CSE 148 Final Report

Mitchell Bizzigotti and Zuo Yang

March 25, 2025

Introduction

For our design project we picked 4 optimizations to implement on top of the baseline processor: branch prediction, register renaming, out-of-order execution, and hardware prefetching. We were able to be successful in implementing the first 3 optimizations, however in focusing most of the time working on them, prefetching is not fully working in our processor.

With all our optimizations put together and compared, we were on average able to get a 35.6% decrease in CPI over the baseline in all benchmarks.

Branch Predictor

The first optimization that our group did is the branch predictor. And we chose YAGS predictor as our predictor to implement. We also found another paper [1] that is supposed to improve the YAGS predictor's performance.

Original YAGS

In the original YAGS paper [2], the author mentioned bimode predictor. A bimode predictor is implemented based on the fact that branches are generally biased toward either taken or not taken as results. Therefore, storing branches based on their biases is a natural way to categorize them to improve the efficiency of checking the result. A bimode predictor is consisted by a choice PHT and two direction caches. The two caches are labeled as taken and not taken that can store the respective branch results. The purpose of this is to reduce destructive aliasing. YAGS predictor is built on top of the bimode predictor. In bimode predictor, large spaces are wasted in the taken and not taken caches. In order to save space, the YAGS predictor is implemented so that the caches only store the biases instead of every branch result. When the branch result does not agree with the choice PHT in the middle, this means it is a bias, so that we can store it in the respective cache. In this way, we do not need that many cache spaces since we are only storing the biases.

Improved YAGS predictor

There are some limitations on YAGS predictor, first is that increasing the associativity has little or no improvement. In the original paper, it is assumed that is due to the lack of aliasing. Second is that the spacing is still very large and can be further refined. In another paper we found suggests that the idea of spliting into two caches is too artificial and we can merge just the two caches in YAGS predictor into one multi-set cache to further reduce the space requirement. And we chose to use LRU as our replacement policy for the cache as suggested by the paper. Our improved YAGS predictor utilized bitwise xor to combine the address and global history. Our design uses ADDR_WIDTH bits of global history to xor with the PC. Also, we used 8 bits of tag from the lsb of the combined value. For the sizes of the direction caches. We used the size of the choice PHT/associativity.

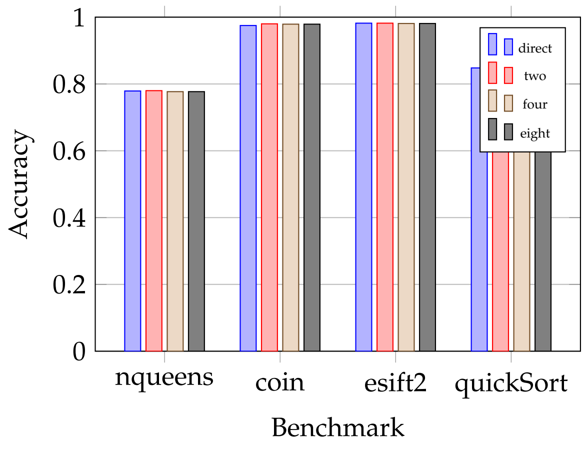
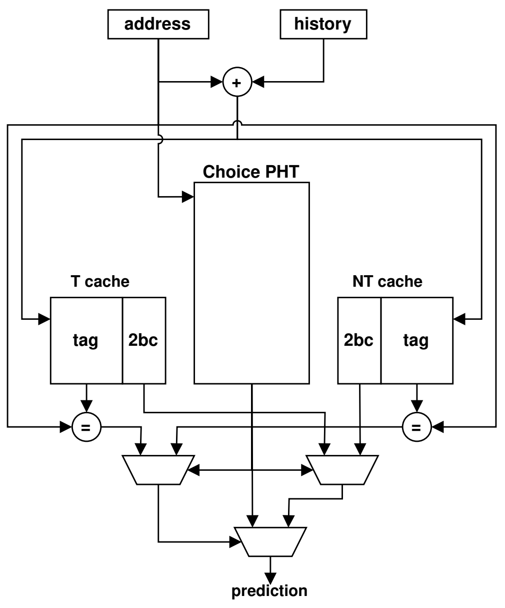


