Speculative Optimizations on Superscalar

Shuusei Yoshida, Chad Wall
Department of Computer Science
and Engineering
University of California - San Diego
La Jolla, CA 92093
{s1yoshid, cmwall}@ucsd.edu

***Abstract*—For the MIPS R2000 processor, we have implemented super scaling, the Yet Another Global Scheme (YAGS) branch predictor [1], load value prediction [2], and an i-cache file stream buffer [3]. The reasoning behind these choices was to observe how branch prediction, value prediction, and a file stream buffer would affect the CPI on a superscalar machine. Specifically, when upgrading to super scaling, the two big issues which generally rise up are an increase in pipeline hazards and an increase in demand for instructions. Our hypothesis was that the YAGS branch predictor would help decrease the amount of branch hazards, load value prediction would help decrease the amount of LW hazards, and a file stream buffer for the i-cache would help increase the rate at which instructions were fetched. What we firstly found when implementing the superscalar was that super scaling reduced the CPI by a considerable amount, but only when the i-cache was large enough. Secondly, when implementing the optimizations, we found that YAGS considerably decreased the CPI all around, load value prediction marginally decreased CPI (and in fact marginally increased for some benchmarks), and the file stream buffer slightly increased CPI.**

Keywords— Superscalar, YAGS Branch Prediction, Load Value Prediction, Stream Buffer

I. Introduction

The H&P 5-stage pipeline design is known to all in the field of CPU design. It is a very straightforward design to learn from and understand how pipelining works. However, it is more of a baseline design and does not utilize any features that would allow it to improve performance. We were challenged to try and optimize the 5-stage pipeline design by implementing various hardware features such as superscaling, out-of-order executions, etc. and to test them on test programs provided for us.

The first and most significant optimization we implemented on the baseline design is 2-way superscaling. The goal was to achieve a CPI as close to 0.5 as close as possible, as that was the theoretical max CPI for a 2-way super scaling machine. The main issues faced with the superscalar design are an increase in pipeline hazards and an increase of demand for reading instructions and data. The optimizations introduced to combat those issues are hardware prefetching, load value prediction, and a better branch predictor.

In this paper, we take a look at the hardware features we implemented and their respective performance results on various test programs provided by the course. Because we were most interested in observing the interactions of optimizations with the 2-way superscalar variant of the MIPS R2000, all features were tested individually on the superscalar design, not the baseline design. Section 2 further discusses the implementation and performance results of adding a 2-way superscalar feature to the baseline processor. Section 3 further discusses the implementation and performance results of adding a YAGS branch predictor to the superscalar processor. Section 4 further discusses the implementation and performance results of adding a load value prediction feature to the superscalar processor. Section 5 further discusses the implementation and performance results of adding a stream buffer to the instruction cache of the superscalar design. The combined performance of all previously discussed features on the given tests are explained in Section 6.

II. Superscalar

When thinking about what optimizations to implement into the baseline design, we thought about either out-of-order execution or superscaling as each should lead us to much better performance. We decided to go with superscaling because it felt like the natural next step after learning about pipelines. We had experience designing and implementing the H&P 5-stage pipeline, but not yet had any with superscalar designs so we were interested in learning how to implement a superscalar 5-stage pipeline machine. We went with a 2-way superscalar design as once we figured out how to implement that we could move on to adding more lanes. The second lane is a copy of the first as we had hoped to implement more specific pipelines later given the time.

A. Changes From Baseline to Superscalar

To create the second lane, most of the components from the baseline design were doubled and the input/output of most components were also doubled. To handle new hazards between the two lanes, the hazard controller was expanded. The forward unit was expanded to handle forwarding between the two lanes. Components that were doubled include most inputs/outputs of each module in the cpu core, all pipeline registers, and the ALU.

Beginning with the IF stage, the fetch unit is modified so that instead of incrementing PC + 4 every cycle, we would increment by PC + 8 if we managed to grab both instructions, but increment by PC + 4 again if we only managed to grab one instruction. This is because in the instruction cache, for the sake of a more simple design, we do not grab the second lane instruction if it is in a different cache block from the first lane instruction. The instruction cache is modified to handle outputting two instructions per cycle with the above mentioned exception. Predicting and

resolving branches/jumps have been moved to the IF stage to handle the new superscalar design.

In the ID stage, extra cases in the forward unit are added to allow for the forwarding of data from the first lane EX stage to the second lane DEC stage, the first lane MEM stage to the second lane DEC stage, the first lane WB stage to the second lane DEC stage, and vice versa for the second lane. The forward unit also handles LW hazards and inner hazards (when both lane instructions are using the same registers or both are memory accesses) of the two lanes.

In the EX stage, the ALU is doubled to allow for two arithmetic instructions to run at the same time. The exception for this is arithmetic instructions that have a memory access in which case only one of those instructions are run. This is due to the fact that the data cache does not have parallelization and this can only support one read and write at a time.

In the MEM stage, not much is changed as only one instruction can enter each cycle much like in the baseline design.

The WB stage has also not changed much as all of the write-back processes can be doubled via extra input/output connections.

B. Results

The data gathered for the 2-way superscalar design was taken by simulating the machine through Modelsim and using the fast SDRAM test bench. The benchmarks used were nqueens, qsort, esift2, and coin, provided by the baseline CPU folder. The cache data gathered only uses the nqueens benchmark for the sake of a short data collection time frame.

