Jamshed Ashurov
CSE 148
Professor Dean Tullsen
June 10th, 2022

Computer Architecture Project Report (Spring - 2022)

***Abstract* -- In CSE 141 & 141L, we are introduced to the realm of computer architecture. In beginning the learning, we were taught the basics of the standard 5-stage pipeline CPU. Of course, along with its easily comprehensible structure, comes with its plethora of drawbacks in performance when it comes to certain prospects of programs.**

I. Introduction

Now, in CSE 148, we are tasked with undertaking the drawbacks previously mentioned in CSE 141 as the new material to study upon.

A. *Drawbacks*

Among the numerous drawbacks the basic architecture provides, we narrowed down the biggest problems seen from our perspective:

1) The cache's inability to adapt to different types of serial memory, thus references back to disk memory occur incurring large amounts of stalled cycles for one instruction.
2) The cache's inability to fully utilize the entire size of the cache due to direct mapping, evaluating to an increase of conflict misses due to different memory being hashed to the same cache line.
3) The CPU's inability to adapt to the pattern of correct branch instructions, which results in the pipeline to be flushed in every account the branch was indeed mispredicted. While this seems inconsequential in one instance, this builds quickly in some programs.
4) The CPU's inability to adapt to make use of stalled cycles for certain instructions to work on simpler instructions, resulting in wasted cycles where nothing at all is accomplished.

With these drawbacks in mind, we chose optimizations to our baseline CPU design that would best try to help the CPU adapt the most to these situations.

B. *Optimizations*

The optimizations we implemented include:

- Set-Dueling with DRRIP & SHiP
- Victim Cache addition
- TAGE branch predictor
- Out Of Order Execution pipeline configuration

Each optimization was the separate attempt to alleviate the said drawbacks, respectively. Our initial goal was to create each optimization in which works cooperatively with each other to produce the lowest possible CPI, which we aimed to achieve to get the closest to 1 CPI(mainly because we were told to have d_cache and i_cache at 8KB) on average with the default benchmarks given: nqueens, qsort, coin, & esift2.

II. Optimizations

A. *DRRIP/Set Dueling + SHiP*

The cache set-dueling optimization aims towards helping the reduction of cache misses by dynamically choosing the better cache eviction policy. DRRIP is a version of set dueling where the first policy is SRRIP and the second policy is BRRIP. SRRIP/BRRIP use the idea of re-reference interval to insert and evict from the cache with the only difference between the two being that during insertion

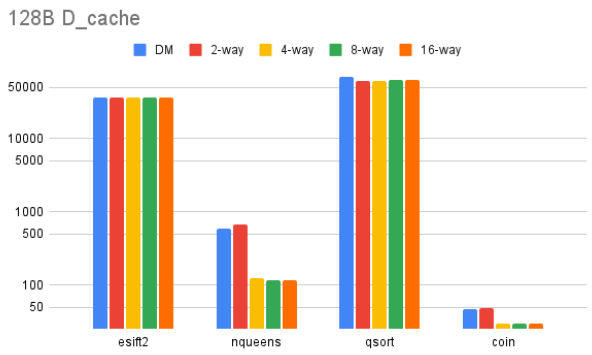
BRRIP uses *long* re-reference interval with probability 1/n and *distant* re-reference interval all the other times and SRRIP uses *long* re-reference interval every time.

Furthermore, to additionally benefit from the reduction of cache misses, we implemented SHiP, specifically SHiP-PC. SHiP-PC makes a prediction of the re-reference interval based on the program counter value of the memory instruction. Every time a line is evicted from the cache, the predictor remembers if the line was re-referenced before it was evicted, so that when this line is inserted into the cache again, a *distant* re-reference interval will be used.

The reason why we used DRRIP instead of DIP is that DRRIP offers better performance compared to DIP[1]. Similarly, implementing SHiP with DRRIP offers better performance compared to standalone DRRIP[3].

For DRRIP we used suggestions that were given in [1] to set M = 2. For the SHiP-PC predictor we used suggestions from [3] to make the SHCT counters to be bi-modal and use the upper 14 bits of the program counter of the memory instruction to use as an index for the SHCT table.

Graph 2.0 D_cache Associativity



Let's first look how associativity will affect the amount of d_cache misses. The graph above gives the amount of d_cache misses for direct mapped(DM), 2 way, 4 way, 8 way, and 16 way associative 128B d_caches.