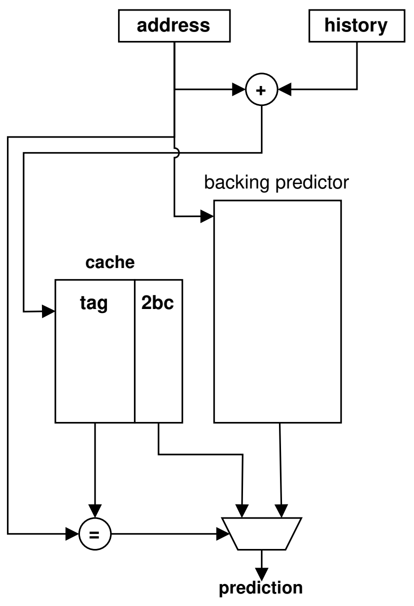
Figure 1: *Accuracy for different associativity*

Results

Associativity Now let's examine the results. We have gathered the results for four different branch predictors. They are always not taken, Gshare, original YAGS, and improved YAGS. We examined associativity size of 1(direct), 2, 4, and 8. Based on Figure 1, we can see that even with the change in associativity of the direction caches in the improved YAGS, the accuracy hardly changes. This result is not consistent with the paper[1], but is consistent with [2], the original YAGS paper. This suggests that there is not much aliasing in the direction caches in our four benchmarks.



(a) *Original YAGS Predictor*



(b) *YAGS with unified direction cache*

Figure 2: *Difference between original and improved YAGS predictor*

Choice PHT size Next we are going to examine different sizes of the choice PHT. Ideally, we want to choose a large enough pht size so that there are as little aliasing happens as possible. But we can't choose infinite. So we used 2^{ADDR_WIDTH} as our size of choice PHT. This way all of the address can have an unique place in the choice PHT. And the result is consistent with our expectations. In Figure 3, we examined the PHT size of the total number of addresses divided by 10^8, 10^4, 10^2, and the total number of addresses. The result is consistent with our expectations. Even though for coins and esift2, the accuracy decreased as pht size increased in some interval. Eventually the pht size of all addresses becomes the most accurate one.

