The Illinois Out-of-Order Machine (IllOOM): A RISC-V Processor for ECE 411

Thomas Vigh
Univ. of Illinois Urbana-Champaign
Urbana, IL
tvigh2@illinois.edu

Ben Pegg
Univ. of Illinois Urbana-Champaign
Urbana, IL
benpegg2@illinois.edu

***Abstract*—This report details the design, implementation, and optimization of *IllOOM*, our RISC-V processor for ECE 411 mp_ooo.**

I. Introduction

IllOOM is our out-of-order RISC-V processor targeting the RV32IM ISA based on Explicit Register Renaming architecture. In-order processors are restricted by false dependencies because they process instructions sequentially according to program order. An out-of-order processor remedies this by executing instructions as soon as their operands are ready, allowing for greater instruction-level parallelism and ultimately quicker program execution. To exploit this out-of-order execution, *IllOOM* implements a number of advanced features such as an N-way parameterizable superscalar frontend (implemented as 4-way), 2-way dispatch and commit, a GShare branch predictor, a pipelined and set/way-parameterizable instruction cache, split load-store queue with store to load forwarding, a writethrough post-commit store buffer, a return address stack, and other optimizations. Completing this mp gave us a better understanding of modern computer architecture concepts and improved our understanding of performance, power, and area tradeoffs involved with optimizing a design. *IllOOM* draws its namesake (and certain architectural choices) from *BOOM*, the Berkeley Out-of-Order Machine [1].

II. Overview

A. Project Overview

There were two primary objectives of this project. The first was to successfully build a RISC-V processor that supported out-of-order execution. The design choices for this baseline processor and the milestones that guided our work are outlined in section III. Overall, this processor was built to prioritize correctness and ease of understanding; our team used this as an exercise to solidify our understanding of processor design and register renaming and out-of-order architectures as well as to build a strong foundation upon which we could make future optimizations.

The second objective of this project was the design competition, where we competed with other teams to make the most performant processor based upon the geometric mean of metric PD^4 across various benchmarks. We routinely scoreboarded our processor, using the baseline as a starting point, to determine performance critical bottlenecks, and pursued various advanced features such as superscalar execution, Load/Store forwarding, and more, to remedy them.

Throughout both stages, individual members often developed sections of the architecture and advanced features and brought them together as a team for integration and validation. Performance metrics were taken before and after proposed changes, and the comparison of the two was used to determine if the change was pursued further. Work sharing was managed through GitHub, utilizing pull requests and issues greatly. We aimed to keep our 'main' branch up-to-date and clean at all times, developing new features and architecture changes on branches and using pull requests and reviews to discuss trade-offs before integrating back into 'main'. Throughout development, issues were used to document areas for future optimizations and quality-of-life updates. As mentioned before, we focused our first iteration on correctness and ease of understanding over performance, so these issues and branches were a good way to keep track of future work while we were laying the foundation.

B. Design Overview

As seen in Fig. 1, *IllOOM* has three distinct stages: frontend, midcore, and backend. The frontend manages instruction fetching, predicts control flow with a GShare branch predictor and return address stack, and enqueues instructions to the instruction queue to be dispatched.

The midcore is comprised of the main register renaming architecture that allows for out-or-order execution. Instructions are dequeued from the instruction queue and decoded before the free list and register alias table are used to rename them. Then, they are dispatched to the re-order buffer and scheduler which routes them to specific reservation stations based upon their needed execution unit. When an instruction has its operands ready, it is woken up and reads its operand values from the physical register file before entering an execution unit. Our processor has five types of execution units: Arithmetic Logic Unit, Multiplier Unit, Divide/Remainder Unit, Address Generation Unit, and Load/Store Generation Unit. After execution, the results and required tracking and status information are broadcast on the central data bus. The reservation stations, RAT, and ROB tap this bus to see execution

