Final Design Report

Anthony Chen (A16516603), Pethaperumal Natarajan (A16242691)

Professor Tullsen

20 March 2024

1. Design Goals
When we were first given the baseline processor design, our primary goal was to maximize the performance of our processor by minimizing the CPI across all provided benchmarks. The CPI of the baseline are shown in the table below:

Benchmark	*CPI*
NQueens	*14.5653*
Coin	*1.00674*
Esift2	*2.71329*
Quicksort	*2.41499*

Additionally, upon designing our own statistical measurements, we quickly found that the biggest issue came from cache accesses. Cache misses resulted in a combined loss of several million cycles for more memory intensive benchmarks such as *Esift2*. Hence, we chose to focus on optimizations that would allow us to minimize cache miss losses.

Our four optimizations included the following: branch prediction, hardware prefetching, value prediction, and cache set-dueling.

2.1 Branch Predictor - Design

Branch misprediction results in a loss of a few cycles. The baseline CPU design has an average branch miss rate of around 40%. This would mean that we have lost a few cycles in 40% of the instruction the cpu runs. So, our goal is to minimize the branch miss rate by implementing the perceptron branch predictor. Perceptron is a simple neural network system that trains during the first part of the program execution and uses the trained perceptron to predict for the rest of the program. The equations to train perceptron is below:

Training perceptron:

```
if $\text{sign}(y_{out}) \neq t$ or $|y_{out}| \leq \theta$ then
    for $i := 0$ to $n$ do
        $w_i := w_i + tx_i$
    end for
end if
```

Predicting output:

$$y = w_0 + \sum_{i=1}^{n} x_i w_i .$$

In order to further improve the perceptron, global history is tracked, and the instruction program counter is hashed to use one of the 256 perceptrons. Initially, we used the hyperparameters given in the paper, but the perceptron performed poorly. We then reduced the hyperparameters such that the weight is 6 bits, number of perceptrons is 256 and global history register is 7 bits. This gave us a massive improvement in branch miss rate.

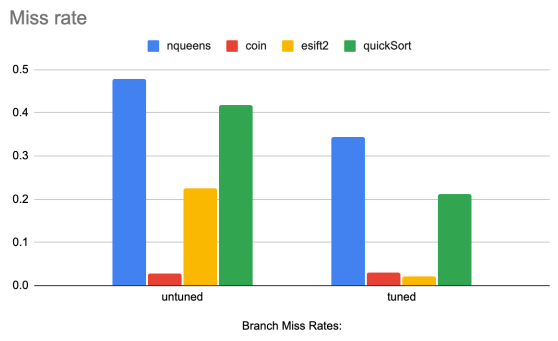


Figure 2.1 Tuned vs Untuned Branch Miss Rate

As shown in Figure 2.1 above, the tuned perceptron has much better results compared to the hyperparameters given in the paper. We believe the improvements are because our programs are short, and fewer perceptrons would mean that each perceptron would be able to train in a short amount of time and it will start predicting sooner.

2.2 Branch Predictor - Results

As shown in figure 2.2, perceptron performs much better than all the other branch predictors used. Comparing perceptron branch miss rate to the 2-bit branch predictor (2nd to last in the chart below), nqueens improved by around 120%, coin improved by around 900%, esift2 improved by 1100% and quickSort improved by 300%.

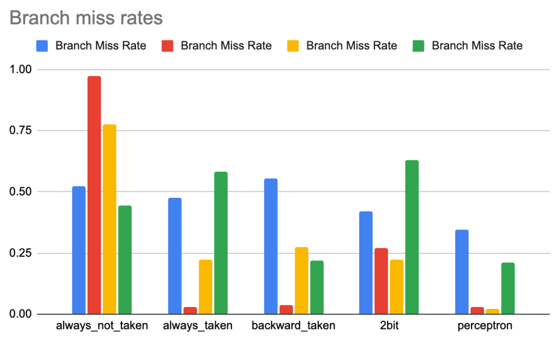


Figure 2.2 Branch Miss Rates

However, the CPI of perceptrons did not improve too much.

