Exploring Optimizations to 5 Stage In-Order Processor

CHI CHOW
University of California, San Diego, cschow@ucsd.edu
CONNOR KUCZYNSKI
University of California, San Diego, ckuczyns@ucsd.edu

ABSTRACT

The baseline processor design consists of 5 stages: Instruction Fetch, Instruction Decode, Execute, Memory Access, and Writeback using the MIPS ISA. Pipelining improves performance over a single cycle, single instruction processor as it allows for multiple parts of instructions to be executed in parallel. We focused our research on finding ways of masking latency from memory operations. Specifically, we targeted instruction cache misses through prefetching, data cache misses through set-dueling and run-ahead execution. An accurate branch predictor also improves performance by avoiding unnecessary instruction execution. In this paper, we overview our implementation for each of the four optimizations. We then examine the results of each optimization individually over the baseline and then finally review avenue's of further optimization. Overall, we saw on average an IPC performance increase over the baseline by 5%-15%.

CCS CONCEPTS • Advanced Processor Design • Computer Architecture • Memory optimization

Additional Keywords and Phrases: Branch prediction, IPC, Prefetching, Runahead execution

1 INTRODUCTION

1.1 Branch Predictor Motivation

The ideal processor (assuming no superscalar execution) executes 1 instruction per cycle in the steady state. For our specific baseline, this means that each stage executes one instruction all in parallel. Furthermore, this ideal processor has perfect branch prediction so that no stalls occur. The reason why accurate branch prediction is important is because, for each stage of the pipeline to do useful work, the fetch unit must fetch the next instruction before the execute stage resolves the branch. Branch prediction occurs in the decode stage and the following instruction after each branch instruction is a NOP according to the MIPS ISA. If a branch is mispredicted, then the pipeline must be flushed, as the processor is executing the incorrect instruction in the pipeline. This clearly leads to losses of instruction per cycle (IPC). The alternative to branch prediction is to speculatively execute both the TAKEN and NOT TAKEN result of the branch. More research is needed to conclude what is the better strategy in our use case. Executing the two results in parallel would require changing the pipeline and adding new architecture and complexity compared to a branch predictor. The deeper the pipeline, the more costly mispredictions are on performance. This is due to having more instruction in flight so more work has to be flushed as a result of a misprediction. Therefore, an accurate branch predictor will make an even larger performance difference on modern processors compared to the standard 5 stage pipeline.

1.2 Instruction Prefetcher Motivation

The baseline processor struggles (particularly with the NQueens) with instruction cache misses. In order to understand the benefits of prefetching, we need to first understand how the instruction cache services a miss. The instruction cache has the highest priority ID in the baseline so it will send the request to the memory arbiter. Then the entire pipeline will stall for around 20 cycles until the data is retrieved and loaded into the cache. Details of how exactly this works will be examined in the 3.1.2 Branch Predictor Design. These cache misses are detrimental to performance because of how only 1 cache line is requested per cache miss. Several of the benchmarks do not stress even the benchmark instruction cache very well due to the large size of the cache itself. Increasing associativity leads to large parts of the program fitting into the instruction cache. Most of the misses therefore are compulsory in nature which make up a small percentage of the cache misses. The pattern we see at the beginning of execution is a cache miss occurring, normal execution of four instructions (this is because a cache line contains 4 words) and then another miss leading to another stalling of 20 cycles. A prefetcher would really help at solving this problem as it will serve cache misses in parallel with the instruction cache. If the prefetcher requests only one cache line per miss then we effectively improved our throughput of cachelines per miss from 1 to 2. Additionally, the instruction cache will have priority over the prefetcher as that is on the critical path of execution. Overall, the main motivation of the prefetcher is that it enables better throughput of the memory subsystem. We really want the instruction cache to be able to service all the requests from the fetch unit without any stalling. Ideally, the cache is large enough to fit the entire program in so then the problem becomes trivial. However, instruction cache typically sits on-chip, meaning they have really low access latency compared to going to the large off-chip memory. Effectively, we face an area constraint where we are limited to a specific cache size. The role of the prefetcher is to speculate on future cache requests when a miss occurs. The reason we can make these requests is because the memory subsystem is underutilized. Unless, misses from the instruction or data caches are being serviced, it is ideal hardware. Prefetching takes advantage of this by making memory requests when the instruction cache and data cache do not make a request this cycle. Furthermore, instruction requests are fairly predictable with a fixed instruction size so it is possible to have a high probability requesting relevant data.

1.3 Set-Dueling Motivation

The baseline processor uses a 4KB, 2-way set associative data cache with LRU. This configuration performs decently well for the benchmarks we have. However, implementing DIP (dynamic insertion policy) via set dueling will provide us the ability to use other cache insertion policies when LRU is not the best option. In our implementation, we used BIP (bimodal insertion policy), to help with cache misses in thrashing workloads. Implementing DIP serves as a great proof of concept in terms of combining two or more cache insertion policies, which allows the processor to accommodate and perform well with different types of workloads.

1.3 Runahead Execution Motivation

As an in-order processor with only 1 read/write port on the instruction cache and data cache, the baseline processor spends many cycles stalling for data to be loaded from memory. Cache misses must be serviced sequentially in the baseline processor. Runahead execution allows us to look into future instructions while stalling for a load, to find loads that can be run in parallel, and prefetch data into the data cache. With runahead execution, cache misses can be serviced in parallel, decreasing the need for stalling for memory load operations. One of the major benefits of runahead execution is that it does not require too much extra physical resources to run, as opposed to out-of-order execution and superscalar execution.