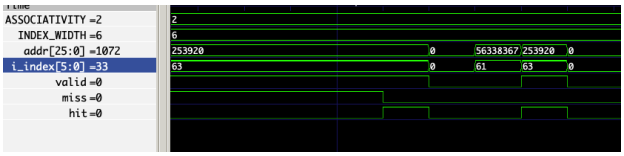
Graph 2.0 shows that using a 2-way set-associative cache in fact decreases the performance by increasing the number of d_cache misses by 0.01%, 15%, 2% for esift2, nqueens, and coin respectively and decreases the number of d_cache misses of qsort by 11%.

Now starting from a 4-way set-associative cache, we see a good amount of reduction of d_cache misses by -0.01%, 79%, 10%, 36% for 4-way and by -0.01%, 80%, 9%, 36% for 8-way and -0.01%, 80%, 8%, 36% for a 16-way set associative cache for esift2, nqueens, qsort and coin respectively.

Hence, using either an 8-way or 16-way set-associative cache will lead to the best performance(although 8-way is slightly better overall).

Now as for set-dueling, replacement policies, and signature based predictor, unfortunately there were not fruitful for the benchmarks for the following reason:

We found an anomaly in the benchmarks(Figure 2.0) that shows an instruction that had a d_cache miss followed by a hit by the same instruction 2 cycles later. The reason why it is problematic is that if we use a re-reference based replacement policy , the cache line will be promoted to top priority immediately after the miss which defeats the purpose of re-reference interval based replacement policies.



Since we use DRRIP for set-dueling, our optimization will not work. Hence, I would say that DIP would be the best set-dueling optimization for the benchmarks because DIP

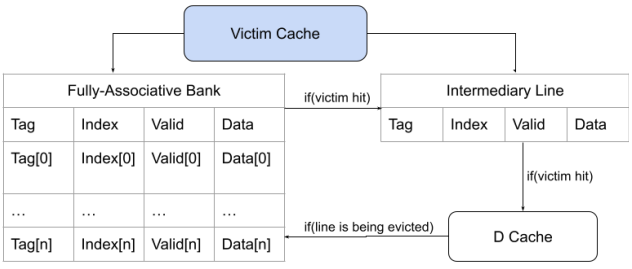
does not use re-reference intervals for its eviction and insertion policies.

B. *Victim Cache*

The victim cache optimization aims towards helping the reduction of cache misses due to conflict misses. Conflict misses are where two or more separate memory references relate to the same cache line, where if the multiple references were referred to the opposite each time, the cache would lose the previous memory reference in favor of the more recent reference, this is especially true for direct mapping caches as they only reserve one line in the cache for references. And each time having to go back to the extremely high latency main memory for the data, carving huge chunks of stalls in the program. In this same scenario, the cache isn't even completely filled, meaning there is indeed space for the memory references, thus wasted in the process.

The victim cache combats this by taking a small portion of the current cache space to house a fully associative cache & separate intermediary line which is able to reliably store any reference anywhere in the cache. The references stored there are decided by the most recently evicted line from the primary cache.

Figure 2.1 *High Level Representation of Victim Cache*



The biggest nuance in implementing this optimization is the utilization of mutual exclusion between the main & victim caches, and the shortcut of writing to memory whenever a reference is evicted to the victim cache instead of whenever a reference is evicted from the victim cache. These nuances allowed the implementation to be minimally invasive to the baseline code. Doing this allowed the least amount of edge-cases created when implementing, and also resulted in debugging code to be extremely more efficient as it was clear where and when the victim cache was incorrectly behaving. Following along, the biggest obstacle was correctly rewiring the cache line configuration to allow the victim cache to truly work optimally.

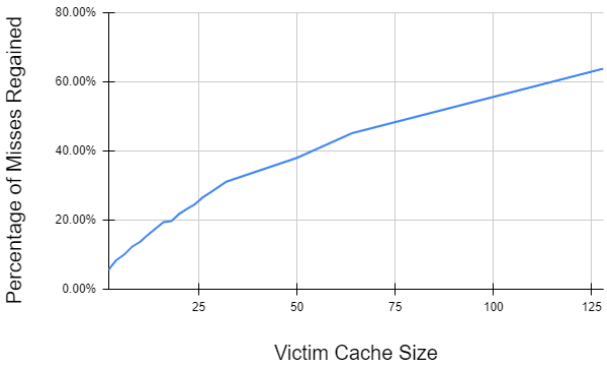
The results for this optimization are included in Table 2.1, using a default size of 16 lines which resulted in an astonishing 0.05%, 23%, 36%, 86% decrease of CPI for esift2, nqueens, qsort, and coin respectively from the baseline design. We then continued the development of the victim cache by finding the optimal size for performance, shown in Graph 2.1, where it seems that the will always linearly increase the accuracy of victim hits, the pay-off of having the fully associative victim cache takes over the increased performance of accuracy. Of course when making the victim cache so large, we are in effect promoting the main cache storage to being the victim cache, defeating the purpose of still having a direct-mapped or n-way associative set cache in the first place, but the nice sweet spot is ~30 lines of space.

