed in Figure 4, we are able to fit 4096 branches without seeing any significant resteers when the alignment is between 2^0 and 2^4 (included), for a higher number of branches, the resteer rate steadily grows as more and more branches need to be evicted from the BTB.

For the higher branch alignments, the number of branches that we can fit far less branches and the resteer rate rapidly reaches 100%. This is probably caused by the fact that these branches are not uniformly distributed over all available sets but are all mapped to the same few sets which cause an eviction almost every branch. This behavior could be an indication about which bits are used for the set determination.

We conclude that the BTB has 4096 entries.

4.1.2 Number of ways. Based on the fact that most systems are either 2-way or 4-way set associative [9], as well as the results of the experiments [3] that Matt Godbolt ran on previous Intel chip generations (Arrendale, Ivy Bridge and Haswell) which determined the BTB to be 4-way set associative, it is highly probable that the BTB in our processor will be as well especially as the performance of the BPU improved in each generation.

We can confirm this further by running the test intended to determine the number of ways (using large alignments in order to have the branches

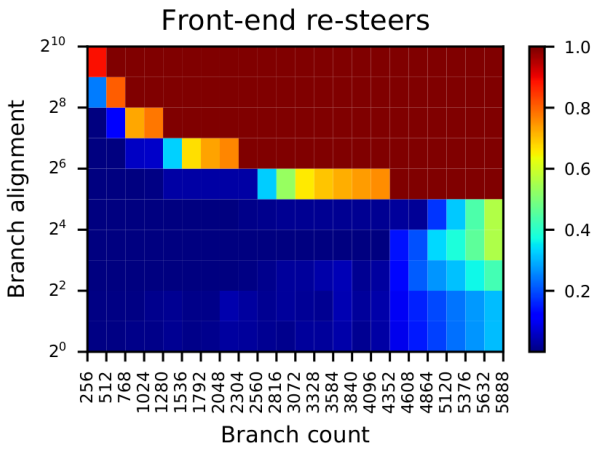

Figure 4: Broadwell capacity test results

all map to the same set). We can observe on Figure 5 that there were no resteers with 3 branches which should be the case if the BTB was 2-way set associative.

By reducing the found eviction sets to their minimum, we can see that 5 branches are required at minimum which confirms the 4-way associativity.

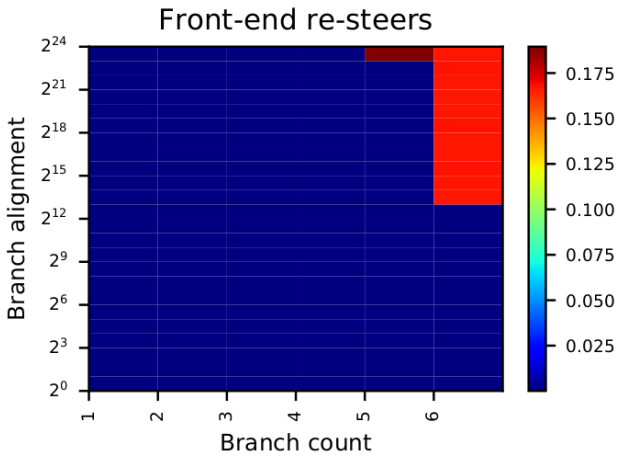

Figure 5: Broadwell naive number of ways

4.1.3 Number of sets. With a size of 4096 entries and a 4 way set-associate cache, we have $\frac{4096}{4} = 1024$ sets.

4.1.4 Index and tag determination. XOR-ing consecutive bits from the branch addresses does not

yield a usable hash function, no matter which subset of bits are used. The 5 branches do not map to a same set. The hash function is therefore something more elaborate that we were unable to find as of now. Below, are the binary representations of the addresses of the 5 branches that constitute an eviction set on our Broadwell processor.

Eviction set:
00000000 00000000 10000000 00000010
00000000 00000000 10100000 00000010
00000000 00000000 11000000 00000010
00000000 00000000 11100000 00000010
00000000 00000001 00000000 00000010

As we do not know how the index is computed, it is difficult to determine the tag as well given that it would probably be a subset of the remaining unused bits.

