ECE411 Out of Order Processor Report

Maxwell Krieger
mk85@illinois.edu

Eddie Dzieza
edzie3@illinois.edu

Nicholas Costello
nc35@illinois.edu

I. Introduction

In this project, we designed and implemented an ERR-style RISC-V RV32IM out-of-order processor to explore fundamental computer architecture concepts such as instruction-level parallelism, dynamic scheduling, branch prediction, and memory hierarchy. To develop an advanced understanding of design considerations utilized in industry, we extended our 'baseline' design to explore a series of advanced features in an attempt to increase performance. Not only did this deepen our understanding of computer architecture generally, but it prepared us for future work within the field by developing key design intuition of trade-offs. This project was the culmination of knowledge presented in our academic careers thus far, marking a pivotal milestone and an excellent learning experience. The report is organized as follows: "Design Description" outlines the architecture and design goals, "Advanced Features" details the design and advanced features, "Contributions" describes contributions, and "Conclusion" summarizes key takeaways.

II. Project Overview

Our primary design goal was to reference existing literature and industry-standard techniques to guide our original architecture, helping us build relevant skills for future careers. We also aimed to explore advanced features and micro-optimizations to deepen our understanding of architectural trade-offs between power, area, and delay, along with increasing IPC. Furthermore, we heavily considered ways to reduce power and area to beat the competition metric PD^4. With these goals, we successfully exceeded baseline performance across all five benchmarks. To support collaboration, we used GitHub and the VS Code LiveShare extension, which allowed real-time co-editing and minimized merge conflicts. We followed a parallelized workflow, evenly dividing tasks among our three team members to accelerate progress. For details, see the "Contributions" section.

III. Design Description

Our baseline processor was developed over three main checkpoints. Checkpoint one focused on instruction fetch, including a line buffer for the multicycle cache, fetch stage logic, an arbiter for BMEM access, and a fetch-decode FIFO. In checkpoint two, we built the core of an out-of-order RV32IM processor, implementing stages like decode, dispatch/rename, issue, execute, writeback, and commit, along with key structures such as the register alias table, reorder buffer, free list, merged register file, and common data bus. We also integrated sequential multiply and divide units. Checkpoint three completed the baseline by adding branch resolution and flushing logic, as well as an in-order load-store unit to support all RV32IM instructions.

A. Checkpoint One

In checkpoint one, we established the processor's initial architecture and implemented the instruction fetch infrastructure. The fetch unit, adapted from mp_pipeline, was modified to interface with the line buffer, which improved fetch latency by staging data between the cache and fetch logic. Upon a cache hit, the instruction cache writes directly to the line buffer. A shared memory arbiter was also developed to route four-cycle memory bursts from BMEM to the instruction cache. Fetched instructions were placed into the fetch-decode FIFO to decouple it from the stall of the decode unit. The following section details these key components.

Fifo: To support buffering and flow control, we implemented a transparent parameterized FIFO as a circular queue. Data is written at the tail and read from the head, following FIFO order. Internally, it uses a register array with head and tail pointers extended by one bit to distinguish full and empty states. The FIFO is full when the head and tail match with opposite MSBs, and empty when equal. Flags are present to prevent overflow and underflow. If the read enable is asserted and the FIFO is not empty, the head increments. If write enable is asserted and the FIFO is not full, data is written and the tail increments. The output, fifo_out, continuously reflects the head and is valid when fifo_empty is low. In checkpoint one, the FIFO is used for pending instructions in the fetch-decode queue.

Fetch Unit: The fetch unit supplies instructions to the processor's frontend by managing the program counter (PC), requesting instructions from memory, coordinating with a line buffer, and generating fetch packets for the instruction queue. On reset, the PC is initialized to 0xAAAAA000. When the fetch-decode FIFO is full, the unit stalls to avoid unnecessary fetches. If the line buffer contains the needed cache line, the instruction is extracted using a bit offset from the lowest five bits of PC, and the PC is incremented. If this increment crosses a cache line boundary, a memory read is triggered. If no valid memory response is available and the line buffer misses, the fetch unit issues a read request and holds state. When a valid instruction is returned from memory, the instruction and PC are inserted into the fetch-decode FIFO, and the PC is updated.

Cache Line Adapter: The adapter acts as an interface between burst memory and both the instruction and data caches, managing 32-byte cache line transfers using a simple FSM and burst protocol. It handles both reads and writes by splitting or assembling data into four 64-bit chunks. Although the data cache was not connected in this checkpoint, the module is designed to prioritize data cache requests over instruction cache requests, which was added in checkpoint two. Later, in advanced features, the cache line adapter also arbitrates the prefetcher memory requests, which have the lowest priority.

Line Buffer: The line buffer temporarily stores the most recently fetched instruction cache line. This buffer reduces fetch latency and arbiter traffic by allowing next-cycle response on instruction requests. To support the linebuffer, the instruction cache from mp_cache feeds the most recently accessed instruction line into the linebuffer, allowing fast reuse. This optimization reduces cache port usage and fetch latency for instructions within the same cache line.

B. Checkpoint Two

Checkpoint two established the foundation of our RV32IM processor. This included the stages needed to execute register and immediate instructions, as well as the proper explicit register renaming structures. IP integration was also explored as sequential multiplication and division units were inserted. Below describes the key components and implementation for this checkpoint.

Decode: Following the fetch stage of our processor is the decode stage. The decode stage observes the instruction in the fetch-decode FIFO and decodes it into signals to be used and referenced in later stages of the processor. This occurs only if the fetch-decode FIFO is not empty and if the FIFO that is decoupling decode and dispatch/rename is not full. The decode stage logic will input the desired signals to execute an instruction in the decode-dispatch/rename FIFO. These signals are instruction-dependent but include information such as opcode, rs1/rs2/rd addresses, immediate values, ALU operation selection, ALU inputs, signed multiplication or division, and return types such as the upper or lower 32 bits of multiplication or the division remainder.

Dispatch-Rename: The dispatch/rename stage is what allows instructions to become in flight. It will take a valid instruction from the decode-dispatch/rename FIFO and notify the proper structures to support the execution of the instruction. These structures include the register alias table (RAT), free list, and reorder buffer (ROB), which are explained below.