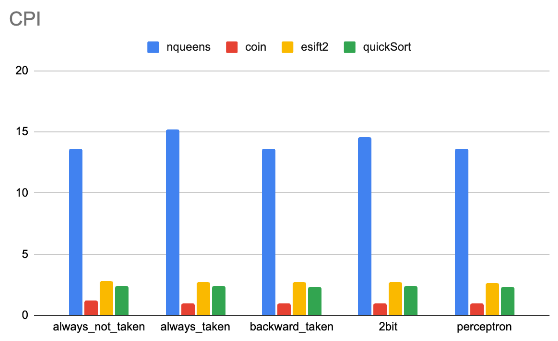


Figure 2.3 CPI for different branch predictors

Program	**Speedup**
nqueens	1.0705
coin	0.999
esift2	1.0123
quickSort	1.0417

Figure 2.3 Speedup for different programs

We believe we do not see big improvements in CPI because the cpu is slowed down by I-Cache and D-Cache misses despite having a more accurate branch predictor.

3.1 Hardware Prefetching - Design

Despite the performance improvements from a much more accurate branch predictor, *NQueens* still stands out with a high instruction cache miss rate, which is one of the main reasons why its CPI is so much higher than the other benchmarks. Because of its large program size, *NQueens* is unable to fit its entire program into the instruction cache, resulting in a large number of capacity misses. Hence an instruction cache stream buffer was used to both reduce the number of compulsory and capacity misses.

The stream buffer unit took inputs from the fetch unit and the i-cache to determine whether it needed to send a memory request or if it could forward the i-cache output. Another AXI port was created to support simultaneous memory reads from both i-cache and stream buffer.

Upon encountering an read miss in both i-cache and stream buffer, the i-cache would fetch the missed cache line, and the stream buffer would fetch the following cache line. A hit in either stream buffer or i-cache would be treated like a normal cache hit, resulting in no additional fetching.

3.2 Hardware Prefetching - Results

This initial design of a direct-mapped buffer with a depth of 8 cache lines resulted in significant decreases i-cache misses in *NQueens*. Even *Coin, Esift,* and *Quicksort* saw slight improvements due to the stream buffer reducing the amount of compulsory misses.

	Baseline	*Stream Buffer*
NQueens	*114265*	*87869*
Coin	*28*	*20*
Esift2	*24*	*15*
Quicksort	*43*	*38*

Fig 3.1. Number of I-Cache misses

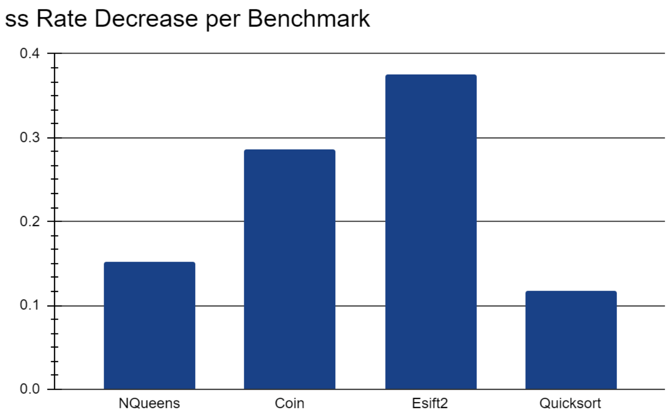


Fig 3.2. (above) Percentage decrease of miss rates

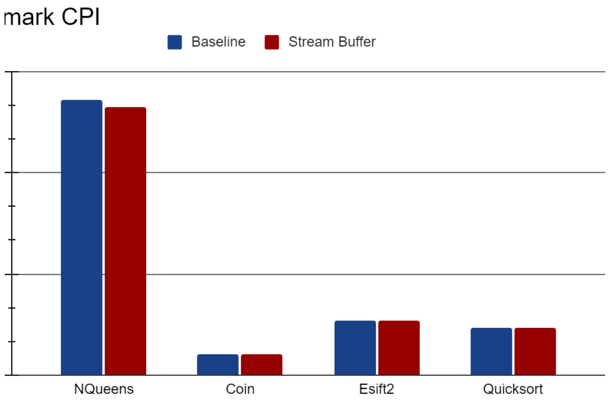


Fig 3.3. Baseline vs Stream Buffer CPI

Despite *NQueens* having the second lowest percentage miss rate decrease amongst all benchmarks, it sees the highest improvements in CPI due to it having a higher ratio of i-cache misses to total instructions.