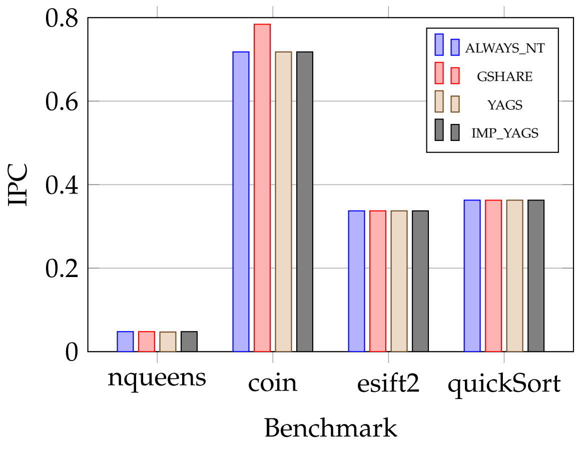


Figure 4: *IPC Comparison for Different Branch Predictors*

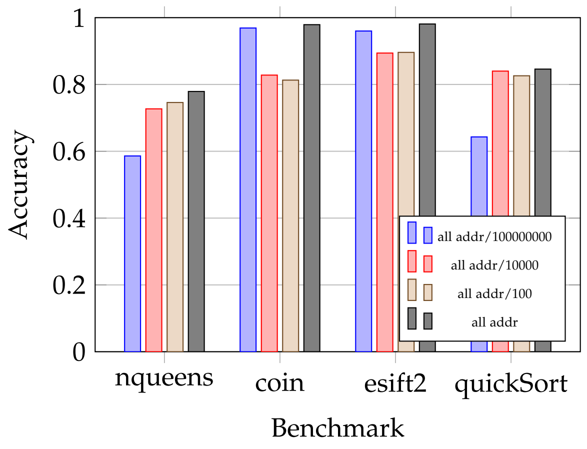


Figure 3: *Accuracy for different pht sizes*

Final Result Finally, we can use our knowledge from above to design our parameters for our improved YAGS predictor. We chose the saturation counter size of 2, PHT size of 2^{ADDR_WIDTH}, tag bits of 8, associativity size of 4, and global history size of ADDR_WIDTH. Also we used xor to combine the address and global history, and LRU replacement policy to update the direction cache. One drawback on our design is that although we find the most accurate model of our YAGS predictor, the IPC does not improve significantly as shown in Figure 4. Our assumption is that there are for loops in the combinational logic that can slow down IPC. Also Since we are implementing OOO, wrong branches will be executed either earlier or later that can mess up the predictor. To show improvement of the YAGS predictor. We also implemented a Gshare predictor to compare with our YAGS predictor. We used 2 bit saturation counter and PHT size of 2^{ADDR_WIDTH} to avoid aliasing. In Figure 5, we can see that there are always taken, Gshare, original YAGS, and our improved YAGS predictor. We can see that altough sometimes the original YAGS is less accurate that Gshare, our improved YAGS predictor is more accurate than gshare in all four benchmarks.

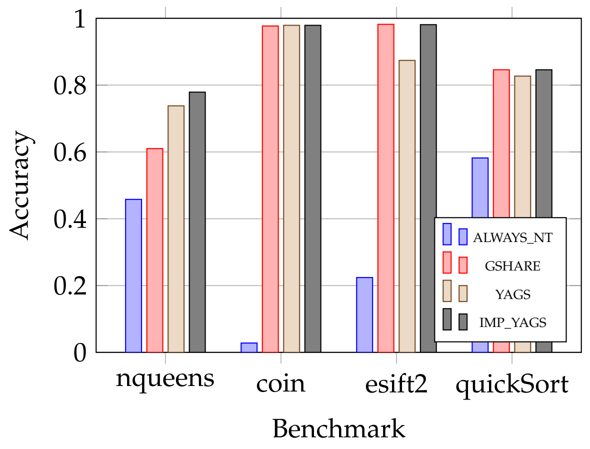


Figure 5: *Accuracy for Different Branch Predictors*

Out-of-Order Execution

The second optimization that we did was out-of-order execution. Our design is inspired from the MIPS R1000 paper [3], however with some simplifications to ensure that this optimization was able to completed within the time frame of this course.

On completion of out-of-order, our pipeline now has the following stages: fetch, decode, rename, issue & execute, and commit. The first two stages of the pipeline, fetch and decode, did not need to be modified in any major way from the baseline implementation, most of the changes were in the rest of the pipeline, and almost everything past the decode stage had to replaced to enable out of order execution. The final structure of our out-of-order pipeline can be seen in Figure 6.

The rename stage is the simplest part of the pipeline, with the only difficulty coming in allocating a register from the free list. When a decoded instruction comes into this stage, the physical registers for the instruction's source operands are simply looked up using the register mapping table. But, for the destination register, we need allocate a free register from the register free list. The register free list is implemented as a queue, so that when the pipeline needs to be flushed we can simply move the head pointer back to the tail, not having to do any check-pointing.

The commit stage is the second simplest part of the pipeline. The job of the commit queue (active list in MIPS R1000) is to store the results of every execution that happens (which may not be in program order), and commit each instruction in order. This is achieved by assigning each instruction a unique commit index. On an instruction committing, it will write it's results to the register file, push the old register it is replacing to the free list, mark the next store buffer entry for eviction (only for stores), and write the new committed mapping to the check-pointed register mapping table.