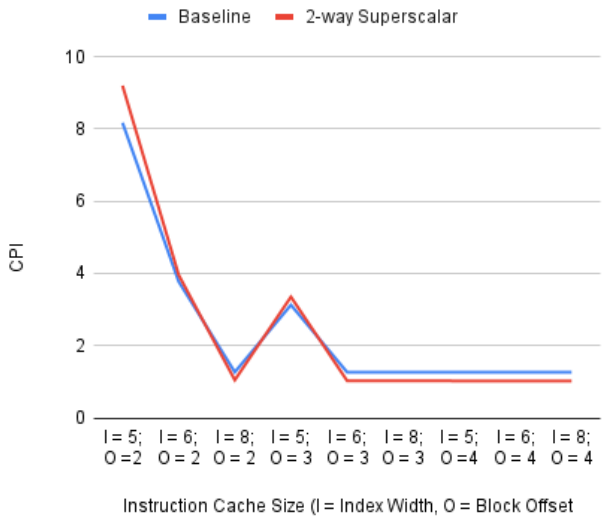

Figure 1: Cache Effect on Baseline and 2-way Superscalar

When first testing the 2-way superscalar design, we noticed that it was somewhat slower than the baseline design. While puzzled, we believed that the cause of this issue was with the cache, specifically the instruction cache. If the instruction cache was too small, the strain of fetching two instructions each cycle would be too much to the point that we were only fetching one instruction per cycle in a 2-way superscalar design. This combined with a lower cycle time due to adding extra components would give us a CPI higher than the baseline design. As such, we tested the cache sizes and looking at Figure 1, the cache size indeed seemed to be the issue. As the instruction cache grows, the performance of the 2-way superscalar design grows and eventually begins to perform better than the baseline design. For this reason, we use the largest cache size configuration for testing all other optimizations. We also tested data cache sizes, but because of the restrictions of one memory access per cycle, there was little to no benefits gained from increasing the data cache.

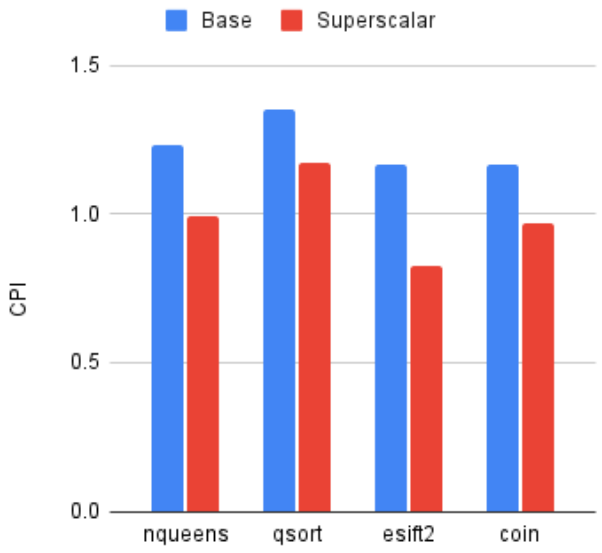

Figure 2: Baseline and 2-way Superscalar Performance

With the cache size testing we did using nqueens, we used the best instruction and data cache configurations for testing the other benchmarks. Figure 2 shows the improvement in performance of the 2-way superscalar design over the baseline design across all benchmarks. We get a 1.25x speedup on nqueens, a 1.15x speedup with qsort, 1.42x speedup with esift2, and a 1.2x speedup with coin. The worst performing benchmark is qsort and this is to be expected as qsort has the most branches thus many instructions would land in different cache blocks forcing the superscalar design to only fetch one instruction many times.

The biggest overheads of our 2-way superscalar design is not being able to fetch two instructions in different cache blocks per cycle and not being able to do two memory accesses per cycle. Even with these overheads we get a decent speedup over the baseline design, but by removing even one of these overheads, a much larger speedup can be achieved.

III. Branch Prediction

The obvious choice for an optimization to observe on the R2000 superscalar machine we created was making a better branch predictor. This is because in the base superscalar, the prediction method set up is always not taken. The results of the runtimes for this are found in table I and show that every benchmark aside from qsort has a prediction rate of less than 50%. This means they are losing more cycles to incorrect branch predictions than they are gaining from correct branch predictions, which is not a good result for

any branch predictor to have because it is actively worsening the CPI.

TABLE I. Base Superscalar Prediction Rates (uses always not taken). Taken with instruction and data cache sizes of index width = 8 and offset width = 4.

	Branch Prediction Correctness
nqueens	39.67%
qsort	58.96%
esift2	22.35%
coin	3.99%

A. YAGS Branch Predictor

The YAGS branch predictor in the paper worked by taking the branch instruction's address and adding it with the global history register. This value is routed to a pattern history table which determines the main prediction. Then the value used to index the primary table is used to index the secondary table of the predictor, which is the "not taken" table if the primary output is taken and vice versa. The table includes a tag which represents the address of the branch. If the tag and address match, then the branch prediction from that secondary table is used, otherwise, the branch prediction is taken from the primary table. The primary table is updated on every feedback except for when it provided a wrong prediction but the secondary table it used provided the correct prediction. The "not taken" table is updated when the primary table predicted taken, but the result was "not taken" and vice versa for the "taken" table. Additionally, secondary tables are updated if their value was used. This is how we implemented our YAGS branch predictor into our machine. The only minor changes we made were that the tag size of the secondary tables were set to be equal to the index size, and that the secondary tables are directly mapped, as these changes seemed to have little to no sizeable effect on results in the original paper. Instead results relayed more on the size of the caches.