Different buffer depths were tested to determine the optimal performance.

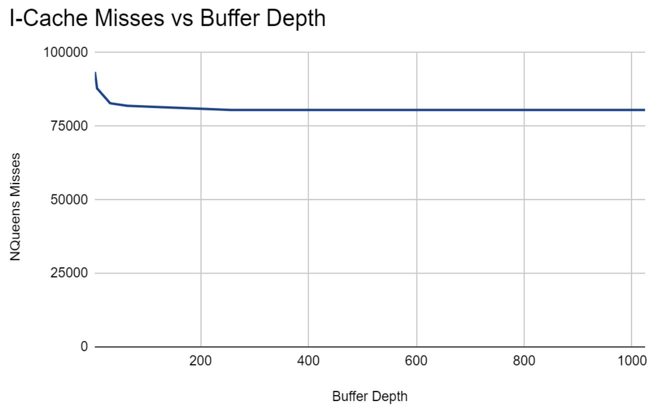


Fig 3.4. I-Cache misses with different buffer depths

However, it is clear that the buffer depth has significant diminishing returns largely due to the buffer being directly mapped, resulting in a large amount of instructions hashing into the same buffer table index. Having a fully associative buffer may increase the performance of larger buffers. However, associativity was not tested due to research time constraints and because of the theoretical increase in access times.

Overall, a i-cache stream buffer with depth 8, or table size of 128 bytes, was determined to be the best performance for size. This is the prefetch unit used for the rest of this paper.

4.1 Value Prediction - Design

The i-cache stream buffer addresses i-cache misses, but does not affect the d-cache. A glance at the table below reveals the d-cache's huge potential for improvement, especially for *Esift* and *Quicksort*.

Benchmark	*D-Cache Misses*
NQueens	*97*
Coin	*29*
Esift2	*113157*
Quicksort	*43529*

Figure 4.1. D-Cache misses for baseline + stream buffer

Instead of stalling every time a load miss is encountered, the value predictor will predict the outcome of the load and continue executing in hopes that the prediction was correct. In the best case scenario of predicting accurately while having sparse

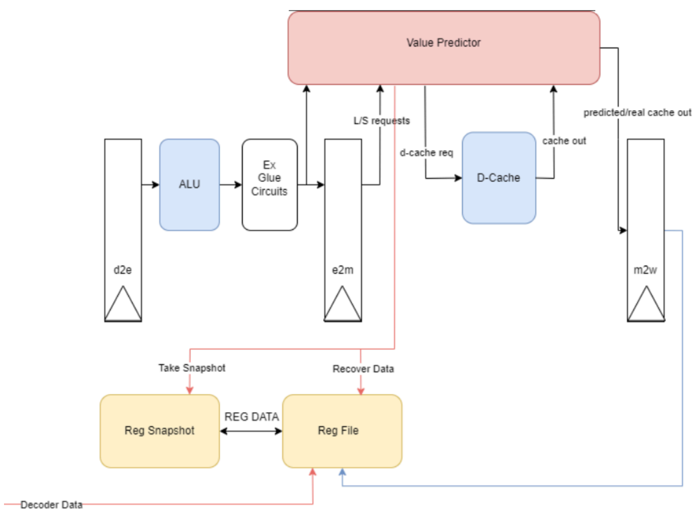


Figure 4.2. High level of affected areas by value predictor

memory accesses, value prediction would be able to save the cycle cost of a load.

Encountering additional memory accesses during speculative execution results in a pipeline stall, because speculative memory stores are expensive to undo and speculative loads require a lot more checkpointing hardware with potentially diminishing performance for its cost.

