CSE 148: Advanced Processor Architecture Design Project Spring 2022

Final Report

Instructor: Dean Tullsen

Brian Lin, Jiawen Xu

1 Motivation

The original plans for optimizations included improved branch prediction, register renaming, out-of-order execution, superscalar execution, and super-pipelining. Due to time constraint, we change our goals to optimize the fully-pipelined MIPS CPU using improved branch prediction, victim cache implementation, stream buffers, and exploitation on value locality.

The benchmarks are fixed, instruction count is relatively fixed. Therefore, our primary metrics for improvement are CPI and cycle time.

Besides, we also have considered instruction reuse, and dual path execution. However, the architecture of the processor did not use multiple-cycle operations, and we will not see too much improvement from this optimization. Also, dual path execution is not reasonable without out-of-order execution. Since we changed our initial plan to Hardware prefetching (Victim cache and stream buffer), we decided not to implement dual path execution as well.

2 Implementations and Architectures

2.1 Branch Predictor

2.1.1 Bimodal Branch Predictor

The very first option of branch predictor we implemented is the bimodal branch predictor. We use a 2-bit counter to identify the states including {T, t, n, N} which takes two mis-prediction to change from strong taken to strong non-taken or the other way around. Based on the functionality counter, we can then use a table of counters for different program counter input. We map the input PC to the index of the counter table; however, for smaller tables, some branches will need to share the same counter, and the accuracy will decrease [1]. In our implementation, we use a 512-entry counter table and map the first 9 bit of PC to the index of the counter table. Figure 1 shows the block diagram of the architecture of bimodal branch predictor and table 1 shows the CPI of the baseline and bimodal branch predictor.

2.1.2 Local Branch Predictor

Based on Bimodal Branch Prediction, we developed a local branch predictor. The Local Branch Predictor contains two tables - history table and counter table. The counter tables is an array of 2-bit counters same as what we used for bimodal branch prediction. The history table contains the history of the past outcome from the feedback, which is represented as a shift register in our implementation[1]. Every time when we make a prediction, we will map the request PC to the history table and the output of history table will be the index of the counter table. After we select the counter from the counter table, the rest will be the same as bimodal branch predictor to get the output prediction. Also, at the same time, if the feedback input is valid, we will use the feedback pc and outcome to update the history table. In our implementation (the Verilog code), we label it as ”old index” and ”new index” to identify which index we use for output prediction or history table update. Our size for history table is 2048 bytes and we take the first 11 bits of the pc as index for history table. Figure 2 shows the block diagram of the architecture of local branch predictor and table 2 shows the CPI of the baseline and local branch predictor.

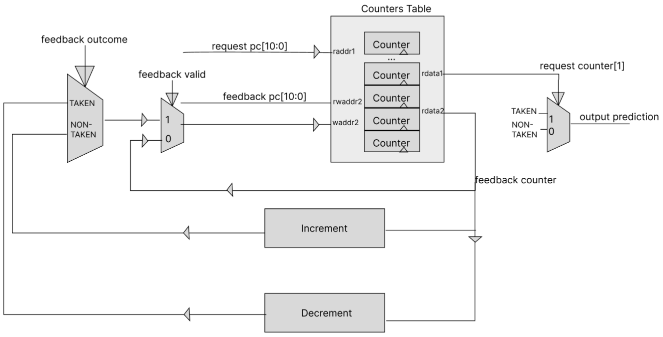


Figure 1: Block Diagram of Bimodal Branch Predictor

	nqueens	coin	qsort	esift
baseline	30.9326	7.12898	4.59141	2.27794
bimodal	30.4717	7.12731	4.50603	2.25367