The issue & execute stage is the most complex part of our out-of-order pipeline. In this stage, we have two instruction queues, two execution units, and a memory unit to manage requests sent to the data cache. The reason that we had two instruction queues was to be prepared for superscalar, and to also separate instruction that needed strict ordering (load/stores) with those that did not. Although, because our pipeline is not superscalar, that means that we need to select which one of the instruction queues to take from; the logic for this is simple always prefer to take a memory instruction, otherwise take a non-memory instruction. The job of the memory unit is to combine the requests from the load-store execution unit and store queue into one request that is sent to the data cache. This is something unique to our design, different from how the MIPS R1000 handles loads/store, as they do not use a store queue (store buffer). The memory unit stores a current request and when the request completes checks to see if a load request is available, and if not then checks to see if a store request is available.

Branching

How branching is handled in our pipeline is novel, but not in the way that benefits performance. In the attempt to get the out-of-order working as fast as possible, we made a major simplification to the branching in our pipeline, by giving every branch instruction a prediction. This is only really helpful for the `jr` instruction. Jumps to registers (`jr`) are easily handled in the baseline implementation because registers are read in the decode stage. However, in our out-of-order pipeline, registers are only read right before execution and cannot always be read because registers are only valid when the corresponding valid bit is set. Therefore, we just treat `jr` as a branch instruction that is not taken, and in the commit stage, when the jump address is fully resolved, we are then able to load the new program counter. It is still the case in our pipeline that `jr` is treated as a not taken prediction to only be resolved in the commit stage; a solution to this would be to use a BTB for branch target prediction, instead of paying the cost for every `jr` instruction. But lucky in the benchmarks `jr` was a rare instruction, with `j` and `b` instructions being far more common.

