Reverse Engineering of the Branch Target Buffer of the Apple M1 Processor

Lucas Cendes
EPFL
lucas.lopescendes@epfl.ch

Atri Bhattacharyya
HexHive laboratory, EPFL
atri.bhattacharyya@epfl.ch

***Abstract*—Information regarding the microarchitecture of the Apple M1 processor has been relatively scarce, and, as a result, there have been significant efforts in reverse engineering it. In this semester project, we focused on reverse-engineering the Branch Target Buffer (BTB) unit used in the Firestorm cores of the Apple M1 processor. Our results show that the BTB is a directly-mapped structure containing 2048 sets and a 1-entry eviction cache. In addition, the hashing function used to index into the different sets in the BTB seems to use bits 2 to 30 of the virtual address of the branch instructions.**

I. Introduction

On November 2020, Apple announced that it would start supplying its products with its new ARM-based Apple M1 SoC chip instead of Intel x86-based processors. The main processing unit of this chip is composed of 4 performance Firestorm cores clocked at 3.2 GHz and 4 low-powered Icestorm cores aimed toward power efficiency. All 8 cores can be used simultaneously, although the performance of the Firestorm cores is higher than the Icestorm cores. The Apple M1 manages to match the performance of x86-based Intel and AMD processors while consuming less power [1].

Information regarding the microarchitecture of the Apple M1 has been scarce, which has led to considerable efforts at reverse engineering and documenting it. The Asahi Linux project, aimed at porting the Linux kernel to the Apple M1, has provided ample documentation regarding the processor's system registers. In particular, considerable information regarding the Performance Monitor Counter (PMC) registers has been documented by the Asahi Linux team [2]. In addition, Dougall Johnson has made considerable efforts at benchmarking and documenting the microarchitecture of the Firestorm cores in the Apple M1. His work mainly focused on investigating the cores' integer, load/store, floating point, and SIMD units and providing benchmarks for each supported instruction. His work has also provided some insight regarding the configuration of the PMC registers [3].

Some reverse engineering work on the branch target buffer (BTB) has already been done. Marek Majkowski has performed a couple of experiments to show how the overall number of branches across a working set affects the latency of the individual branch instructions. For the Apple M1, he reported that the lowest latency is achieved when all branches are arranged in a 4096-instruction-long block of code with equal spacing between each instruction. This arrangement could correspond to either 1024 branches that are each 4 instructions apart or 512 branches that are each 8 instructions apart [4]. Andrej Gorjan has designed some experiments to test if the Apple M1 uses a simple program counter (PC) based BTB and has demonstrated that this is not the case. His experiments use the PMC units of the Apple M1 to measure the number of indirect branch mispredictions encountered in a block of code [5].

This semester project aims to identify the size and associativity of the branch target buffer (BTB) of the Firestorm cores of the Apple M1 and provide insights into the hashing function used to index into BTB. Section II explains the overall setup used to perform the experiments described in this report. Section III describes the experiments done to test if there is any aliasing between consecutive branches in the BTB. Section IV describes the process of finding the overall size of the BTB. Section V describes the procedure used to obtain two eviction sets of indirect branch addresses and discusses a possible hypothesis that explains the sizes of the eviction sets. Section VI goes through different attempts to improve the precision of the PMC measurements. Section VII explains the experiments performed to figure out which bits of the virtual address of branches are used in the hashing function of the BTB. Section VIII explains the process done to verify the hypotheses raised in section V regarding the associativity of the BTB. Section IX describes some address bit-flipping experiments performed to extract information regarding the hashing function used in the BTB. Finally, section X summarizes the overall results obtained in the semester project and concludes the report.

II. Experimental Setup

The computer used for the experiments is a Mac mini from 2020 with an 8-core Mac M1 processor. Four of the cores of the M1 processor use the Icestorm microarchitecture, and the other four cores of the processor use the Firestorm microarchitecture. The `lscpu` command reports that the Icestorm cores run at a maximum clock frequency of 2.064 GHz, while the Firestorm cores can achieve a maximum frequency of 2.988 GHz. These frequencies differ from official numbers released by Apple when the processor was announced [1]. The operating system used is a Debian distribution running version 6.0.0 RC5 of the Asahi Linux kernel.

To use the performance monitor counters (PMC) of the Mac M1 processor, they must first be made accessi-

ble in user mode. This configuration is done through the SYS_APL_PMCR0_EL1 register that contains the configuration flags for all 10 PMC units in the processor [2]. Since this register is only accessible in kernel mode by default, it must be configured using a Linux kernel module. The module designed for this project creates a sysfs file that can be opened by processes running in user mode. Any values written to this file will be written to the SYS_APL_PMCR0_EL1 register by the kernel module. At the start of every experiment, the sysfs file created by the kernel module is opened, and the appropriate flags of the SYS_APL_PMCR0_EL1 are set to ensure that all 10 PMC counters are enabled and that all PMC registers are accessible in user mode. From this point on, all PMC registers can be accessed directly by the experiments, and the sysfs are not needed anymore.

The number of BTB misses can be recorded by configuring one of the PMCs to count indirect branch misprediction events. To do this, we must first configure register SYS_APL_PMCR1_EL1 to enable one of the PMCs to count events in user mode, which corresponds to exception level 0 in an ARM processor [2]. We have decided to use PMC 5 for our experiments, but any PMC other than 0 and 1 could have been used. Afterward, we must set the SYS_APL_PMESR0_EL1 register to select the indirect branch misprediction event for PMC 5. The exact value to which the SYS_APL_PMESR0_EL1 needs to be set was determined through the experiments described in [5].