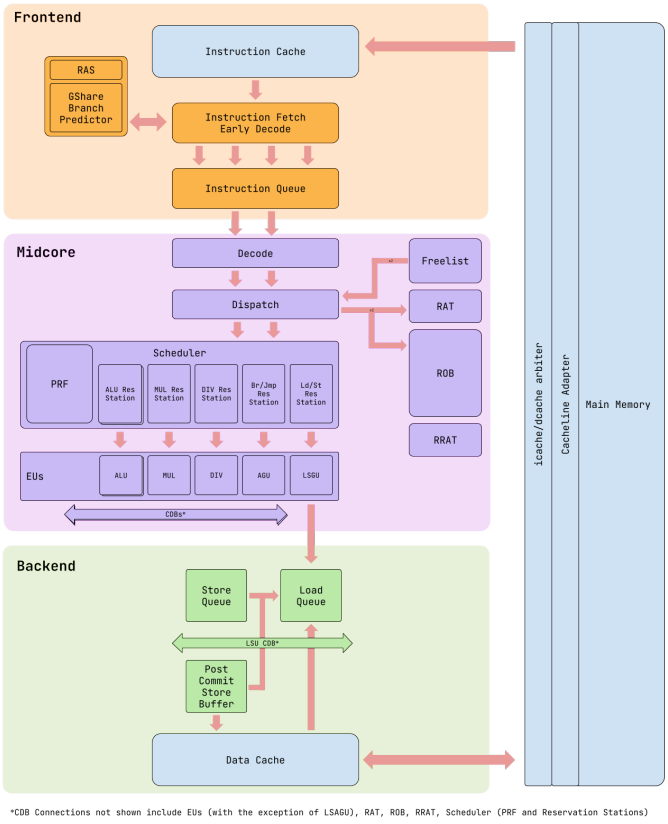

Fig. 1. *IllOOM* Block Diagram

information. Finally in the midcore, instructions that are retired from the ROB get sent to the retirement register alias table which is used to track non-speculative register mappings and used to update the free list.

Data management is handled in the backend, where split load store queues interact with a data cache. A writethrough post-commit store buffer is used to extend the life of forwardable data from the store queue to load queue, essentially acting as a small ”L0” cache. It also allows for quicker store commits by masking some of the data cache latency.

III. Milestones

A. Checkpoint 1

For checkpoint 1, we created our frontend. This included a parameterizable queue (implemented as a circular buffer), a cacheline adapter to handle bursts from our DRAM, and a working instruction fetch that would fetch instructions into our queue. We also created a preliminary block diagram and solidified some of our central architectural choices. Making some of the larger architectural choices earlier on, like our decision to implement funnel-shaped superscalar execution with a wide frontend to a narrow midcore and backend, proved to be helpful later on. Our baseline design was implemented with the knowledge that it would later need to be expanded to support superscalar execution and various other advanced features, which meant that we tried to implement as much parameterization into our baseline design as possible. This was useful in the advanced features stage, because it minimized the amount of HDL we had to rewrite to accommodate a superscalar design, and we had already considered a lot of the challenges that would be associated with implementing it.

B. Checkpoint 2

Checkpoint 2 involved building out most of the core to support all RV32IM reg and imm instructions, with the exception of memory and branch/jump instructions. For our midcore, we decided on a segmented approach, where each execution unit would have a dedicated reservation station and CDB. This was implemented with the idea in mind that we would analyze the performance, area, and timing tradeoffs of this approach later on when optimizing. It also simplified our design, because we didn’t have to deal with CDB arbitration logic or complex reservation station to EU routing. Another part of checkpoint 2 was implementing the Synopsys multiplier and divider IPs. We decided to implement the IPs as a sequential divider and a pipelined multiplier after guidance from our mentor CA. Our biggest challenge with implementing checkpoint 2 was during the integration phase, where we were trying to piece together all of our modules. Although we were communicative throughout all of checkpoint 2, most of the time working collaboratively in the same room, integration still took longer than we expected due to small edge cases that could not be caught by our module-level testbenches. With this in mind, we dedicated more time to integration for checkpoint 3, consolidating modules into larger pieces as soon as they were ready so that we could identify issues earlier on.

C. Checkpoint 3

For checkpoint 3, we implemented the rest of the RV32IM instructions (memory and branch/jump instructions). The two largest parts of this were implementing our backend, and modifying our processor to support speculative execution. For our backend, in our baseline design, we implemented split load store queues. Although a unified load store queue would have simplified our design, we knew that we were planning on using a split design anyways, which would require rewriting most of the logic of a unified design. To simplify the implementation, we did not include any of the more sophisticated features in our baseline, such as store to load queue forwarding or adding a post-commit store buffer. The biggest challenge involved with the load store queues was getting RAW hazard detection to work correctly, as there were many edge cases associated with it. Because *IllOOM* doesn’t support speculative loads, every load must be RAW-checked before it can broadcast from the load queue head. This means that, for each load, we must check every older store instruction for a matching address. RAW checking is handled by creating a store queue snapshot for every load queue entry when it is dispatched. One edge case which took a while to debug was our logic for byte-coverage. We could have ”false” RAW hazards, where we have a store and load with matching addresses, but the bytes that those store and load instructions cover don’t intersect.