Table 1: Performance(CPI) of Bimodal Branch Predictor

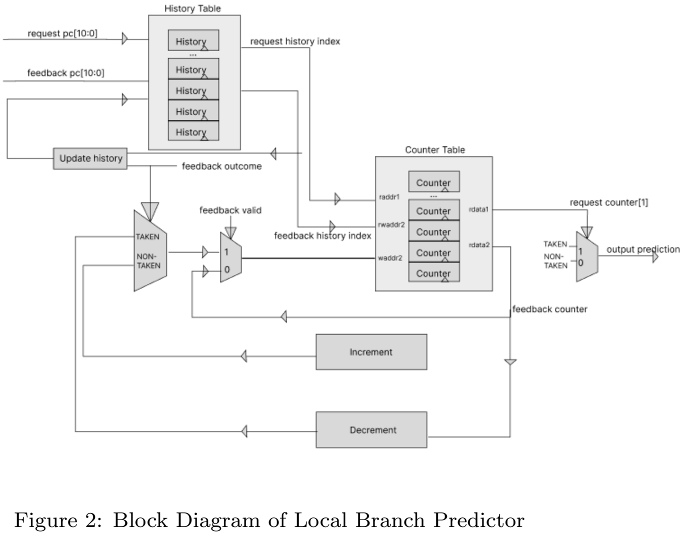


Figure 2: Block Diagram of Local Branch Predictor

	nqueens	coin	qsort	esift
baseline	30.9326	7.12898	4.59141	2.27794
local	30.9486	7.12485	4.50185	2.2537

Table 2: Performance(CPI) of Local Branch Predictor

2.1.3 Global Branch Predictor

Global branch predictor use a separate shift register, which takes one bit as MSB and shift all the bits to the right for one bit. We then use the updated output of the shift register (we call it ”new index”) for the index of the counter table to get the prediction, and use the output before updating (we call it ”old index”) for feedback update. Figure 3 shows the block diagram of the architecture of global branch predictor and table 3 shows the CPI of the baseline and global branch predictor.

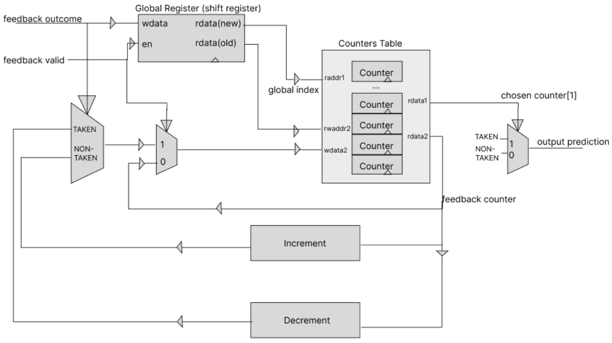


Figure 3: Block Diagram of Global Branch Predictor

	nqueens	coin	qsort	esift
baseline	30.9326	7.12898	4.59141	2.27794
global	31.4006	7.12731	4.50603	2.25367

Table 3: Performance(CPI) of Global Branch Predictor

2.1.4 GSelect Branch Predictor

The Gselect branch predictor use a shift register as GR and we combine some bits of the GR and some bits of PC as our index[1]. In this way we can use both the pc address and the global register for branch prediction. Figure 4 shows the block diagram of the architecture of gselect branch predictor and table 4 shows the CPI of the baseline and gselect branch predictor.

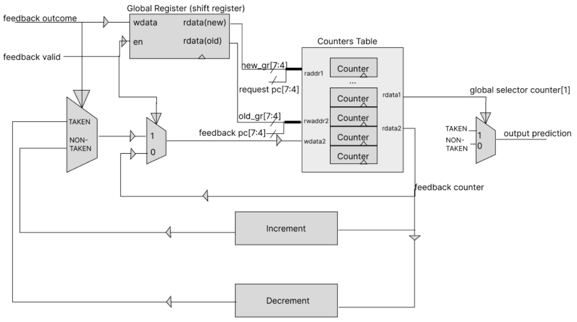


Figure 4: Block Diagram of GSelect Branch Predictor

	nqueens	coin	qsort	esift
baseline	30.9326	7.12898	4.59141	2.27794
gselect	30.9123	7.12573	4.49797	2.25345

Table 4: Performance(CPI) of GSelect Branch Predictor

2.1.5 Combining Branch Predictor

We combine Gselect and Local branch predictor into combine branch predictor where two sub-predictors are designed as modules of the combining branch predictor. We use a two bit saturating counter to decide which predictor to use. We have to modify the interface so that we can have the feedbacks from two branch predictors. We compare the feedback prediction and outcome to decide if we should switch the branch predictor If they are both good or bad, we don’t change. If only one of the branch predictor is good, we will either increment or decrement the counter accordingly We will output the MSB of the counter as the selection of the branch predictor. Figure 5 shows the block diagram of the architecture of combining branch predictor and table 5 shows the CPI of the baseline and combining branch predictor.

