Reverse engineering the Apple M1 front end

Andrej Gorjan
andrej.gorjan@epfl.ch

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

January 7, 2022

Abstract

The Apple M1 has generated a lot of interest as one of the first desktop class ARM based processors. Apple has achieved an impressive level of performance per watt but has not disclosed any details about how they did it. Efforts have been made to reverse engineer the microarchitecture, but have thus far not not focused on branch prediction. In this Semester Project, we use simple benchmarks to uncover some elementary details about the predictors in the M1.

1 Introduction

In 2020, Apple announced that it will be replacing Intel processors in its desktop computers with in-house, ARM based chips. The first of these processors is the M1, which matches the performance of offerings from Intel and AMD at a lower power consumption [1]. As of the date of writing, Apple has only released some high level information about the microarchitecture such as the number of cores and sizes of caches.

The lack of published information led to efforts to reverse engineer the M1, which are described in section 2. This work focused more on aspects like the width of the pipeline, number of execution units and sizes of various buffers and queues. We were not able to find any published research into the front end of the M1, particularly into its branch predictors.

In this Semester Project, we attempt to uncover details of the branch predictors implemented in the M1 processor, specifically branch outcome prediction and branch target prediction. In section 2, we overview existing reverse engineering work. Section 3 covers our testing methodology. In sections 4 and 5 we present our findings for the outcome and target prediction. We make our conclusions in section 6.

2 Related work

While we were not able to find any papers about the microarchitecture of the M1, a few people have been making measurements and sharing their findings.

Andrei Frumusanu[2] tested Apple's A14 processor, which shares its architecture with the M1, specifically the Firestorm high performance core. His results indicate that Apple implemented a surprisingly wide architecture decoding up to 8 instructions every cycle. His work seems to agree with that of Dougall Johnson[3], who provides detailed benchmarks for each instruction in the ISA.

Although they did not look into the branch predictors of the M1, their results were quite useful for distinguishing the two types of cores on the processor. Furthermore, Johnson published the source code he used to get his results. Beyond studying it, a kernel extension he wrote was crucial for accessing performance counters. More details on the use of their work can be found in section 3.

One person that did measure some aspects of branch target prediction was Marek Majkowski[4]. He was interested in how the number of branches impacts their latency and the cost of misprediction. He concluded that latency is lowest when branches are placed in a "4096" pattern. That is *1024 branches whose addresses differ by 4*, *512 branches whose addresses differ by 8* and so on. He made no further claims about the size or associativity of the branch target buffer.

Majkowski relied on benchmarking code adapted from the Masters thesis of Vladimir Uzelac[5], who described methods of reverse engineering branch prediction units. The benchmarks in the thesis were a great reference for us and some of the code we use was modeled directly after them. An article by Milenkovic[6] uses a similar approach.

Maynard Handley shared an extensive but unfinished document[7] in which he discusses the microarchitecture of the M1, including branch prediction. He shows no experimental results, but rather describes the goals and limitations of the circuitry. The basis of the document are Apple's patents, including one from 2015[8] that shows an implementation of the TAGE predictor[9].

Finally, the Asahi Linux project[10] aims to bring the Linux operating system to Apple silicon devices. During the course of their work, collaborators compiled a collection of documents describing the hardware in the M1 processor. Their list of system registers was particularly helpful when taking measurements.

3 Workflow

The results of the experiments in this report were gathered on a 2020 Mac Mini with Apple's M1 processor. Initially, we were planning to run Linux on it which would give us more low level access to the hardware. In particular, we would be able pin threads on certain cores and set their frequencies. Asahi Linux seemed promising, but due to it being in the early stages of development we were not able to get it to work on the Mac. In the end, we decided to move on with out work on macOS.

3.1 Performance counters

It is common for manufacturers to equip their processors with hardware that is used to evaluate performance. This is usually in the form of *Performance Monitor Counters* or PMCs. These counters are incremented whenever a certain event occurs in the processor, such as a branch being mispredicted. While intended to optimize software, they can be immensely useful for reverse engineering. For example, Uzelac relied on PMCs to gather the measurements in his thesis. We will use the PMCs on the M1 to gather our measurements as well.