Register Alias Table: The RAT is what allows false dependencies (WAW and WAR) to be eliminated. The RAT has 32 entries, with each index representing an architectural register. Every entry contains a physical register mapping along with a ready bit to indicate if the merged register file holds the desired value. Our RAT initializes each architectural register to ready and maps it to physical register zero in order to hard-wire the data of each register to zero. When a register is renamed, its physical address will be updated, and it will be marked as not ready. If, instead, an instruction is written back on the common data bus and the physical address at the architectural register index matches, it will be marked as ready. Priority is always given to rename over writeback, as renamed registers are what the following instructions are dependent on.

Free List: The freelist holds a reference to all physical registers and indicates if they are currently mapped. This is done through a 64-bit vector where each register holds their own bit. A priority mux is used to decipher the free-list, indicating the first free physical register (observes the first 0 bit) and returns it to dispatch/rename for possible register renaming. It is important to note that the free list returning physical register zero represented that there were no free registers. This is because physical register zero was allocated for x0 at all times. Dispatch/rename will notify the freelist if it uses the provided free physical register, and the free list will then mark it as in use. If the retired register alias table kicks a physical register, it will then be marked as free again.

Reorder Buffer: The ROB enables instructions to commit in order by acting as a circular FIFO. Instructions are enqueued to the ROB via dispatch/rename with instruction information, including their destination and physical register addresses. Once instructions are written back on the common data bus, they will be marked as done in the ROB, enabling them to be committed.

The dispatch/rename stage uses these structures to dispatch decoded instructions to the proper issue queue (ALU, MUL, or DIV) based on decoded signals. Dispatch occurs only if the target issue queue, ROB, and free list (for instructions with destination registers) are not full; otherwise, the stage stalls. The process starts by identifying the instruction type and the issue queue from the opcode. If rs1 or rs2 are used, their renamed physical addresses and readiness are read from the RAT and checked against the common data bus for writeback updates. If all required registers are ready, the instruction is marked ready; if not, it is marked not ready. If the instruction writes to a destination register, a free physical register is allocated from the free list, and the RAT and free list are updated accordingly. Lastly, the instruction is enqueued to the ROB, and all information needed for execution and writeback is sent to the corresponding issue queue.

Issue Stage: To reduce the critical path of our design, since out-of-order cores are typically bound by their issue-stage logic, we aimed to optimize this path early to avoid later restructuring. To accomplish this, the issue stage is divided into four key components: a priority mux, issue queue, priority decoder, and Physical Register File (PRF). A formal description of each component can be seen below.

Priority Mux: The priority mux handles the arbitration of requests from dispatch/rename to the issue queues in the issue stage. Internally, this unit holds an array of bits corresponding to the status of each of the issue queue entries. An asserted bit in the priority mux array for an entry denotes it as ‘free’, while a deassertion means it is ‘occupied’. Combinationally, the priority mux will output the index of a free entry based on its index in the issue queue. Lower indices are given priority as this offers the quickest arbitration logic for this stage. If all entries are occupied, the priority mux will assert its full signal, restricting incoming requests to this queue. When an instruction leaves the issue queue, its corresponding index in the priority mux array will be freed for future issuing.

Issue Queue: The issue queue is a table structure that houses all incoming instructions from the dispatch/rename stage. In this checkpoint, there are three total issue queues for each of the possible instruction types: ALU, MUL, and DIV. If the priority mux selects an entry, the instruction will be loaded into the queue at the specified index. Once in the queue, every instruction will probe the bus as part of its 'wakeup' condition. Once an instruction has been woken up, it will send its status to the priority decoder, allowing the decoder to arbitrate between issue requests and forward them to the execute stage. If the instruction has been selected by the priority decoder for execution, the instruction in the queue will be invalidated, and a request to the PRF will be made for its operands' data.

Priority Decoder: In a similar structure to the priority mux, this unit holds an array of bits specifying which instruction from the issue queue is ready. If the bit for a specific issue queue index is asserted, that instruction is assumed to be 'ready'. The inverse is also true for deasserted values. Reminiscent of the priority mux, arbitration of incoming ready requests is done based on indices in the issue queue, where lower indices are given priority. If the execution lane for the corresponding instruction type is not stalled, and if there is a ready instruction, it will be forwarded to the execute unit. Simultaneously, the priority decoder will broadcast the index it removed for coherency across the system. The decoder will also clear the corresponding index in its bit array to mark this instruction as not 'ready', as it has been forwarded to the next stage.

Physical Register File: As part of the explicit register renaming (ERR) microarchitecture, a merged physical register file was introduced in our design that is separate from the ROB. Naturally, we chose to implement the 'read-after-issue' structure for register reads as this allows for smaller widths in our issue queues. Due to this, register data is not stored in the issue queues themselves, but instead read out of the PRF when it is forwarded to the execute stage. The PRF consists of sixty-four 32-bit registers, each of which is addressable by a 6-bit physical address. To handle incoming read and write requests, the PRF has seven input ports. Six ports are 'read-only' for the two possible operands of the issue stage, while the final port is 'write-only' and takes incoming write requests from the common data bus (CDB). Every clock cycle, the PRF will check if it has received a broadcast from the CDB, updating the data at the corresponding register if it has. It will also check if it is receiving a read request from the issue queues, forwarding the data at the corresponding physical address to the execute stage if applicable.

Execute: The execute stage computes arithmetic, logical, and AUIPC instructions. The design is comprised of three different computational units: one for ALU, MUL, and DIV instructions. The execute stage registers the outputs of issue and inputs to writeback. In addition to these registers, there are registers to facilitate temporary data storage and stall handling. Multiplication and division are done through the use of Synopsys sequential IPs. To facilitate this, a register is used to track when a multiplication or division operation is in progress. These signals affect the stall logic being sent to the issue stage. The execute stage receives stall signals for ALU, MUL, and DIV from the arbiter in writeback if there is contention. When a stall is asserted, the associated data and unit are stalled and stored. While stalling, or a unit is actively being used to execute an instruction, a stall signal is fed back to the issue queues in the issue stage. To further clarify the interaction between the units, view Fig. 1.