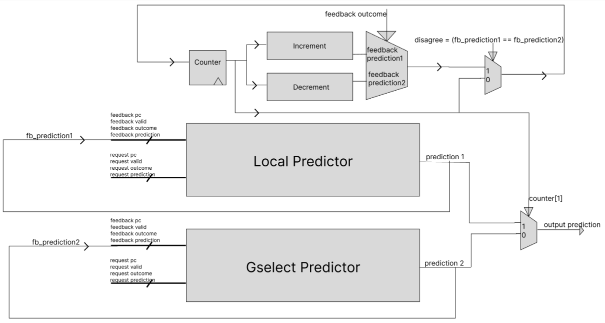


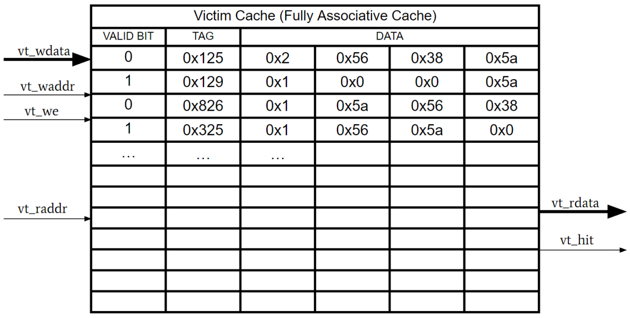
Figure 5: Block Diagram of Combine Branch Predictor

	nqueens	coin	qsort	esift
baseline	30.9326	7.12898	4.59141	2.27794
combine	30.8877	7.12485	4.50185	2.2537

Table 5: Performance(CPI) of Combine Branch Predictor

2.2 Victim Cache

The implementation of our Victim Cache is a D-cache with 32-entry victim cache and I-cache with 64-entry victim cache. Victim cache is a small fully associative cache that is loaded with the victim line from the direct mapped cache, and victim data (victim line) is when a miss occurs, the data in the databank before it is replaced by the memory.[2]



Victim Cache (Fully Associative Cache)

VALID BIT	TAG	DATA				
0	0x125	0x2	0x56	0x38	0x5a	
1	0x129	0x1	0x0	0x0	0x5a	
0	0x826	0x1	0x5a	0x56	0x38	
1	0x325	0x1	0x56	0x5a	0x0	
…	…	…				

Figure 6: Architecture of Victim Cache

As show in Figure 6, we use a fully associative cache for our victim cache with lists of valid bit, tags, and cache line data. The way we accommodate data is similar to a circular array list with a counter saturated to the total entry size of the cache. The saturated counter will be the index of the location for the new write-data if the tag does not appear in the tag list. If the write-data has tag that is already in the victim cache, we will overwrite the existed data line instead of adding the data as a new entry.

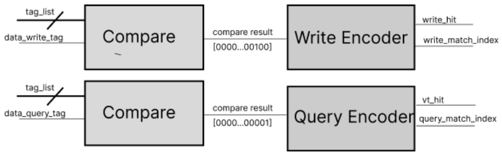


Figure 7: Block Diagram of Encoders in Victim Cache

In the fully associative cache, we need to compare all the tags with the target tag. We use one-hot encoding for our compare result such as ”000...00100” when there is a victim hit in the 3rd entry. As shown in Figure 7, We then use the one-hot-encoded comparing result as input to an encoder, and the output will become the index of the matching index, which is 2 in this example. We have two encoders in our victim cache. The first encoder is to check if the input write-data has matching tag. If we find the input data tag in our victim cache we will use the matching index to overwrite data; otherwise, we will use the saturated counter to add a new entry. The second encoder is to check if the query address (read-address) exists in our victim cache. We will output the vt_hit signal from encoder by checking if there is any matching tag for query data in victim cache, and we will use the query_match_index to output the cacheline from victim cache and write it to databank.

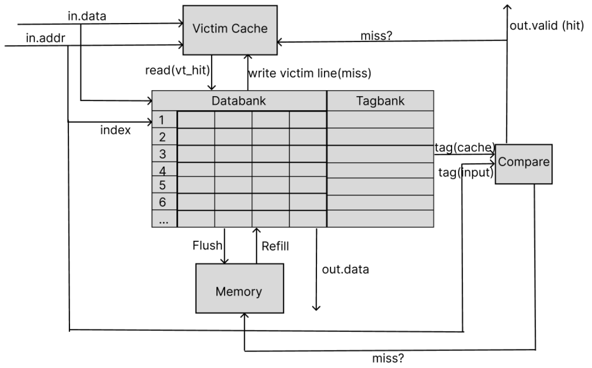


Figure 8: Block Diagram of Victim Cache

The way the processor to load a cache line in the baseline design is to load it byte by byte with multiple cycle using an enable signal for each offset. If the line size of the cache line is 4, then we will need 4 cycles to write the data to the cache databank by setting write-enable signal to 0, 1, 2, 3 and the databank module will write the write-data to the corresponding offset. For the victim cache implementation, we use the same way to write a cache line from victim cache to databank, so we add a new state to the initial state machine to write cache line data to databank within multiple cycles.