Apple's open source code features many references to PMCs and other registers associated with them. Further details were gathered from the Asahi Linux documentation and Johnson's work. The M1 contains 10 PMCs labeled PMC0 to PMC9. PMC0 always counts clock cycles and PMC1 always counts retired instructions. The remaining counters can be configured to track different events. The configuration is performed by writing certain values to other system registers:

- `PMCR0` is the main control register that enables PMCs and their interrupts
- `PMCR1` enables counters in different processor modes (EL0, EL1)
- `PMESR0` selects events for PMCs 2, 3, 4 and 5
- `PMESR1` selects events for PMCs 6, 7, 8 and 9

By default, these registers are not accessible from user space as one must first set bit 30 in the `PMCR0` register. Fortunately, a kernel extension developed by Johnson exposes PMC configuration registers as `sysctl` parameters. Along with the extension, his code also contains some configuration values to write into the registers, but he never explains them. We felt somewhat hesitant to proceed with our experiments without being certain that the PMC configuration was correct.

Apple provides no official guide to working with the registers and instead instructs developers to use the profiling component of the Xcode IDE. We decided to extract the values for the registers from the IDE by writing a simple program that prints out their values. When we "profile" this program, we obtain the proper configuration for tracking each one of the supported events.

The configuration registers are reset to default values during a context switch. On one hand, this makes gathering data more time consuming since many runs fail. It also prevents us from running very long experiments. On the other, it is very obvious when the OS de-schedules our code while it is running. The next time our code will attempt to access the PMCs, it will instantly fail. This makes the lack of core pinning somewhat less critical.

In our experiments the PMCs are read with the `mrs` instruction, which may be executed out of order. We prevent this with the `isb` serialization instruction that causes all instructions in the pipeline to complete. The instructions that follow continue to be fetched only after the completion of `isb`.

3.2 Identifying cores

One of the key features of the M1 is that it contains both high performance *Firestorm* and power efficient *Icestorm* cores. It is reasonable to expect that the predictors of the two types of cores will be different. To properly interpret our results, it is important to be able to tell on which core they were gathered.

As mentioned previously, Apple does not provide a way to pin threads to cores. Instead, we can provide the scheduler with a *hint* as to which core the thread should be scheduled on. This hint is usually considered, but not always, so we need a definitive way to distinguish cores.

Existing research into the M1 microarchitecture indicates that Firestorm cores have 4 floating point execution units while Icestorm cores only have 2. By measuring how many cycles it takes to execute a large number of floating point instructions, we should be able to easily tell which core we are running on. We created a function that measures the amount of clocks needed to complete a loop with 1000 iterations. The body of the loop contains 8 floating point instructions that have no dependencies between them or between consecutive runs of the loop. An Icestorm core should take twice as long to execute this loop than a Firestorm core, which is exactly what we see in Figure 1.
Because the peaks are very distinct, we could have chosen multiple values to be the "limit" but we settled on 3500. At the beginning of our experiments, we run the same loop with floating point instructions. If it takes less than 3500 cycles to complete, we know we are on a Firestorm core and if it takes more, we are certainly on an Icestorm core.

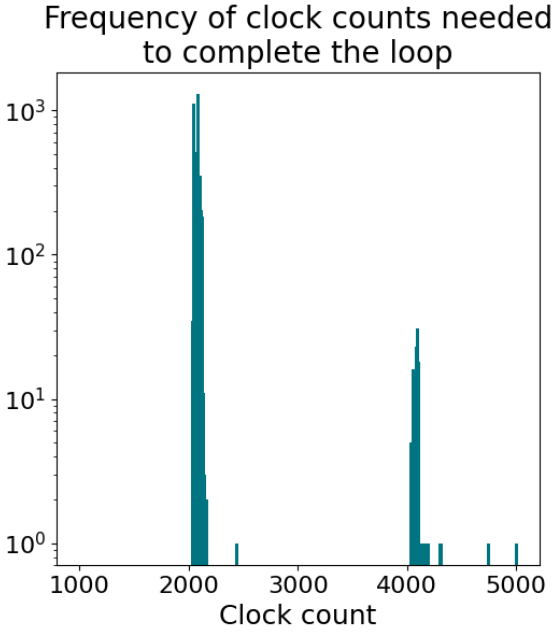

Figure 1: A peak at 2000 cycles per loop indicates the Firestorm core and one at 4000 cycles is the Icestorm core.

4 Branch outcome predictor

Branch outcome prediction is what most people think of when discussing the topic. The task is simply to predict whether a conditional branch will be taken or not taken. Some highly accurate branch predictors rely on information about past branching behavior to make this decision. There are two common approaches:

- `BRANCH HISTORY REGISTER - BHR`
  A shift register is used to "remember" the past outcomes of branches. It can be local to a certain branch or global (shared between all branches).

- `PATH HISTORY REGISTER - PHR`
  This register stores information about the sequence of addresses that led the processor to the branch. For example, addresses of basic blocks on the execution path could be continuously folded in to the register using some hash function.