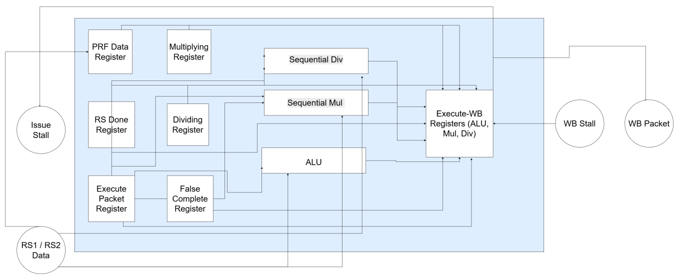

Fig. 1. Interaction between the units

Writeback and Common Data Bus: The writeback stage notifies the processor when an instruction has completed execution. It arbitrates between the ALU, MUL, and DIV units to broadcast the completed instruction using the common data bus. Since there is only one bus, priority is given to DIV, then MUL, then ALU, giving preference to higher-latency instructions. The common data bus connects to multiple structures and stages, including dispatch/rename, issue queues, the ROB, and the RAT. It broadcasts the completed instruction's destination register value, its physical and architectural register index, and the instruction's ROB index, allowing the rest of the processor to update accordingly.

Commit and Retired Register Alias Table: The commit stage ensures instructions are committed in order. If the instruction at the head of the ROB is marked as done, it is dequeued and committed. During this process, the commit stage also updates the Retired Register Alias Table (RRAT) to maintain the architectural state of the processor. The RRAT stores the current architectural register mappings. When an instruction that writes to a destination register commits, its physical register index is written to the RRAT. The old physical register previously mapped is then released and returned to the free list.

C. Checkpoint Three

As part of this checkpoint, we extended upon the framework of checkpoint two to support memory (load/store) and control (branch, jal, jalr) operations. Due to a high degree of parameterization and modularity in checkpoint two, much of the existing codebase was reused in completing this checkpoint. For each of the new instruction types, they flow through the fetch, decode, and dispatch/rename pipeline as normal, but follow their own issue and execution paths. Below, a detailed description of the infrastructure and design considerations for implementing these instruction types is presented.

Load-Store Unit: To implement the Load Store Unit (LSU), we first modified the decode stage by adding cases for load and store opcodes, handling each of their operand requirements. In the dispatch/rename stage, an additional dispatch lane was added to send memory instructions to a unified load-store queue (LSQ) in the issue stage. We used a naive, in-order LSQ design where all instructions snoop the bus for their non-ready operands. Once in the queue, a load can issue if ready, and a store can issue only if it is ready and at the head of the ROB. Once issued, it proceeds to its address calculation stage, which computes its address and memory mask. To decouple memory requests and improve scalability, we added a second LSQ between the address calculation stage and the memory unit. An instruction is dequeued from this queue only if the memory unit is free, the queue is non-empty, and if the instruction is a store, it must be at the head of the ROB. In the memory unit, slightly different processes are conducted for load and store instructions. Both instruction types forward their computed address and mask to the external memory port and wait for a DMEM response. Stores forward their packet to writeback with no destination register, while loads extract and forward the loaded value per the RISC-V spec. At this stage, we used a single writeback bus, prioritizing memory (the longest-latency instruction) at the top of the priority mux. Existing stall logic from checkpoint two was reused to block other units during bus contention.

Control Operations: In supporting control operations (branches, jal, and jalr), a high degree of reuse from checkpoint two was leveraged. To generate a solid baseline for later branch extensions, our team opted for a static not-taken branch prediction model. Similar to the other instructions in our pipeline, control instructions follow an identical path from the fetch, decode, and dispatch/rename stages, but with new arbitration lanes for the different types. These instructions are also issued out-of-order, utilizing a unified 'control' issue-queue that houses support for all pending control operations. Once an instruction enters the execute stage, it will follow the same flow as normal ALU instructions, but will instead utilize a 'special' ALU to handle the edge cases of control operations. This unit will uniquely calculate the branch PC for the incoming instruction and simultaneously calculate the return value for jal and jalr instructions (if applicable). Once completed, the branch requests are forwarded to the writeback unit, where they are given second priority on the bus. When a branch instruction becomes the head of the ROB, its branch result will be seen by the CPU at commit. If any of the control operations have a result of 'taken', a flush signal is broadcast to the rest of the CPU since the branch is mispredicted. To every unit in our CPU, we logically OR a branch reset signal 'br_rst' with our normal reset, completely flushing all in-flight instructions. ROB entries beyond this point are also flushed by moving the tail of the ROB one index past the head index.

IV. Advanced Features

In our RISC-V RV32IM processor, we implemented an SRAM Gshare, a split load-store queue, early branch recovery, a return address stack, a next-line prefetcher, pipelined cache, and age order issuing. Our final design for the competition included all features except for the next-line prefetcher and age-order issuing. The following sections will discuss the design of each feature. Please refer to Fig. 6 for PD^4 and Fig. 7 for IPC on the given benchmarks. Features were built on top of each other, meaning the (power) * (delay)^4 will reflect that unless the feature did not make the final design (next-line prefetcher and age order issuing). The dotted red line on each plot shows the ECE411 baseline that our final design was attempting to be lower than. This was achieved on every test. It is important to note that we did not take power-saving metrics (such as clock gating) into consideration until the addition of EBR. Therefore, for earlier features, the improvement in IPC is more significant than the potential increase in PD^4. SRAM GShareTo start, we implemented a Gshare branch predictor using a 7-bit Global History Register (GHR) and a 128-entry Pattern History Table (PHT) stored in dual-ported SRAM. The predictor uses an XOR-based indexing scheme that combines segments of the program counter (PC) and the GHR to access the PHT. The specific equation used for indexing is: index = (PC[31:25] xor PC[24:18] xor PC[17:11] xor PC[10:4] xor global_history_register). On average, this indexing equation is the one we found to reduce aliasing the most. The pattern history table stores 2-bit saturating counters, which are updated based on the actual branch outcome and the original prediction value fed from the ROB (with early branch recovery, it comes from the branch's writeback bus). Because of the one cycle SRAM latency, fetch initiates a read request with the next PC value to the SRAM so that the appropriate prediction is read from the PHT on the following clock cycle. This means if the current fetched instruction is a branch instruction, then it will use the read prediction value from the SRAM to determine the next PC value. If the leading bit is one, it is predicted to be taken and sets PC to be PC + branch_immediate value on the next cycle. Otherwise, the branch is predicted to be not taken and updates PC normally to PC + 4. Lastly, we initialized the 2-bit saturating counters to be weakly not taken ('01) as this produced the most accurate branch prediction on average.

Testing was conducted using both small branch test files and larger benchmark programs. The small tests were specifically designed to exercise all four states of the 2-bit counter: strongly taken, weakly taken, strongly not taken, and weakly not taken. Larger benchmarks, such as CoreMark, were used to validate functionality at scale. Verification was done through waveform analysis to confirm correct state transitions, SRAM interfacing, and index generation.