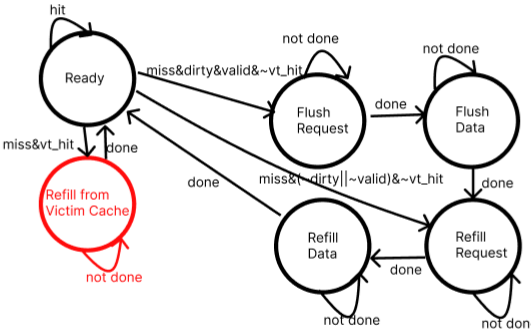


Figure 9: State Diagram of Victim Cache for D-cache

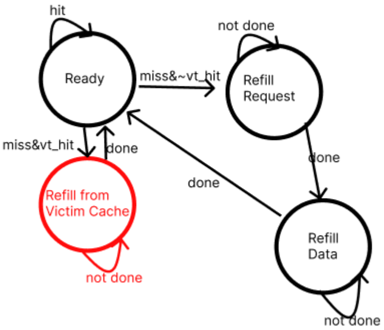


Figure 10: State Diagram of Victim Cache for I-cache

The state machine shown in Figure 8 (D-cache) and Figure 9 (I-cache) can explain how we write the data from Victim cache to databank. If a miss occurs, we will also check if there is a hit in victim cache (labeled as ”vt_hit”). If there is a hit in victim cache and miss in the main cache, then we will fetch the cache line from victim cache instead of memory, which will take less cycles to complete. Also, whenever there is a miss occurs, we will write the victim cacheline into victim cache regardlessly in ready state. The way we write the data from databank to victim cache only needs one cycle. We use a bus whose bus width is same as the cacheline size, so we do not have to use another state to finish this process. In other word, we need one cycle to write from databank to victim cache in ready state and four cycles to write from victim cache to databank.

We tested our victim cache line on four different benchmark and compared it with our baseline implementation (including the previous implementation from branch predictor).

Implementations	Benchmark (metric: CPI)			
-	nqueens	coin	qsort	esift
baseline	30.8877	7.12485	4.50185	2.2537
Victim cache - size 16				
victim cache (I-cache only)	20.7487	7.12485	4.50185	2.2537
victim cache (D-cache only)	29.4734	5.34386	4.44093	2.2537
victim cache (D-cache and I-cache)	18.6795	5.34386	4.44093	2.2537
Victim cache - size 32				
victim cache (I-cache only)	16.2103	7.12485	4.50185	2.2537
victim cache (D-cache only)	29.438	4.02328	4.3953	2.2537
victim cache (D-cache and I-cache)	13.8215	4.02328	4.3953	2.2537

Table 5: CPI result four 4 benchmarks and with 32-entry/64-entry victim cache for I-cache or D-cache or both

Table 5 shows the CPI result for 32-entry and 64-entry victim cache for I-cache or D-cache or both. Benchmark nqueens and coin is improved significantly, while qsort has slight improvement and esift doesn't have any. Also, we notice that nqueens is more affected by I-cache and coin is more affected by D-cache. We also notice that the size of victim cache affects the performance of the benchmark. We want to find the threshold of the size where it reaches the maximum improvement, so we design victim cache with 4-entry, 8-entry, 16-entry, 32-entry, 64-entry, 128-entry and we plot the performance of different size as shown in Figure 11.

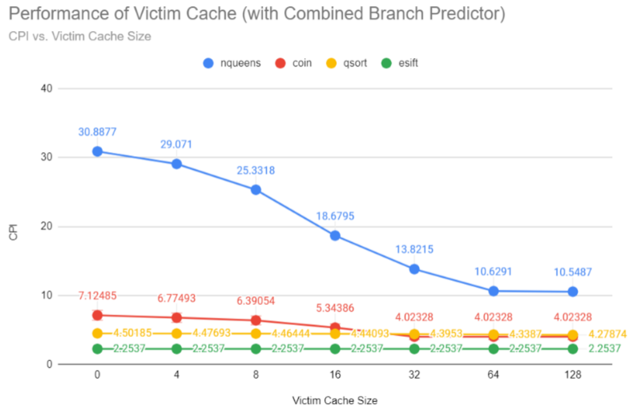


Figure 11: CPI result four 4 benchmarks and with multiple sizes of victim cache

At the end, we chose a 32-entry victim cache for D-cache and 64-entry victim cache for I-cache for our final implementation, and the performance is listed in Table 6.