Table 2.1 Victim Cache Performance

	esift2	nqueens	qsort	coin
V_cache Hits	32	964	26888	34

	esift2	nqueens	qsort	coin
CPI	2.12778	23.814	2.94798	1.01032

Graph 2.1 Optimal Victim Cache Size



C. *TAGE*

TAGE branch predictor is a branch predictor that uses tagged predictor components indexed by using different history lengths[4]. These history lengths form a geometric series. When making a prediction, all the components of TAGE and the bi-modal base predictor make a prediction and the prediction from the component of the highest geometric history length becomes the main prediction. Upon branch resolution, the component that gave the prediction is updated.

The implementation of TAGE was challenging because of the fact that I had to fit the process of making a prediction and update in just 1 cycle. Furthermore, TAGE requires a lot of comparators and priority encoders to choose the component for the main prediction, choose the component for alternate prediction and choose the component entry for eviction. We currently use the previous pc to make sure that we do not continuously make a prediction and update GHR when the pipeline stalls, but given more time a more elegant solution is certainly possible.

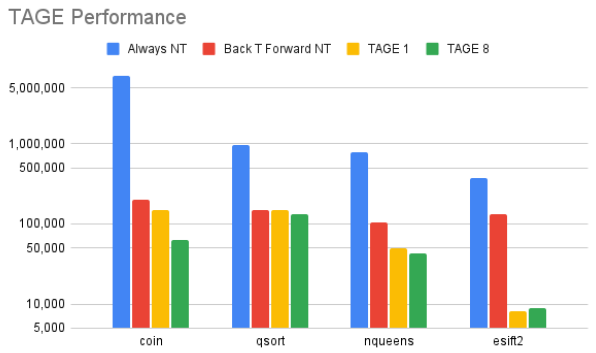
TAGE 1:

- 4K Base Predictor
- Table Len: 2048
- GHR 10
- L1 = 10
- TAG WIDTH = 12

TAGE 8:

- 8K Base Predictor
- Table Len: All tables 512
- GHR 256
- L1 = 2
- Alpha = 2
- TAG WIDTH = 10 & 12

Graph 2.2 TAGE Performance(mispredictions)



Graph 2.2 shows that using a simple backward taken and forward not taken branch predictor decreased the number of mispredictions significantly: 65%, 86.5%, 85%, 97% for esift2, nqueens, qsort, and coin respectively.

Using TAGE 1 further decreased the number of branch mispredictions: 94%, 52%, 2%, 26% for esift2, nqueens, qsort, and coin respectively.

Lastly, using TAGE 8 resulted in the following reduction of branch misses: -9.5%, 15%, 9.5%, 58.5% for esift2, nqueens, qsort, and coin respectively.

The results above show us the abundance of for loops in the benchmarks as we saw the biggest decrease in branch misses with backward taken and forward not taken branch predictor. Furthermore, while we see a good amount of improvement in performance when using a 1 component TAGE only coin saw a big

improvement when using an 8 component TAGE. Hence, we decided to compare different versions of TAGE.:

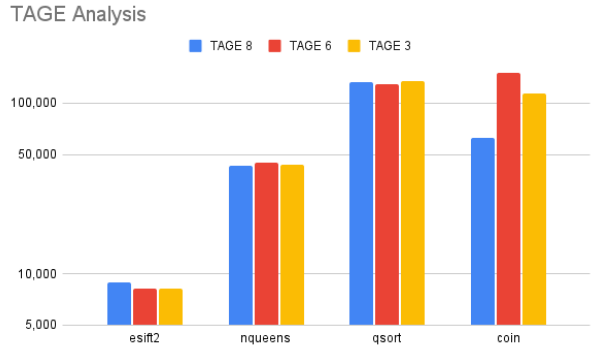
TAGE 3:

- 8K Base Predictor
- Table Len: 2048, 1024, 512
- GHR 256
- L1 = 10
- Alpha = 5
- TAG WIDTH = 12

TAGE 6:

- 8K Base Predictor
- Table Len: 1024, 512, 512, 512, 512, 512
- GHR 256
- L1 = 10
- Alpha = 2
- TAG WIDTH = 8 & 12

Graph 2.3 TAGE Analysis(misprediction)



Graph 2.3 shows that compared to TAGE 8 TAGE 6 had: 8.5%, -5%, 3% decrease in branch mispredictions for esift2, nqueens, and qsort respectively and more than double increase of branch misses for coin.