2 RELATED WORK

1. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers (1990)

The paper suggests three methods of improving cache performance: miss caching, victim caching, and stream buffers. Miss cache enables the performance benefits of being directly mapped cache while also having the benefits of an associative cache. Victim caching is a technique where evicted cache lines from a cache miss are stored in a small fully-associative cache that reduces conflict misses that occur in a directly mapped cache. The technique we use in our processor is stream buffers which prefetches data to remove capacity and compulsory misses. The paper uses a FIFO queue and checks the head entry for a hit, while we use a cache as the underlying structure to improve hit rate.

2. Dynamic Branch Prediction with Perceptrons (2001)

The paper suggests a branch predictor using perceptrons. The perceptron branch predictor leverages the perceptron as the method for learning correlations between branch outcomes from global history and the current branch. The perceptron can be described as follows:

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

In the use case of branch prediction, each x represents a branch outcome in the global history with the value 1 if taken and -1 if not taken. If y is greater than 0, then the branch prediction is taken, else it's not taken. There exists a table of perceptrons that use the lower bits of the PC. The number of PC bits used is a hyperparameter determining the number of perceptrons used. Overall, the perceptron enables looking at a much wider range of global history compared to GShare. Patterns that correlate to a wide range of global history will see benefits over GShare.

3. Set-Dueling-Controller Adaptive Insertion for High-Performance Caching (2008)

The paper discusses benefits of the dynamic insertion policy in caches. DIP estimates the number of cache misses caused by two competing insertion policies, and chooses the one that causes the fewest cache misses. Even though LRU is a good cache insertion policy for many use cases, it leads to thrashing in memory intensive workloads. The paper proposes to combine LRU and BIP (bimodal insertion policy), which handles memory intensive workloads better by keeping some of the working set in memory. DIP is implemented via set dueling, using a PSEL counter and threshold to determine which cache insertion policy to use. The main advantage to using set dueling is that it requires very little hardware overhead (15 bits and very little additional logic).

4. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors (2003)

The paper proposes runahead execution as an alternative to large instruction windows for out-of-order processors, showing that a processor with a smaller instruction window and runahead execution, can achieve similar performance to a processor with a larger instruction window but no runahead execution. The main idea behind runahead execution is that independent load/store instructions are serviced in parallel, and provide very accurate prefetches into the caches. The rest of the paper discusses in detail about the specific implementation of runahead execution in an out-of-order processor. This includes hardware requirements, as well as runahead mode, which processes instructions differently compared to normal mode.

3 TECHNICAL MATERIAL

3.1 METHODS

3.2.1 Baseline System Design

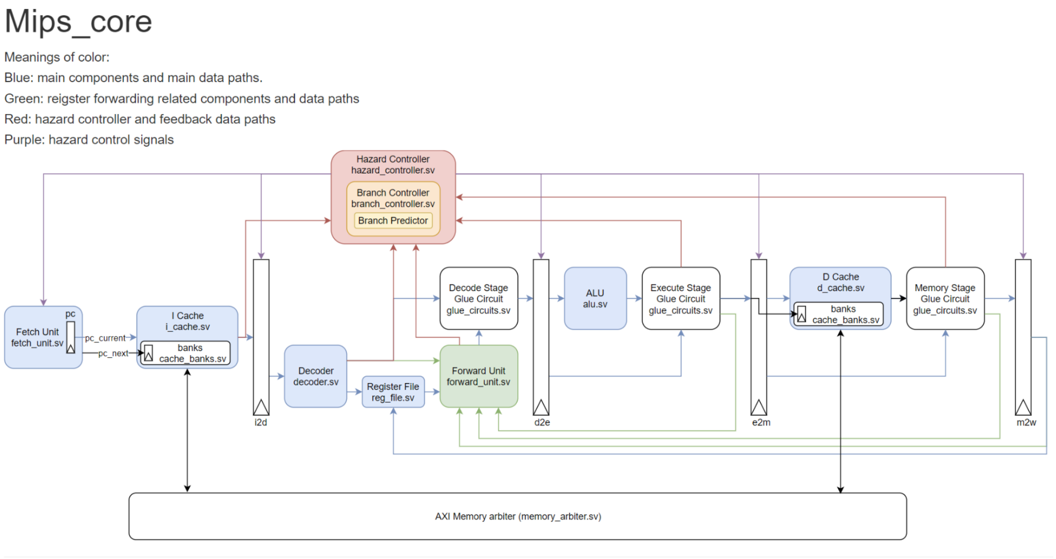


Figure 1: The CSE 148 baseline 5 stage processor design adapted from Hennessy and Patterson's “Computer Organization and Design”

3.2.2 Branch Predictor Design

First we will talk about why the perceptron branch predictor is more desirable than something like GShare. The major reason why the perceptron is able to use a large global history is due to the amount of memory required grows nlog(n) with respect to the amount of history. First there is a linear relationship between the history length and the number of bits required to represent a single weight. For a single perceptron, the number bits needed to represent the weight is equal to the floor of (log base 2 of 1.93 times the history length + 14) plus 1 for the sign bit [1]. A perceptron contains a global history length number of weights. Therefore, the total number of bits required to represent a perceptron grows nlog(n) with respect to the global history length. The last major factor on the memory is the number of entries in the perceptron table. Finally, we need to add the bits required for the global history register. GShare grows exponentially with the length of global history. This is due to the Pattern History Table (PHT) being indexed by an XOR of the lower PC bits and the global history register. To map everything global history to a single entry in PHT requires $2^{\text{global history}}$ bits.