2.3 Stream Buffer

We also implement the stream buffer for our optimization, which is to build a support hardware unit that observes the cache miss stream, recognizes patterns, and begins prefetching future misses [2].

Although the stream buffers and victim caches are both considered hardware prefecthing, the motivation behing interchanging and using both asynchronously is to provide the results if one prefetch technique was chosen over the other.

Stream buffer consists of a series of entries, each consisting of a tag, an available bit, and a data line. When a miss occurs, the stream buffer begins prefetching successive lines starting at the miss target. When a line is moved from a stream buffer to the cache, the entries in the stream buffer can shift up by one and a new successive address is fetched [2].

We only implement stream buffer for I-cache because I-cache is read-only while D-cache has both read and write operations. Our design of stream buffer does not fit in the combination of read and write operation, because it is hard to keep track of the sequential queue if we want to update the data that is contained in the stream buffer. Figure 12 briefly shows our idea of stream buffer. Our stream buffer will takes the pattern of sequence of data as input to our stream buffer during operations.

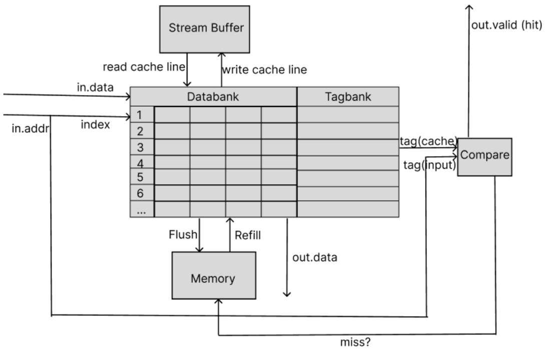


Figure 12: Block Diagram for Stream Buffer in I-cache

As we are developing the stream buffer feature, we notice that there is a hold-time constraint that occurs during the stream buffer module and the main cache module. The stream-buffer-hit signal is only up for 1 cycle because at the next cycle, the head entry will be removed and the output stream-buffer-hit-signal will be 0, which is before the combinational logic in I-cache finish executing. We have been looking for ways to solve this situation but at the end, we choose to slightly modify the design of stream buffer.

Our version of stream buffer is still a circular array list where data is inserted sequentially from the tail. However, in our stream buffer, we do not only check the head entry. Instead, we check all the elements by using the encoder implemented previously for victim cache.

This implementation in reality will require more transistors due to the encoder, but it has higher performance comparing to a typical stream buffer because it increases the chance of getting the right "pattern" because the query order do not have to be in order any more.

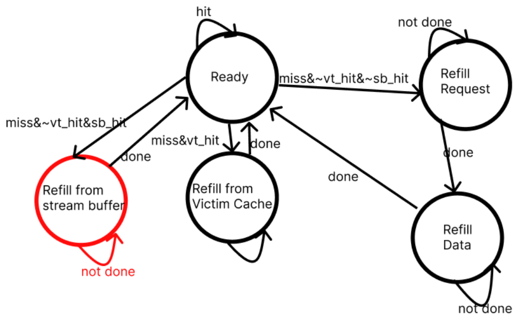


Figure 13: state diagram for Stream buffer implementation

As shown in Figure 13, the way writing data to stream buffer and outputting data from stream buffer to databank is similar to what we do for victim cache. We add one more state in the state machine which will be transitioned when neither hit in databank nor hit in victim cache but hit in stream buffer.

	nqueens	coin	qsort	esift
baseline	30.8877	7.12485	4.50185	2.2537
stream buffer(only)	13.6408	7.12485	4.50185	2.2537
victim cache(only)	10.6292	4.02328	4.3953	2.2537
stream buffer and victim cache	10.5887	4.02328	4.3953	2.2537

Table 6: Performance(CPI) of 64-entry stream buffer with or without victim cache (all run with combine BP)

Table 6 shows the performance of the (32-entry D-cache and 64-entry I-cache) victim cache and (64-entry) stream buffer implementation. Note that all the data is run with combine branch predictor implemented including the baseline. We can see stream buffer mainly improve the performance of nqueens benchmark while victim cache improves nqueens, coin, and qsort. The reason of limited improvement of stream buffer is because we only implement it with I-cache, and nqueens is mainly affected by I-cache while coin and qsort are affected by D-cache. Also, although stream buffer itself can improve nqueens significantly, we notice that when we combine stream buffer and victim cache, the CPI is barely improve comparing to either stream buffer or victim cache.