All experiments starting from section IV follow the same template in which all indirect branches tested are put inside a loop that runs for a number of iterations. The experiments described in section IV use 10 loop iterations, and all experiments in later sections use 3. The general template corresponds to the following assembly code:

```
    mov  x1, <loop iterations>
    ...
init:
    adr  x10, start
    isb
    msr  x5,  PMC5
    isb
start:
    <loop body>
    dsb  sy
    subs x1, x1, 1
    beq  end
    b    init
    ...
end:
    isb
    mrs  x6, PMC5
    isb
    sub  <num_misses>, x6, x5
```

The mrs instruction in the assembly code above is used to retrieve the current value of the PMC. The number of indirect branch misses for the experiment is obtained by subtracting the value of the PMC obtained before the loop starts from the value of the PMC measured after the end of the loop. In addition to the code above, all experiments from sections V to IX flush the BTB before the loop is executed. The following assembly code accomplishes this flushing operation:

```
    mov x3, 12
    dsb sy
    adr x10, .
    add x10, x10, x2
    br  x10
    dsb sy
    <repeat last 3 instructions 8192 times>
```

This series of instructions fills up the BTB and ensures that results from previous experiments do not affect the current experiment.

The cset command was used to reserve a core exclusively for the experiments. Core 7, which uses the firestorm microarchitecture, was used for all experiments and the cset command ensures that no other processes are running in that core.

III. BTB Aliasing

This experiment was used to determine if there is any aliasing between consecutive branches, which would mean the hashing function does not use the least significant bits of the virtual address of the branches. This possibility can be tested by designing experiments with an increasing series of consecutive branch instructions. These experiments are implemented by loading the address of the first branch instruction into register x10. Then, the address of every register n used in the indirect branch instruction was set to the value of register $n-1$ plus 4. Then, finally, a series of consecutive indirect branch instructions was executed, with each one using a different register as its argument. The exact ARM assembly instructions used were the following:

```
    adr x10, start
    add x11, x10, 4
    add x12, x11, 4
    ...
start:
    br  x11
    br  x12
    ....
```

A maximum of 16 consecutive branches were used, with every branch always jumping to the next consecutive instruction. If there were any aliasing, executing a number of consecutive instructions greater than the associativity of the BTB would always result in conflict misses.

Since no indirect branch misses were ever recorded for any of the runs of this experiment, and the associativity of the BTB is unlikely to be 16 or greater, we can safely conclude that there is no aliasing in the BTB. In addition, since the ARM architecture requires instructions to be 4-byte aligned, meaning that the 2 least significant bits of the address of every

instruction are always set to 0, we can determine that the least significant bit used in the BTB hashing function is bit 2 of the 48-bit virtual address of the respective branch instruction.

IV. BTB Size

Experiments that fill up the BTB in different patterns were performed to determine its size. Subsection IV-A describes a series of experiments in which an alternating series of branch instructions of different sizes was executed. This alternating series of instructions was crafted in a way in which half of the indirect branch instructions map to the BTB sets with an even index, and the other half maps to the odd sets in the BTB. Subsection IV-B describes experiments in which the branch instructions are separated by an arbitrary number of no-operation instructions.

A. Alternating Branch Pattern

This experiment mainly involves executing an alternating sequence of indirect branch instructions to fill up the BTB and determine its capacity. This experiment is the first one that uses the general loop structure described in section II. Before the loop starts, register `x10` is initialized with the address of the start of the loop, and register `x11` is initialized with the address of the instruction halfway into the loop body and is incremented by 4 rights afterward. The first half of the body starts with an add instruction that increments `x10` by 8 and a branch instruction that uses `x10` as its target. This pattern is repeated for the first $\frac{n}{2}$ indirect branch instructions. The second half of the loop inverts this pattern by starting with an indirect branch to register `x11` followed by an add instruction that increments `x11` by 8. The general pattern of the assembly code and the expected BTB mappings are shown in Figure 1 This specific pattern of branch instructions ensures that no

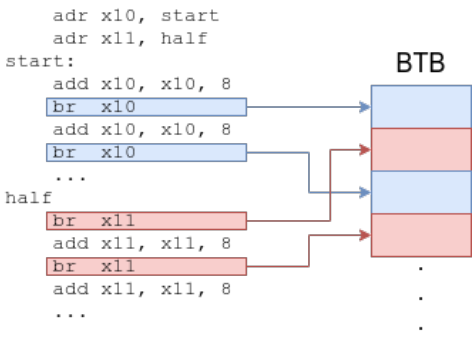

Figure 1. Assembly code and expected BTB mappings for the experiment using an alternating sequence of branches. The first $\frac{n}{2}$ indirect branch instructions map to the blue BTB entries, and the other indirect branch instructions map to the red ones.

entries of the BTB are skipped and minimizes the number of conflict misses in the BTB. This experiment was done for an increasing number of branches inside the loop, and the results are shown in Figure 2 and Figure 3

The results shown in Figure 2 indicate a BTB capacity of 2048 elements. This conclusion can be drawn because the

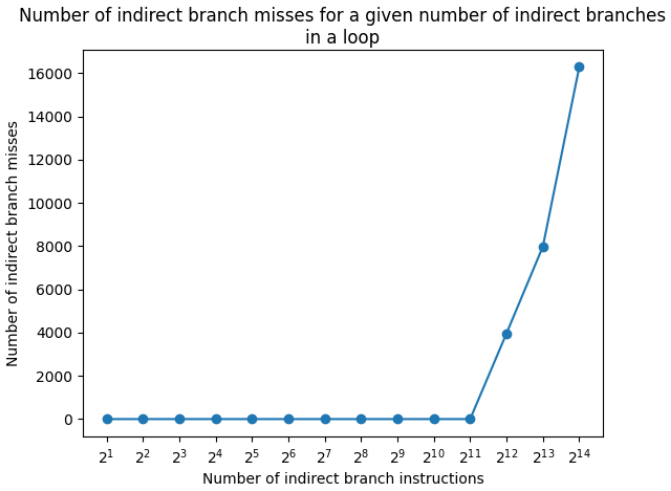

Figure 2. Logarithmic-scale graph of the number of indirect branch misses for each power-of-two number of branches inside a loop. The number of misses is 0 for 2048 branches or lower. For any higher power-of-two number of branches, the number of misses is close to the total number of misses in the loop