Performance was measured using a SystemVerilog-based tracker that logs total committed branches and mispredictions. Prediction accuracy served as a direct reflection of how effectively the design captured and tracked branch history-based behavior.

Gshare increased area by 15,000µm² and power by 20mW. GShare led to improvement in IPC amongst all tests, as seen in Fig. 7, with the most noticeable increase of 0.08 in Compression. Dual-port SRAM allowed for simultaneous reads and writes, improving throughput. The combined GHR and PC hashing method provided better index distribution and reduced aliasing compared to simpler PC-only indexing. The bar graph in Fig. 2 shows the accuracy improvement after integrating Gshare into our design. Gating was later addressed in future versions of our advanced features, making the increase in power negligible.

GShare is less effective in situations where multiple branches have similar PC values. This is because the XOR-ing has led to uncorrelated predictor entries being aliased. Furthermore, in programs with frequent alternating branch outcomes, XORing may fail to distinguish them, leading to consistent mispredictions. A workload that GShare would perform extremely well in is a CSV text parser. A text parser will often loop over characters and use conditional branches to check for commas, quotes, or brackets. The outcome of each branch depends on previous characters, allowing for the global history register to successfully capture useful patterns. In programs where the logic is consistent and repeated, GShare can learn and predict the correct paths with extremely high accuracy.

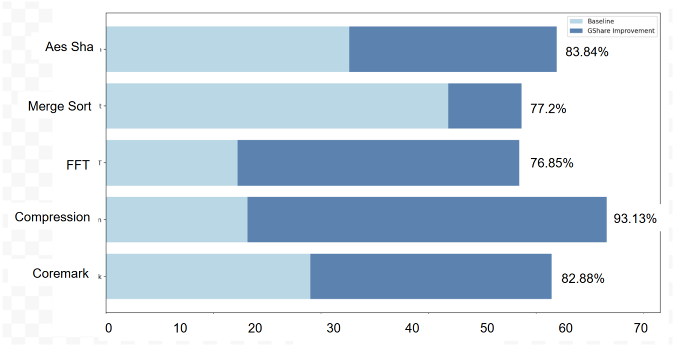

Fig. 2. Baseline vs GShare branch prediction accuracy

Split Load-Store Queue: As there is a large disparity between load and store instructions across every benchmark, we implemented out-of-order loads between in-order stores to gain a potential speedup. In integrating this feature, much of the pre-existing infrastructure of the load-store unit was repurposed. The difference between the new LSU and the previous one was a decoupling of loads and stores from the combined LSQ of the previous checkpoint.

In decoupling these instruction types, the issue stage saw a new issue queue for load instructions, while stores still maintained the same FIFO structure. When dispatched to the LSU, loads now followed their own execution path, having both an address calculation module and a final holding queue for memory. Once inside the final queue, load instructions are only sent to memory if their store tag has been broadcasted, while stores are sent if they are at the head of the ROB. If a store instruction is at the head of the ROB, it is given arbitration priority in the mem unit for execution.

To allow loads to fire between in-order stores, each store instruction is given a unique 'tag' once it enters the dispatch/rename stage. In a similar structure to the 'freelist', this tag is a binary number that is assigned to each store instruction. When a load instruction passes into the dispatch/rename stage, it is assigned the tag of the most recent store instruction, binding it to this instruction. When a store instruction finishes executing in the memory unit, it will broadcast this tag to the load-store unit, the load issue queue, and the dispatch/rename stage, effectively waking up pending load instructions.

To initially test the split LSQ design, directed RISC-V test programs were utilized. This tested specific edge cases in out-of-order load requests, enabling strict verification of this feature's baseline. Once the baseline was established, the five provided benchmark suites were leveraged to conduct application testing. This revealed bugs in the unit's integration with our existing framework, which found other edge cases in the design that were later fixed with waveform debugging.

Through program counters, we identified that each of the five benchmarks had significantly more load instructions than store instructions, meaning decoupled issuing of load instructions would naturally lend a performance increase. For memory-intensive benchmark programs, such as FFT and AES_SHA, the split LSQ proved to be a solid advanced feature that saw marginal improvements. More specifically, AES_SHA had a dynamic total of 299,972 memory operations, with 67.06% of them being loads. As this benchmark had the largest disparity of the five, it saw an IPC improvement of 0.027 with this feature, leading to a noticeable performance improvement. Paired with the IPC improvement, the delay also decreased by 8.36%, decreasing our PD^4 value significantly as power remained constant.

In general, our core received performance improvements across the board with the inclusion of the Split-LSQ, but specific improvements in IPC, delay, and power consumption in these benchmarks can be seen in Fig. 6 and 7. Workloads with a lack of nested loads between store instructions would not see much of an improvement since they would be tightly bound to the in-order stores. If there is a significant amount of loads between stores, even larger improvements could be seen with this feature.

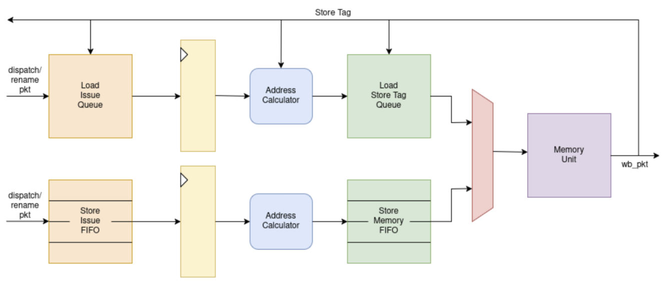

Fig. 3. Split load store queue data path

Early Branch Recovery: Next, we decided to implement early branch recovery (EBR) as branches would get stalled in the ROB behind unfinished instructions, leading to a large increase in delay with mispredicted branches. Early branch recovery is optimal in reducing latency and increasing IPC for programs regularly encountering this case. This is because it allows mispredicted branches to flush dependent instructions on writeback, allowing the processor to be filled with the proper instructions at a faster rate. This would not be good for programs with little to no branches as it consumes more area and power than it helps with delay. However, it is generally a great feature to have as branches typically construct about 20% of programs, and all of the five provided benchmarks are very branch-heavy.

In order to implement EBR, both checkpointing and branch stack masking were required. To begin, a branch mask register was added to the dispatch/rename stage to account for all in-flight branches. This was a three-bit register, meaning that only three branches could be in the issue and execute stages at a given time. Each time a branch instruction is dispatched, it is tagged with an available index slot in the branch mask. This available tag index is deciphered by means of a priority mux. This index will also be tagged (set to '1) in the branch mask on the next clock cycle to notify all subsequent instructions that they are dependent on this branch. Furthermore, every instruction that is dispatched will also receive a copy of the current branch mask to observe each branch instruction it is dependent on.