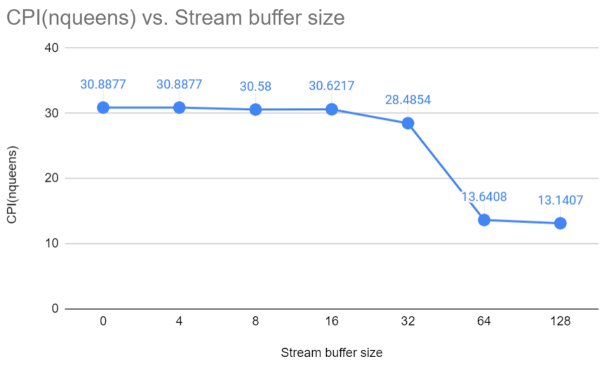


Figure 14: CPI result for nqueens benchmark with multiple sizes of stream buffer

Figure 14 shows how the size of stream buffer affect the performance. The plot shows that at size 64, there is a significant turning point of performance. With this result, we decided to choose size 64 for our stream buffer implementation.

2.4 Value Prediction

Load Value Prediction is a way to identify instructions with predictable outcomes. If the instruction is stalled, we should provide the predicted outcome and proceed.

2.4.1 Load Value Prediction Table

The Load Value Prediction Table (LVPT) is used to predict the value being loaded from memory. It is indexed by the load instruction address and is not tagged. The LVPT is also direct-mapped, and it can be both constructive and destructive interference can occur between loads that map to the same entry. Our implementation of LVPT includes both load table and confidence table[3].

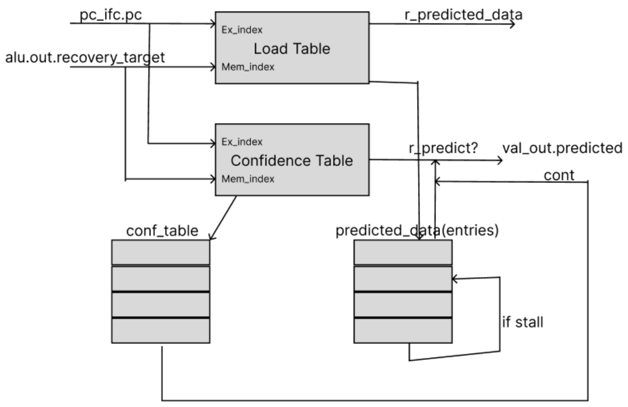


Figure 14: Block Diagram for value Prediction

The Load PC is used to index into the LVPT and LCT to find a value to predict and to determine whether or not a prediction should be made [3]. When the load completes, the predicted and actual values are compared, the LVPT and confidence table are updated, and dependent instructions are reissued if necessary. An index of memory is used to see if the value prediction matches the previous predicted value. An index of execution is used, based on if the confidence meet the threshold requirement, to load the predicted value from the LVPT as the new predicted value. If a stall is detected in that, the value is reverted back to the recovery target [3].

2.4.2 Confidence Table

A table of all saturating counters are saved for estimation to calculate when a prediction will most likely be correct. The counters are incremented when the prediction is correct and decremented otherwise. A threshold will be used to compare as an estimation. If the confidence is below the threshold, the predictor is prevented from making a prediction. A recovery technique is implemented to execute the instruction that depend on the misprediction load value again. If there is a stall in the execution pipeline that is detected, then the predictive data is not counted as predicted and the branch is reverted back to the recovery target [4].