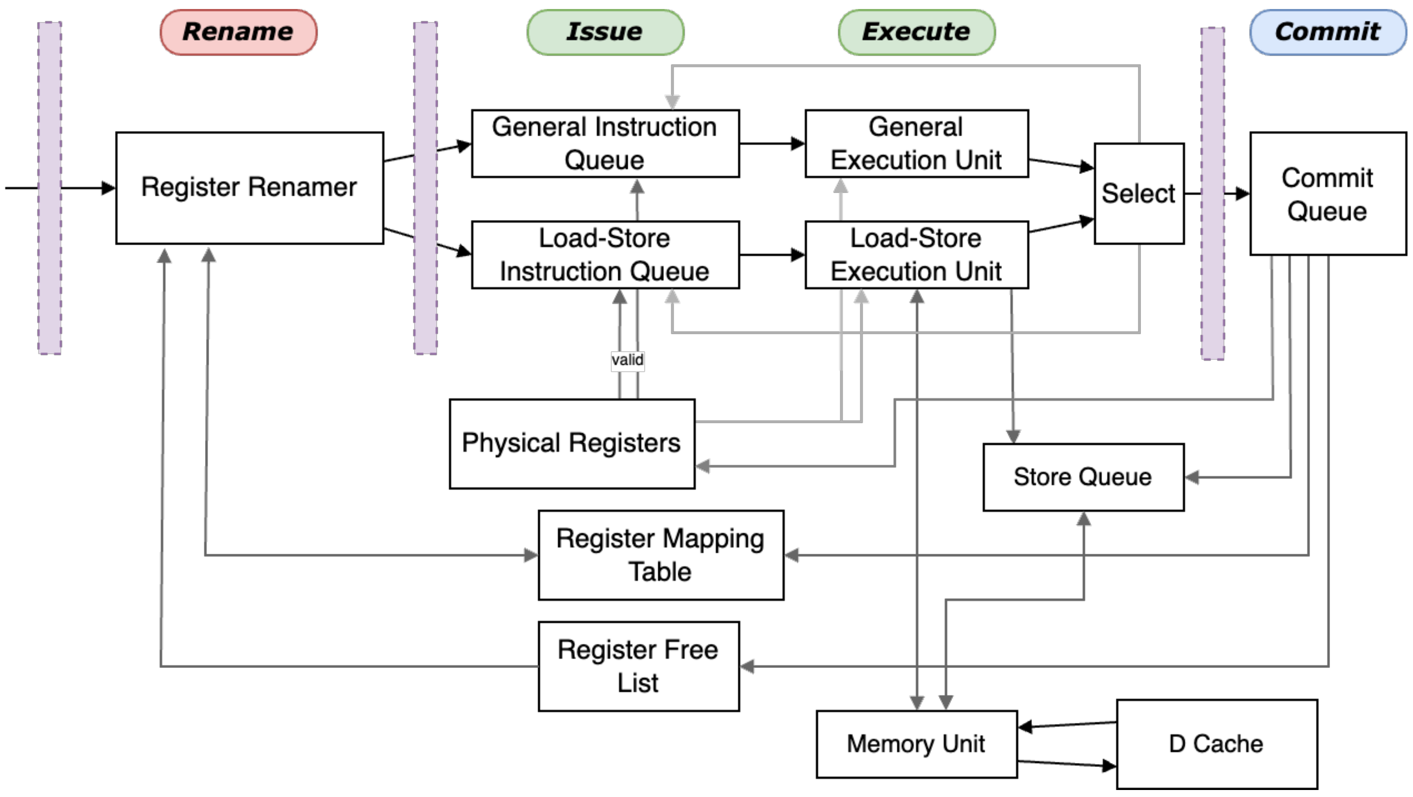


Figure 6: *Out-of-Order simplified pipeline diagram*

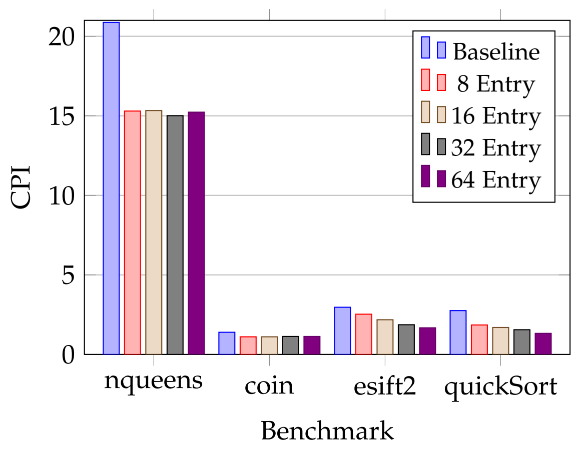


Figure 7: *CPI Comparison for different Instruction Window sizes (with Blocking Data Cache)*

Analysis

For out-of-order, there are many parameters that can be tweaked, but perhaps the one that should have the biggest effect on performance is instruction window size. Figure 7 shows the result of how varying the instruction window size effects CPI performance. As expected, as we made the instruction window bigger, the performance went up. But, not for every benchmark, "nqueens" and "coin" did not see any significant change in performance with differing instruction windows; for these benchmarks, there were other factors that were more impactful. "nqueens" was heavily limited by the instruction cache, having a very large number of cache misses, rarely filling up the instruction window, and "coin" is already nearing the maximum possible performance of 1 CPI.

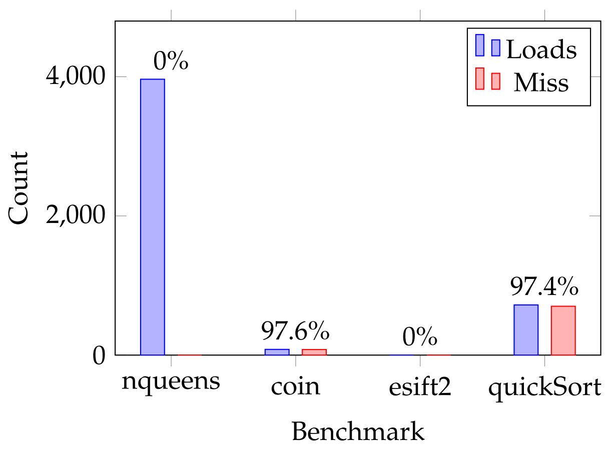


Figure 8: *Percentage of total out-of-order* ***loads*** *(only counting loads that are executed out-of-order), that resulted in a cache* ***miss***