Both of these methods aim to exploit any correlation between past behavior and the outcome of the branch. Since these kinds of predictors have proven to be so effective, we believe it is reasonable to assume that M1's branch prediction also uses such an approach. Specifically in the experiments that follow, we will assume the presence of a `BHR`.

A sequence of experiments described in [6] can help us determine some simple details of the branch predictor, like the length of the `BHR`. Suppose we are executing a program, whose control flow is depicted in Figure 2. Implementing such a program in `aarch64` assembly produces two conditional branches, one for the loop and another inside the body for the `if` statement. Let us say that '1' means that a conditional branch was taken and '0' that it was not taken. The pattern of the internal branch may look like

$$\underbrace{1111...1}_{\text{m-1}}0$$

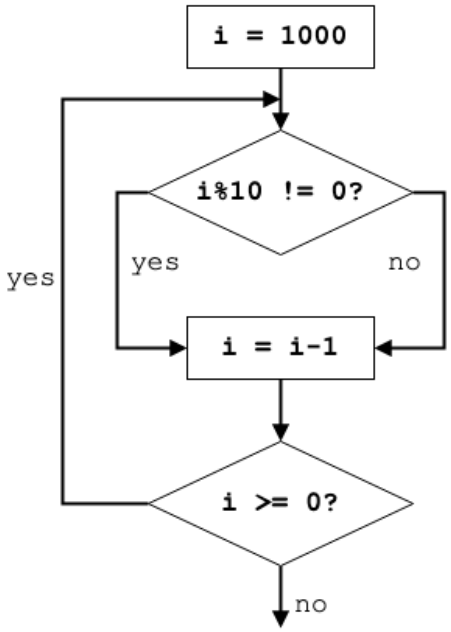

Figure 2:

Suppose that we are executing this code on a processor with a local `BHR` of length 4, as depicted in Figure 3. We can see that the pattern of the branch can be up to 1 greater than the length of the `BHR` for us to be able to accurately predict the outcome based on history alone.

If the processor has a global `BHR` the maximum possible length of the pattern (m) is halved. After every `if` statement branch, we have one from the loop consuming space, as depicted in Figure 4.

Let us look at how the value of m effects the number of mispredictions on the M1. We write the loop mentioned earlier in assembly language to guarantee we get only two condi-

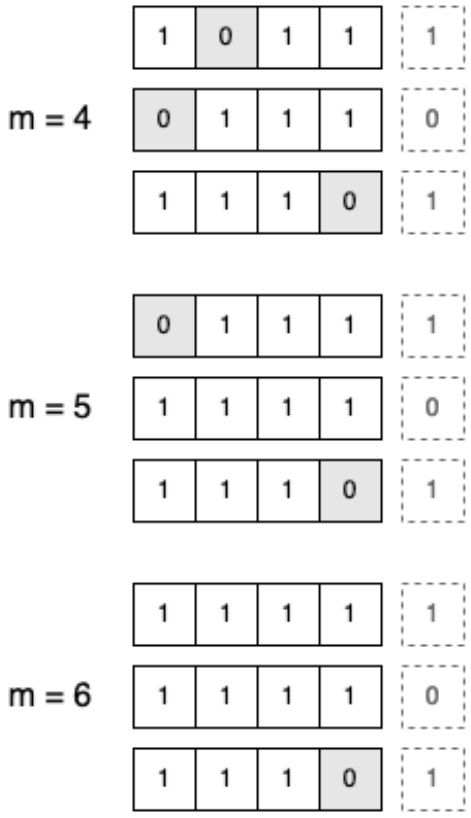

Figure 3: The `BHR` is represented as a series of cells storing a '1' or '0' depending on the branch history. The next outcome is in the dashed cell on the right. In the $m = 6$ case, the same "state" of the `BHR` is followed by different outcomes.

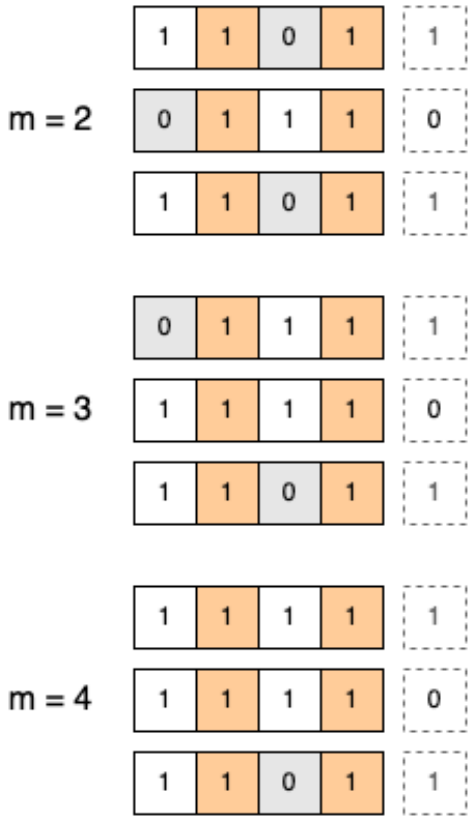

Figure 4: When the `BHR` is global, other branches (shaded orange) will consume space, reducing the longest possible pattern it can capture for a certain branch.