In addition to the branch mask, the branch stack was implemented to achieve checkpointing. Checkpointing is saving the current state of the processor in case a flush needs to occur. When a branch is first dispatched, it will screenshot the processor's current state into the stack index correlating to its branch mask tag. Upon initialization, the freelist, RAT, ROB tail, current store tag, pre-address calculator store fifo tail, and post-address calculator store fifo tail were saved. However, because older instructions can be written back before a younger branch instruction, or because data structures dependent on older instructions could change, some of these checkpoints needed to be updated. To begin, if the common data bus broadcasts an instruction that writes to a destination register, all of the RATs in branch stack indices that the instruction is not dependent on, determined by the branch mask of the instruction at that index being zero, may be updated. This is only if the physical address broadcasted at the architectural register matches the one in the stack entries RAT, then the ready bit must be set. Along with that, if a register is kicked from the RRAT, it must be marked as free in every stack entry's free list. Similarly, if a store occurs, it must be marked as free in each stack entry's store tag list. This is because they are not dependent on any branch as they have committed. Lastly, the store FIFOs before and after address calculation were the most difficult to checkpoint. This is because particular update logic was required in case older instructions moved from one queue to another before a branch was written back. In order to update the pre-address calculation store FIFO checkpoint, it was tracked if the head index ever equaled the checkpointed tail. If this were the case, the stack would hold a flush bit notifying to empty this FIFO upon a branch misprediction, as no older stores are located in this FIFO. In contrast, the checkpointed post address calculation store FIFO tail would be incremented for a stack index if a store not dependent on that branch index was inserted into this FIFO. This is because it is an older instruction that shouldn't be flushed.

The last major change to implement EBR was the branch writeback logic. When a branch is written back, its branch tag would be freed from the branch mask along with its branch stack entry being cleared on the next cycle. For correctly predicted branches, all instructions in the execute stage and issue queues would have their branch mask copies cleared of this branch tag to notify that they are no longer dependent on it. Next, if a branch was mispredicted, it must now flush the processor of all younger instructions. This means that any instructions that are in the issue or execute stages with a branch mask dependent on the written back branch tag must be flushed. Along with that, the branch stack is read to set the appropriate structures to their checkpointed values. The flushing logic for the front half of the processor remained the same. It is important to note that jal instructions were not tagged as they had a 100% prediction accuracy.

In order to test EBR, we began with the random testbench to ensure ideal behavior still occurred because the logic for multiple different stages and structures was edited. Next, we tested the same basic branch tests used in checkpoint three to make sure simple branches worked. We also monitored Verdi to see if early flushing occurred. Then, we used test cases with branching and loads to test out-of-order branch flushing. Lastly, we ran the benchmarks to test EBR across a larger variety of cases.

EBR yielded a significant positive contribution to the performance of our processor. Clock gating was added at the same time as this feature, so power was reduced significantly. However, EBR does consume extra power even after the additional gating. The main contribution of EBR was decreasing delay and increasing IPC, and it did so significantly. Across the five benchmarks, delay decreased anywhere from 500 microseconds to 2000 microseconds while IPC increased anywhere from .07 to .27. On average, IPC increased by .17 and delay decreased by 646 microseconds. This led to an overall decrease in PD^4, providing a boost to the processor's performance.

Return Address Stack: Following EBR, we decided to implement a RAS as we had zero percent accuracy on jalr instructions. This was a big bottleneck as it meant every time we encountered a jalr instruction, we would have to partially flush our processor, which significantly increases the program delay every time a jalr is seen.

The RAS is a part of our fetch stage and is a circular stack of depth four, with each entry containing a valid bit and a desired return address. To predict jalr instructions, the RAS uses RISC-V conventions for calling and returning from functions. This means on function calls, which are indicated by a jal or jalr instruction with destination register x1 (ra), PC + 4 is pushed to the top of the RAS as this is where the called function returns. On return instructions, which are a jalr instruction with x1 as the rs1, the return address is popped off of the stack to allow the PC to become that value. Being a circular stack, older entries can be overwritten if it gets filled; however, this is more effective than not pushing on newer instructions, as those will be popped first. It is important to note that by RISC-V convention, the same functionality can be carried out with x5 instead of x1, but we saw no improvement on the benchmarks so it was not included. Lastly, to account for early branch recovery, we would checkpoint the top stack entry along with the stack pointer. In order to test our RAS, we made simple function call test cases to see if the desired behavior was achieved in Verdi. Then, we ran it on the benchmarks to check accuracy and for any bugs that occurred.

The first measurement we accounted for was the accuracy of our jalr predictions. We began with a RAS of depth thirty-two and slowly shrunk it down until we received a similar accuracy with a RAS of depth four, as seen in Fig. 4 to reduce the negative effects of power and area. The RAS accomplished an average accuracy of 48%, which is significantly better than the previous 0% accuracy. The delay was improved as it decreased on average by 51 microseconds, and IPC improved on average by .01. The power increased to 35 mW, leading to an overall increase in PD^4.

The RAS is good for workloads that contain a lot of function calls, but not those that are super deep, as return addresses will be overwritten. Programs which do not have many jalr's, such as Compression with five, are also negatively affected by the RAS as it consumes extra power and leads to almost no delay improvement.

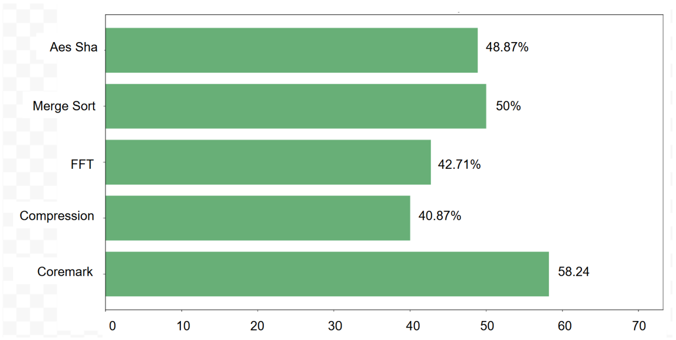

Fig. 4. Prediction accuracy of return address stack

Next-line Prefetcher: Thereafter, we implemented a next-line prefetcher to reduce cache miss penalties and to improve instruction throughput by using a single 32-byte register, which triggered prefetching of the next line on every cache miss. After a cache miss, a read signal would be asserted, causing the arbiter to begin fetching the next line if no instruction or data cache accesses were in progress. Initial testing was done by attaching the prefetcher to the minimal fetch stage (no branches) from checkpoint one. It was later integrated into the checkpoint three fetch.