B. Updates to Superscalar for Branch Prediction

When implementing branch prediction onto a superscalar machine, the branching system in the pipeline changes. Notably with MIPS, each jump has a branch delay slot (BDS) which should always run regardless of if the jump occurs or not. This normally allows for branches to be run in the ID stage, as the instruction in the IF will always be the BDS instruction and therefore always be valid. However, with super scaling, the BDS could be in the pipeline below the branch, which means when the instructions get to the ID stage, the IF stage instructions are invalid. Because of this, branches and jumps need to be resolved in the IF stage. To do this, instructions are decoded in the IF stage and if they are a branch which is predicted as taken or a jump, the PC changes. This however can cause issues too if the BDS is not paired with the branch and is in the next set of instructions. If this is the case, then predictions are still done in the IF stage, but now the PC jump is stalled to occur in the DEC stage. Additionally, it is inferred that the BDS in this situation will always be in the 1st pipeline, so we disregard the 2nd instruction and only accept the 1st one if there should be a jump. All this information needs to be moved through the pipeline so that during the resolution of the branch in the EX stage, if it needs to be recovered because of a misprediction, it knows which instructions to wipe as the BDS could either be paired with the branch in EX or still be in the ID stage. Additionally, the GHR needs to be carried to the EX stage so that in the event of a misprediction, it repairs the GHR so that it reflects the actual history of branches. To do this, a separate pipeline containing this information moves from the IF to the EX stage, always being in the same stage as the branch instruction and knowing which pipeline it is in.

C. Results

The data gathered for branch prediction was taken by simulating the machine through Modelsim and using the fast SDRAM test bench. The machine used was the baseline superscalar, which had, for both the i-cache and d-cache, an index width of 8 (256 entries) and a block offset width of 4 (16 instructions per line). The benchmarks used were nqueens, qsort, esift2, and coin, provided by the baseline CPU folder. The branch predictor was configured in four separate ways on each of the benchmarks, either the index (& tag size) used were 4 bits (16 entries), 6 bits (64 entries), 8 bits (256 entries), or 10 bits (1024 entries).

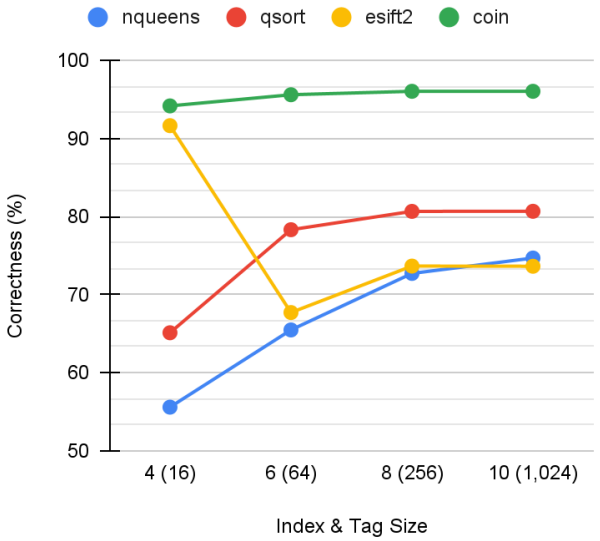

Figure 3. YAGS Branch Prediction Correctness Rate

On figure 3, you can see the rate of correctness for each configuration. In general, the prediction rates are noticeably higher than the correctness rates for the baseline, which was always not taken. Specifically when going from baseline to index width 4, nqueens goes from 39.67% to 55.57% (x1.4 higher), qsort goes from 58.96% to 65.10% (x1.1 higher), esift2 goes from 22.35% to 91.63% (x4.1 higher), and coin goes from 3.99% to 94.16% (x23.6 higher). What is notable is that the correctness rate for esift2 when the index and tag width equal 4 is much higher than any other configuration for esift2 and breaks the general inclination for the correctness to have mitigating returns as index and tag increases. Beyond that, the benchmarks share the same pattern of correctness in which the percentage of correct predictions rises at a decreasing rate. It is increasing most likely due to a decrease in aliasing caused by the increased

table sizes, but is decreasing in rate due to less and less instances of said aliasing as the size increases.

TABLE II. Percent of instructions that are branches for each benchmark.

	Inst Count	Branch Count	Percentage
nqueens	964950	160661	16.64%
qsort	3794394	846954	22.32%
esift2	4058282	476523	11.74%
coin	42803374	7269774	16.98%

Before looking into the CPI results of branch prediction, the actual percent of instructions as branches should be observed in table II, as that will play a role in the change in CPI achieved for each benchmark. Notably, esift has the lowest ratio of branches while qsort has the highest ratio.