However, there are some patterns that GShare learns better than perceptron like linearly inseparable functions[1]. Therefore, we decided to combine these two ideas and see if we can combine the strengths of perceptron like long pattern recognition while also using GShare that can better learn linearly inseparable functions. The final design acts like a tournament predictor where, when the perceptron has low confidence in its prediction, it uses GShare to predict and when there is a high level of confidence perceptron is used. Several values were tried and the threshold for confidence that gave best average performance over the benchmarks was around 30. We also experimented with using a static predictor (like the branch predictor always_backward_taken_forward_not_take) to improve warmup but this did not have a noticeable effect.

The main components of the perceptron are the global history register (GHR) and the perceptron table (with 64 elements). One problem that needed to be addressed was that we need to speculate what the new GHR value should be before we know the result of the branch. The GHR needs to be repaired if the feedback prediction does not equal the feedback outcome. Each time the feedback signal is high, the entry in the perceptron table used for prediction is updated with the new weights. The low bits of the PC are used to index into the perceptron table. Then the dot product is performed between that specific perceptron and the global history. If the value is positive the prediction is TAKEN else it's NOT TAKEN. The absolute value of the result is the confidence used to determine between the perceptron and GShare predictors.

Training is performed by first checking if the prediction is incorrect or if the max amount training has already been performed to prevent overfitting. Weights that agree with outcome and GHR are strengthened while weights that disagree are weakened. This is where the built in confidence for the perceptron comes from as weights near 0 did not correlate well with the GHR.

3.2.3 Instruction Cache Prefetcher Design

ICACHE:

The key problem we want to avoid when designing the prefetcher is slowing down the critical path. We want to make the common case; therefore, all prefetcher requests have lowest priority among the instruction cache and data cache. Additionally, if data does hit in the prefetcher, we do not want to wait for the data to be available in the instruction cache to use it. This is why in our design, if a miss occurs in the instruction cache but hits in the instruction cache, we service the request first and then load the data from the prefetcher into the instruction cache after. Furthermore, the instruction cache can still service requests while the data from the prefetcher is transferred to the instruction cache. The instruction cache has 4 states: STATE_READY, STATE_REFILL_REQUEST, STATE_REFILL_FROM_PREFETCHER, and STATE_REFILL_DATA. The instruction tag and index are sent a cycle early allowing for the hit signal and data to be ready at the positive edge of the next cycle. Reading from the instruction cache is decoupled from writing. The registers storing the current missed tag and index are updated in the ready state on a miss. The instruction cache also uses 2 way associativity with an LRU policy to select the least recently used way. If a cache miss occurs in ready and there is a miss in the prefetcher, then the state updates to refill request the next cycle, this talks to memory arbiter directly (or in our case indirectly through the directory) and waits for a ready signal to then transition to the refill data state. If however, there is a hit in the prefetcher, then the state refill request will be fulfilled by the prefetcher and transition to the refill from the prefetcher state. Cache lines are written serially one word per cycle which dictate how the refill requests work. Additionally, the reason why the data banks can only write one word per cycle is because of this memory limitation.

PREFETCHER:

The underlying structure of the prefetcher is an 8 entry cache that replaces 4 entries on every instruction cache miss. The reason for 8 entries versus only 4 is to increase the size of the window of the number of local instructions we can see. The prefetcher effectively is a small cache that stores values that are near the current program counter. It leverages the locality of programs to reduce the number of cache misses. In the ready state, and if there is an instruction miss and prefetcher miss, then the next four cache lines are requested while the instruction cache requests the line missed (the current PC). Each cache request acts as its own state machine in parallel with state ready, state refill request, and state refill data. Each request has its own ID (2-5 inclusive). The reason why the ID is needed is so that each state machine knows if the data coming back belongs to them. All four state machines can be waiting for data at the same time. They are waiting on the valid signal from their respective directory before filling the prefetcher.

DIRECTORY:

The directory has two main jobs. One is to direct traffic for the memory arbiter, the instruction cache and the prefetcher. On a valid request, the address is registered and we also record what wire we should be reading for the requested data. The second is to prevent redundant memory requests by registering requests from the two agents, the instruction cache and the prefetcher. In our design, there are four directory modules. This is because we know that the prefetcher by design will not make the same memory requests. Therefore, we only need to handle traffic between the instruction cache and each individual prefetcher request (4 in total). The benefit we see from the directory is reduced number of memory requests as well as reducing the latency of a miss in the instruction cache, if the same outgoing request has already been made (but not yet received) from the prefetcher. When the data comes back from the arbiter, the entry is freed from the table.

3.2.4 Data Cache Set Dueling Design

Data cache set dueling is implemented with a few additions to the baseline data cache. In the baseline, the data cache is a 4KB, 2-way set associative cache, using the LRU insertion policy. Our cache set dueling implementation uses DIP (dynamic insertion policy) to combine LRU and BIP (bimodal insertion policy), which ideally reduces data cache misses in thrashing workloads.

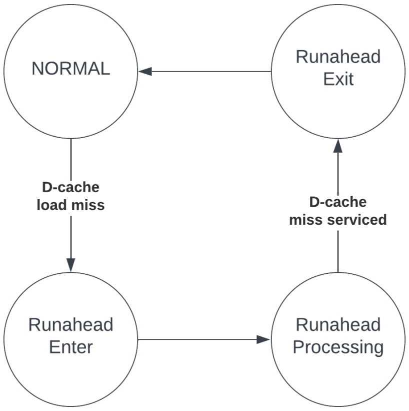
To choose between the two insertion policies, we dedicate ¼ of sets to always using LRU, and ¼ of sets to always use BIP. The follower sets will use the cache insertion policy that causes fewer cache misses. This is done with a 10-bit PSEL counter as mentioned in the referenced work. Whenever a cache miss happens in a dedicated LRU/BIP set, the PSEL counter is incremented if it is a dedicated LRU set, and decremented if it is a dedicated BIP set. The follower sets use LRU if the PSEL counter is less than half of its maximum value, and they use BIP if the PSEL counter is more than half of its maximum value. The PSEL counter starts at 0, which means that LRU is prioritized.