For example, if we have a store half to the upper two bytes and a load byte from the least significant byte, this should not be flagged as a RAW. However, if we have a store half to the middle two bytes and a load half from the two lowest bytes, this should be flagged as a RAW. Another edge case was making sure to flag the right store instruction as a RAW, because a load can have multiple valid RAWs. The correct approach was to make sure that we flag the youngest store with byte coverage that is older than the current load.

IV. Advanced Design Features

A. Superscalar

One of the main advanced features we implemented in our processor was superscalar execution. In our baseline processor, one of the main bottlenecks, outside of branch prediction accuracy and cache response times, was simply the number of instructions we could push through our processor at any given cycle. Multiple separate execution units allowed for different instructions to be executed at the same time if their operands became ready in the same cycle and they targeted different EUs. However, single instruction dispatch and commit limited how much we could take advantage of the ILP of our architecture. Therefore, we decided to implement a parameterizable superscalar processor.

Our processor was superscalar with parameters to specify the number of ways we supported in the frontend, midcore, and commit path, all separately. This level of parameterization allowed for us to use our scoreboarding to execute design space exploration to determine the optimal parameters. In the end, we went with a funnel-shaped design. We used a 4-way frontend that could fetch and enqueue 4 instructions each cycle into the instruction queue. We had a 2-wide midcore that would decode, rename, and dispatch strictly 2 instructions each cycle. Our 2-way commit path also allowed us to retire up to 2 instructions each cycle. This funnel-shape, with a wider frontend than midcore, rarely stalled the core due to instruction queue starvation. The 4-wide frontend led to critical path issues, but these were remedied and as described in section H.

When evaluating performance, power, and area tradeoffs, our main comparison was between 2-way and 3-way dispatch. With 2-way, we saw a large increase in IPC from our baseline scalar processor. The ability to rename and dispatch twice the number of instructions each cycle affords much greater ILP. Along with our wider frontend and midcore width, we incorporated a second ALU. In our scoreboarding, noticed a bottleneck where multiple instructions in the ALU reservation station were ready to wake up at the same time. Adding a second ALU helped to resolve this bottleneck, and we saw a big IPC increase on benchmarks with a heavy reliance on ALU instructions.

When going to 3-way dispatch, we continued to see improvements in overall IPC across all the benchmarks, but significantly less than the improvement from 1-way to 2-way. After validating both 2-way and 3-way in simulation to ensure logic was sound and edge cases were tested, we moved onto synthesis to see the effect on area, power, and timing. Here, we discovered that going from 2-way to 3-way increased the rename/dispatch logic to such an extent that it dominated our critical path. We were able to meet timing at 625 MHz with 2-way dispatch, but could only meet at 590 MHz for 3-way. Seeing as this was a more than 5% reduction in frequency for a less than 5% increase in IPC, we ultimately decided to go with 2-way dispatch for our final design.

Superscalar execution was the advanced feature that provided the greatest increase in performance across the board. This is largely due to the fact that it improves performance in all workloads. Programs with high densities of branches and jumps can benefit from faster pipeline fill on mispredictions. Programs with high densities of data memory accesses benefit from increased forwarding opportunities if more load/store instructions are decoded and resolved in parallel. Because of the performance increase across all workloads, superscalar architecture was our most important advanced feature.

B. Branch Prediction

After completing our baseline processor with a static always taken branch predictor, we noticed, obviously, that our performance was being heavily limited by flushing penalties from mispredicting control flow instructions. The next step was to implement a branch predictor. We decided to go with a two-level design: Gshare.

We implemented gshare with an 8-bit global history register that is XORed with 8 bits of the PC to index into a 256-entry pattern history table. The specific bits of the PC we used in the XOR function could be modified to tune our branch predictor; ultimately, we landed on `PC[9:2]`. Additionally, we implemented the PHT as 2-bit saturating counters in order to minimize the thrashing of a singular mispredict. This PHT was initialized to weakly-taken.