tional branches. We set the iteration count to $1000 \times m$ to make sure each pattern repeats an equal amount of times. This will give each run of the benchmark an equal number of opportunities to mispredict the last branch. We configure a PMC to count the number of retired conditional branches that were mispredicted. Then, we sample the counter before and after the loop and calculate the difference.

Since the loop branch is taken every single time except at the very end, it will cause only a small number of mispredictions after the predictor learns to classify it as always taken. Thus, the difference will be a good estimation of the number of mispredictions of

the `if` statement branch. The average number of mispredictions over 100 runs of such an experiment is depicted in Figures 5 and 6.

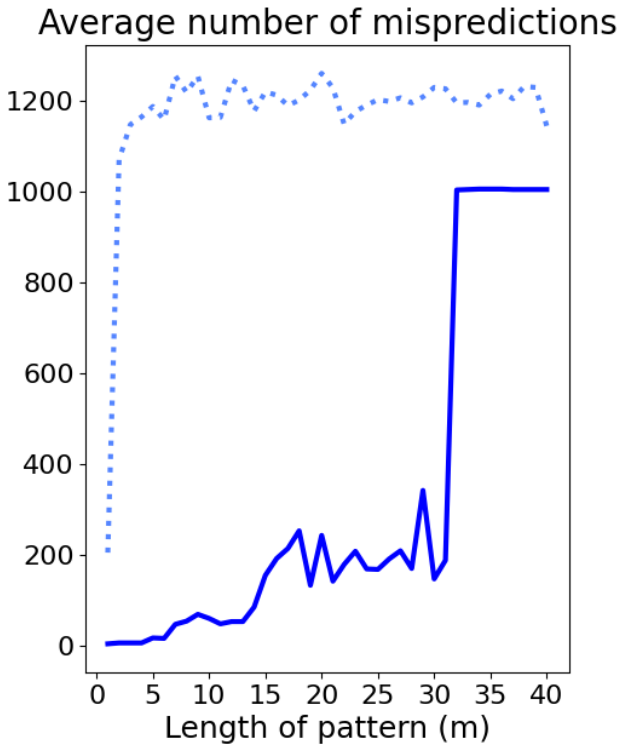

Figure 5: Results for an Icestorm core. The solid blue line represents the number of mispredictions after running the loop. The dotted light blue line is the same code, but with 31 conditional branches executed before the `if` statement branch.

Both graphs show a clear pattern, where the number of mispredictions is low, as long as the length of the pattern does not exceed a certain length. After that, we consistently get around 1000 mispredictions, which means that on average, one branch is mispredicted per one 111..110 pattern.

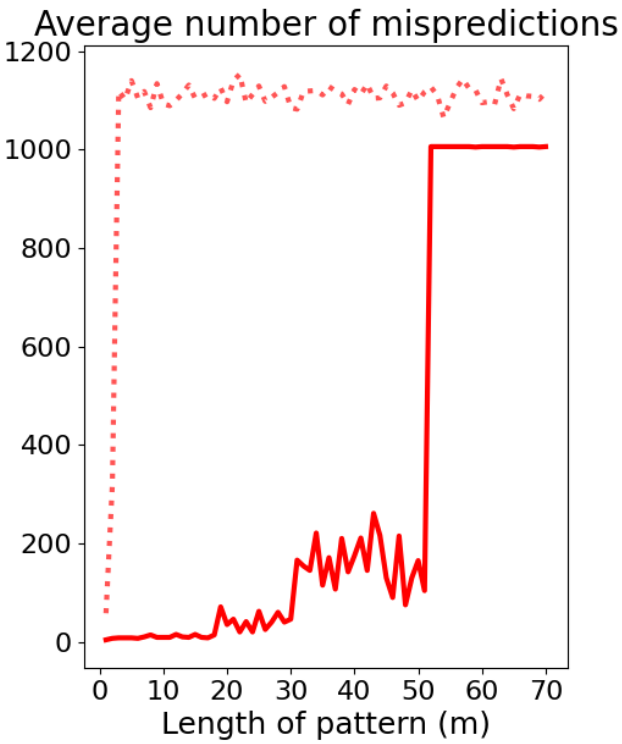

Figure 6: Results for a Firestorm core. The solid blue line shows the number of mispredictions that occur when running the loop. We get the lighter, dotted line when we execute 51 conditional branches just before the `if` statement branch.