In the set of provided tests, this feature helped improve IPC across most benchmarks, as shown in Fig. 7. The most significant improvement was noted on Coremark with an increase of 0.02, where the prefetcher was successfully utilized for 19,308 accesses. However, this improvement came at a significant cost: power consumption increased by approximately 20 mW, which negatively impacted our overall energy efficiency metric, notably degrading the PD^4 measure as seen in Fig. 6. While the next-line prefetcher demonstrated potential for reducing delay, the power overhead and limited effectiveness in certain workloads ultimately made it impractical for our final design.

Since the prefetcher was assigned the lowest priority in the cache port arbitration scheme, it was often starved of access in load/store-heavy applications. It also performed poorly in branch-heavy code, where frequent control flow changes invalidated many prefetches before they could be used. The prefetcher would prove more effective in workloads with a significant amount of spatial parallelism, along with minimal branches. Examples of this include matrix and gpu operations.

Pipelined Cache: With the naive mp_cache implementation, there is an intrinsic one-cycle delay between each load-store request. Since load operations have the longest latency compared to other instructions, we implemented a pipelined data cache to improve the amortized access to memory.

In implementing the pipelined cache, the specification of mp_cache_pp was followed. This led to the creation of a two-stage pipelined cache, supporting the coalescing of WAW, WAR, and RAR requests in back-to-back cycles, but a one-cycle delay for RAW hazards. The pipelined cache remained the same internal configuration as that of mp_cache, having four ways, sixteen sets, and 32-byte cache lines. Each cycle, an incoming request is latched by a pipeline register, allowing it to be serviced by the memory unit in the second stage. To determine the hit, writeback, and allocation conditions, a similar state machine to that of mp_cache was utilized. Instead of a traditional 'Idle' state, the base state is the 'compare' state, performing a tag comparison only if there is a new request or if there was previously an allocation. Despite the simple and familiar structure, additional considerations need to be taken to allow for coalesced requests.

The most notable modification was the inclusion of dual-ported PLRU, dirty, and valid arrays. Since flip-flop array reads and writes have a one-cycle latency, this is crucial to ensure that a request in the current stage can finish while we are simultaneously priming an incoming request. While this is necessary for requests of different set addresses, forwarding is crucial if these requests are coming from the same set address. By forwarding the PLRU, dirty, and valid data if the incoming request is from the same set, we ensure that we do not need to stall between requests unless it is a true hazard, RAW.

To receive the maximum throughput increase offered by a pipelined cache, our LSU needed some additional modifications. Generating coalesced responses requires coalesced requests, which were not necessary in the naive cache model since requests required a one-cycle delay inherently. To add this support, the request to the memory unit in our LSU now contained a dynamic input instead of our traditional load-store arbitration. If the memory was not stalling, a new request would always be sent. Simultaneously, the request is registered for use if memory is stalling. If memory is stalling, the registered input will be fed into the memory unit, allowing a new request to occupy the 'raw' input. Not only does this ensure that the current request is always held constant until serviced, but it allows a back-to-back memory operation to occur when memory does eventually respond. This enables pipelined requests of all valid types, allowing our core to receive the maximum speedup from our pipelined cache architecture.

To test this design, the same testbenches utilized in mp_cache were used. Slight modifications were made to ensure the incoming memory model matched that of the mp_cache_pp specification, specifically coalescing requests. Once this was established, the remainder of the testing was done against the five given benchmarks and a working base implementation.

As seen from Fig. 5, performance counting supports that many cycles were saved in programs with high memory utilization, such as FFT and AES_SHA. While the IPC and delay improvements can be seen in Fig. 6 and Fig. 7, the most notable improvement in IPC was in FFT, with approximately a 0.03 IPC improvement and the saving of 86,738 cycles. Despite this being the benchmark with the most notable improvement, an average IPC improvement of 0.022 has been achieved across the five benchmarks. This greatly improved our PD^4 value across the suites, enabling us to achieve an average delay decrease of 3.41% at the same power. Overall, this advanced feature greatly improved our performance across the provided benchmarks, allowing us to safely surpass the performance of the baseline core.

Workloads where these improvements are particularly noticeable are ones with a large number of back-to-back load/store operations. If the requests are streamlined, this means we can issue a new request nearly every cycle, so long as it is not a read-after-write request. On the other hand, programs with sporadic load/store requests throughout the program will not see any benefit with this data cache since requests are not coming in back-to-back. Similarly, if a program is more compute-bound than memory-bound, such as performing many ALU, multiply, or divide instructions, the improvements with this data cache will not be seen.

Age Order Issuing: As mentioned in the literature, age-order issuing can save a large amount of delay because older instructions tend to contribute the most to dependencies in a program. To explore the potential speedup offered by a traditionally 'better' issue structure, we decided to implement this feature in our design. As will be shown below, the increase in critical path to the issue stage brought by this feature made it unsustainable for our final design, thus we opted not to include it. To implement age order issuing, much of the logic from the issue stage was repurposed. Instead of a traditional priority decoder, modifications were made inside this unit to enforce the age-order structure. Upon receiving a new request, the priority mux will forward the issue queue index it placed the instruction into to the age-order decoder. The decoder will then push this address to an internal FIFO, maintaining a strict order of when each instruction was issued for the corresponding issue queue. Note that the priority mux and issue queue logic are the same, including the wakeup logic. Every cycle, the priority decoder will check the ready bits of the instructions placed in the FIFO from head to tail, issuing based on age priority. When an instruction is ready for execution, its address will be sent back to the issue queue and priority mux to maintain coherency. Inside the age-order FIFO, it will invalidate the entries that have been sent to execute, collapsing on that index to enforce a unified structure. This process is enforced for every entry and has additional support for early branch recovery invalidations.

To test this unit, additional RISC-V-directed test programs were developed to ensure base functionality. Since the design would simply break if instructions were not being issued properly, this verified that directed tests would be effective for most cases. Once the baseline was developed, additional testing was performed on the five benchmarks, which were completed with little friction.