4.1.5 Replacement policy. By running the following scenario, we expect 1 resteer as it simply fills a set of the BTB and the fifth branch can not be predicted as it is not in the BTB causing a resteer:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$

It is indeed the case as we are only running the sequence of our eviction set once. Now, we can try repeating J5 at the end of the sequence. We expect to remain at 1 resteer as we assume that on the first J5, another branch gets evicted and on the repetition of J5, we should have a hit.

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5 \rightarrow J5$$

However, we have now 2 resteers! So no branch got evicted, J5 misses twice.
We want to see if this trend continues with an additional repetition of J5:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5 \rightarrow J5 \rightarrow J5$$

We still have two resteers so J5 entered the BTB on the second time. This hints at a policy that avoids evicting *hot* branches. Once the BTB is full and an additional branch maps to the same set, one of the previous branches get marked "to evict" but remains in the cache. Only the second time around, once already marked, does it get evicted.

We now want to know which branch got evicted:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$
$$\rightarrow J5 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$

We remain at 2 resteers which shows that the branch J1 got evicted as the branches J2, J3 and J4 did not create a new resteer (alternatively, we could run a similar scenario but with a repetition of J1 and observe 3 total resteers).

Therefore we can determine that the replacement policy is LRU with a strike system that only evicts an entry the second time around.

4.2 Skylake

4.2.1 Total capacity. The number of branches we can fit without any significant resteers, as can be seen on Figure 6, are very similar to the results from the Broadwell CPU therefore we can safely assume that the BTB size is the same with 4096 total entries.

However there are some significant differences:

- **The intensity of the increase in resteers**: when no more branches can be fit into the BTB, the resteer rate goes up way faster than on the Broadwell CPU, we assume that this is due to a new replacement policy.
- **Branch alignment of 2^0 and 2^1**: We have a constant resteer rate of approximately 50%. With such a small alignment, 1 or 2 bytes for each branch instruction, the BTB is completely flooded and seems to behave fundamentally differently to the one from Broadwell. This is probably caused by the new replacement policy. See section 4.2.5 for more details.

4.2.2 Number of ways. The same reasoning and test that were used for Broadwell (Section 4.1.2) can be applied here. It results in the same observations except that the resteer rate is around 28% instead of 16% which we trace back to the new replacement policy.

4.2.3 Number of sets. With a size of 4096 entries and a 4 way set-associate cache, we have $\frac{4096}{4} = 1024$ sets.

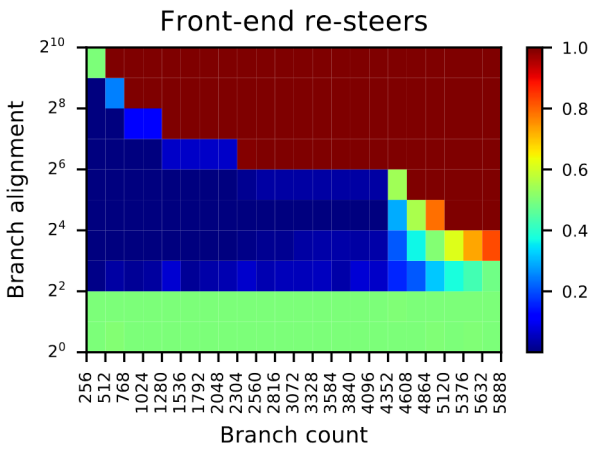

Figure 6: Skylake capacity test results

4.2.4 Index and tag determination. The 5 branches that form an eviction set for the Broadwell processor, do as well for the Skylake one. This strongly suggests that the set determination is the same for both processors. This could be verified once the hash functions are found.

4.2.5 Replacement policy. First, we run the the basic scenario with the 5 branches from the eviction set sequentially:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$

We have 2 resteers opposed to the 1 on Broadwell so we know the replacement policy is different. This second resteer is caused by the fact that we repeat our benchmark 100 times to train the BTB so our initial state is the same as after a single run of the benchmark. Presumably, we have an eviction on J5 and as well on one of the other branches.

To confirm this suspicion, we repeat the entire sequence:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$
$$\rightarrow J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$

As expected, it results in 4 resteers. To find which branches cause the evictions, we modify the sequence so that J1 is not repeated:

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$
$$\rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow J5$$

The resteer count is down to 2 again so J1 is indeed the source of a resteer. This shows us that the eviction is immediate and not delayed with a strike system like Broadwell (Section 4.1.5). We can exclude the use of Least Recently Used (LRU) or variations of Pseudo-LRU, Most Recently Used (MRU) and First in First out (FIFO) with our experiments so far. Last in First out (LIFO) could be a potential candidate by having J4 and J5 evict each other on each repetition.

