Exploring Writeback Designs for Efficiently Leveraging Parallel-Execution Units in FPGA-Based Soft-Processors

Eric Matthews, Yuhui Gao and Lesley Shannon
School of Engineering Science, Simon Fraser University
{ematthew, yuhui_gao, lshannon}@sfu.ca

***Abstract*—Maximizing processor performance depends on maximizing the product of instruction throughput and clock frequency. Writeback mechanisms and forwarding networks heavily impact both of these properties along with the resource usage and scalability of the processor design. Furthermore, these mechanisms are typically multiplexer heavy which can make their implementation resource inefficient on FPGAs.**

In this paper, we explore multiple different writeback and result storage mechanisms using an FPGA-based RISC-V soft-processor (Taiga), exploring both exception-safe and non-exception-safe designs. Writeback mechanisms based on per-unit result storage and centralized storage are explored while leveraging FPGA specific resources such as LUTRAMs. We evaluate the designs based on their impact on instruction throughput, processor frequency, and scalability of both simultaneous instructions in-flight and the number of execution units. As each design has different characteristics, we focus on comparing and contrasting the designs. We find that across all designs, average IPC can vary by up to 11%, with a few designs reaching the maximum IPC of one for some benchmarks. Clock frequency is found to vary by up to 20% across the designs, but is not significantly impacted when increasing the number of execution units. Scaling up the instructions in-flight is found to have the greatest variability, with LUT usage increasing by 3% to 93% across the different designs. Overall, we find that under current constraints, a commit-buffer design provides the highest combination of performance and performance per LUT.

I. Introduction

Recently, with the emergence of the RISC-V ISA [1], there has been an increase in the number of soft-processor designs [2]–[5] for FPGAs. Two of these newer processors, both FPGA-based, ORCA [2] and Taiga [3] implement more flexible execution pipelines by structuring their designs with parallel execution units. An advantage of these designs is that it can be easier to integrate custom accelerators with varying latencies and achieve higher Instruction Level Parallelism (ILP), and thus performance, than a fixed-pipeline based design. However, parallel-execution units present additional design challenges over fixed-pipeline designs as they allow for both increased number of instructions in-flight and for the simultaneous completion of instructions. Handling simultaneously completing instructions is the responsibility of the writeback network.

While most components of a processor require careful design to not negatively impact clock frequency in an optimized design, the writeback network is of particular importance due to its high connectivity. As it interfaces with both the issue logic and all of the execution units, it can potentially impact many critical paths within the processor's pipeline and the routability of the design. Additionally, this network, along with the forwarding of results requires careful design as it is typically multiplexer heavy which are more costly for FPGA-based designs [6].

In this paper, our goal is to explore writeback storage and forwarding mechanisms for FPGA-based parallel-execution units to evaluate how to effectively leverage their execution resources. As such, it important that other processor components, such as the execution units and branch predictor are not the limiting factor in the processor's performance. Thus, based off of the study by Matthews et al. which found Taiga to have higher IPC and runtime performance than ORCA [7], we have selected the Taiga processor as the baseline for this work. Even though we have selected Taiga for this work, the outcomes of this study would be applicable to any other FPGA-based soft-processor design with parallel-execution units including ORCA [2].

Our paper explores three different writeback storage and forwarding mechanisms: (*per-unit ID-buffers*, *commit-buffer*, and *ID-banked register file*) in addition to the baseline sourced from Taiga. We find that our exception-safe *commit-buffer* design is able to achieve a 6% increase in IPC over the non-exception-safe commit variant of Taiga [7]. The performance improvement further increases to 11% when compared against the exception-safe variant. In terms of performance per LUT, the *commit-buffer* design ties the baseline in resource efficiency while providing higher throughput. Additionally, we explore the scalability of the designs focusing on their ability to support additional execution units and greater numbers of in-flight instructions. Specifically, in this paper we present:

- three new writeback and storage mechanisms: *commit-buffer*, *per-unit ID-buffers*, and *ID-banked register file*,
- a thorough analysis of IPC, frequency and resource usage of the designs, and
- an exploration of the scalability of the designs.