As seen from Fig. 6, the PD^4 characteristic for this model proved very ineffective. Since the issue queue now had to reorder a queue every time an instruction was issued, compared to clearing a single bit, this drastically shifted the critical path of our design to this unit. If we aimed to target a higher frequency than 476.19 MHz, this may have been a better advanced feature since our issue stage would have more pipelining and scalability. As this was not the case, the baseline frequency this feature allowed us to target was 376.19 MHz. Additionally, since our queue depths were sixteen for both ALU and load/store instructions, and three for the remaining instructions, the benefits of an age-order structure were not seen in our design. As seen in Fig. 7, the IPC changes were very insignificant, leading to the same IPC at a much worse delay. If the ROB depth of our design was much larger (> 32), there may have been more benefits in programs that have a high degree of dependencies, which was not seen in our case.

V. Additional Observations and Features

Attempted Third CDB for Pipelined Multiply Support: To improve performance on programs with heavy multiply usage, such as the FFT test, we experimented with adding a third CDB to the processor's backend. This additional CDB was designed to prioritize multiplication operations, aiming to better support a pipelined MUL unit and reduce execution bottlenecks. However, while the architectural change was functional, it came with significant trade-offs. It resulted in increased area and power consumption, due to added arbitration logic and bus routing complexity, but no observable IPC improvement, even in MUL-intensive tests like FFT. The existing scheduling

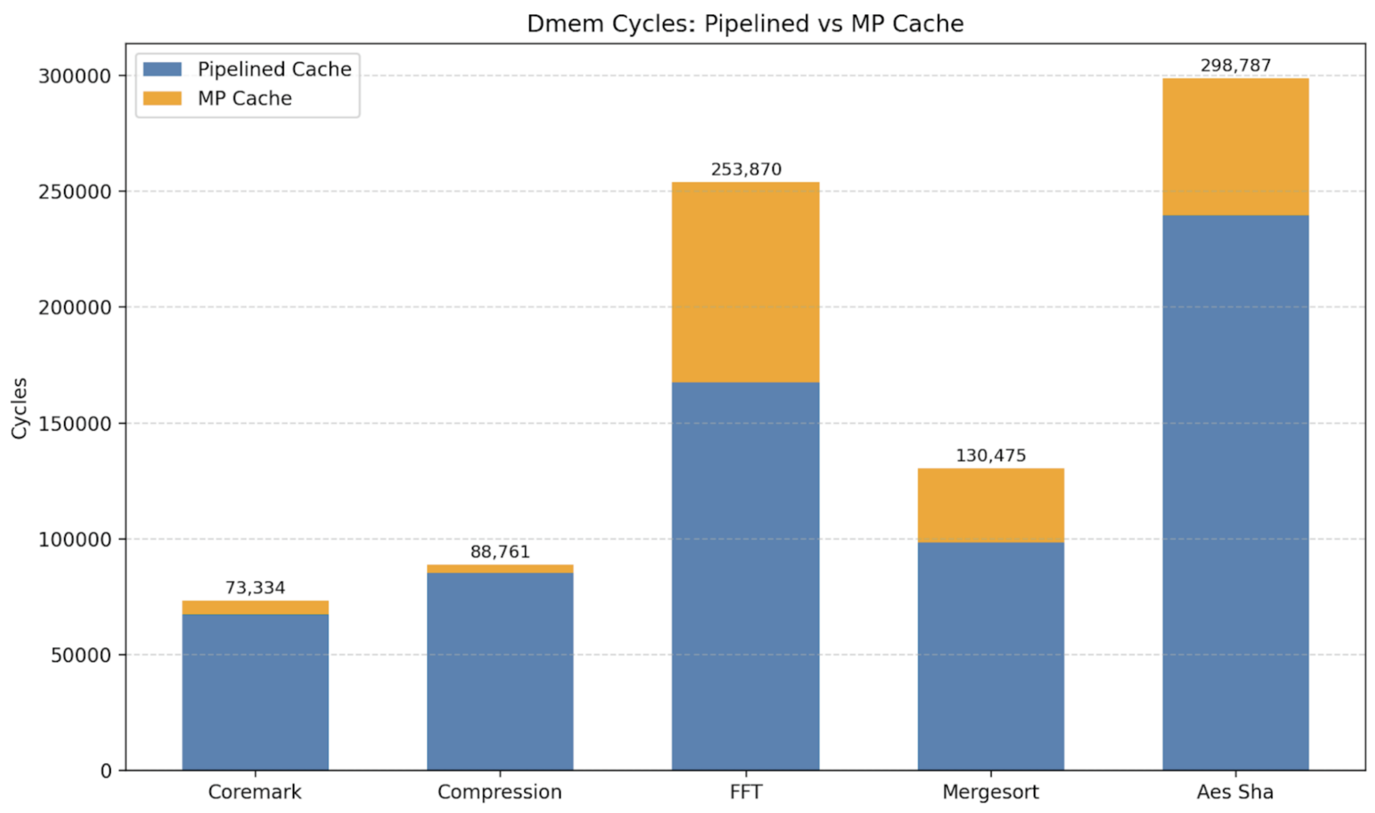

Fig. 5. Visualizing the Number of Cycles Saved between each new dmem Response

and execution logic already utilizes available CDB bandwidth effectively in most cases.

Jal Pre-Computed in Fetch: To improve the accuracy of jal instructions and increase IPC, the jump address for jal was pre-computed in the fetch stage. This required adding partial decode logic in fetch to correctly set the PC for jal instructions, since jal is an unconditional branch with a target address computed as PC + jal_imm. This optimization had a negligible impact on power and area, requiring only an additional multiplexer.

VI. Contributions

This project was a collaborative effort, with each team member contributing significantly to different components of the processor design. Max was responsible for implementing the fetch stage, the next-line prefetcher, the Gshare branch predictor, and the initial memory unit. He also developed the execute stage and assisted with the design and debugging of the split load/store queue (LSQ) memory unit. Nick focused on the remainder of the frontend, including instruction decode, rename, and dispatch logic. He also designed and implemented early branch recovery, which included branch masking and checkpointing logic, the RAS, and helped contribute to the Gshare branch predictor. Eddie developed the split LSQ memory unit, designed and implemented the pipelined cache, and added age-order instruction issue logic. He also designed the issue stage and the physical register file (PRF). In addition to their primary responsibilities, all team members contributed to debugging and system-wide integration efforts throughout the project.