The GShare branch predictor was alongside our fetch stage to make predictions based on the current pc and redirect control flow if predicted taken and the target was resolved or cached. Our integration in the fetch stage allows for two control flow redirects within an N-wide fetch. Originally, we used a large branch target buffer to cache past branch targets and use them in future branch predictions. Provided a gshare ”taken” prediction, the BTB would be used to redirect control flow if it stored the target of the pc being predicted on. Because we implemented GShare and the BTB as flipflops, the large number of entires that the BTB would hold caused a large increase in area usage. Additionally, due to the nature of a BTB, we would have to contend with unavoidable compulsory misses. Although the BTB would help us with target prediction, looking at the opportunity cost, we decided to remove it.

In its place, we opted for a pre-decoder inside of our fetch stage. This pre-decoder would detect branch and jump instructions and compute their ’PC + offset’ targets. This way, as long as the branch predictor predicts correctly, we can always redirect control flow the first time we see a new branch. This allowed for better accuracy as we avoided the

TABLE I
GSHARE BRANCH PREDICTOR ACCURACY

Benchmark	Accuracy
coremark	82%
aes_sha	69%
compression	99%
fft	72%
mergesort	70%
image	79%
Average	78.5%

compulsary miss problem while also reducing our area. This pre-decoder in the fetch stage did contribute to a critical path as we pushed frequency, and our solution to this is discussed in section H. Removing the BTB did not help in all cases, however, because JALR instructions depend on a register operand which often is not resolved at the time of branch prediction. This means that a pre-decoder is unable to resolve the target. Profiling the provided benchmarks, we noted that most JALR instructions were used for function calls in the CALL/RET format. Therefore, we implemented a return address stack for this exact pattern. When detecting a CALL, we would push the PC+4 onto the RAS. When detecting a RET, we would pop from this RAS and redirect to what we predicted was the correct position.

Using our GShare branch predictor design with a pre-decoder and RAS, we achieved a prediction accuracy on the provided benchmarks of: 82% on coremark, 69% on aes_sha, 99% on compression, 79% on image, 70% of mergesort, and 72% on fft. This is notably better than our static always taken predictor, which was correct only slightly better than half the time. However, we note that our branch predictor was one of the largest opportunities for further improvement of our processor. We attempted to reduce PHT indexing aliasing by XOR-folding more of the PC into the indexing scheme, but we saw a large decrease in accuracy (down to around 60%), so we reverted to the standard `PC[9:2]`.

C. Pipelined Instruction Cache

Profiling the provided benchmarks and reasoning about common programming style, we realized that the working set of instruction memory would be relatively small and often reused due to the dependence on for-loop constructs. To take advantage of this, we aimed to improve the throughput of our instruction cache, specifically on sequential cache hits. To achieve this, we implemented a two stage pipelined instruction cache. This allowed us to increase the throughput of our cache, getting back-to-back cycle responses on sequential cache hits, while maintaining our one-cycle latency on all cache hits. This increased throughput helped to decrease any starvation in the instruction queue.

D. Partial Early Branch Recovery

To mask some of our shortcomings in branch predictor accuracy and our large pipeline stage flush penalty, we opted for a partial version of early branch recovery. This partial early branch recovery was implemented to redirect only the frontend of our processor as soon as a branch instruction is executed, before it is committed. We implemented this partial approach instead of full EBR due to the large area overhead associated with full EBR. In a full EBR design, each in-flight instruction must be tagged with a branch mask, and we would have to manage passing around clear or squash masks on each branch resolution.

In our baseline processor, when a branch instruction is determined to have been mispredicted, it is marked in the ROB, but a flush is only broadcast once this instruction reaches the head of the ROB. This is to simplify branch recovery, as once the mispredicted instruction is at the head of the ROB, we can be certain that all in-flight instructions are speculative and must be squashed. This is opposed to the alternative of analyzing each in-flight instruction to determine if it is older or younger. This naive approach allowed for much less overhead, but also caused a larger penalty on each mispredction.