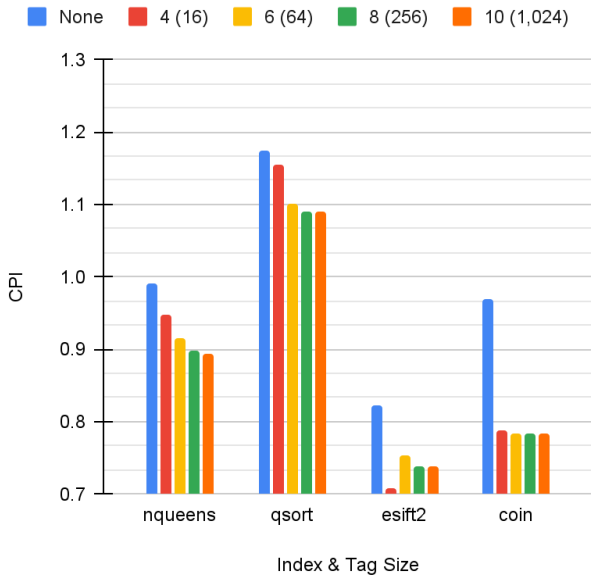

Figure 4: YAGS Branch Prediction CPI

Now looking at the CPIs for each instruction, one can note that they decrease for each benchmark to varying degrees. Specifically for going from always not taken to YAGS with an index & tag size of 4, we get that nqueen's CPI goes from 0.9901 to 0.9468 (-0.0433 cycles per instruction), qsort's CPI goes from 1.1746 to 1.1542 (-0.0204 cycles per instruction), esift2's CPI goes from 0.8228 to 0.7072 (-0.1156 cycles per instruction), and coin's CPI goes from 0.9683 to 0.7886 (-0.1455 cycles per instruction). It seems then that the least affected benchmark is qsort which at first sounds off since it seems to have the most branches per instruction; however, looking back to figure 3, this makes more sense as the prediction rate increase is relatively small compared to the others. The most affected seems to be esift2, which makes the most sense as it receives the highest increase in prediction out of the whole set of benchmarks. Continuing to look on, the decreases in CPI mirrors the increases in the rate of correct predictions. The best index & tag sizes for each of the benchmarks were 10 for nqueens (CPI = 0.8938), 8 for qsort (CPI = 1.0901), 8 and 10 for esift2 (CPI = 0.739), and 8 and 10 for coin (CPI = 0.7837).

To conclude, branch prediction gives a considerable increase in CPI for the benchmarks which makes sense considering that branches make up 11-22% of the instructions in the benchmarks as well as the fact that there is a sizable increase in prediction rates across the board.

IV. Load Value Prediction

With the implementation of superscalar, the amount of memory hazards (specifically LW hazards) drastically spiked. This is because with scalar, the only LW hazard possible is between the ID instruction and the EX instruction. But in a 2-way superscalar, there are 5 possible LW hazards between the ID and EX stage(the first ID instruction could be dependent on either EX instruction or the second ID could be dependent on either EX instruction or even the first ID). On table III, it shows the number of LW hazards between the ID and EX stage (LW hazards) for baseline and superscalar, and on table IV it shows the number of hazards between the two ID instructions (inner hazard).

TABLE III. Number of LW Hazards

	nqueens	qsort	esift2	coin
Baseline	0	0	0	0
Super scalar	119319	389399	138238	7007655

TABLE IV. Number of Inner Hazards

	nqueens	qsort	esift2	coin
Super scalar	123289	1095875	319902	3088487

(NOTE: LW hazard counts were gathered by counting the number of times there was any number of LW hazards between the ID stage instructions and the EX stage instructions. Additionally inner hazards were gathered by counting the number of times there was any number of hazards between two instructions in the ID stage (including inner LW hazards). As such, the data is off when it comes to giving an accurate count of the actual number of hazards. However, the data is still valuable because it does show a rough estimate of the number of hazards which occured, and looking forward, shows a downward trend when load value prediction is implemented.)

Normally, the solution to this problem is through the usage of Out of Order Execution. The two are usually put together and for good reason; they simply synergize extremely well together. However, our team decided to take a more unique approach to the situation. We instead focused on LW hazards and attempted to mitigate them by using the methods found in a paper on load value prediction [2].

A. Load Value Prediction Functionality & Implementation

The load value predictor in our implementation shares many features of the load value predictor used in the paper. Notably, both have a Load Value Prediction Table (LVPT)

which houses the most recently loaded value from the address which it is indexed to, and the Load Classifier Table (LCT), which is an array of 2-bit saturating counters. In the IF stage, the PC is used as an index for the LCT and LVPT, grabbing the value in the LVPT as the prediction and determining if the value should be predicted by looking at the upper bit of the value obtained by the LCT. When a value is predicted, it is treated like a forwarded value. For example, if an instruction in the ID stage depended on the loaded value of an instruction in the EX stage, if that EX instruction has predicted a value, it is forwarded to the ID instruction to be used. Once a LW instruction gets to the MEM stage, the loaded value overwrites the old value in the LVPT, and if it predicted a value back in the IF stage, the predicted value is compared to the loaded value. If they are the same, then increment the value in the LCT and continue. However, if it was a wrong prediction, firstly the CPU decrements the LCT. Then it wipes any instruction which came before it and resets the PC to the instruction after the load. Additionally, the GHR in the branch predictor is reset to what it was at the time of the initial prediction so as not to break it. The only aspect which was not included from the paper was the Constant Verification Unit, which acted as a cache for the predicted value to check if it was correct without going into the data cache. The reason why we did not include it was because we felt that any improvement it brought could be overshadowed by simply improving the d-cache to be able to hold more data at a given time.