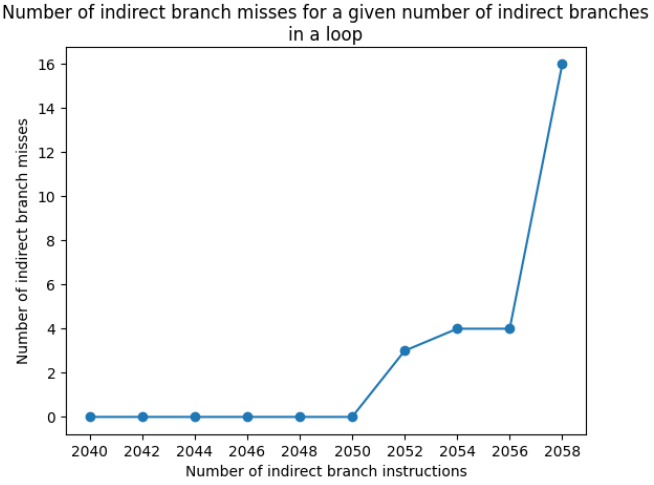

Figure 3. Graph of the number of indirect branch misses for a number of branches ranging from 2040 to 2058. The number of misses is 0 for a total of 2050 branches or lower.

largest power-of-2 number of branches that result in no indirect branch misses is 2^{11}. Furthermore, for all other powers of 2 greater than 2^{11}, the number of indirect branch misses nearly equals the total number of indirect branches in this experiment. This observation indicates that all of those powers of 2 are multiples of the total capacity of the BTB, which serves as further evidence that the BTB indeed has 2048.

Figure 3, on the other hand, may seem to suggest that the BTB actually contains a total of 2050 entries, but this is not contradictory to the hypothesis stated in the previous paragraph. Instead, this observation indicates that the BTB may contain an eviction cache in addition to its 2048 main entries. These results, however, cannot be used to determine

the exact size of this eviction cache due to the lack of precision of the PMC measurements. This issue will be addressed in the later experiments.

B. Branches Separated by NOPs

Experiments containing a sequence of indirect branches separated by a variable number of no-operation (NOP) instructions were designed to investigate the placement policy used in the BTB and test if the results obtained in subsection IV-A are accurate. In these experiments, the loop will start with an add instruction that will increment the branch target register, followed by a number of NOP instructions and an indirect branch instruction. The amount by which the branch target register is incremented depends on the number of no-operation instructions. Every branch is guaranteed to jump to the instruction immediately after it. This procedure corresponds to the following assembly instructions:

```
    mov x2, <(number of NOPs + 1) * 4>
start:
    add x10, x10, x2
    nop
    ...
    br  x10
    add x10, x10, x2
    nop
    ...
    br  x10
```

There are two different variations of this experiment: one in which the number of NOPs between each branch is always the same and another in which the number of NOPs between each instruction is randomized. For the experiments with a constant number of NOP instructions, a low even number of NOPs will result in the same results as the even and odd branch experiments described in **??**. This observation means that these experiments result in the same pattern as the one shown in Figure 1. An odd number of NOPs, however, will start registering a high number of indirect branch misses for 2048 branches since it results in the pattern shown in Figure 4. This phenomenon occurs because an odd number of NOPs will result in a sequence of branches whose address differs by an even multiple of four. This results in only half of the entries of the BTB being accessed since the branches will only map to the even entries of the BTB. Because of that, all branches executed after half of the BTB entries are filled will result in a BTB miss. In addition to the previously mentioned results, certain numbers of NOPs might result in a high number of misses for 1024 and 512 branches, as shown in figures 5 and 6

Figure 5 and Figure 6 show that certain branch separations maximize the number of conflict misses. In addition, these conflict misses seem to become more common as the instructions are spaced further apart. This observation ultimately provides an insight into what the hashing function is, and this consistent manner of inducing conflict misses will help us find eviction sets.

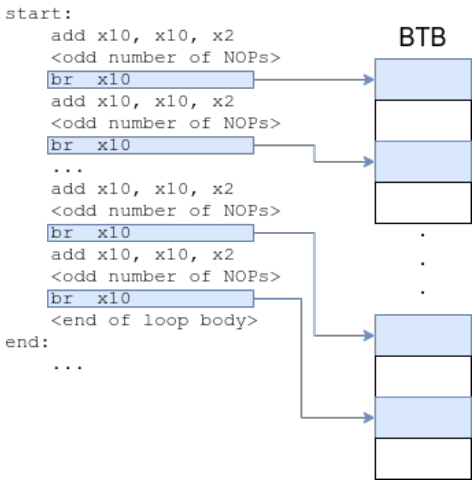

Figure 4. Assembly code and expected BTB mappings for the experiments in which the branches are separated with a constant odd number of NOP instructions. Only the blue entries will be accessed, and the white entries will be left unused. As a result of that, any branch instruction executed after half of the BTB sets are filled up will evict the entry used for one of the previous branch instructions