We experimented with different PSEL counter widths, as well as different cache policy thresholds to switch between LRU and BIP. However, we found no significant difference in performance when we changed these parameters. We observed that increasing the PSEL counter width simply means that LRU/BIP is more prioritized, and DIP biases towards either cache insertion policy quickly regardless of the threshold. Therefore, we decided to keep the settings recommended from the referenced work.

3.2.5 Runahead Execution Design

Our implementation of runahead execution is quite different from what was proposed in the referenced work. The paper mainly discusses the implementation of runahead execution in an out-of-order processor, and utilizing existing out-of-order processor resources such as the instruction window, scheduling window, and store buffer. However, since we are implementing runahead execution in an in-order processor, we have to make some modifications. The following shows implementation requirements, and how we planned to apply them to our own design.

- Non-blocking cache
    - Being able to service cache misses in parallel is crucial to obtaining the main benefits of runahead execution. This means that a future load from memory can happen while a previous load is stalling.
    - In our implementation, an additional read port is added to the data cache. During normal processing, only the original read/write port is used. This additional read port is only used during runahead mode. This acts as a proof of concept that some new data can be prefetched into the data cache during runahead mode.
- Entering/exiting runahead mode
    - To manage the progress of normal and runahead mode, we use different states (NORMAL, RUNAHEAD_ENTER, RUNAHEAD_PROCESSING, RUNAHEAD_EXIT). These states allow different parts of the processor to behave differently in runahead mode. We find that using a single bit to distinguish between normal/runahead mode (as proposed from the paper) was difficult in terms of checkpointing and propagating invalid values.

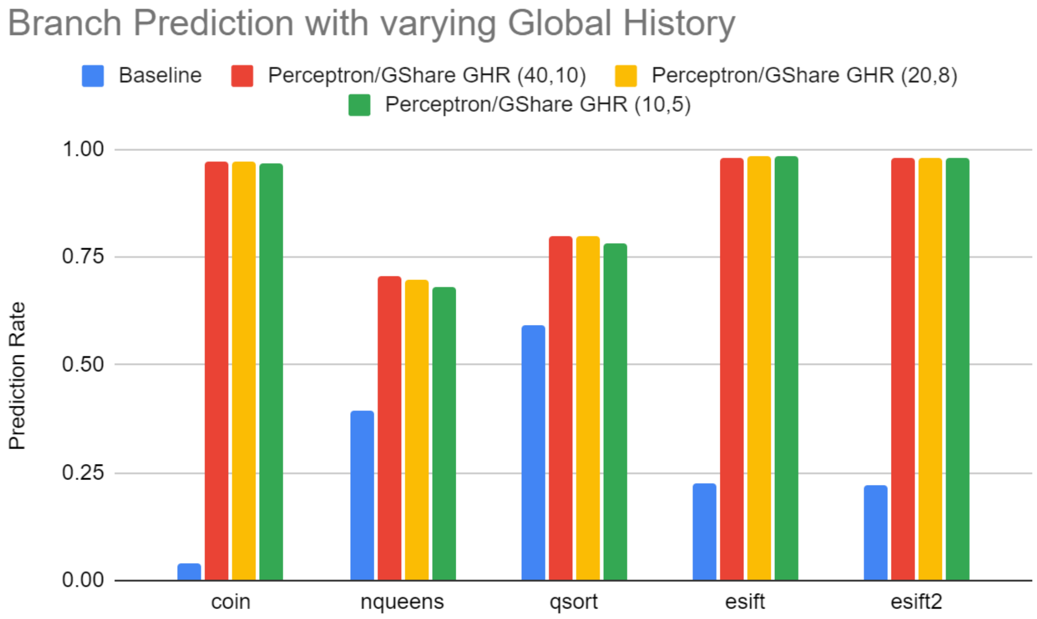


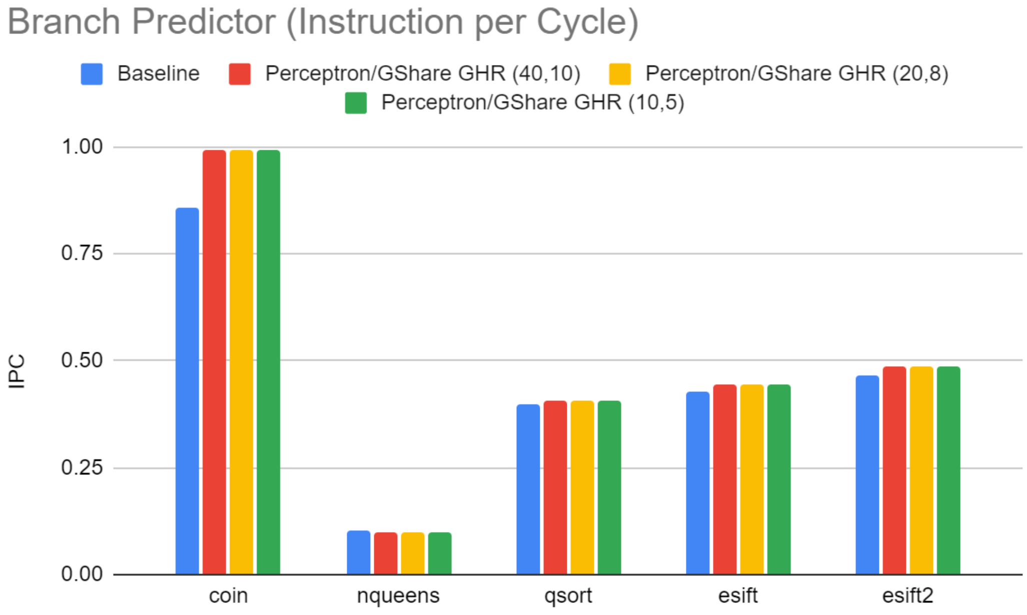
    - NORMAL: normal processing, no changes to the baseline behavior; enter runahead mode when there is a data cache load miss.
    - RUNAHEAD_ENTER: checkpoint register file and PC, and then enter the RUNAHEAD_PROCESSING state.
    - RUNAHEAD_PROCESSING: special handling of instructions (loads, stores, branches), and propagating invalid values from inputs. When the load that caused entry to runahead mode is served, enter the RUNAHEAD_EXIT state.