The value predictor module is inserted into the hazard control to give it access to pipeline stalls and flushes. It detects memory accesses from the output of the *e2m* pipeline register, with the output of *ex-glue-circuits* used for edge cases of back-to-back memory accesses. The d-cache is disconnected from the rest of the pipeline because the rest of the pipeline does not need to differentiate between speculative or normal execution, as the value predictor handles all related logic.

For checkpointing value prediction, a new register snapshot module was used to copy all register data from the register file. However, because the pipeline registers are not checkpointed, restoring from an incorrect prediction results in a net loss of 3 cycles compared to the baseline design, making value prediction potentially very costly.

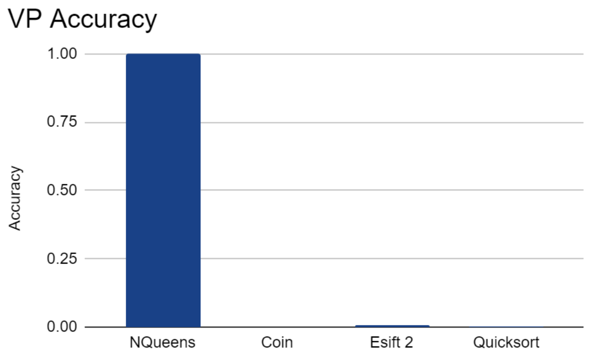
4.2 Value Prediction - Results

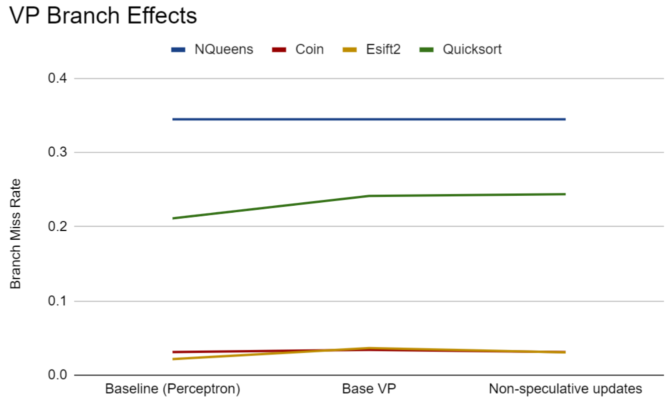
An initial design of only predicting 0's was used to compare d-cache miss results.

	Baseline	*Value Prediction*
NQueens	97	96
Coin	29	29
Esift2	113157	112375
Quicksort	43529	23789

Fig 4.3. D-cache Misses

Here correct value predictions are counted as a cache hit. However, the results from *Quicksort* are misleading; despite only correctly predicting once, *Quicksort* still sees a tremendous decrease in d-cache misses. This is because d-cache misses are calculated using an edge detector, and the entire duration of a speculative execution counts as a d-cache miss, condensing





otherwise separate d-cache misses.

Fig 4.4. Value prediction hit rates for each benchmark

The value prediction accuracy is 100% for *NQueens* because it only occurs once since every other memory access is a store followed by a load hit. Despite *Esift* having a seemingly low accuracy, it is accurate enough to have an overall speedup. *Coin's* CPI stays relatively the same because it only value-predicts a few times. Only *Quicksort* misses enough predictions to have a significant detriment to CPI.

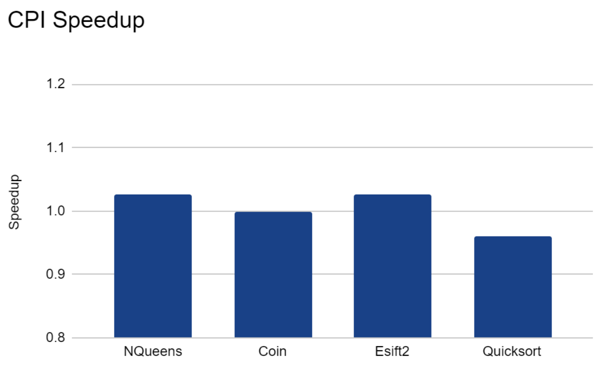


Fig 4.5. CPI speedup from value prediction

Another aspect of the pipeline affected by VP is branch prediction accuracy. Because there are branch conditions dependent on loaded data, speculatively execution can negatively impact branch predictor accuracy. Thus, a design that prevented speculative updates to our branch table was tested.