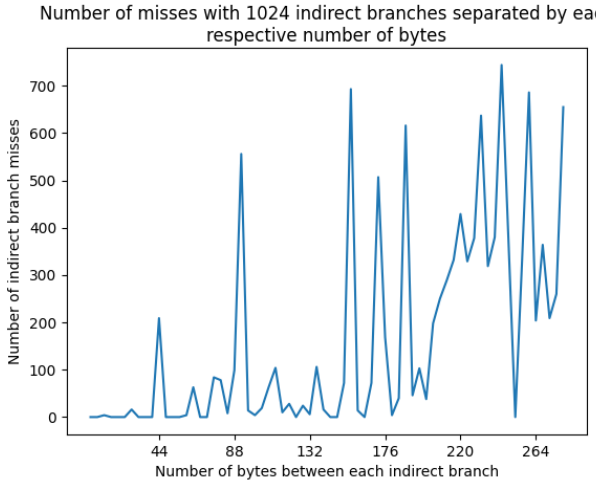

Figure 5. Graph for the number of indirect branch misses for a total of 1024 branches separated by an increasing number of bytes. The first significant peak in the number of misses occurs when branches are 44 bytes apart from each other. The frequency of the peaks increases as the separation between the branches increases

Adding a random number of NOPs between each branch always in a non-zero number of misses being recorded for any number of branches higher than 1024, although the exact number of misses obtained each time the experiment is run varies. This result indicates a high number of conflict misses, possibly due to the nature of the hashing function.

V. Finding an Eviction Set

To facilitate the process of finding an eviction set, we have decided to start with a loop containing branches that result in a high number of conflict misses in the BTB. Therefore, we decided to base ourselves on the results found in Figure 5 and

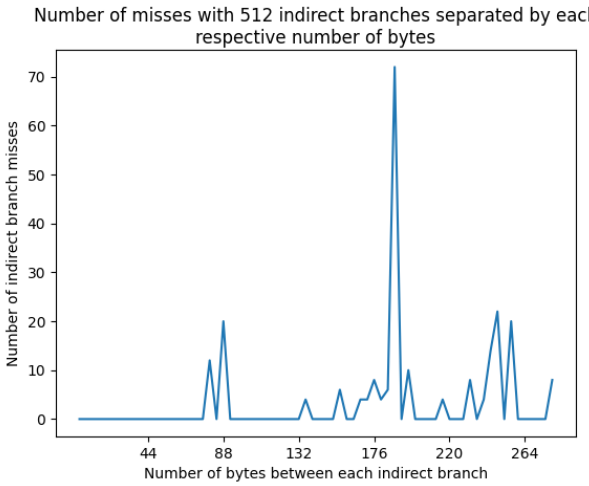

Figure 6. Graph for the number of indirect branch misses for a total of 512 branches separated by an increasing number of bytes

used a loop containing 1024 branches, each 44 bytes apart. The reason for this choice is the relatively high number of conflicts in this configuration and the simplicity of using a loop with a smaller number of instructions.

The first step to finding an eviction set in the loop configuration described in the previous paragraph is finding a branch that is a part of an arbitrary eviction set. To accomplish this, we decided to measure the number of indirect branch misses right after each branch instruction inside the loop and use the first branch that results in a BTB miss. This procedure was done by having a block of assembly code right after each instruction that would branch out in the last iteration of the loop right after the desired branch was executed. The assembly code used was the following:

```
dsb sy
eor x15, x1,  1
eor x16, x4,  x3
orr x16, x16, x15
cbz x16, end
add x4,  x4,  1
```

In the assembly code above, `x3` is a command line argument containing the index of the desired branch, and `x4` is the index of the last branch executed. By running the experiment, we have determined that the first branch that results in a BTB miss is the 26th branch of the loop.

To find the eviction set containing the 26th branch of the loop, we iterate through all branches of the loop, replacing each one with a NOP instruction and then checking the number of misses for the resulting loop body right after the 26th branch. To improve the accuracy of the result, we repeat the experiment for each branch a total of 5 times. If any of the five trials differ from the previous ones, we restart the experiment for the given branch. If we record at least one miss in all five trials, we conclude that the given branch is not part of the eviction set, and we keep the NOP instruction at that place of the loop. Otherwise, if no misses are recorded in any of the five trials, we determine that the given branch is a part of the eviction set and put the branch back into the program. After iterating through all 1024 branches in the loop, we determine our initial eviction set and restart the experiment for all branches in that eviction set. This process is repeated 50 times and can be described by the following pseudo-code:

```
for each branch:
    remove branch
    while run < 5:
        run program, get BTB misses
        if misses in run-1 != run:
            run = 0
        else:
            run++
    if misses == 0:
        put branch back
        evict_set.append(branch)
for i in 50:
    for branch in evict_set:
        evict_set.remove(branch)
        remove branch from program
        while run < 5:
            run program, get BTB misses
            if misses in run-1 != run:
                run = 0
            else:
                run++
        if misses == 0:
            put branch back
            evict_set.append(branch)
```

After running the experiment, we find an eviction set containing branches at addresses [0x1a018, 0x1d018, 0x1d048, 0x23018, 0x23048, 0x24008].

If we attempt to run a loop that only contains the branches in the eviction set obtained previously, the number of BTB misses obtained will be 0 most of the time. This observation could be explained by the fact that the indirect branches are always executed in the order in which they appear in the loop, and every indirect branch jumps to the instruction immediately after it. This predictable ordering of indirect branches allows the processor to bypass the BTB and still correctly predict the target of the indirect branches. To test this hypothesis and increase the precision of the PMC measurements, we decided to shuffle the order in which the branches are executed.