    - RUNAHEAD_EXIT: restore register file and PC checkpoint, and return to normal processing.
- Register file checkpoint, and recovery
    - In our implementation, a single register file checkpoint is kept during normal mode. Any changes to the register file in normal mode are also made to the checkpoint. During runahead mode, the checkpoint register file is not updated.
    - When the processor exits runahead mode, the checkpoint is restored.
- Propagation of invalid bits
    - An invalid bit is attached to every register in the register file. These invalid bits are only used during runahead mode, and are reset when exiting runahead mode. if any instruction sources a register with an invalid value, the resulting value will be invalid as well.
    - The first invalid value is introduced when there is a data cache load miss in normal mode. A writeback sets the invalid bit in the register file, and runahead execution begins.
    - Since the baseline processor supports register forwarding, invalid bits that result from the execution, memory, and write back stage are propagated through the forward unit as well.
- Runahead cache
    - The runahead cache is used to store the results of runahead stores, instead of dropping store instructions in runahead mode. This results in fewer invalid values for future instructions in runahead mode that depend on these store instructions.
    - We reused the data cache implementation for the runahead cache, since there are very few differences between the runahead cache and the data cache. For the runahead cache, no stores are written to memory, and there is no write back.
    - INV and STO bits are attached to each byte in the runahead cache, to help with the propagation of invalid values.

3.2 RESULTS

3.2.1 Branch Predictor



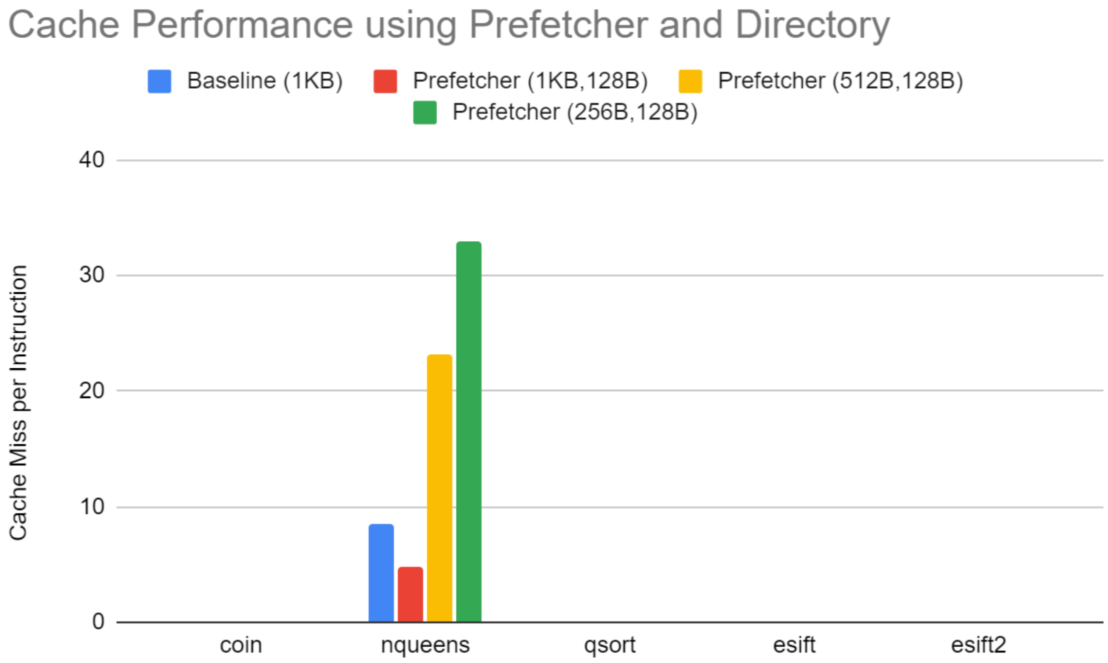


The baseline predictor uses a static predictor ALWAYS_NOT_TAKEN. We can see that our branch predictor outperforms the baseline on average by 625.68%. In particular coin, esift, and esift2 greatly outperform the baseline with each having around 98% accuracy in branch prediction. It is interesting however that there is not a significant increase in IPC. On average, there is an increase of 4.68% in IPC over all the benchmarks.