Fig 4.6. Effects of VP on branch miss rate

The number of branch misses either stayed the same or increased going from no VP to VP. However, preventing speculative updates to our branch predictor was able to decrease the misses from VP.

Finally, different prediction schemes were tested, including the LVPT. However, the performances of these prediction schemes were about the same or worse compared to predict 0 on all benchmarks, likely due to the lack of value locality on these specific benchmarks.

Overall, the value prediction did not significantly increase the processor's performance, but better results can be achieved with a more accurate predictor or a more efficient checkpoint restoration mechanism.

5.1 Cache Set Dueling - Design

From our previous optimizations, we found out the D-Cache misses were one of the key factors increasing the CPI. Our goal with cache set dueling is to decrease D-Cache miss.

Cache set dueling is a very effective optimization because it does not require significant hardware overhead. Cache set dueling divides the D-Cache indexes into 3 sets. The first set uses the first cache policy, the second set uses the second cache policy and the third set picks one of the 2 cache policies depending on which performs better.

The cache policies we chose to experiment with the 2-way associative D-cache are least recently used replacement policy (LRU), most recently used replacement policy (MRU), and bimodal insertion policy (BIP). From the experiments, we found LRU to have the best overall performance.

LRU evicts the least recently used way and inserts the new data in the most recently used way. MRU evicts the most recently used way and inserts new data as the most recently used way. LRU Insertion Policy (LIP) evicts the least recently used way and places new data as least recently used as well. Once the placed data is accessed again, it is then promoted to the most recently used spot. BIP is similar to LRU, but it would occasionally place the new data as most recently used to adapt to changes in the working set.

LRU displayed the best result in every program except for esift2. MRU showed the best results for esift2, but it was significantly worse for other programs. BIP was also better than LRU for esift2, but it was very slightly worse for all the other programs. Since LRU performed the best in almost every program, we expected improving miss rates for all programs to be difficult. Since cache set dueling gives results which would be the combination of 2 policies, we can expect the results to be a midpoint between the 2 products. We thought it would not be possible to get results better than the policy that would work best for a program.

So, we decided to move forward with cache set dueling with LRU and BIP because they had the best results for all 4 programs. We believed that by using both of these, we would be able to improve at LRU with esift2 and minimize performance decrease with the other programs that work best with LRU.

5.2 Cache Set Dueling - Results

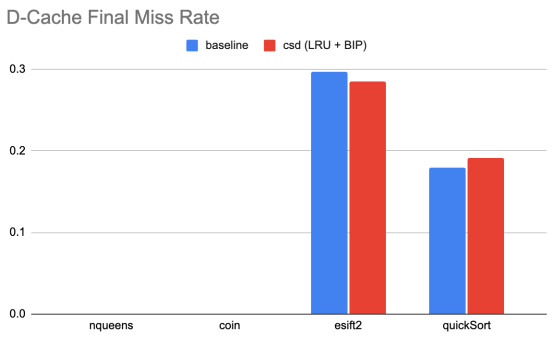


Figure 5.1 D-Cache Miss Rates for different programs

	nqueens	coin
baseline	0.000388014	2.61E-06
csd (LRU + BIP)	0.000388014	2.61E-06

Cache set dueling with LRU and BIP managed to maintain the branch miss rate for nqueens and coin. This was possible because the first set uses LRU which is the same as baseline. The third set uses the policy that performs better, which is the LRU. In the second set, BIP worked almost as well as LRU, so the results were the same. However, for the esift2, there is an improvement on the miss rate. BIP performs better for esift2, so esift2 gained an advantage from the cache set dueling. However, quickSort has an increased miss rate because LRU performed much better than BIP. As a result of using both LRU and BIP, quickSort has an increased miss rate.