The remainder of this paper is organized as follows. Section II covers related work on FPGA-based soft-processors and provides background on the Taiga processor. Section III presents the writeback storage and forwarding mechanisms studied in

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00025

this paper. Section IV presents a comparison and analysis of the performance characteristics of the various designs. Finally, section V presents an analysis of the scalability of the designs before the conclusion of the paper.

II. Background

For fixed-pipeline processors, researchers have explored many aspects of their design. General architecture studies have been performed on fixed-pipelines [8] along with studying the impact of forwarding on deeply pipelined DSP-based processors [9]. For parallel-execution unit designs, there is an advantage in forwarding networks in that results are not propagated through additional stages if they complete early, such as common ALU operations. As such, the writeback network itself can be leveraged for forwarding results.

Much of the soft-processor research focuses on processor features that are independent of how the execution logic is structured. This includes work on features such as branch predictors [10] and more complex functionality such as multi-threading [11] [12] and runahead execution [13]. As this paper focuses on exploring writeback mechanisms to leverage parallel-execution units, these works are orthogonal and complementary to the focus of this paper.

Superscalar out-of-order processors also feature parallel execution units, however, the mapping of ASIC designs to FPGA fabrics has been found to be inefficient [14] [15]. Research on FPGA optimized components of superscalar out-of-order processors such as the reorder buffer [16], instruction schedulers [17] or memory hierarchies [18] has resulted in better scaling and higher operating frequencies, however they still require large amounts of FPGA resources.

The Taiga processor, which has been selected as the baseline for this work, supports a mechanism called *"early-commit"* [7] for increasing throughput of its execution units, however, it is not exception safe, and as we find in this paper, its exception-safe variant comes at a cost in throughput. As some embedded system designs will require exception-safe operation, in this work we explore writeback mechanisms that are inherently exception safe without performance loss.

A. Taiga Overview

The Taiga processor [3] is a single-in-order issue processor. It supports an early-commit behaviour allowing instructions to complete out-of-order with both exception-safe and non-exception-safe variants. A high-level overview of the processor is presented in Figure 1. The figure highlights the main stages of the processor's pipeline and the performance characteristics of each of the execution units. As can be seen in the figure, there are six execution units in total (Br, ALU, Mul, Div, CSR, LS), however, the branch unit (Br) does not have a connection to the writeback logic and the CSR unit and Load Store unit share a connection. For each unit, the upper number represents the latency of the unit and the lower number the rate at which the unit's datapath can begin a subsequent instruction.

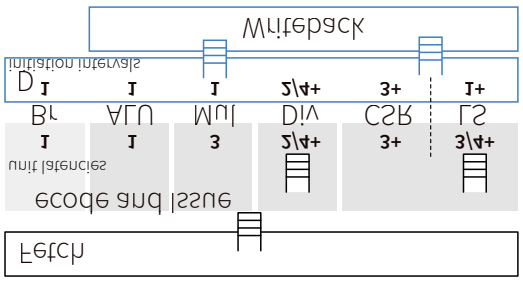

Fig. 1. Taiga Block Diagram. Upper unit numbers represent the unit's latency. Lower unit numbers represent the rate at which a unit can start additional requests. Plus symbols indicate a minimum bound and slashes indicate their are multiple latency paths within the unit. Blue outlined sections indicate components modified in this paper.

Taiga was sourced from its repository [19], specifically, commit[1] with fixes back-ported. We have highlighted in blue the components of the processor that are modified in this work. This includes the register file and register bypassing inside of the issue logic as well as unit result storage and the writeback infrastructure.

III. Writeback Designs

A major difference between fixed-pipeline designs and parallel-execution unit designs is that parallel designs can support both simultaneous and out-of-order completion. The actual IPC improvements that can be extracted from parallel execution depend on the constraints differing designs place on the execution units and the application itself.