B. Results

The data gathered for branch prediction was taken by simulating the machine through Modelsim and using the fast SDRAM test bench. The machine used was the baseline superscalar, which had, for both the i-cache and d-cache, an index width of 8 (256 entries) and a block offset width of 4 (16 instructions per line). The benchmarks used were nqueens, qsort, esift2, and coin, provided by the baseline CPU folder. The load value predictor was configured in four separate ways on each of the benchmarks, either the index for the LCT and LVPT were 4 bits (16 entries), 6 bits (64 entries), 8 bits (256 entries), or 10 bits (1024 entries).

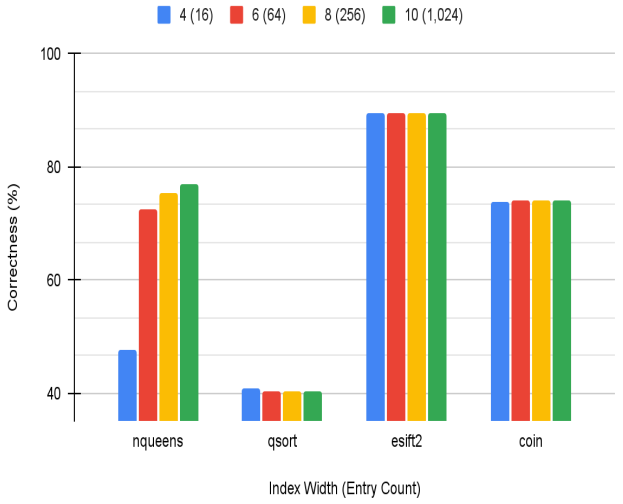

Figure 5: Load Value Prediction Correctness Rate

For prediction rates, we are generally looking for something higher than 50% because that means it gives a correct prediction more often than it gives an incorrect prediction. The results on Figure 5 show three very different conclusions. For nqueens, it shows that setting the index to 4 is less than 50% (47.64%), but as the index sizes increase, the correctness rates increase at diminishing rates. This is most likely because of aliasing in the load value prediction. For qsort, it is in fact less than 50% across all configurations and in fact decreases as the index gets higher (40.89% to 40.35% to 40.28%). For esift2, it already has a high prediction rate but it doesn't change based on index width, staying at 89.58%. Finally, coin is almost like esift2 as it starts relatively high at 73.91%, but there shows no signs of significant change in correctness as the index increases, going from 73.91% on index width 4 to 73.96% on index width 6, and finally stagnating at 74.01% on index width 8 and 10.

TABLE V. Number of LW Hazards for Load Value Prediction

	nqueens	qsort	esift2	coin
4 (16)	86804	394939	89745	3586079
6 (64)	51934	393878	89745	3561158
8 (256)	41000	393845	89745	3543351
10 (1,024)	30572	393845	89745	3543351

TABLE VI. Number of Inner Hazards for Load Value Prediction

	nqueens	qsort	esift2	coin
4 (16)	140791	1101693	281155	3856736
6 (64)	139973	1101636	281155	3875156
8 (256)	139952	1101554	281155	3874703
10 (1,024)	140082	1101554	281155	3874703

In tables V and VI, we can see the number of LW and inner hazards which still occur after load value prediction, though for some there is an overall decrease in the number of hazards. For nqueens, the total hazard counts are consistently lower than the baseline superscalar and drop as the index width increases. Qsort starts with a higher hazard count (1,485,274 at base superscalar vs 1,496,632 at index width 4), and while it drops slightly as the index increases, it never goes under what it originally was. Esift2 is immediately has a lower hazard count with load value prediction (458,140 at base superscalar vs 370,900 at index width 4), but it never changes. Finally, coin has a considerable decrease in total hazards (10,096,142 at base superscalar vs 7,442,815 at index width 4), but its hazard count remains largely unchanged as index width increases.

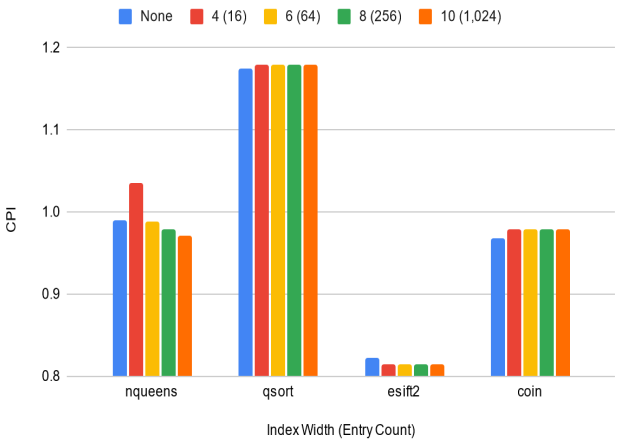

Figure 6: Load Value Prediction CPI

In general, the conclusions shown in figure 6 are that for nqueens, the cpi is initially higher than base superscalar, but eventually at index 6, its CPi becomes less than the base. Qsort minorly increases in CPI mostly due to its poor correctness rates. Esift2 sees a slight decrease in CPI when the index width is 4 and remains the same CPI for the other index width sizes. The surprising result is that despite the good correctness rates and the decrease in hazards, the coin benchmark's CPI actually increases. The best index sizes for each of the benchmarks (if it had to include the optimization) were 10 for nqueens (CPI = 0.9721), any size for qsort (CPI = 1.1799), 6, 8, and 10 for esift2 (CPI = 0.8146), and 8 and 10 for coin (CPI = 0.9788).