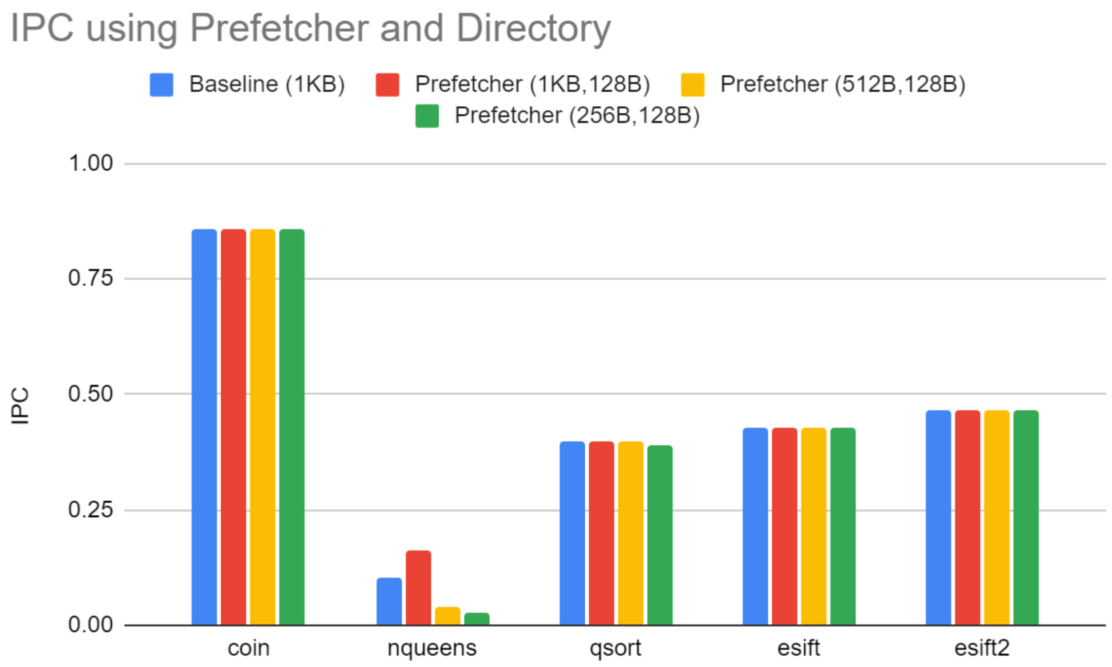
For exploration, we experimented with different GHR lengths and how they affected performance. The most significant difference seen is in NQueens where branch prediction accuracy drops from 70% to 67% looking at smaller history. This was somewhat unexpected seeing how little the performance dropped compared to the number of bits of history recorded. One reason for this is because the benchmarks themselves are not super complex, meaning that the patterns that correlate well are all close to each other in time. This explains why even though we are looking at more history, it doesn't necessarily lead to any meaningful performance increase.

One interesting observation is that NQueens is the shortest benchmark. This means that warm up time may explain why the branch predictor is not able to predict accurately compared to the much longer benchmarks like coin. We tried isolating this problem by ignoring the first 50% of the program instructions and did not see any significant differences in NQueens, meaning that the problem likely isn't warm up time. Benchmarks that stressed more complex longer history patterns would be interesting to study in the future.

3.2.2 Instruction Cache Prefetcher



Cache Miss (ICache, Prefetcher cache)	Baseline (1KB)	Prefetcher (1KB,128B)	Prefetcher (512B,128B)	Prefetcher (256B,128B)
coin	7.92E-05	6.34E-05	6.34E-05	0.000186504
nqueens	8.47918	4.87534	23.1592	32.9571
qsort	0.00112297	0.000861798	0.000861798	0.000209398
esift	0.000259038	0.000209398	0.000209398	0.000209398
esift2	0.000673684	0.000539391	0.000539391	0.000539391



IPC (ICache, Prefetcher cache)	Baseline (1KB)	Prefetcher (1KB,128B)	Prefetcher (512B,128B)	Prefetcher (256B,128B)
coin	0.857176	0.857187	0.857187	0.857102
nqueens	0.102896	0.163546	0.0409914	0.0292514
qsort	0.399384	0.399305	0.399305	0.389045
esift	0.427528	0.427536	0.427536	0.427536
esift2	0.464629	0.464654	0.464654	0.464654

Overall we see an average decrease in cache misses per instruction of 25% over the baseline. We measured cache misses using the built in stat given in the baseline design and then divided by the number of instructions. This gives us a better understanding of the efficiency of the instruction cache compared to just the absolute number of misses. Interestingly, we see that the working set for qsort, esift, and esift2 is very small as most of the misses are entirely compulsory. There was an increase in performance as we expected because prefetching does help reduce the number of compulsory misses. One interesting benchmark is NQueens where we see more capacity and conflict misses. The improvement in NQueens shows that the prefetcher also helps reduce conflict misses. However, the prefetcher is limited in that it assumes a stride of just one instruction. We tried using a cache instead of FIFO queue unlike in the paper to be able to handle more strides while only allocating one stream buffer. Because the working set is so small for all the benchmarks besides NQueens, we only see significant IPC improvement on NQueens of 58.94% over the baseline using the same instruction cache size. The increased associativity in the instruction cache played a major role in reducing conflict misses.

Prefetcher Only

Cache Miss	Baseline (1KB)	Prefetcher (1KB,128B)	Prefetcher (512B,128B)	Prefetcher (256B,128B)
coin	7.92E-05	6.82E-05	6.82E-05	0.000190335
nqueens	8.47918	5.21332	25.2103	35.2537
qsort	0.00112297	0.000979339	0.000979339	0.00087271
esift	0.000259038	0.000218597	0.000218597	0.000218597
esift2	0.000673684	0.000600254	0.000600254	0.000600254

IPC	Baseline (1KB)	Prefetcher (1KB,128B)	Prefetcher (512B,128B)	Prefetcher (256B,128B)
coin	0.857176	0.857184	0.857184	0.857102
nqueens	0.102896	0.161421	0.398124	0.0274117
qsort	0.399384	0.399295	0.399295	0.423436
esift	0.427528	0.427368	0.427368	0.427368
esift2	0.464629	0.464511	0.464511	0.464511

For exploration, we did two things. One, we tried reducing the size of the instruction cache to see if that would yield more capacity and conflict misses. Two, we removed the directory module and connected the instruction cache and prefetcher directly with the memory arbiter. For (1), there were no major changes meaning, the working set must be even smaller than a 256B cache. We did see a predictable decrease in performance for NQueens as the cache was reduced in size. For (2), we want to prove that the directory actually made a noticeable difference in performance. For Prefetcher (1KB, 128B) NQueens, increased the number of cache misses per instruction by 6.93%. We theorize the reason for this increase is that we no longer have the case where we can reduce the latency of a miss using the prefetcher. The problem that occurs is that the instruction cache will still wait the entire latency for a miss to be serviced by memory even though the prefetcher made the request earlier. Further study will need to be done in reducing the cache size and observing the other benchmarks.