The order in which the branches inside the loop are visited can be shuffled by pre-computing the difference between the address of the branch instruction and the desired target. In our experiments, we set the desired target to be either 11 instructions before the next branch instruction or, for the last branch to be executed, the target is set to the instruction right after the last branch in the loop. This procedure ensures that the instructions needed to set up a given branch are executed and that only the desired branch is executed. In addition, an unconditional branch instruction that jumps to

the first instruction that needs to be executed is added 11 instructions before the first indirect branch in the loop. This instruction allows us to change the first executed branch in the loop. Furthermore, every branch inside the loop is guaranteed to jump to a `dsb sy` serializing instruction to ensure the precision of the PMC measurements. This structure corresponds to the following assembly code:

```
start:
    ...
    b   fst_br
    dsb sy
    add x10, x10, 8
    add x10, x10, 4
    <add x10, x10, 4 repeated 5 times>
    mov x11, <branch offset>
    [add/sub] x10, x10, x11
    dsb sy
    br  x10
    ...
    <other branches>
    ...
fst_br:
    <first branch to be executed>
    ...
    <other branches>
    ...
    dsb sy
    <add x10, x10, 4 instructions until end>
```

Changing the order in which the branches are visited considerably increases the precision of the PMC measurements, with some specific orderings resulting in more precise results than others. In fact, for the eviction set obtained above, some orderings always result in 6 indirect branch misses. The reason why these specific orderings maximize the number of BTB misses is currently unknown.

With this increased precision in the PMC measurements, we can trim down the eviction set by removing each element individually and measuring the number of misses in the resulting set. When taking these measurements, all possible permutations of the resulting sets are tested, and each experiment is run a total of 5000 times. If the number of misses is lower than 1000 for all possible permutations of the set, we can conclude that the set tested is not an eviction set anymore and that the removed element must be added back in. Using this procedure, we removed the branches at 0x1d018 and 0x23018 and still got an eviction set [0x1d048, 0x23048, 0x1a018, 0x24008] that results in a total of 20000 indirect branch misses. Removing any of the elements in the latter set will result in a low number of misses, which means that this is our final eviction set.

To obtain a different eviction set, we have decided to repeat the same process described in the previous paragraphs with a variation of the original 1024 branch loop in which the 26th branch is always replaced with a NOP. This results in an eviction set containing the indirect branches at [0x1e098, 0x1a0a8, 0x200a8], visited in that order. The number of misses obtained in this set is significantly lower and more inconsistent than in the other set, with the total number of misses obtained ranging from 3000 to 5000. These experiments, therefore, give us the following two eviction sets:

[0x1e098, 0x1a0a8, 0x200a8]
[0x1d048, 0x23048, 0x1a018, 0x24008]

The presence of an eviction cache in the BTB can explain the different sizes of the eviction sets. In particular, these eviction sets suggest that the BTB is directly mapped with a 1-element eviction cache. This hypothesis suggests that the 4-element eviction set is the result of the situation described in Figure 7, meaning that it is composed of elements that map to 2 different BTB sets. The 3-element eviction set, on the other hand, only contains elements mapping to the same eviction set and results in the situation described in Figure 8.

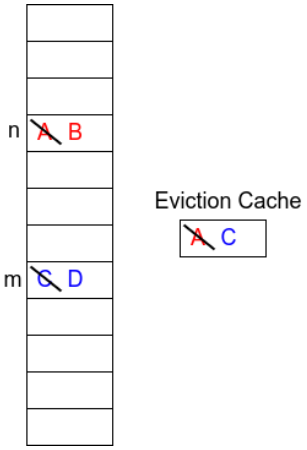

Figure 7. Diagram showing the state of a directly-mapped BTB with a one-entry eviction cache after executing branches with addresses in the eviction set [A, B, C, D]. In this case, addresses A and B map to BTB set n and addresses C and D map to BTB set m. When branch A is executed, it is placed in set n. Then, branch B evicts A from set n, and the latter branch is placed in the eviction cache. Afterward, executing branch C will place that branch in set m. Finally, when branch D is executed, it evicts C from set m, and the latter is placed in the eviction cache. Since the eviction cache already contains A, the latter element is evicted from the BTB.

Attempts were made to prove the hypothesis described in the previous paragraphs by removing one of the elements in the 3-element eviction cache and replacing it with 2 elements in the 4-element eviction cache. If the two elements chosen from the 4-element eviction cache map to the same BTB set, this would result in a situation similar to the one described in Figure 7. However, the actual results were inconsistent and not precise enough to draw any conclusions. This phenomenon could be due to the processor correctly predicting the branch targets without using the BTB. Our unsuccessful attempts to address these issues are described in section VI

VI. Attempts to Improve the Accuracy of the PMC measurements

Different methods were used in an attempt to improve the precision of the PMC measurements. Subsection VI-A

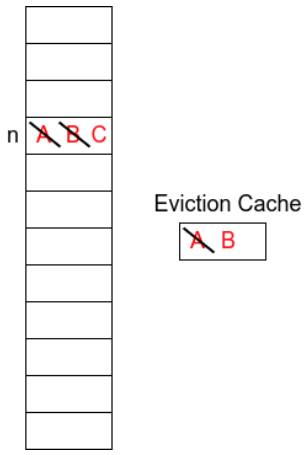

Figure 8. Diagram showing the state of a directly-mapped BTB with a one-entry eviction cache after executing branches with addresses in the eviction set [A, B, C]. All addresses in the eviction cache map to the same BTB set. When branch A is executed, it is placed in set n. Then, branch B evicts A from set n, and the latter is placed in the eviction cache. Finally, C evicts B from set n and places the latter in the eviction cache. Since the eviction cache already contains A, the latter is evicted from the BTB.

explains our attempt at using a linked list of branch offsets to calculate the branch targets. Subsection VI-B describes our attempts to improve the precision of the PMC by making the control flow of each loop iteration different.