To confirm, the following test case should result in 2 resteers (marked in bold):

$$J1 \rightarrow J2 \rightarrow J3 \rightarrow J4 \rightarrow \mathbf{J5}$$
$$\rightarrow J1 \rightarrow J2 \rightarrow J3 \rightarrow \mathbf{J4}$$

However, we observe 3 resteers when running it so it is not a LIFO replacement policy either. A Least Frequent Recently Used (LFRU) policy can also be excluded as it would result in 2 resteers in the tested scenario as well.

The replacement policy remains unknown.

5 CONCLUSION AND FUTURE WORK

We were able to reverse-engineer many undisclosed properties such as the total size, number of ways and replacement policy of the Branch Target Buffer from our two different generations of Intel processors with the help of some micro-benchmarks. Not all properties have been found and will require some additional work but nevertheless some useful insight could be acquired.

Taking into consideration Matt Godbolt's work on the previous generations, we can observe the evolution of the BTB and the increase of complexity in its implementation over several processor models. The reverse-engineering process to determine the microprocessor properties becomes more complex as well and will render the search for exploits based on the speculative execution of branch predictors more difficult.

There are several approaches to pursue to complete the reverse-engineering process.

5.1 Improved analysis of the hash function

As the precise hash function for the determination of the set index remains unknown, an additional effort to find it is required. Working with eviction sets and the eviction relationships between several branches seems to be a promising approach as this method allowed to crack the hash function used in the Sandy Bridge LLC [12].

One approach could be to change how the eviction sets are build to allow better control over the branches' addresses. Being able to flip a single bit of the address would make the search of the hash function easier as we would be able to isolate the effect of each bit of the address.

Another possibility could be to automate the finding of the parameters of the hash function. Create a program that runs a test hash function on a given eviction set and varies automatically which bits are used to find the cases where all 5 branches map to the same set index. One would then need to apply this method over several eviction sets and determine which hash function is common to them all.

5.2 Skylake BTB replacement policy

We need to find an eviction policy that matches the observed behavior. We could try to determine if some randomness is involved, for example an LFU policy with a random eviction when several branches have the same number of uses (instead of the first one). Repeating each experiment 100 times could possibly hide some randomness by averaging it out over all runs.

5.3 Full BPU Analysis

The BTB is only one element among several in the BPU, the other ones need to be reverse-engineered as well to acquire a complete understanding of how the branch prediction process is implemented.

REFERENCES

[1] Agner. 2018. Test programs for measuring clock cycles and performance monitoring. https://www.agner.org/optimize/#testp

[2] Stijn Eyerman, J.E. Smith, and Lieven Eeckhout. 2006. Characterizing the Branch Miss Prediction Penalty, Vol. 2006. 48 – 58. https://doi.org/10.1109/ISPASS.2006.1620789

[3] Matt Godbolt. 2016. The BTB in contemporary Intel chips. https://xania.org/201602/bpu-part-three

[4] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[5] Intel. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide, Part 2.
Sections 19.2 for Skylake and 19.3 for Broadwell. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[6] Intel. 2017. Intel® 64 and IA32 Architectures Performance Monitoring Events
p. 10-41 for Skylake and p. 42-79 for Broadwell. https://software.intel.com/en-us/download/intel-64-and-ia32-architectures-performance-monitoring-events

[7] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[8] C. H. Perleberg and A. J. Smith. 1993. Branch target buffer design and optimization. *IEEE Trans. Comput.* 42, 4 (April 1993), 396–412. https://doi.org/10.1109/12.214687

[9] Yan Solihin. 2015. *Fundamentals of Parallel Multicore Architecture* (1st ed.). Chapman & Hall/CRC.

[10] Vladimir Uzelac. 2008. *Microbenchmarks and mechanisms for reverse engineering of modern branch predictor units.* Master's thesis. University of Alabama, Huntswille, USA.

[11] Pepe Vila, Boris Köpf, and José Francisco Morales. 2018. Theory and Practice of Finding Eviction Sets. arXiv:cs.CR/1810.01497

[12] Zhipeng Wei, Zehan Cui, and Mingyu Chen. 2015. Cracking Intel Sandy Bridge's Cache Hash Function. arXiv:1508.03767