To conclude, load value prediction had mixed results. For nqueens and esift2, it provided minor increases in CPI, which made sense considering their high correctness rates and decrease in hazards. For qsort and coin, it negatively impacted the CPI. For qsort it made sense as it had a low correctness rate and an increase in hazards, but coin had a good correctness rate and a decrease in hazards but still increased in CPI, which is interesting.

V. Instruction Cache Stream Buffer

With the addition of 2-way superscalar to the baseline design. We were now able to fetch double the instructions per cycle from the instruction cache to execute. This, however, means accessing the cache a lot more potentially leading to more capacity misses. Instructions also tend to be fetched in sequential order leading to a higher chance of correctly predicting the next instruction to be fetched. Then the only logical choice of optimization would be to implement a stream buffer attached to the instruction cache.

A. Stream Buffer Functionality

The stream buffer design used in this paper was inspired by the design created by Norman P. Jouppi and it works in a similar way [3]. When a cache miss happens, the first entry of the stream buffer is also checked to see if the cache block is in the stream buffer. If it is, the cache block is moved from the stream buffer to the cache. If not, the stream buffer is flushed of all data and fetches the missing cache block plus an extra number of successive cache blocks from the SDRAM. Like the design by Jouppi, the stream buffer is a simple FIFO queue and only the first entry of the stream buffer is checked. The difference in design is that our stream buffer does not fetch another instruction after moving one to the cache while Jouppi's design does.

B. FSM Design

The stream buffer was designed using a finite-state machine because the baseline design uses a finite-state machine to implement the instruction cache. This is likely due to the way in which the instruction cache receives data from the SDRAM. The baseline design sends word-sized data each clock cycle from the SDRAM to the instruction cache. Because we didn't want to potentially create any issues inside the SDRAM by modifying its code, we decided to use the same process when implementing the stream buffer, thus sending data from the stream buffer to the instruction cache takes a few cycles rather than one. Those few cycles are cancelled out by the fact that receiving data from the SDRAM also takes more than a few cycles, thus it is still a lot faster to send data from the stream buffer than from the SDRAM.

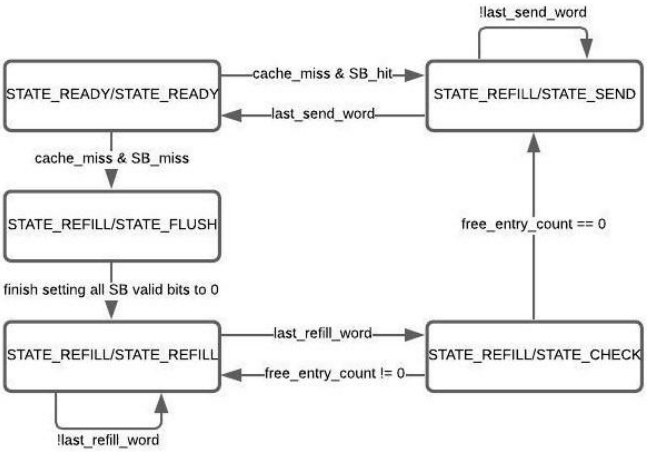

Figure 7: State Diagram of Instruction Cache and Stream Buffer

In Figure 7, the state on the left corresponds to the instruction cache and the state on the right corresponds to the stream buffer. When both are in STATE_READY, we have not yet been given an instruction to fetch or we have a cache hit. If we have a cache miss, the instruction cache moves to STATE_REFILL to wait for incoming data. We then check the stream buffer. If the stream buffer hits, it moves to STATE_SEND and it sends word-sized data each cycle until we send the last word. Then both go back to STATE_READY. If the stream buffer misses, it goes to STATE_FLUSH where the valid bits of all entries in the stream buffer are invalidated. It then moves to STATE_REFILL where the stream buffer receives word-sized data from the SDRAM each cycle until the last word is received. The stream buffer moves to STATE_CHECK to check how many entries it has added. If the stream buffer is not full, it returns to STATE_REFILL to repeat the data grab process. Once the stream buffer is full, it goes to STATE_SEND to send the cache the requested block of data.

C. Results

The data gathered for the stream buffer was taken by simulating the machine through Modelsim and using the fast SDRAM test bench. The machine used was the baseline superscalar, which had, for both the i-cache and d-cache, an index width of 8 (256 entries) and a block offset width of 4 (16 instructions per line). The benchmarks used were

nqueens, qsort, esift2, and coin, provided by the baseline CPU folder. The stream buffer was configured in four separate ways on each of the benchmarks, an index width of 1 bit (2 entries), 2 bits (4 entries), 4 bits (16 entries), or 8 bits (256 entries).