On Icestorm cores, the jump happens between $m = 31$ and $m = 32$, which means that we possibly have a local `BHR` of length 30 or a global `BHR` of length 60. On Firestorm, the transition happens between $m = 51$ and $m = 52$, which would incidate a local `BHR` of

length 50, or a global BHR of length 100.

To determine whether we have a local or global BHR, we add a small loop before the `if` statement branch. This attempts to populate the BHR with branch outcomes that are not correlated with it. It has an iteration count of 31 on Icestorm cores and 51 on Firestorm cores. If there was a local BHR for each branch, we would be able to track the behavior of this small loop and the `if` statement separately, leading to a small number of mispredictions. On the dotted lines in the graphs, we can see that this is not what happens. The number of mispredicitons immediately jumps to an average of more than one misprediction per iteration of the outer loop. From this, we conclude that the M1 has predictors with global BHRs.

When gathering our measurements, we noticed some interesting behavior. If the addresses of the branches of the outer loop and the `if` statement differed by certain amounts, spikes in misprediction counts would occur at smaller pattern lengths. For example, when the difference was a multiple of 32 we would get the spikes shown in Figure 7. Where they occur depends on the difference of addresses but for a fixed difference, they consistently appear at the same positions.

While we have not determined what causes this behavior, we believe the test confirms that the address of the branch instruction is used in prediction computation.

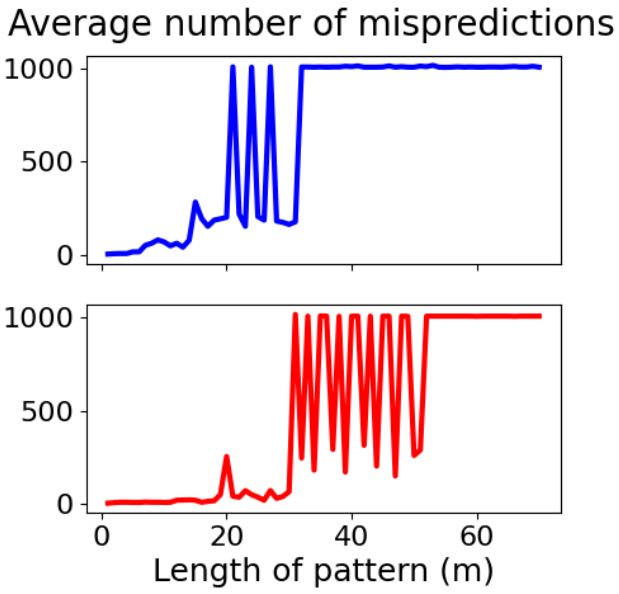

Figure 7: Results of the BHR length measurement where the addresses of the loop and `if` statement branch differ by 32. Icestorm above in blue and Firestorm below in red.

5 Loop predictor

Conditional branches used in loops have a regular pattern of always being either taken or not taken, except for the final iteration of the loop when the opposite outcome occurs. For certain predictors such as the commonly used saturating counter, this makes it very likely that the last branch will be mispredicted. On the other hand, a predictor that is aware of the loop iteration count could easily correctly predict the behavior of a loop branch.

Different kinds of *loop predictors* have been proposed and implemented to more accurately predict loop branches. A good

overview of some different approaches can be found in section 4 of [11].

We can determine whether the M1 processor has some sort of loop predictor with a benchmark involving nested loops, for example:

```
for(i=0; i < 1000; i++) {
  for(j=0; j < COUNT; j++) {
    // ...
 }
}
```

If there is no loop predictor, the processor will be limited by the length of the `BHR`. If we denote this length as L, then as `COUNT` exceeds $L + 1$, we should see a noticeable increase in the number of mispredictions. From the previous section, we know that this limit is 61 on Icestorm cores and 101 on Firestorm cores. Alternatively, if the M1 does have additional loop prediction logic, it should be able to maintain a small misprediction count beyond this limit. The results for different kinds of loops are presented in Figure 8 for an Icestorm core and Figure 9 for a Firestorm core.

For *do-while* loops, the behavior is exactly as we expected for a core lacking a loop predictor. When the pattern of the internal loop exceeds the maximum length, the number of mispredictions increases to an average of one per iteration of the outer loop.