3 Result Analysis

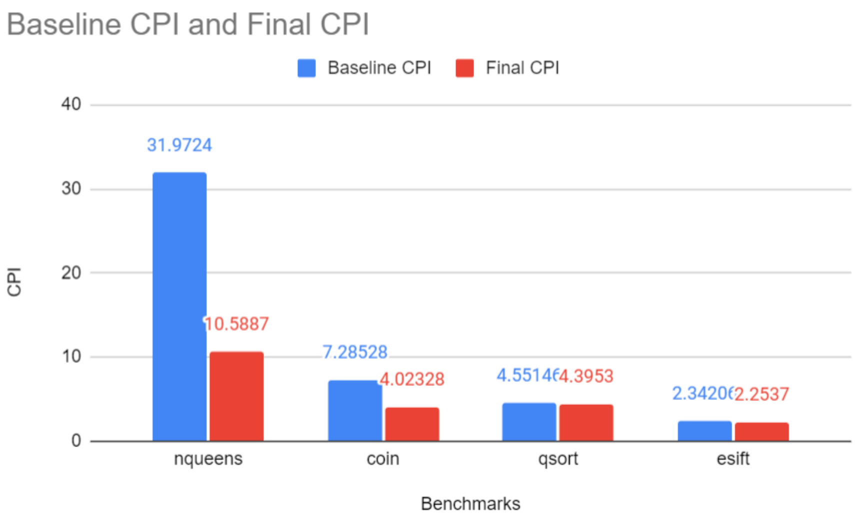


Figure 15: Baseline and Final Result

Our implementation significantly improve the performance of nqueens and coin benchmark, while qsort and esift are slightly improved. From the above architecture and pre-analysis, we know that nqueens is improved by having victim cache and stream buffer in i-cache, while coin and qsort are improved by victim cache in d-cache. Nqueens benchmark get the most benefit out of improving i-cache, and coin benchmark get the most benefit out of improving d-cache. Qsort get benefit from all the implementations except stream buffer, while esift can only be improved by branch predictor. Figure 15 shows us the results of our final implementation comparing with baseline.

From the data summary in section 5, tables 7-11 recorded all the data in detail for baseline, combining branch predictor, victim cache, and stream buffer:

- For combining branch predictor, overload in execution stage decreases while increases in decode stage.

- For victim cache, the count of flush and stall all decrease in prefetch, decode, execute, and memory stages, especially for nqueens and coin benchmarks

- For stream buffer, the count of flush and stall all decrease in prefetch, decode, execute, and memory stages for nqueens benchmark. However, we notice although miss in i-cache is much reduced, but miss in d-cache is increases. It is not likely happen in the stream-buffer-only case since D-cache is not embedded with stream buffer. We might need further improvement on our stream buffer.

- For our final implementation, we can see stall and flush are all decreased in all stages. However, we still notice that the overload in decode stage is still higher than baseline, which also occurs in combining branch predictor. We might need further development in the future for this part.

4 Future Development

Due to time constraint of the quarter system, we couldn't successfully finish some of the tasks that were planned. We will probably keep working on the further implementation as follow:

- Local value prediction is currently 80% completed. Load Table and confidence table have been individually implemented. It is been embedded but did not contribute to the improvement. In the future, we will need to figure out a way to correctly embed the load table and confidence table to our main code and debug the problems that may occur.

- With the result presented in combining branch predictor, we wonder what the reason is for having a higher overload in decode stage.

- Stream buffer is currently only implemented for I-cache because it is designed as a read-only buffer. In the future, we may be able to find a way to resolve the read-write conflicts and use it for d-cache as well.

5 Data Summary

5.1 Baseline

Baseline				
	nqueens	coin	qsort	esift
Total Time	308517525	3118347715	172700175	257136785
Cycle Count	30851752	311834771	172700175	257136785
Instruction Count	964950	42803374	3794394	10979095
CPI	31.9724	7.28528	4.55146	2.34206
IPC	0.031277	0.137263	0.21971	0.426975
Stats				
ic_miss	22579904	3388	4261	2844
dc_miss	3233921	327	327	218
ex_overload	775826	6980073	973978	991411
mem_flush	10585716	261902912	12668879	12933836
dec_stall	12599706	261903130	12669097	12934054
if_stall	29664935	261905865	12672487	12936569
dec_flush	3233921	327	327	218
mem_stall	10585716	261902912	12668879	12933836
if_flush	23345452	6983460	978236	994253
dc_miss	10585716	261902912	12668879	12933836
dec_overload	2936	1	28830	224
ex_stall	10585716	261902912	12668879	12933836

Table 7: Baseline Design Performance and Simulation Stats

5.2 Combing Branch Predictor Only

Branch Prediction				
	nqueens	coin	qsort	esift
Total Time	298051105	3049677425	170817765	247435505
Cycle Count	29805110	304967742	17081776	24743550
Instruction Count	964950	42803374	3794394	10979095
CPI	30.8877	7.12485	4.50185	2.2537
IPC	0.0323753	0.140354	0.222131	0.443715
Stats				
ic_miss	21560645	3497	4261	2844
dc_miss	3233921	327	327	218
ex_overload	84705	112935	180532	21283
mem_flush	10586611	261902912	12668879	12933836
dec_stall	12600601	261903130	12669097	12934054
if_stall	28674047	261905974	12672486	12936569
dec_flush	3233921	327	327	218
mem_stall	10586611	261902912	12668879	12933836
if_flush	21630217	116431	184680	24125
dc_miss	10586611	261902912	12668879	12933836
dec_overload	5098205	20461300	1003450	1093081
ex_stall	10586611	261902912	12668879	12933836