A. *Offset List*

One possible reason for the BTB to be bypassed is that the offsets used to compute the branch targets are stored as literals in a `mov` instructions and, because of that, the processor might be able to infer the branch targets by using these offsets. Based on this hypothesis, an attempt was made to improve the accuracy of the PMC measurements by storing the offsets in a circular linked list. To prevent prefetching, each node of the list stores an offset to the address of the next node instead of storing a direct pointer. The pointer to the next node in the list is then computed by adding the offset to the address of the current node. In addition, the `dc civac` ARM instruction is used to flush the branch offsets from all levels of the data cache right after they are accessed. This process is accomplished through the following assembly code:

```
ldr x11, [x3]
add x10, x10, x11
ldr x12, x3,  [x3, #8]
dc  civac, x3
add x3,  x3, x12
dsb sy
br  x10
```

In the assembly code above, the `x3` register is initially set to the front of the linked list and is subsequently set to the next element of the list at each iteration.

Using a linked list to store the branch offsets had no impact on the accuracy of the results. This most likely means that instruction literals are not used to optimize the computation of branch targets, and, therefore, the usage of a linked list does not prevent the BTB from being bypassed. However, due to the limitations in the length of literals in ARM instructions, the linked list approach makes the experiments more scalable for branches that are further apart from each other and facilitates the computation of branch targets inside the loop. Because of that, all subsequent experiments use the same circular linked list approach to store the branch offsets instead of storing them directly as literals in `mov` instructions.

B. *Non-identical Loop iterations*

Another possible factor that could explain the bypass of the BTB is the processor correctly identifying that the branches are inside a loop and repeating the same control flow observed in the first iteration of the loop without having to access the BTB. If that were the case, having non-identical loop iterations might prevent that from happening, thus forcing the processor to use the BTB. One simple way to accomplish this would be to jump to a different block of code at each iteration of the loop and then, at the end of that block of code, branch back to the beginning of the loop to start the next iteration. To test this, a block of code composed of 5000 random `mul`, `sdiv`, and `adr` instructions was added right after the end of the loop. This block of code was divided into 2 sections, with each ending with an unconditional branch instruction that would jump back to the beginning of the loop. The first 2 iterations of the loop would each access a different section, and the last iteration would bypass the block of code entirely by jumping out of the loop. This procedure corresponds to the following assembly code:

```
    mov x1, 3
    <additional initialization>
init:
    <loop body>
    dsb  sy
    subs x1, x1, 1
    beq  end
    cmp  x1, 2
    beq  rand1
    bne  rand2
rand1:
    <random mul, sdiv and adr instructions>
    b    init
rand2:
    <random mul, sdiv and adr instructions>
    b    init
end:
...
```

The addition of this random block of code at the end of each loop iteration had no impact on the precision of the PMC measurements. This observation means that, even if the original hypothesis was true, this simple method of jumping to a different random block of code at each loop iteration does not make the loop iterations sufficiently different to prevent the

processor from using the control flow of the first loop iteration to bypass the BTB.

Ensuring that the order in which the indirect branches are executed is different at each iteration of the loop would bypass any mechanism that identifies the control flow of a loop iteration and repeats it in later iterations. This feat can be accomplished by adding two conditional direct branches right after the target of each indirect branch, resulting in different branches being taken in each of the 3 iterations of the loop. This procedure requires different branch offsets to be used in each iteration of the loop, meaning that the size of the linked list used to store the offsets would have to be tripled. This change in the order of execution of branch instructions can be accomplished through the following assembly code:

```
    tst x1, 3
    ...
start:
    ...
branch1:
    dsb sy
    beq branch2
    blt branch3
    ...
    br  x10
branch2:
    dsb sy
    ...
branch3:
    dsb sy
    ...
    <other branches>
    subs x1, x1, 1
    beq  end
    cmp  x1, 2
    ...
end:
    ...
```

In the assembly code above, none of the direct conditional branches would be taken in the first iteration of the loop since the flags would be cleared by the `tst` instruction executed before the loop start. Then, after the first iteration is over, the `cmp` instruction would set the zero flag, meaning that all `beq` branches would be executed in the second iteration of the loop. Finally, in the last iteration of the loop, the `cmp` instruction would clear the zero flag and set the negative flag, which means that only the `blt` branches would be taken. This design ultimately allows the order in which the branches are executed in each loop iteration to be different while keeping the target of every indirect branch constant across all loop iterations. As a result, executing an indirect branch with an entry in the BTB would still result in a hit since the actual target of the branch would not change across loop iterations.

This strategy of modifying the control flow at each loop iteration had the exact opposite effect since it lowered the precision of the PMC measurements. In fact, for eviction sets that would have previously resulted in a high number of misses, implementing this loop randomization strategy would result in 0 misses being recorded across all 5000 iterations of the experiment. The exact reason for this is not entirely known, but it does indicate that this strategy resulted in the BTB always being bypassed. As a result, this strategy cannot be used in further experiments, and the hypothesis regarding the reproduction of the control flow across all loop iterations seems false.

VII. Address Bits Used in Hashing Function

Since the size of the BTB is considerably smaller than the 48-bit virtual address space of the Mac M1, we can already assume that not all bits in the virtual address space of a branch are used in the hashing function of the BTB. This observation means that two different branches at different addresses will always end up in the same BTB set as long as the virtual address bits used in the hashing function of both branches are set to the same value. The aliasing experiment described in section III has already demonstrated that the least significant bit used in the hashing function is bit 2. With that in mind, we can start examining the most significant bits of the address to determine the most significant bit used in the hashing function.