Our partial EBR approach works as follows. As soon as a branch is determined to have been mispredicted by the AGU, a signal and correct target is sent to the frontend. The frontend immediately clears the entire instruction queue and begins fetching new instructions from the updated target. The dispatch stage stalls, halting any consumption from the instruction queue, until it sees the usual flush command from the ROB. This method does not avoid the full performance penalty of waiting for the mispredicted instruction to reach the head of the ROB. It does, however, reduce this penalty. This is because the frontend has been redirected early and will already have sent out an instruction cache request for the correct target, as well as potentially already having enqueued corrected instructions into the instruction queue.

We found this partial EBR feature improved performance on sporadic, branch-heavy workloads, where our branch predictor often mispredicted. It also did not affect our critical path or hurt IPC, so it had a negligible impact on performance of other benchmarks. In the future, with branch predictor accuracy still being a large bottleneck, we would opt to optimize our branch predictor or implement full EBR.

E. Split Load Store Queues and Writethrough Post-Commit Store Buffer with Forwarding

Our backend is reliant on a split load store queue design, where the store queue is connected to a post-commit store buffer. Our post-commit store buffer is used to extend the life of forward-able data from the store queue to load queue, essentially acting as a small ”L0” cache. Implementing a post-commit store buffer allows us to hide some of the data cache latency, because, assuming the PCSB can evict something, stores can CDB broadcast before the data cache is ready to be written to. This helps resolve some data cache stalling in the scenario where stores in the store queue are ready to broadcast, but are being stalled while waiting for data cache to be accessible. Because the PCSB is writethrough and forwardable to the load queue, it also increases the number of forwarding opportunities. PCSB entries are only evicted

TABLE II
FORWARDING EVENTS

Benchmark	Store-to-Load	PCSB-to-Load	Forward Rate
coremark	1182	10227	0.1863
aes_sha	2967	15697	0.0927
compression	0	35000	0.7565
fft	0	19657	0.1546
mergesort	416	17380	0.2249
image	6552	6333	0.4705
Average			0.3143

TABLE III
DATA CACHE LATENCY

Benchmark	Hits	Misses	Hit Rate	Avg. Latency (Cycles)
coremark	68545	137	0.998	1.08
aes_sha	276153	5707	0.980	1.61
compression	52992	785	0.985	1.88
fft	231336	3425	0.985	1.62
mergesort	122449	2501	0.980	1.82
image	28496	21	0.999	1.02
Average			0.984	1.50

when the PCSB is full and receiving upstream pressure from the store queue. This means that a load that is younger in sequential program order can be forwarded to from a much older store that has already been CDB broadcasted and sent to the data cache.

Another design decision we made for the load store queue was forwarding to any load queue entry. To accommodate for this without blowing up our area or bottlenecking our timing, we made our load and store queues shallow (depth of four), which we could accommodate for with our more aggressive forwarding logic. This meant that by the time a load reached the load queue head, there was a higher likelihood that we could bypass the data cache altogether and broadcast to the CDB.

As seen in Table II, across all 6 released benchmarks, we have an average forward rate of 0.3143 (the rate at which a load instruction gets forwarded to). We can look at a specific benchmark, such as coremark, with a forward rate below our average at 0.1863 and an average data cache latency of 1.08 from Table III, which is also below our overall average. With $1182 + 10227 = 11409$ total forwards and an AMAT of 1.08 cycles, that means we save $11409 * 1.08 = 12321.72$ cycles. At a frequency of 614.3 MHz, this amounts to $12321.72/614.3 = 20.058$ µS in total latency saved. On the extreme end, in our best benchmark, compression, we have a total of 35,000 forwards and an average datacache latency of 1.88. This means we save $35000 * 1.88 = 65800$ cycles, which, at 614.3 MHz, amounts to $65800/614.3 = 107.114$ µS in total latency saved. At a total program latency of 441.2 µS, this means that we achieved an approximate speedup of 19.54% from forwarding alone. We also noticed that PCSB forwarding events occur significantly more than store-to-load forwarding events, with some benchmarks having zero store-to-load forwards. This is most likely caused by our modification to the store queue depth, which greatly reduces the likelihood of a forwardable store before the time it exits the store queue.