For *for* and *while* loops, the results are not so straight forward, since the limits appear to be 60 and 100 instead of 62 and 102. If we disassemble the benchmarking program, we see that *clang* places the condition check for *for* and *while* loops before the loop body. With

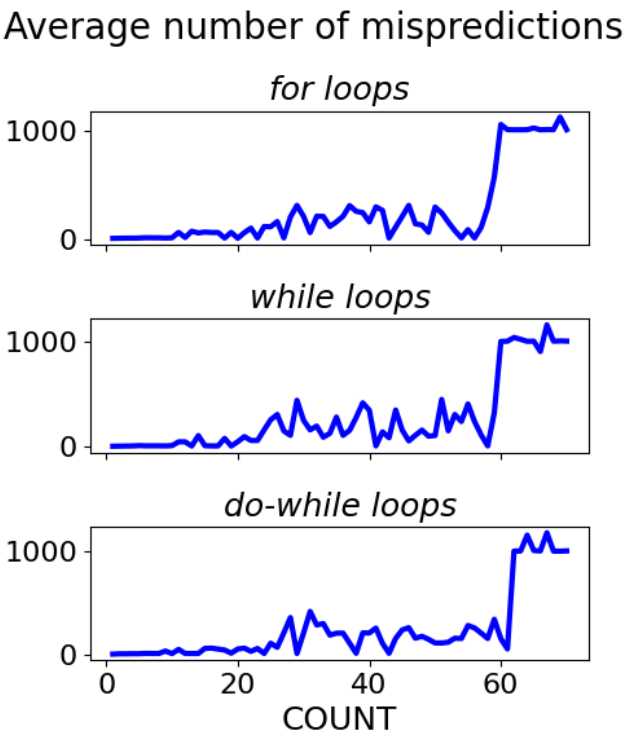

Figure 8: The number of mispredictions when running a nested loop on an Icestorm core. $COUNT$ is the number of iterations of the internal loop.

do-while loops, the check is at the end, which explains the difference seen on the graphs.

We conclude that the M1 processor does not contain a *loop predictor*, but relies on the `BHR` to accurately predict loop branches.

6 Branch target predictor

When the fetch unit retrieves a branch instruction from the cache, it must know at what address to continue fetching from. In

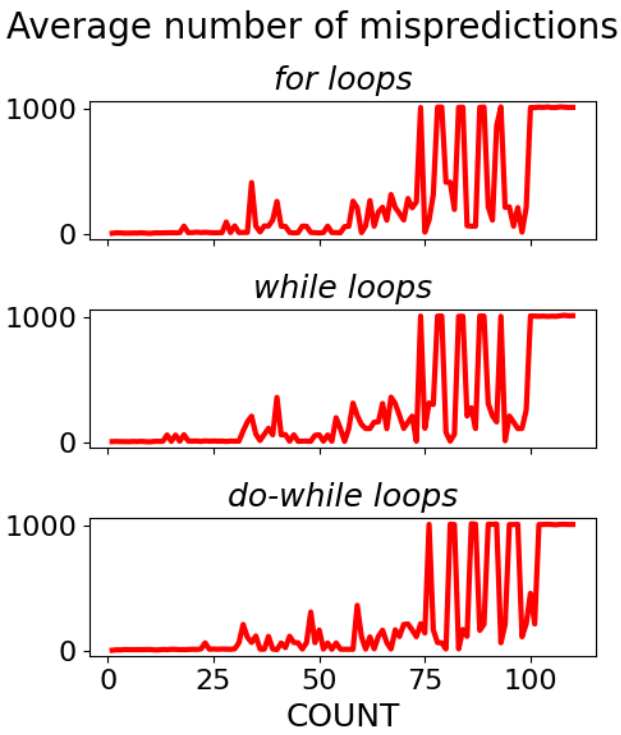

Figure 9: The number of mispredictions of nested loops on a Firestorm cores. We again see the spikes before the limit, just like in Figure 7

.

some cases, the decode unit may provide an address for PC relative branches in adequate time. With other instructions like indirect branches the target address is resolved too late for it to be worth waiting.

Processor designers try to get around this problem by using *branch target buffers*. In their most simple implementation BTBs are cache like structures, indexed by the address of the branch. Like caches, they may be set associative with varying set sizes and replacement policies. Stored within is the target address that the branch jumped to the last time it was executed. Like with outcome predictors, there are more sophisticated designs that attempt to achieve higher accuracy.