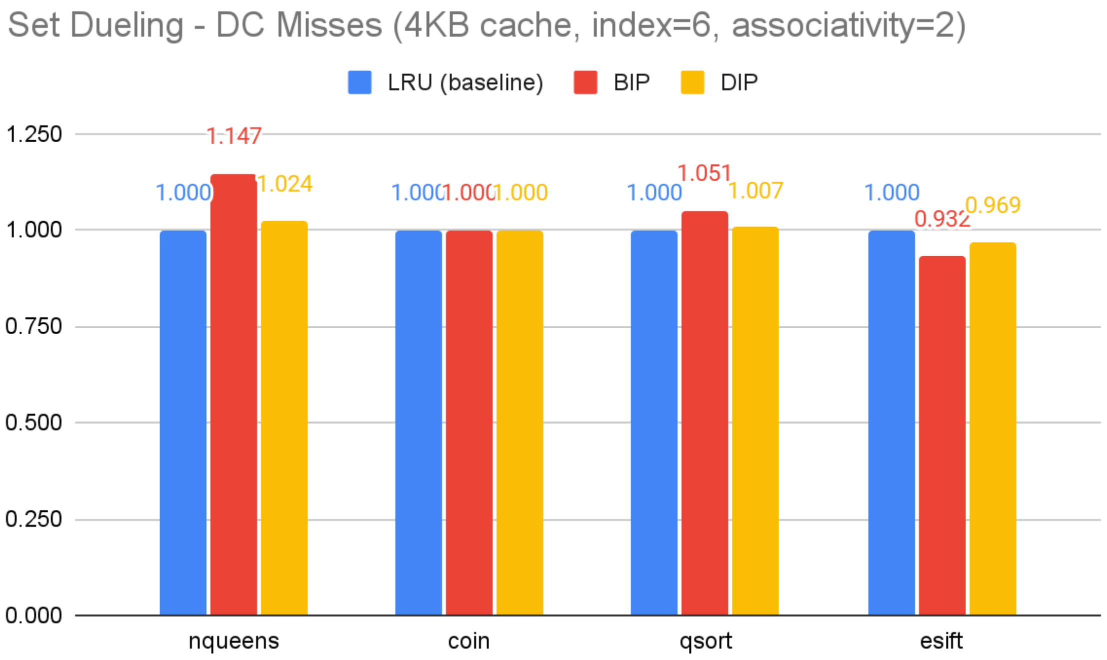
3.2.3 Data Cache Set Dueling

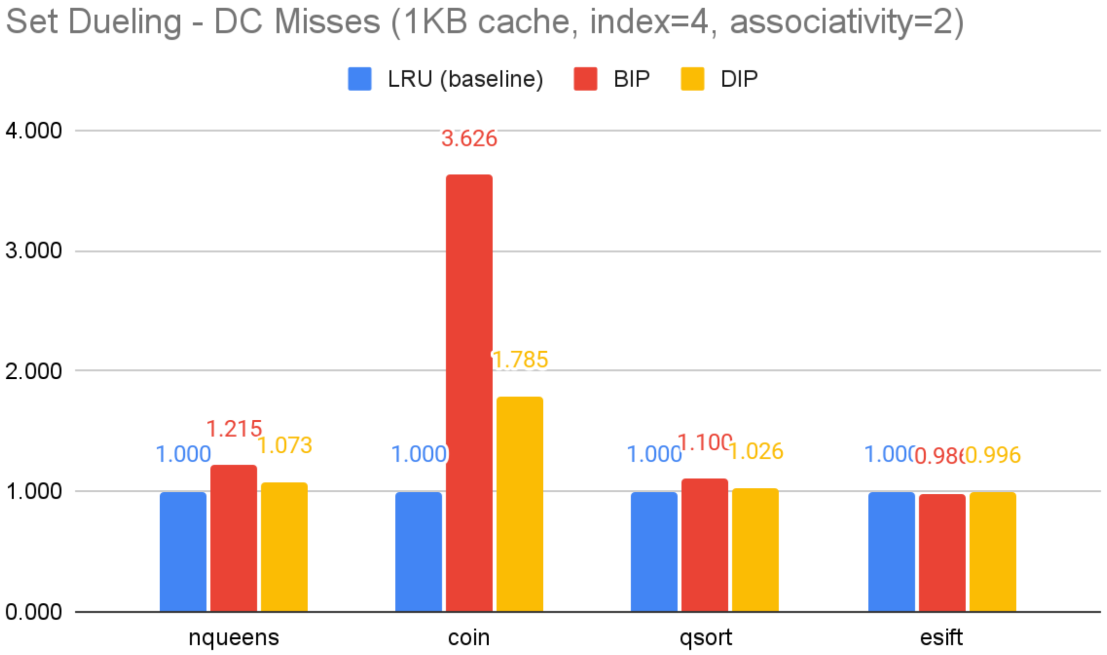
Before implementing data cache set dueling, we expected it to have little benefit with the existing benchmarks and data cache size. This is because none of the benchmarks produces a thrashing workload for the baseline data cache.