F. Load/Store-Specific Execution Unit

In our baseline processor, we implemented a single AGU which would handle all loads, stores, branches, and jumps. When we looked at our scoreboarding data, we noticed that this was a large bottleneck on benchmarks in programs with either a large number of control flow instructions or memory instructions. Additionally, combining both into a single AGU is inherently an overgeneralized solution, as control flow instructions output to the CDB, while memory instructions output to the LSU. To alleviate this, we created a separate execution unit that we called the Load Store Generation Unit (LSGU), which is essentially just a small ALU specialized for address and data generation for memory instructions. By decoupling these operations, the primary AGU is now dedicated to resolving control flow instructions, updating the branch predictor, and broadcasting misprediction on the CDB.

G. Store Address and Data Disambiguation

In addition to our dedicated LSGU, we also implemented store address and data disambiguation. This means that a store's target memory address and data could be scheduled and executed as entirely independent operations. As soon as the LSGU computed either operand, it routed the result directly into the store queue, allowing the queue to mark the address or data as ready without waiting for both parts to finish simultaneously. By resolving store addresses as early as possible without having to wait for data, we could identify a potential RAW sooner, reducing the amount of time a load has to sit in the load queue waiting for older stores to be issued in order to be RAW checked.

H. Increasing Frequency to 614 MHz

After implementing many of the above advanced features, we attempted to increase our frequency with the goal of reducing the overall delay of our processor and ultimately the PD^4 metric. We knew that one way to increase frequency would be to pipeline parts of our processor with the longest combinational logic paths. Pipelining would reduce our IPC because each instruction would take another full cycle to traverse through the processor, and flushing and redirects could lead to a longer penalty. We believed, however, that our previous advanced features improved our IPC to a point that the next best optimization would be to push our frequency.

The first critical path that limited our frequency was in the fetch stage. As we increased our fetch stage from 2-way to 4-way and implemented a pre-decoder for control flow target generation, we no longer met our 500 MHz clock. So, we decided to pipeline our fetch stage, splitting into instruction fetch from linebuffer and pre-decode in stage 1, and branch prediction and instruction queue enqueue in stage 2. This pipeline allowed us to push frequency to around 600 MHz.

Our next critical path was in the bulk decode/rename/dispatch logic, spanning from instruction

queue dequeue, through the decode and rename stages, and eventually being routed to the required reservation stations. We again considered pipelining this dispatch stage, but ultimately decided not to because of the potential hit to IPC. Instead, we were able to reduce this critical path by shrinking our reservation stations. This decreased our upstream dispatch logic as well as our downstream issue arbiter logic. We decided to go this route instead of pipelining because, when analyzing our scoreboarding across all benchmarks, we noticed that most of the reservation stations were never reaching capacity. As such, decreasing their sizes would likely have a small to negligible impact on overall performance.

Our final critical path was from our reservation station to certain execution units, through the arbiters and PRF. At first, we tried pipelining specific execution units. Once we would fix one, the critical path would just shift to a different execution unit. Looking at the timing reports, we noticed that the middle of the critical path was always on the output ports of our PRF. Therefore, we decided to convert our PRF from combinational reads to registered reads, placing a register directly in the center of our critical path and removing the scheduler to EU timing bottleneck.

With these three reductions, we were able to push our processor to around 614 MHz which resulted in significant improvements to our overall PD^4.

I. Miscellaneous Optimizations

Along with the aforementioned architectural features, we also made many smaller optimizations with the goal of optimizing PPA. One of these was sharing CDBs across multiple executions units. Our divider was implemented as a sequential divider with 14 cycles of latency, which meant that even at theoretical maximum usage, it would only CDB broadcast results once every 14 cycles. When adding the second ALU, we connected it to the divider's CDB and placed a priority arbiter to prioritize the occasional DIV broadcast. This greatly reduced our area because each additional CDB results in many additional ports on the ROB, RAT, PRF, and each reservation station.

Another smaller optimization we implemented was tuning the size of our ROB and the number of physical registers our processor had. Our baseline design used a 64-entry ROB and 64 physical registers, resulting in a maximum 32-entry free list. When scoreboarding our processor, we noticed that our ROB would rarely reach above half full, as it was draining at almost the same rate it was being filled. Additionally, flushes would cause the ROB to reset to empty. With this in mind, we reduced our ROB depth to 32-entries and our number of physical registers to 48 (maximum 16 free at a time). This reduction came with a large decrease in both area and power. Although this change had the potential to cause dispatch to stall due to the ROB being full or the free list being empty, the previous change to a 2-wide commit meant that we very rarely reached this scenario.