One result that is surprising is how a non-blocking data cache did not have a sizable impact on performance, even with instruction windows as large as 64 entry. With a non-blocking data cache, it can be expected to see a large boost in performance, because the processor can now do multiple load misses in parellel and hide much of the load miss penalty. However, the biggest performance increase that we measured with a non-blocking data cache is a 1.41%

increase in IPC in the benchmark "esift2". This finding is particularly strange because looking at Figure 8, there are zero load-misses happening in parallel for this benchmark, so the performance shouldn't be any different.

We hypothesize that the very small performance increase must have come from how the non-blocking data cache works. In our implementation, there are multiple fetch units from within the data cache that store a request address and associated cache line it fetched from main memory. So if a speculative load happens on a branch that is mispredicted, then the data cache will retain the fetch request for future loads. This speculation does come at a cost, in our data cache there are only 8 fetch units, so doing speculative loads can create a bigger penalty, as future cache misses need to wait for previous misses to finish before giving the request to a fetch unit. This could explain why on "nqueens" there was a 0.3% decrease in IPC. Getting the non-blocking data cache to work with out-of-order has been the biggest challenge for this optimization, as memory instructions have strict dependency requirements that must be meet, therefore one cannot just use the same out-of-order infrastructure for memory operations.

While we were able to get a little closer to load-miss parallelism with our design, it is not fully there. For true load-miss parallelism, modifications must be made to the pipeline to either keep track of memory dependencies to schedule around them, or be speculative and do fixes on the commit stage.

Out-of-order works well with a store buffer, because otherwise the pipeline would have to stall on both loads and stores; with a store buffer, stores can be "free" by just saving the store into the store buffer to be processed at a later time. The store buffer also helps with loads that directly after stores, enabling a complete bypass of the data cache. But we were curious to see how much the store buffer helps with these subsequent loads to the same address. In our measurements, we saw that Store-to-Load forwarding was actually very very rare in practice, and can be seen in Figure 9, where on average less than 0.6% of stores were able to be forwarded directly.