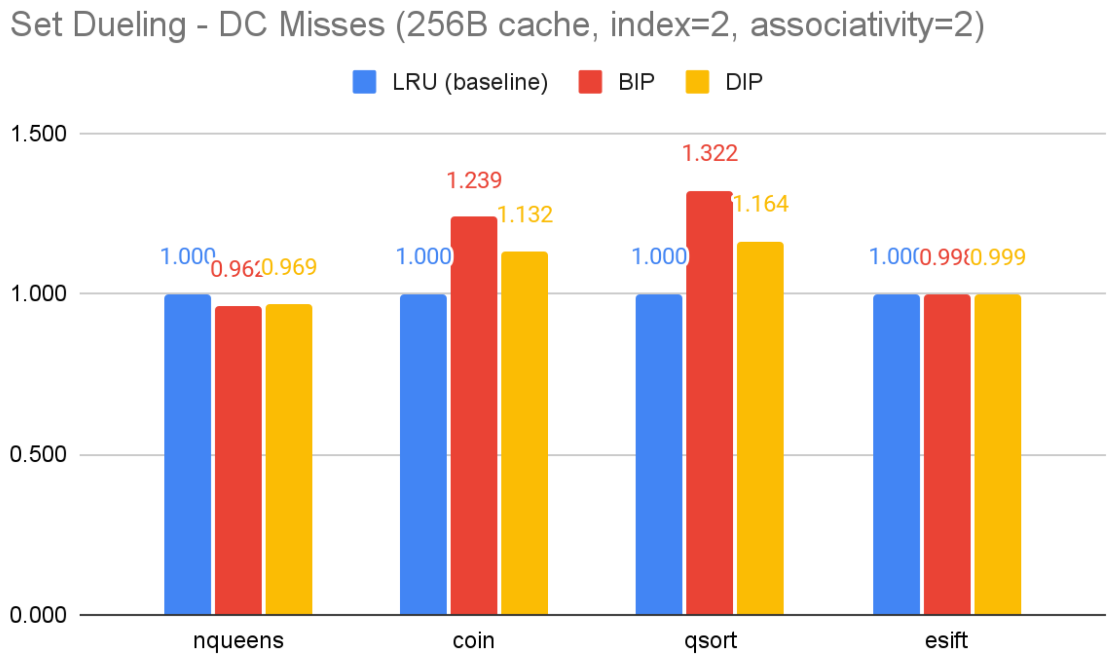
Therefore, The set dueling implementation is tested on the four benchmarks with varying cache sizes (4KB, 1KB, 256B). For each cache size, the LRU (baseline), BIP, and DIP insertion policies are individually tested in order to visualize the effect of set dueling. The number of data cache misses are recorded, and the results are as follows:

Cache Miss	4KB, index=6, LRU	4KB, index=6, BIP	4KB, index=6, DIP	1KB, index=4, LRU	1KB, index=4, BIP	1KB, index=4, DIP	256B, index=2, LRU	256B, index=2, BIP	256B, index=2, DIP
nqueens	10053	11535	10294	79596	96678	85375	1.18E+07	1.13E+07	1.14E+07
coin	3277	3277	3277	5637	20441	10062	2.22E+08	2.75E+08	2.51E+08
qsort	4.90E+06	5.15E+06	4.93E+06	6.76E+06	7.43E+06	6.93E+06	9.43E+06	1.25E+07	1.10E+07
esift	1.29E+07	1.20E+07	1.25E+07	1.29E+07	1.28E+07	1.29E+07	1.29E+07	1.29E+07	1.29E+07

To better visualize the effects of different cache insertion policies, we can refer to the following figures. The cache miss data is normalized to the LRU (baseline) insertion policy, which means that the value of each column represents the relative number of cache misses for each insertion policy. More specifically, for a value of >1, the insertion policy has more cache misses than the baseline, and for a value of <1, the insertion policy has fewer cache misses than the baseline.


Set Dueling - DC Misses (4KB cache, index=6, associativity=2)


Set Dueling - DC Misses (1KB cache, index=4, associativity=2)



The results show that LRU is generally a better insertion policy compared to BIP across different cache sizes. It also goes against the expectation that BIP causes fewer cache misses at smaller cache sizes. BIP outperforms LRU only for the esift benchmark, and one specific nqueens test. Comparing the LRU and BIP results to DIP, we can see that the performance of DIP is bounded by the performance of LRU and BIP. Using DIP always results in more cache misses than the better policy, and fewer cache misses than the worse policy.

We also observed that DIP often does not use BIP at larger cache sizes (4KB, 1KB), since there are fewer cache misses. However, we still suffer a performance penalty because ¼ of the cache sets are dedicated to use BIP, which is an inferior policy compared to LRU in this case.

3.2.4 Runahead Execution

Unfortunately, the runahead execution implementation does not work correctly at the time of writing…

4. CONCLUSION

Overall, we were able to get an IPC improvement of 5%-15% of all three optimizations. However, the average IPC improvement is skewed by the nqueens benchmark, as shown below. We saw that the branch predictor, prefetcher, and set-dueling all targeted orthogonal aspects of the processor. Future work will involve finishing implementing runahead execution to further improve data cache performance as well as experimentation with multi-stream buffers of varying stride lengths.

IPC	baseline	optimized	Improvement
nqueens	0.102896	0.834916	711.42%
coin	0.857176	0.991874	15.71%
qsort	0.399384	0.405894	1.63%
esift	0.427528	0.451727	5.66%
esift2	0.464629	0.491899	5.87%
Average			148.06%