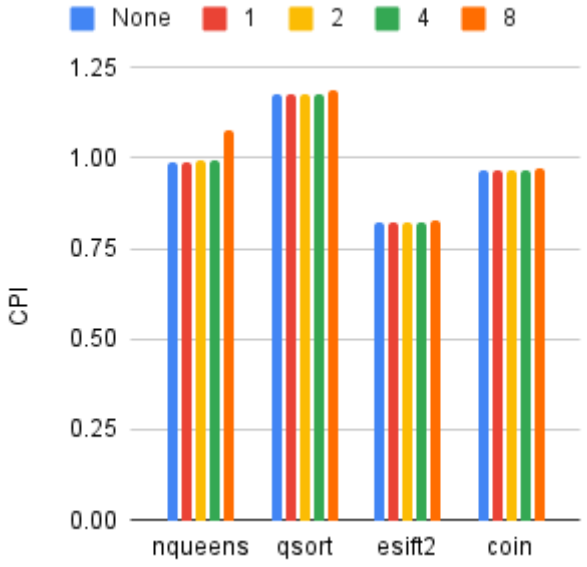

Figure 8: Stream Buffer CPI

From Figure 8, we can see that the CPI of each configuration increases very slightly. This means that the stream buffer actually increases the total runtime of each test program. This is likely due to the fact that on a stream buffer miss, the stream buffer grabs the maximum amount of entries possible forcing the instruction cache to wait not just for the requested data, but the prefetching as well. It may also be due to the fact that once we remove entries we never grab new entries. These are things that we would've liked to improve upon, but given the time frame could not get to. Improving these things should improve the CPI, but because of the fact that there is no parallelization in memory it may be of little to no benefit.

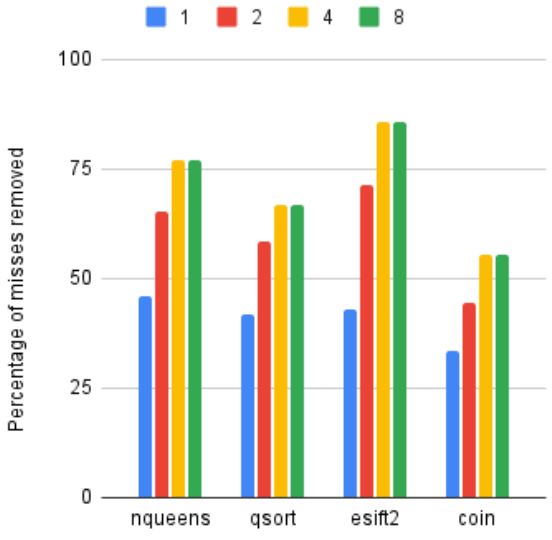

Figure 9: Stream Buffer Performance

Taking a look at Figure 9, there is a significant percentage of misses that have been removed due to the use of a stream buffer. Having used the most optimal configuration for the instruction cache which happens to be the largest configuration, that alone has removed a significant percentage of misses. But even then we see a large improvement in miss rates with the worst result still being about 33%. The four bit and eight bit configurations have the same percentage of misses removed which shows that there may be misses that span more than 256 entries. Larger stream buffers may be required to test these results even further.

Although stream buffers perform very well in removing misses from the instruction cache, our implementation of it combined with not having parallelization in memory nullifies its benefits. With a more optimized implementation, however, even without parallelization in memory we should see enough of a performance improvement which will decrease the CPI.

VI. Overall Data Analysis

The data gathered for branch prediction was taken by simulating the machine through Modelsim and using the fast SDRAM test bench. The machine used was the baseline superscalar, which had, for both the i-cache and d-cache, an index width of 8 (256 entries) and a block offset width of 4 (16 instructions per line). The benchmarks used were nqueens, qsort, esift2, and coin, provided by the baseline CPU folder. Branch prediction is configured to be as large as was tested in the Branch Prediction section (index = 10) as this generally showed the best results, load value prediction is configured to be as large as was tested in the Load Value Prediction section (index = 10) as this generally showed the best results, and the file stream buffer is configured to be as small as possible (index = 1) as this generally showed the best results.

TABLE VII. CPI Values for Optimization Combinations

Optimizations	nqueens	qsort	esift2	coin
Base	1.2329	1.3490	1.1667	1.1665
Superscalar	0.9901	1.1746	0.8228	0.9683
Superscalar and Branch Prediction	0.8937	1.0905	0.7390	0.7836
Superscalar and Load Value Prediction	0.9722	1.1799	0.8146	0.9788
Superscalar, Branch Prediction and Load Value Prediction*	0.8676	1.0960	0.7320	0.7793
Superscalar and Stream Buffer	0.9907	1.1746	0.8229	0.9684
Superscalar, Stream Buffer, and Branch Prediction	0.8943	1.0905	0.7391	0.7837
Superscalar, Stream Buffer, and Load Value Prediction	0.9727	1.1800	0.8146	0.9788
All Four Optimizations*	0.8676	1.0962	0.7320	0.7793

* These two tests which had both branch and load value prediction were simulated and ran on modelsim but technically could not be fitted onto the FPGA Cyclone V board and as such while they are synthesizable, they are not fittable.