TABLE IV
BASELINE VS. ADVANCED IPC COMPARISON

Benchmark	Baseline	Advanced
coremark_im	0.442	0.584
aes_sha	0.457	0.432
compression	0.537	1.587
fft	0.636	0.707
image	0.358	0.506
mergesort	0.531	0.679

TABLE V
BASELINE VS. ADVANCED DELAY COMPARISON

Benchmark	Baseline (µS)	Advanced (µS)
coremark_im	6592.0	813.3
aes_sha	16971.0	2926.5
compression	8008.4	441.2
fft	18240.7	2672.8
image	13203.5	1520.6
mergesort	9080.6	1155.4

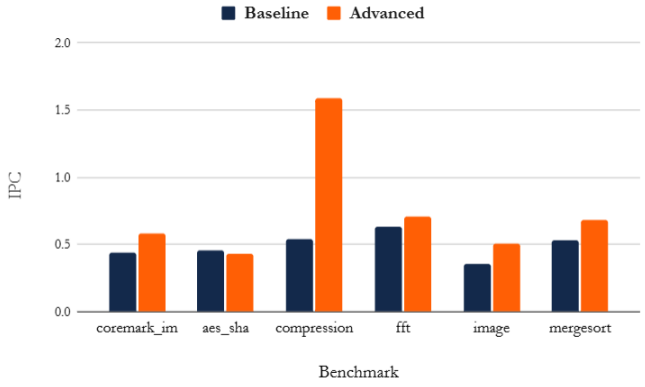

Fig. 2. IPC across released benchmarks on our baseline design and final design

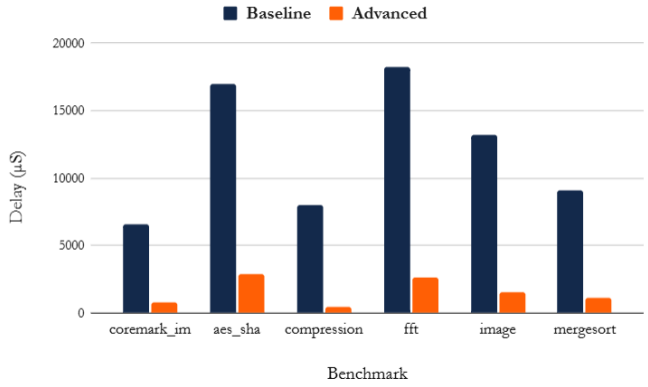

Fig. 3. Program delay across released benchmarks on our baseline design and final design

V. ADDITIONAL OBSERVATIONS

We noticed that our two biggest bottlenecks in our design were our data cache and our branch predictor. Given more

time, we would have done more design space exploration with the parameters in our GShare implementation. Because we didn't choose to implement full EBR, the programs with a large number of control instructions are even more reliant on strong branch predictor performance, one area where we were lacking. We did end up with about 13k μm^2 of free area below the 300k μm^2 limit, which could have gone to increasing the size of our PHT, allowing us to index into it with a larger GHR and PC. The other bottleneck that we weren't able to resolve was our data cache. We experimented with a pipelined data cache, but due to our load store queue arbitration logic, we weren't able to fully saturate the data cache with sequential hits to take advantage of the back-to-back hits that a pipelined cache provides. Additionally, a pipelined data cache increased our area significantly, so we decided not to use it. Given more time, we would have tried implementing a non-blocking data cache which can use MSHRs to support multiple outstanding requests and provides miss coalescing.

VI. Conclusion

Overall, we are very proud of what we were able to accomplish with *IllOOM*. We were able to implement most of the advanced features that we had set out to implement from the beginning, and optimizing our design taught us a lot about thinking outside the box to manage the tradeoff between performance, power, and area. Although we weren't able to achieve our goal of a top three finish, we were still happy with our final result of fifth, considering the high number of competitive processors this semester.

Acknowledgment

We would be remiss not to thank the 24-hour Circle K for keeping us energized during long nights spent at the ECEB. On a more serious note, we would like to thank our CA mentor, Pratyay Rudravaram, for guidance throughout the project. His input helped guide a large number of our architectural choices, and he was always available to answer our questions. We would also like to thank the ECE 411 course staff for their support throughout the semester, and for providing all of the tooling for mp_ooo. Finally, we would like to thank Professor Rakesh Kumar for a great experience in ECE 411.

References

[1] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun. 2015.