We can check whether the M1 has a simple address based BTB using one of Uzelac's benchmarks. The core assumption he makes is that the BTB is indexed by a range of bits from the branch address. If this is true, a benchmark where we vary the number of branches and the interval between them should show a recognizable pattern.

Suppose we are working with a processor where the address bits 7 to 3 are used as an index into a 2 way set associative BTB. Furthermore, we are repeatedly executing a block of code that has exactly 64 instructions (the size of the BTB). Figure 10 demonstrates what would happen if the addresses of the instructions differed by 2, 4, 8 and 16.

During our execution of the block with 64 instructions, the sequence of addresses of branches will go through all possible permutations of 6 bits at a certain position. The position depends on the distance between instructions. For example, if the distance is 2 as in the first example in the figure, the sequence of addresses would be

$$\begin{array}{c} 000000000000 \\ 000000000010 \\ \vdots \\ 000001111110 \end{array}$$

In this case, there would be 4 addresses with

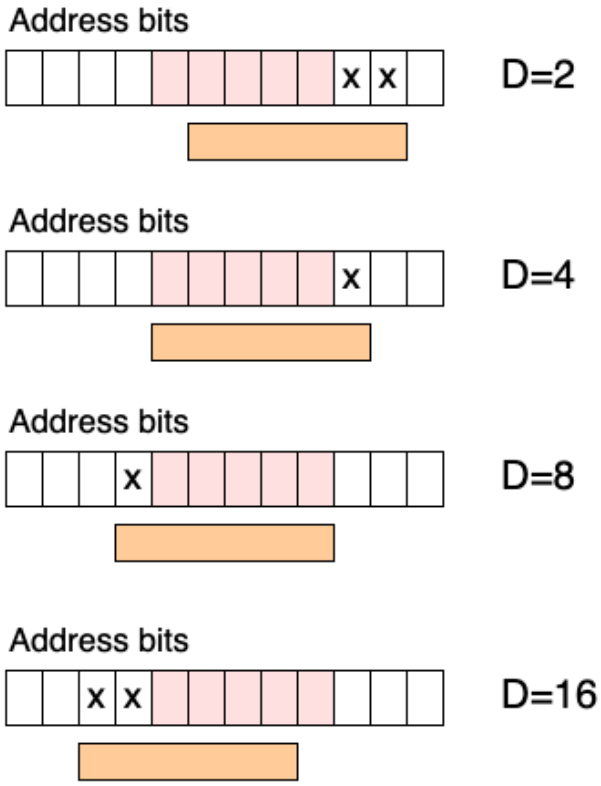

Figure 10:

the same BTB index, which would not work in our 2 way cache. On the other hand, if the distance is 4 or 8 only two among the 64 branches would have the same index and everything would fit into the BTB. Then at a distance of 16 and over, we would again see more than 2 instructions sharing an index.

What really determines whether all branches will fit into the BTB is the number of bits in the "overhang" marked with x. Let B denote the total size of the BTB, W denote its wayness and O the number of overhanging bits. If $2^O \leq W$ then B branches can fit into the BTB. For every additional bit added to the overhang, the maximum number of branches is reduced by half ($B/2, B/4, B/8$, $B/8$, etc.). Such a pattern should be easy to spot in a benchmark and prove our assumption right.

We created a python script that prepares ARM assembly code with branches at certain intervals just as we described. Conveniently, one of the possible events that a PMC can count is the number of retired indirect branches that were mispredicted. This made indirect branches an obvious choice.

The script places this block of code into the body of a loop that is ran 1000 times. Before and after the loop, we sample the PMC and calculate the difference. We confirmed with experiments, that PC relative branches never influenced the value of the PMC set up to track mispredicted indirect branches. However, we realize that the loop branch may still be influencing the result to a certain degree if it is inserted into the BTB. Because the number of executed PC relative branches is much smaller than the number of executed indirect branches, we believe the difference of the PMC is stil a good estimation. The average numbers are displayed in Figures 11 and 12.