Table 8: Performance and Simulation Stats Optimized by Improved, Combined Branch Prediction

5.3 (32-entry D-cache 64-entry I-cache) Victim Cache Only

Victim Cache				
	nqueens	coin	qsort	esift
Total Time	137110715	1790769285	168657425	257136785
Cycle Count	13711071	179076928	16865742	25713678
Instruction Count	964950	42803374	3794394	10979095
CPI	14.2091	4.18371	4.44491	2.34206
IPC	0.0703774	0.239022	0.224976	0.426975
Stats				
`ic_miss`	5077114	3388	4261	2844
`dc_miss`	660775	327	327	218
`ex_overload`	775826	6980073	973978	991411
`mem_flush`	8034854	129144972	12264604	12933836
`dec_stall`	8525577	129145190	12264822	12934054
`if_stall`	12524254	129148022	12268212	12936569
`dec_flush`	660775	327	327	218
`mem_stall`	8034854	129144972	12264604	12933836
`if_flush`	5842662	6983460	978236	994253
`dc_miss`	8034854	129144972	12264604	12933836
`dec_overload`	2936	1	28830	224
`ex_stall`	8034854	129144972	12264604	12933836

Table 9: Performance and Simulation Stats Optimized by Inclusion of a Victim Cache

5.4 (64-entry) Stream Buffer Only

Stream Buffer				
	nqueens	coin	qsort	esift
Total Time	128246725	3118347715	172700175	257136785
Cycle Count	12824672	311834771	172700175	257136785
Instruction Count	964950	42803374	3794394	10979095
CPI	13.2905	7.28528	4.55146	2.34206
IPC	0.0752417	0.137263	0.21971	0.426975
Stats				
`ic_miss`	1293718	3388	4261	2844
`dc_miss`	10532033	327	327	218
`ex_overload`	775826	6980073	973978	991411
`mem_flush`	10532033	261902912	12668879	12933836
`dec_stall`	10647218	261903130	12669097	12934054
`if_stall`	11637855	261905865	12672487	12936569
`dec_flush`	181928	327	327	218
`mem_stall`	10532033	261902912	12668879	12933836
`if_flush`	2059266	6983460	978236	994253
`dc_miss`	10532033	261902912	12668879	12933836
`dec_overload`	2936	1	28830	224
`ex_stall`	10532033	261902912	12668879	12933836

Table 10: Performance and Simulation Stats Optimized by Inclusion of Stream Buffers

5.5 Total Result

Total Results				
	nqueens	coin	qsort	esift
Total Time	102175365	1722098995	166775015	247435505
Cycle Count	10217536	172209899	16677501	24743550
Instruction Count	964950	42803374	3794394	10979095
CPI	10.5887	4.02328	4.3953	2.2537
IPC	0.0944406	0.248554	0.227516	0.443715
Stats				
`ic_miss`	1239363	3497	4261	2844
`dc_miss`	660775	327	327	218
`ex_overload`	84705	112935	180429	21283
`mem_flush`	8031438	129144972	12264604	12933836
`dec_stall`	8146211	129145190	12264822	12934054
`if_stall`	9086473	129148022	12268211	12936569
`dec_flush`	181516	327	327	218
`mem_stall`	8031438	129144972	12264604	12933836
`if_flush`	1313467	116431	184577	24125
`dc_miss`	8031438	129144972	12264604	12933836
`dec_overload`	2803580	112935	1003450	1093081
`ex_stall`	8031438	129144972	12264604	12933836

Table 11: Performance and Simulation Stats Optimized by Inclusion of all optimization

6 Reference

[1] Combining branch predictors - HP labs (available at https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf).

[2] Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. IEEE Xplore (available at https://ieeexplore.ieee.org/document/134547/).

[3] Value Locality and Load Value Prediction
https://pharm.ece.wisc.edu/mikko/oldpapers/asplos7.pdf).

[4] Load Value Prediction Using Prediction Outcome Histories (available at
https://userweb.cs.txstate.edu/ burtscher/papers/tr98.pdf).