Return Address Predictor

Zih-Bo Yu(于子博), 111550016

***Abstract*— In this report, I will first introduce the existing branch predictor in Aquila, then test its effectiveness by modifying some parameters, and finally explain the implementation of the return address predictor.**

I. BRANCH PREDICTOR

A. Implementation

Overall, the branch predictor receives signals from three different CPU stages: the program counter, decode, and execute stages. If the PC signal from the program counter is already recorded in the Branch History Table (BHT) as a hit and is determined to be taken, branching will occur in the next clock cycle. The signal from the decode stage handles the initial encounter with a branch (not hit) and records it in the BHT. The signal from the execute stage verifies the relationship between the hit prediction and the actual result, adjusting the branch likelihood accordingly.

B. Analysis of BPU

BPU entry number	Coremark score
0(disable BPU)	89.150397
32	99.884134
64(default)	100.739427
128	100.922431
256	101.157239
512	101.185899
1024	101.189994
2048	101.204332
4096	101.204332

From the table above, it can be observed that when the entry number reaches 2048, the Coremark score stabilizes. This is reasonable, as exceeding this size prevents BHT overflow from occurring.

BPU entry number	Jal hit rate	Branch hit rate
0(disable BPU)	X	X
32	75.68%	67.07%
64(default)	91.35%	75.03%
128	93.20%	75.58%
256	98.29%	78.04%
512	98.96%	78.27%
1024	99.13%	78.33%
2048	99.46%	78.38%
4096	99.48%	78.38%

We can observe a positive correlation between the increase in hit rate and the Coremark score. For example, from sizes 32 to 64 and 128 to 256, both the scores and hit rates show a significant improvement. Additionally, the hit rate for JAL is higher than for branches because, in Coremark, the number of branch instructions is significantly greater than that of JAL instructions. Branches with different addresses are more likely to map to the same BHT entry, causing misses.

II. RETURN ADDRESS PREDICTIOR

A. Architecture

The return address predictor (RAP) is quite similar to the branch predictor, but RAP has several important characteristics:

1. If only a single stack is used, we won't know the return address when a JAL instruction is read, making it impossible to jump immediately after the fetch like a BPU. To improve this, a return history table is needed, similar to the BHT, which stores encountered `ret` instructions. Later, when this address is read, we know it is a `ret` instruction.

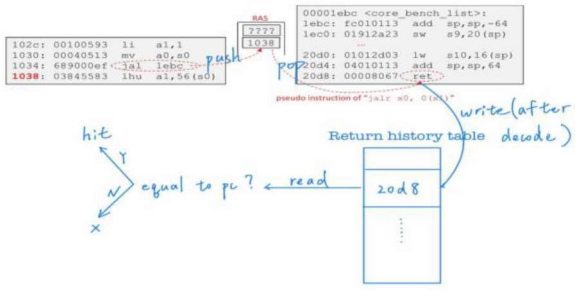
2. Reason for using a circular buffer: In theory, as long as the stack is large enough, there should be no overflow issues since each pushed address will eventually be popped upon return. However, tail-call optimization may push an address without a corresponding pop, potentially accumulating too many elements and causing overflow, and even resulting in incorrect popped addresses. Using a circular buffer solves this problem by overwriting the earliest elements when full, which, combined with the misprediction mechanism, ensures accurate results.

3. Some key points require special attention when implementing RAP:

RAP stalls involve not only stall_pipeline but also stall_data_hazard.

If RAP is a hit, it needs to mask the original PC calculated from the EXE stage, and the next PC should be PC + 4.

Pay close attention to the flush timing, as incorrect flushing can lead to execution errors that are difficult to debug.



B. Analysis of RAP

RAP (size=128)	tail-call optimization	Coremark score (on board)	Improve (with/with out RAP)
○	X	101.030511	0.792%
X	X	100.236558	
○	○	101.200235	
X	○	100.698850	0.498%

From the results above, it can be observed that if tail-call optimization is removed, the improvement of RAP increases further, but the score decreases. I believe this is because tail-call optimization increases the likelihood of misprediction when predicting return addresses. This is due to the fact that not every pushed address in the buffer will have a corresponding pop, leading to some useless elements remaining in the buffer until it is full (in this implementation), at which point they can be overwritten.

RAP buffer size	with tail-call opt.	without tail-call opt.
16	101.200235	101.022346
32	101.202283	101.030511
64	101.200235	101.028470
128	101.202283	101.030511
256	101.202283	101.032553
512	101.204332	101.030511

The table above shows the results obtained after adjusting the buffer size in RAP (on board). Some strange phenomena can be observed, such as increasing the size does not necessarily yield a better score, and scores for different sizes are sometimes the same.

Regarding the issue of identical scores, I found that even with the same settings, the score can still vary. For example, when size=128, the score is sometimes 101.202283 and other times 101.200235. The exact reasons may include cache misses or slight frequency variations. Therefore, changing the sizes from 16 to 512 (512 sometimes get 101.202283) showed no improvement, regardless of tail-call optimization.

Changing the size of the Circular Buffer does not necessarily improve benchmark scores, and the reasons might include the following:

With tail-call optimization enabled, multiple return addresses may remain in the Circular Buffer without being correctly popped, as they are never used. Regardless of the buffer size, these unused addresses occupy space, and increasing the buffer size could delay the time it takes to overwrite them, potentially leading to more mispredictions.

In Coremark, the function call depth may not be very large, meaning buffer overflow or overwriting doesn't occur frequently. Thus, even if the Circular Buffer size is increased, it won't significantly affect return address prediction accuracy or scores.

III. IMPROVE THE RAP

In my implementation, if a misprediction occurs, there is no pop operation; theoretically, mispredictions should only happen in tail-call optimization and when the number of JALs between a JAL and a RET exceeds the buffer size (as I believe). I'm very curious about the potential improvement if a pop is added during misprediction—an extra pop should address the excessive push issue in tail-call optimization.

	Ret_instr_cnt	Ret_hit_cnt	Coremark score
Added 1 pop	38180	45377	101.200235
Original	38178	45377	101.200235

Unfortunately, the score did not improve. I then calculated the number of ret instructions and the RAP hit rate, finding only two additional hits. I'm not certain if this is because tail-call optimization can skip over multiple ret instructions consecutively, or mispredictions occur due to the number of JALs between a JAL and a RET exceeding the buffer size. If that's the case, solving the tail-call optimization issue might require additional pops.

However, regardless of the cause of the misprediction, popping multiple times after a misprediction should help improve accuracy. The challenge is determining the number of pops, which ideally should be dynamically decided. This remains solved.

IV. WHAT I HAVE LEARNED

A. Exception Occur

If a bug occurs in this assignment, it will be harder to trace the source. Sometimes, it's due to incorrect flush timing, leading to incorrect register contents. For example, I once encountered an exception caused by unaligned memory access. Although execution continues after handling the exception, the contents of the registers may be incorrect, resulting in unpredictable outcomes.

B. A way to debug

Since the program execution relies on the values in the registers, it's sufficient to identify where the error first occurs. In this assignment, I compared the contents of all registers at critical moments with and without RAP, which helped me identify the errors and make the necessary corrections.