TAGE 3 performed slightly worse than TAGE 8 with: 8.5%, -1%, -1% decrease in branch mispredictions for esift2, nqueens, and qsort respectively and more than double increase of branch misses for coin.

While not documented here, we also used different GHR len in our experiments but they gave similar results to TAGE 3 and TAGE 6.

These numbers tell us that only coin saw the biggest benefit in using a longer GHR length with an emphasis on recent history(alpha = 2, L1 = 2). Since TAGE 3 and TAGE 6 had emphasis on further history, they did not perform as well as TAGE 8.

As for the other 3 benchmarks, they mostly had a history length of about 10, especially esift2 since using L1 = 2 did not perform as well as L1 = 10.

D. *Out-Of-Order Execution*

Out-of-Order execution breaks the sequential implementation of the pipeline to enable executing instruction during stalls and branch mispredictions. While there are different implementations of the out-of-order execution, in our design we tried to implement the pipeline that is as close as to the one being used by Alpha 21264 [5]:

- Fetch, Decode/Rename, Issue, Execute, Load/Store, Commit
- Active list to store the instructions and other global parameters(uses_rw, is_load, is_store, etc)
- Free List to hold the physical registers that are not being used
- Merged Register File to hold the registers that were committed and those that are in flight
- Rename Buffer to hold mapping from the logical to physical registers
- Branch Tables to hold the metadata of the current pipeline when a branch is encountered
- Memory Queue to hold all the in-flight memory instructions that are waiting for their register operands to compute the memory address

- Integer Queue to hold all the other instructions that are waiting for their register operands.
- Load Queue to hold all the load instructions that have computed the address and ready to be dispatched to d_cache.
- Store Queue to hold all the store instructions that have computed the address and the data to be dispatched to d_cache.
- Commit List to hold the ready bits of the all the instructions that have finished execution and retire them in order
- Reclaim List to hold all the physical registers that need to be reclaimed during retirement.

The following are the corner cases that I have encountered while implementing out of order machine :

- Issuing a branch instruction with the delay slot required me to have a buffer in the decode stage that would buffer the branch instruction and wait for the delay slot instruction to arrive. After the delay slot arrives, we can now issue both the delay slot and the branch instruction to the appropriate queues which guarantees that during the misprediction the delay slot is in the one of the issue queues.

- Branch misprediction is probably one of the hardest parts of the out of order machine because it requires checkpointing and squashing the "wrong" instructions from the issues queues and load/store queues. This creates room for a lot of bugs:
    - One of the potential hidden bugs is that there's a branch misprediction during a d_cache miss and the instruction that is in the d_cache is squashed. I have a global memory controller to combat this problem which will invalidate cache output once the "wrong" memory instruction finishes its execution.
    - Instruction squashing from the queues happens with the help of the active list color bit. A > B if two instructions have the same color bit and A > B, and two instructions have a different color bit and A < B.
    - The writes pointers of the queues need to be restored to the pre-branch misprediction state to continue correct execution.
- Instead of committing 1 instruction at a time, we can have a window of instructions that we can commit at a time. Otherwise, the pipeline will always be slower than the in-order pipeline because of the false loads during branch misprediction.

III. Conclusion

After the 10 weeks of working on the optimizations we can confidently proclaim that we did succeed in our goals: getting all of our proposed optimizations to work together. Given the hypothetical thought of how to continue this regardless of time, the next few best optimizations would be to include include the *stream buffer* to improve the instruction cache in the similar fashion to the victim cache, the *runahead pipeline configuration* to work together with the Out Of Order optimization, &

the *superscalar execution optimization* to amplify the performance enhanced already by the rest.

ACKNOWLEDGEMENTS

To Professor Dean Tullsen & TA Sankar Ramesh for all the help and support in making this project a reality.

REFERENCES

[1] High performance cache replacement using re-reference interval prediction (RRIP)

[2] Adaptive Insertion Policies for High Performance Caching

[3] SHiP: Signature-based Hit Predictor for High Performance Caching

[4] A case for (partially)-tagged geometric history length prediction

[5] Processor Microarchitecture An Implementation Perspective