The results shown in table VII are for the most part in line with what we have seen in the previous sections. Notably that superscalar on its own already decreases the CPI by a significant amount. Additionally, Branch Prediction as an inclusion always improves the CPI of whatever configuration it is added into across all benchmarks. Load value prediction without branch prediction follows the same results shown in the load value section.

What is interesting to show is the synergy between load value prediction and branch prediction shown in nqueens, qsort and coin. For nqueens, the CPI of branch prediction is 0.8937 (-0.0964 CPI from superscalar), the load value prediction CPI is 0.9722 (-0.0179 CPI from superscalar), and using both gets the CPI 0.8676 (-0.1225 CPI from superscalar), which is a greater decrease in CPI than the sum of just branch prediction and load value prediction (-0.0964 + -0.0179 = -0.1143). For qsort, the CPI of branch prediction is 1.0905 (-0.0841 CPI from superscalar), the load value prediction CPI is 1.1799 (+0.0053 CPI from superscalar), and using both gets the CPI 1.0960 (-0.0786 CPI from superscalar), which is actually a decrease in CPI from the sum of just branch prediction and load value prediction (-0.0841 + 0.0053 = -0.0788). While there does seem to be some synergy between the two, the CPI is still worse off because of load value prediction as it is still a greater CPI than just having superscalar and branch prediction. For coin, the CPI of branch prediction is 0.7836 (-0.1847 CPI from superscalar), the load value prediction CPI is 0.9788 (+0.0105 CPI from superscalar), and using both gets the CPI 0.7793 (-0.1890 CPI from superscalar), which is a greater decrease in CPI than the sum of just branch prediction and load value prediction (-0.1847 + 0.0105 = -0.1742). This is especially surprising since load value prediction on its own hurts the CPI of coin.

There also seems to be some anti-synergy between the two optimizations when going to the esift2 benchmarks. For esift2, the CPI of branch prediction is 0.7390 (-0.0838 CPI from superscalar), the load value prediction CPI is 0.8146 (-0.0082 CPI from superscalar), and using both gets the CPI 0.7320 (-0.0908 CPI from superscalar), which is actually an increase in CPI from the sum of just branch prediction and load value prediction (-0.0838 + -0.0082 = -0.092).

Looking at the stream buffer optimization, it seems to have very little effect on the CPI across all configurations and benchmarks. Where changes are to be found, they show an increase in CPI, which is in line with the results shown in the Stream Buffer section.

Concluding on the CPI values, the best optimizations to have for each benchmark are superscalar, branch prediction and load value prediction for nqueens (CPI = 0.8676), superscalar and branch prediction for qsort (CPI = 1.0905), superscalar, branch prediction and load value prediction for esift2 (CPI = 0.732), and finally superscalar, branch prediction and load value prediction for coin (CPI = 0.7793). Additionally, one can include the stream buffer to any of these best sets of optimizations and it would not change the CPI, so technically it can be included with the best configuration if looking at CPI alone.

On a final note, we can take a brief look at the clock cycle times on table VIII which, while not the focus of this paper, is still interesting to observe. Notably, the clock cycle speed almost halves once all the optimizations are implemented, only being 52.98% as fast as the baseline clock cycle speed. This is in part due to some inefficiencies in the machine we created. For example, because we decode the instructions at the IF stage instead of having a cache of PCs for branch prediction, the cycle increases due to the long wire paths caused by that. Though at a certain point, even if the inefficient processes were optimized, the cycle speed would still take a toll as the optimizations like branch prediction and load value prediction inherently create longer wire paths.

TABLE VIII. CPI Values for Optimization Combinations

	Clock Cycle Speed (MHz)
Baseline	96.66
All Four Optimizations	51.22

(NOTE: The clock cycle speed taken for all the optimizations was based on a smaller index size for both load value prediction and branch prediction than what was set for the overall results (index width 10 to index width 8). This is primarily because the original could not fit on the FPGA board and so to measure a cycle time the indexes were decreased.)

VII. Conclusion

In conclusion, the optimizations for the most part improve the baseline CPU by a considerable amount when it comes to decreasing the CPI. Specifically, superscalar and branch prediction show the greatest improvements to the CPI and load value prediction shows marginal increases in benchmarks aside from qsort (and coin if you ignore the improvements to CPI given by its synergy with branch prediction). The stream buffer optimization, while not making much of a difference in CPI, still decreases the miss rate of instructions.

In the future, there are some possible improvements that can be made to the optimizations given in this paper. For superscalar, one could increase the number of parallel pipelines to increase instruction throughput. For branch prediction, one could improve the prediction rates by creating a tournament style system with two different predictors, choosing the results of the predictor which is more accurate. For load value prediction, indexing could instead be based on the loaded address so that when a store is made to that address, the value in the LVPT is set to what was stored at that address. Finally, for the stream buffer, allowing the memory to be fetched in parallel for each slot in the buffer could dramatically increase the CPI.

References

[1] Eden, A. N. and T. Mudge. “The YAGS branch prediction scheme.” *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture* (1998): 69-77.

[2] Lipasti, M. et al. “Value locality and load value prediction.” *ASPLOS VII* (1996).

[3] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364-373, doi: 10.1109/ISCA.1990.134547.