VII. Conclusion

We successfully designed and implemented an ERR-style RISC-V RV32IM out-of-order processor, achieving our objectives of exploring advanced computer architecture concepts and exceeding baseline performance on all benchmarks. Notable features include a Gshare branch predictor, split load/store queues, early branch recovery with checkpointing, a next-line prefetcher, a pipelined cache, a return address stack, and age-based instruction issuing. These features enhanced performance, accuracy, and architectural sophistication, demonstrating effective application of industry-standard design techniques and analysis of the tradeoffs between power, area, and performance.

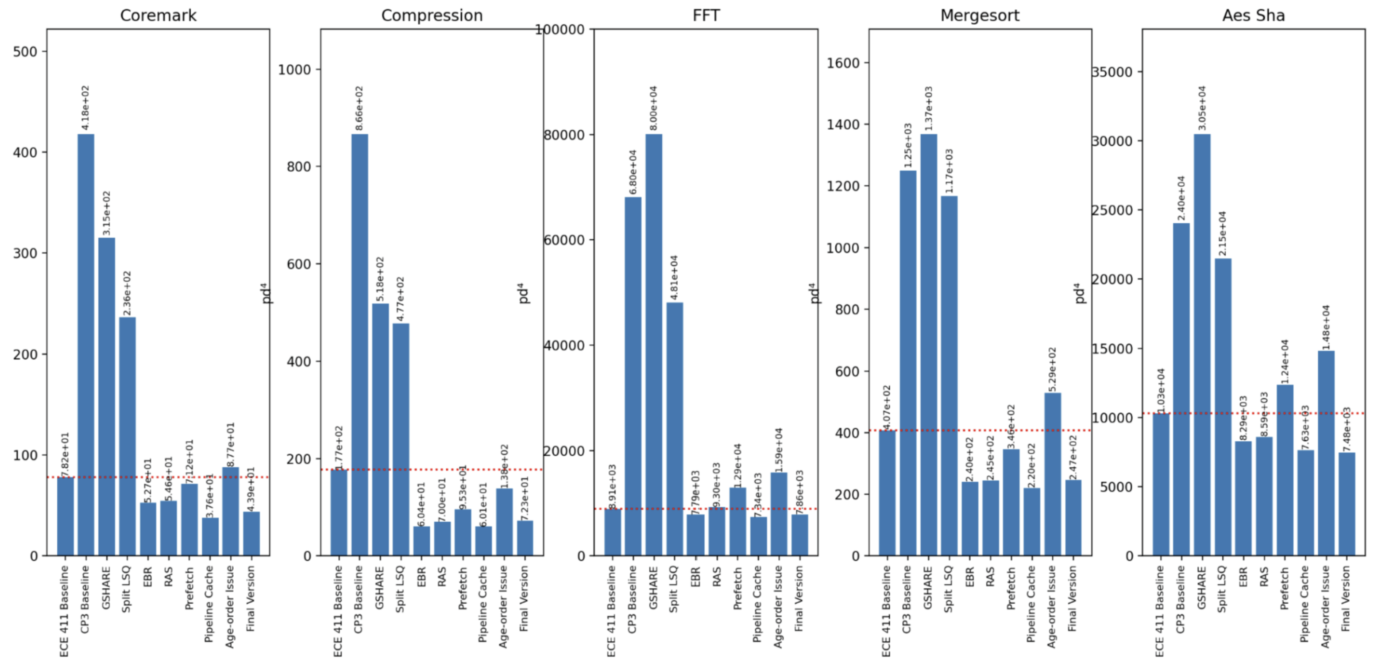

Fig. 6. PD^4 values across all tests

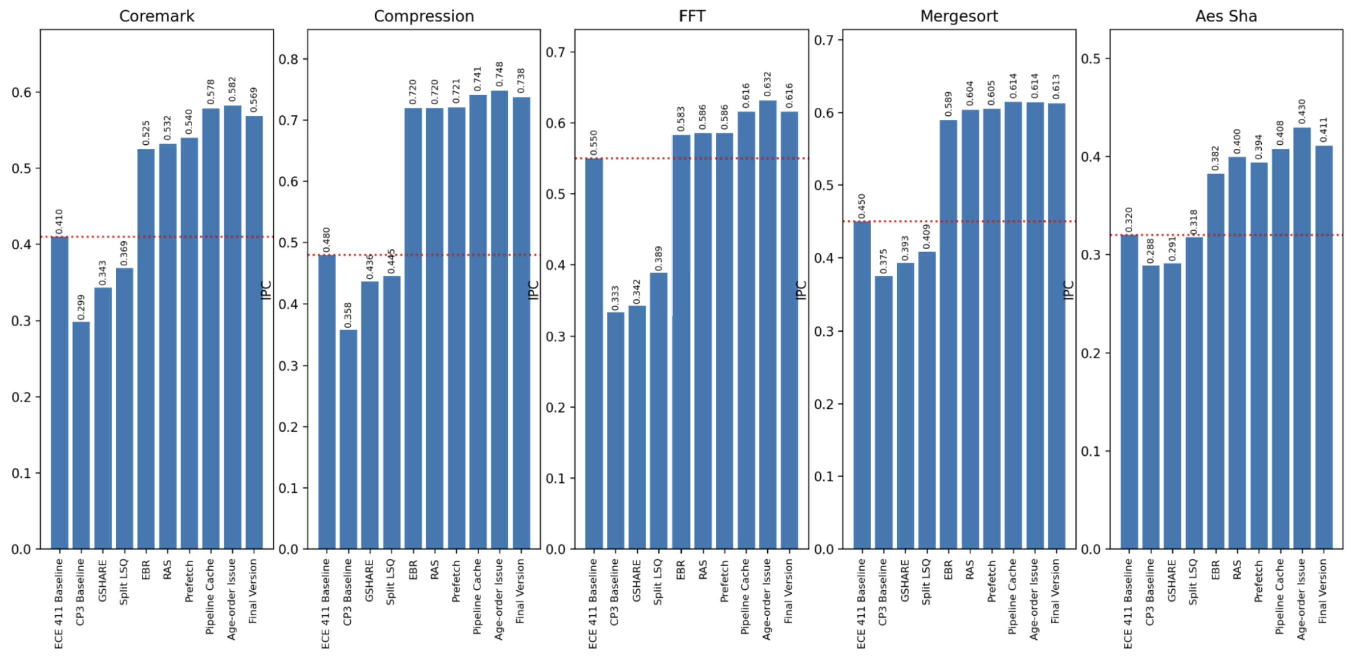

Fig. 7. Instructions per cycle across all tests