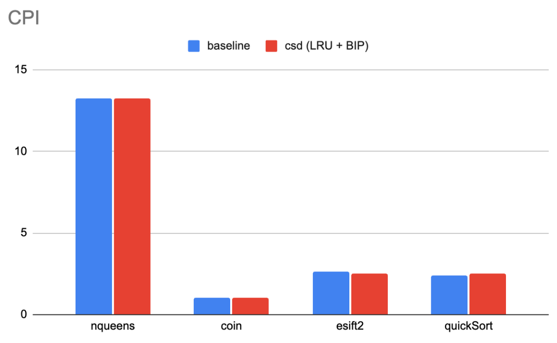


Figure 5.2 CPI for different programs

Similar to the D-Cache miss rate, the CPI remained the same for nqueens and coin, improved for esift2 and worsened for quickSort.

Program	**Speedup**
nqueens	1
coin	1
esift2	1.028
quickSort	0.961

Despite the results not being the best, we believe that a much longer program with different branching patterns will always have an overall better performance with cache set dueling. The nature of the programs we tested are short, so we think that cache set dueling will perform better in general programs.

6. Final Results

The final design gave us a small improvement over the baseline. As shown in Figure 6.2, there is a noticeable improvement in nqueens and esift2. Minor difference on coin and a slight deterioration on quickSort. We think quickSort is performing worse because quickSort performed far better with LRU then BIP, which caused cache set dueling to make it worse. Also, our value prediction also does not predict values accurately for quickSort resulting in more deterioration.

However, our optimizations worked well with the 3 other programs and we can see an improvement in their performances.

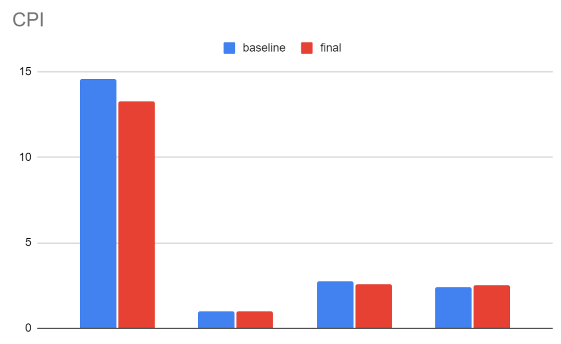


Figure 6.1 Baseline vs Final CPI

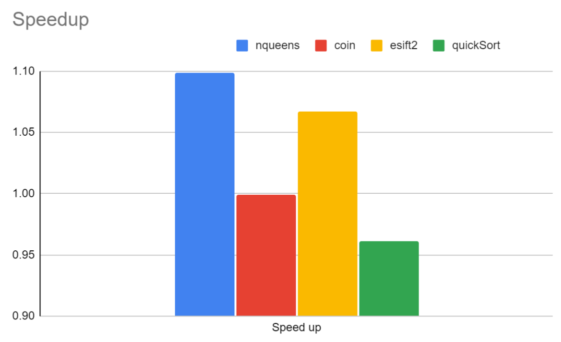


Figure 6.2 Speedup for different programs

Average speed up: 1.0324

7. Works cited

Qureshi, Moinuddin K., et al. "Adaptive Insertion Policies for High Performance Caching." ECE Department, The University of Texas at Austin, {moin, patt}@hps.utexas.edu, Intel Corporation, VSSAD, Hudson, MA, {aamer.jaleel, simon.c.steely.jr, joel.emer}@intel.com.

Jiménez, Daniel A., and Calvin Lin. "Dynamic Branch Prediction with Perceptrons." Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, {djimenez, ling}@cs.utexas.edu.

Jouppi, Norman P. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." Digital Equipment Corporation Western Research Lab, 100 Hamilton Ave., Palo Alto, CA 94301.

Jouppi, Norman P. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." Digital Equipment Corporation Western Research Lab, 100 Hamilton Ave., Palo Alto, CA 94301.