If the M1 had a BTB like the one just described, we would see a recognizable staircase pattern, where the number of intervals with few mispredictions would increase as we decrease the number of branches. Instead, the difference of addresses seems to have no effect on the number of mispredictions. They increase steadily with the number of branches, before we see a sudden jump between 512

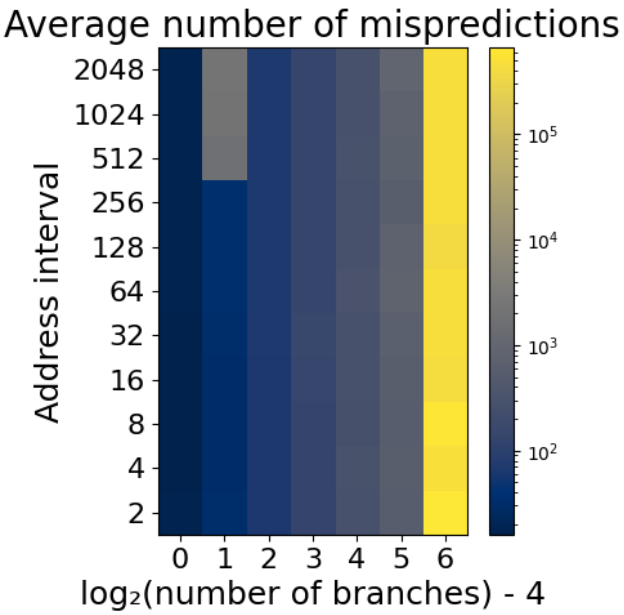

Figure 11: Average number of mispredictions when running Uzelac's benchmark on an Icestorm core. The 'Address interval' indicates by how much the addresses of branches differed. The horizontal axis shows the number of branches. The strange notation is used because the numbers would otherwise overlap in the plot.

and 1024 branches on an Icestorm core. On Firestorm cores, a similar increase can be seen between 2048 and 4096 branches, though it is less prominent.

Since we do not see the pattern we expected based on the description of the simple PC based `BTB`, we feel confident in saying that the M1 does not use this kind of target predictor.

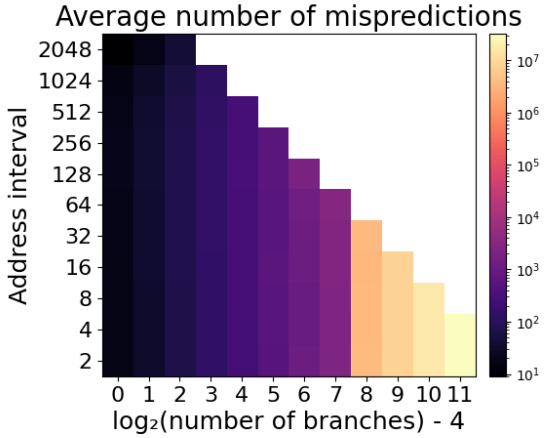

Figure 12: Average number of mispredictions when running Uzelac's benchmark on a Firestorm core. The upper right section is missing data where the benchmark was so large, it was crashing the program.

7 Summary

Little work has been done specifically on reverse engineering the front end of Apple's M1 processor since it release. We combined the knowledge of those looking into other aspects of its microarchitecture with existing benchmarks used to reverse engineer branch predictors. Assuming that both types of cores on the M1 track branch outcome histories, we managed to show that both use a global `BHR` which is 50 bits wide on Icestorm cores and 100 bits wide on Firestorm cores. Neither of them seem to implement some sort of loop prediction mechanism. Regarding branch target prediction, we were able to show that the M1 does not use a simple approach, where a range of bits is used as an index into a cache like structure.

References

[1] A. Frumusanu. *The 2020 M1 Mac Mini Unleashed: Putting Apple Silicon To The Test*. Anandtech, 2020.
https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

[2] A. Frumusanu. *Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14*. Anandtech, 2020.
https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

[3] D. Johnson. *Apple M1 Microarchitecture Research*.
https://dougallj.github.io/applecpu/firestorm.html

[4] M. Majkowski. *Branch predictor: How many "if"s are too many? Including x86 and M1 benchmarks!*. The Cloudflare Blog, 2021.
https://blog.cloudflare.com/branch-predictor/

[5] V. Uzelac, *Microbenchmarks and mechanisms for reverse engineering of modern branch predictor units*. The University of Alabama in Huntsville, Huntsville, Alabama, 2008.

[6] M. Milenkovic, A. Milenkovic and J. Kulick. *Microbenchmarks for determining branch predictor organization*. Software: Practice and Experience vol. 34, 2004.

[7] M. Handley. *M1 Explainer*.
https://drive.google.com/file/d/1WrMYCZMnhsGP4o3H33ioAUKL_bjuJSPt/view

[8] Branch prediction system patent.
https://patents.google.com/patent/US10719327B1

[9] A. Snezec and P. Michaud. *A case for (partially) TAgged GEometric history length branch prediction*. The Journal of Instruction-Level Parallelism vol. 8, 2006.

[10] The Asahi Linux project.
https://asahilinux.org/

[11] S. Mital. *A Survey of Techniques for Dynamic Branch Prediction*. Concurrency and Computation Practice and Experience, 2018.