To determine which bits of the virtual address of the branches are used in the hashing function of the BTB, we used an experiment that would flip each bit of the virtual address of the elements in an eviction set and place the branches at the respective location. In particular, we started with the [0x1e098, 0x1a0a8, 0x200a8] eviction set and flipped the same bit of 2 elements of the set while keeping the other element constant. We used the Linux `mmap` system call to map a page to the higher addresses where the branches will be placed. By setting the page's read, write and execute permissions, we were able to copy the instructions required for executing the branches to the desired location at runtime and then have these instructions executed when needed. The instructions used to execute the branches are identical to the ones shown in subsection VI-A. All possible permutations of the set were tested, with the constant branch always being the first to be executed and the other two branches having one of their bits flipped. The experiment flipped bits 18 to 46 of the virtual address of the branches, and each iteration was repeated a total of 5000 times. The total number of misses across all trials was then recorded.

The results indicate that the highest-order bit used in the hashing function is bit 30, meaning that bits 31 and higher are ignored. This conclusion can be drawn because flipping bits 30 or lower will always result in less than 100 misses, meaning that resulting addresses are not a part of the same BTB set anymore. On the other hand, flipping bits 31 or higher will consistently result in a number of misses of around 5000, which means that all three branches are being mapped to the same set since these higher-order bits are ignored. In addition, these results are consistent across all possible permutations of the set, meaning that the specific order in which the branches are visited does not impact the total number of misses. Thus, flipping one of the higher-order unused bits of

an address increases the precision of the PMC measurements. This observation will be crucial in designing the experiments used to test the associativity of the BTB.

VIII. BTB Associativity

The results from section VII demonstrate that we can construct an arbitrary eviction set as long as we have branches with virtual addresses in which bits 2 to 30 are the same value for all branches and bits 31 to 47 are set to a different value for each branch. In addition, if we build an eviction set using this method, we do not need to check all possible permutations of the set since setting one of the higher-order bits of the branch address will make the results consistent across all possible branch ordering. Based on these observations, we can verify our hypotheses regarding the associativity of the BTB by building eviction sets using the previously mentioned method.

To determine the associativity of the BTB, we designed eviction sets of varying lengths by setting bits 2 to 30 of all branches to the same random value, setting all higher-order bits of the first branch in the set to 0, and setting the higher-order bits of the other branches to different random values. The same method used in the experiments described in section VII was used to place the branches at the appropriate addresses. Each experiment used a unique set of branch addresses and was repeated 5000 times, with the total number of misses across all trials being recorded. If the size of the generated set is large enough to form an eviction set, then all possible sets with that size will result in a number of misses that is higher than 5000. Otherwise, if the set is not large enough, the number of misses will be considerably lower.

The experiment results suggest that at least 3 branches are needed to construct an eviction set. Any set of that size will always result in a number of misses higher than 10000. On the other hand, sets with only 2 elements have an inconsistent number of misses that are usually well below 5000. Sometimes, however, experiments with 2 element sets will result in a number of misses close to or even higher than 5000. This could be caused by the operating system interrupting the process running the experiment and running code that inflates the total number of misses. This result supports our hypothesis of a directly mapped BTB with a one-element eviction set since at least 3 elements mapping to the same set are needed to evict an element from that BTB set. However, further experiments have to be performed to prove this hypothesis.

To test our hypothesis of a directly mapped BTB with a one-entry eviction set, we need to generate a set containing elements that map to 2 different sets so that the pattern shown Figure 7 can be reproduced. To do that, we need to test sets composed of two subsets. The first subset must always contain exactly 2 elements that map to the same BTB set. The second subset would then contain elements that map to a set different from the first subset. The length of the second subset needs to be varied across experiments, and we must ensure that all elements in that subset map to the same set. Each subset was generated using the same method described in the previous experiment. Due to the random nature of the generated addresses, it is possible to obtain a set in which both subsets map to the same BTB set. This phenomenon should not occur very often and can be accounted for in the experiments. As in the previous experiment, each experiment uses a unique set and is repeated a total of 5000 times. If an eviction set is obtained, the number of misses will always be higher than 5000

The results of the second experiment support the hypothesis of a directly mapped BTB with a one-element eviction cache. If we have a 4-element set in which both subsets contain 2 elements, the number of misses will always be 20000. If we, instead, have a 3-element set in which the second subset contains only 1 element, the number of misses is inconsistent, but it often is well below 5000. As with the previous experiment, however, the number of misses for a 3-element set will sometimes result in a number of misses that is close to or higher than 5000. This can be explained by the possibility of both subsets mapping to the same BTB set or, as in the first experiment, the probability of the operating system interrupting the process running the experiment. The high consistency of the results obtained for sets with 4-element supports the hypothesis that the BTB is directly mapped with a 1-element eviction cache.

IX. Branch Address Bit Flip Experiments

A series of experiments were designed to test the effects of flipping certain bits in the virtual address of one of the branches in an eviction set to determine whether the hashing function of the BTB uses simple xor operations between the different bits of a virtual address. The first series of experiments involved flipping each bit between bits 6 and 30 of the virtual address of an indirect branch. The other experiments involved flipping every possible combination of two bits between bits 6 and 30. Three different types of eviction sets were generated for these experiments. The first one was generated using the same method described in section VIII. The second type of set was generated in a similar way to the first set, except that bits 47 to 33 of the address of 2 elements of the set were always set to 0, and bit 32 of both of these elements was always set to 1. The third type of set was based on the 3-element eviction set obtained in the experiments described in section V and contained two elements with their address bit 32 set to 1. Each experiment was run a total of 5000 times, and the total number of misses across all trials was recorded.