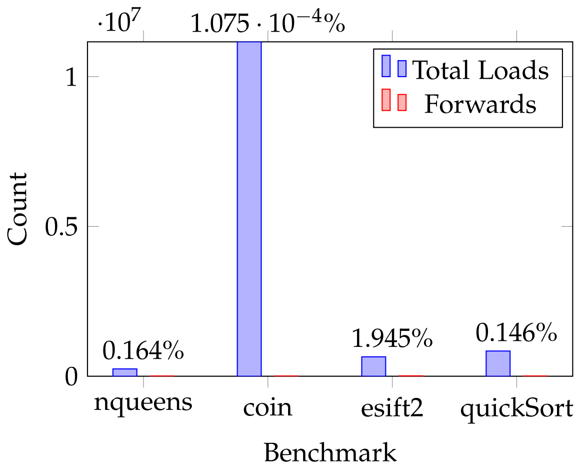


Figure 9: *Percentage of Store-To-Load Forwards by the Store Buffer*

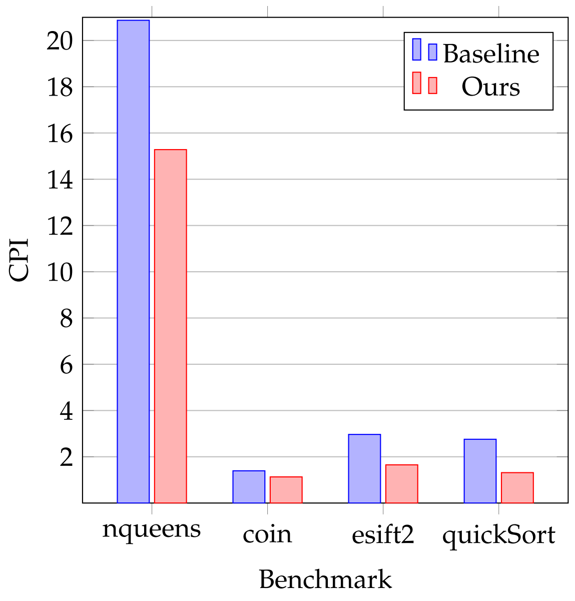


Figure 10: *All optimizations CPI Comparison*

Results

Putting together all the optimization we have working, YAGS predictor, register renaming, and out-of-order execution, we saw a noticeable performance improvement in all benchmarks, which can be seen in Figure 10.

The benchmark that saw the least improvement from our optimizations was "coin", and it is clear why; this benchmark already had performance close to 1 CPI so the only thing that would actually improve the CPI would be to introduce a superscalar design. The benchmarks that saw the greatest improvement were "esift" and "quickSort", and this also makes sense, since they were not limited much by the instruction cache, so they were able to keep the instruction window to near full capacity, taking advantage of the out-of-order pipeline; these two benchmarks being the ones that benefited the most from bigger instruction windows (Figure 7).

Future Improvements

One major limitation with our design of the out-of-order pipeline is the combined issue and execute stage. In real processors, the combinational delay would be too long to combine these two stages without also decreasing the clock rate. In addition, there also needed to be an extra clock cycle for load hits to set the next address correctly, which could have been avoided by placing a pipeline register between these two stages and making the address calculation 2 wide, so that it can calculate the address for both the current and the next memory instruction. The rename stage and the decode stage were also much simpler and could have been combined into one stage. So, if we were to start over, we would combine the decode and rename stages and split up the issue and execute stage into two separate stages. But, we think that generally, our pipeline could have been greatly improved by being more like the MIPS R1000 for memory instruction, using an address queue and special logic to deal with dependencies.

Another limitation with our out-of-order pipeline is with the write back to the register file. Write backs only happen in the commit stage as an instruction gets committed (“graduation"). This introduced a one cycle latency to all instruction that had RAW dependencies, because an instruction in the execute stage would not have the data it needed until the instruction in the commit stage commited one cycle ahead. In an attempt to fix this we added a very simple forward unit in our pipeline to hopefully catch these one cycle delays without rewriting a major part of the pipeline. However, we were not able to get the forward unit to actually work, with an overall 4.15% decrease in IPC, and “quickSort" never completing simulation. The solution to this problem would not easy to implement, just always write to the register file during the execute stage and have a check-pointed register file to restore the state when there is a pipeline flush.

One of the benchmarks that did not see much improvement as “nqueens", due to it’s large number of instruction cache misses. On this benchmark in particular, 0.9997% of the cache misses result in an eviction from cache. This makes “nqueens" a good candidate for cache optimizations; so for future improvements, trying out a victim cache would be a good place to start for optimizing the instruction cache for this benchmark.

Conclusion

Overall, this project was a great experience, turning the high level concepts from papers into real hardware design. Some things that we could have done better are:

- Spending more time at the beginning developing tools (especially for data collection)
- Collecting more data along the way to aid development
- Doing more research outside the papers given during the course

The biggest one is collecting data while developing our optimizations, Not only would this have helped better our understanding of these optimizations, but this would have also helped lead the way in what optimizations we should be trying next to get maximum performance from the benchmarks.

One challenge we faced was the time it took for the simulation to compile, which significantly reduced the speed at which we could iterate.

References

[1] A.N. Eden. “The YAGS branch prediction scheme”. In: *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture* (2002). DOI: 10.1109/MICRO.1998.742770.

[2] Hans Vandierendonck. “Improving the YAGS Predictor”. In: *ELIS Technical Report 2005-003* (2005).

[3] K.C. Yeager. “The Mips R10000 superscalar microprocessor”. In: *IEEE Micro* 16.2 (1996), pp. 28–41. DOI: 10.1109/40.491460.