For the first type of set, in which bits 46 to 32 were set to a random value, every possible bit flip would always result in a set that would result in 15000 misses. The exact reason for this is currently unknown, but it might indicate that either all possible bit flips result in an eviction set or, for some reason, the branch targets for these addresses are not being placed in the BTB. As a result of that, no meaningful conclusions could be extracted by using this type of eviction set.

For the second type of set, in which all elements would have their lower 30 bits set to the same value, and 2 of the elements would have bit 32 of their address set, flipping any bits from 22

to 30 would always result in misses higher than 10000 as long as none of the other bits are flipped. If, however, any bits from 6 to 21 are flipped, the results obtained are inconsistent. This observation holds for both the single and double-bit flipping experiments. These results indicate that the hashing function is most likely not a simple xor function between different bits of the virtual address of a branch. In addition, these results may seem to indicate that bits 22 to 30 are not used in the hashing function. However, this hypothesis is contradicted by running the bit-flipping experiment with the third type of set, in which all three elements contain their lower 30 bits set to one of the addresses present in the 3-element eviction set obtained in section V. More specifically, if we start with the set [0x200a8, 0x10001a0a8, 0x1000200a8] and start flipping the bits of the third element of the set, those bit flips from bits 22 to 30 often result in a number of misses lower than 1000. This, therefore, means that bits 22 to 30 are indeed used in the hashing function of the BTB and that further experiments would have to be performed to draw any conclusions regarding the hashing function itself.

X. Conclusion

The results obtained in this semester project indicate that the branch target buffer (BTB) used in the firestorm cores of the Apple M1 is a directly-mapped cache with 2048 sets and a 1-entry eviction cache. Furthermore, the hashing function used in the BTB only seems to use bits 2 to 30 of the virtual address of the indirect branch instructions, with bits 31 to 48 being ignored. The bit-flipping experiments suggest that the hashing function used in the BTB is not a simple xor operation between different bits of the virtual address of the branch instructions. Further work must be done to determine the exact hashing function used.

The experiments have also resulted in a few observations that do not currently have a plausible explanation. One of them is that the number of misses for a set of branches that make up an eviction set seems to be highly dependent on the order in which the branches are encountered. In particular, different orders result in a highly divergent number of misses. This observation could suggest an additional branch-prediction mechanism that bypasses the BTB if branches are encountered in specific orders. However, this dependence on the ordering of branches is not observed when the upper bits of the branch addresses are set, meaning that, in this case, the number of misses for an eviction set is consistently high for any possible order. Another unexplained phenomenon is the consistently high number of BTB misses encountered in the bit-flipping experiments when 2 of the branches in a 3-element eviction set have different higher-order bit set, which is not the case when both elements have the same higher-order bit set. Further experiments will need to be performed to uncover the mechanism that dictates these phenomena.

References

[1] Andrei Frumusanu. "The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test". In: *AnandTech* (Oct. 17, 2020). URL: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested (visited on 01/15/2023).

[2] Ensar Ilhan. *Asahi Linux Documentation*. Sept. 2, 2022. URL: https://github.com/asahilinux/docs/wiki (visited on 01/15/2023).

[3] Dougall Johnson. *Apple M1 Microarchitecture Research*. URL: https://dougallj.github.io/applecpu/firestorm.html (visited on 01/15/2023).

[4] Marek Majkowski. *Branch predictor: How many "if"s are too many? Including x86 and M1 benchmarks!* The Cloudflare Blog. May 6, 2021. URL: http://blog.cloudflare.com/branch-predictor/ (visited on 01/15/2023).

[5] Andrej Gorjan and Atri Bhattacharyya. *Reverse engineering the Apple M1 front end*. EPFL, Jan. 7, 2022.

[6] Arm Limited. *Arm Armv9-A A64 Instruction Set Architecture*. 2021. URL: https://developer.arm.com/documentation/ddi0602/2021-12/ (visited on 01/18/2023).

[7] Arm Limited. *Learn the architecture - ARMv8-A memory systems*. 2022. URL: https://developer.arm.com/documentation/100941/0101/ (visited on 01/18/2023).

[8] Arm Limited. *Arm A-profile Architecture Registers*. 2022. URL: https://developer.arm.com/documentation/ddi0601/2022-03/ (visited on 01/18/2023).

Appendix

The more uncommon ARM instructions used in the experiments are described in Table I

Instruction	Description
`mrs`	Move System Register instruction used to read an AArch64 System Register into a general-purpose register [6]. This instruction is used to read the PMC registers
`msr`	Move value to an AArch64 System Register [6]. This instruction is used to write into the PMC registers.
`isb`	Instruction Synchronization Barrier instruction that ensures that all context-changing operations after the `isb` instructions only take effect after the `isb` instruction has completed. It also ensures that the effects of all completed context-changing operations before the `isb` instruction are visible to all instructions after the `isb` instruction [7]. According to [2], writes to the PMC control registers need an `isb` instruction to take effect.
`dsb sy`	Data Synchronization Barrier instruction that ensures that all instructions before the `dsb` instruction are completed before the instructions after the `dsb` are executed. This effectively prevents the reordering of instructions across the `dsb` instructions. The `sy` operand makes this instruction a full system barrier [7].
`dc civac`	Clean and invalidate the data cache line mapped to the virtual address contained in the register used as an operand. The contents of the respective cache line are written back to memory if the respective dirty bit is set. All levels of the data cache hierarchy are affected by this instruction [8].

Table I
Description of the uncommon AArch64 instructions used in the experiments.