In this section, we present three new writeback mechanisms called: *per-unit ID-buffers*, *commit-buffer*, and *ID-banked register file*. We also analyze the *early-commit* implementation of Taiga [7], which we use as a baseline. Figure 2 presents all four designs highlighting the datapath aspects that change between the designs. Each system will be discussed in detail in the following sections.

1) Design Commonalities: Taiga uses an ID tracking system for all instructions that write to the register file or memory. On issue, these instructions are assigned an ID that is used to track their order and to handle Read-After-Write (RAW) hazards [7]. IDs for instructions are propagated, along with the instruction, through their execution unit. Figure 3 highlights the interface of the execution units. Signals in red are the control signals that remain the same for all designs. The accepted signal, highlighted in blue, is used only for the *early-commit* design. In all designs, with the exception of the *ID-banked register file* design, the done and ID signals on the writeback side will set a flag marking the ID as done and pending for writeback to the register file. Once the instruction has been committed the flag is cleared and that ID becomes available again for reuse.

While each design changes how results from execution units are stored, the latency and throughput characteristics of each execution unit remains constant across all designs.

[1] gitlab.com/sfu-rcl/Taiga (ae32ecd72fc47b350a5c8232868d0f56baf66628)

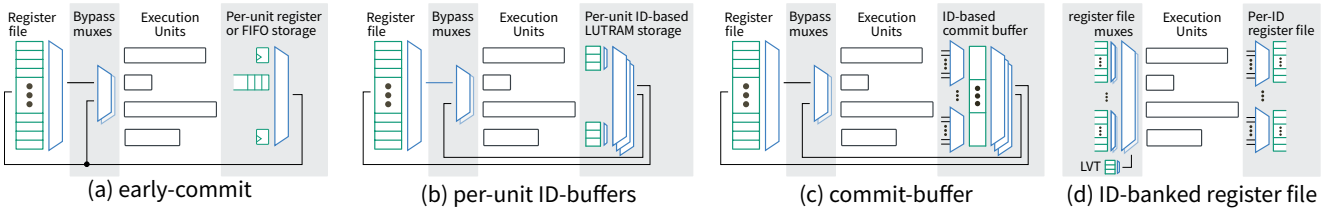

Fig. 2. Writeback and forwarding datapaths. Diagrams show Issue through Writeback stages of the processor. Shaded sections highlight the major datapath differences between the designs. In subfigures (c and d), as writeback storage is ID-based, there is an additional layer of muxes that select between the execution units for each storage element. In subfigure (d), the register file is split between the issue and writeback stages to highlight its connectivity to both stages.

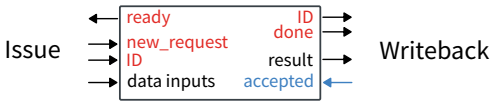

Fig. 3. Execution unit interfaces. Red signals are the control signals and are constant across all designs with the exception of accepted, highlighted in blue, which is used only by the original Taiga design.

2) General Forwarding Considerations: The RISC-V base integer ISA, including the Multiply and Divide extensions, contains instructions with at most two source operands, labelled *rs1* and *rs2*. As such, in any design with register file bypassing, there are two bypass muxes, one for each operand. We added tracing support to Taiga looking at how often the *rs1* vs *rs2* operands utilized a forwarded result and found that it was benchmark dependent. While most benchmarks saw more frequent usage of rs1 forwarding, in some benchmarks it was the other way around.

A. Early-Commit

The baseline for our investigation is the "*early-commit*" from the existing open-source Taiga repository and is shown in Figure 2 (a).

1) Storage Mechanism: In *early-commit*, storage of instruction results is implemented on a per-unit basis. Simple units, such as the ALU, have the result stored in Flip-Flops, whereas the Load Store unit uses a FIFO for the output. Matthews et al. [7] presented the rational for the FIFOs as providing decoupling between the issue and writeback stages as well as providing control signal isolation.

2) Forwarding Support: In this design, not all results are visible at the writeback stage due to the use of FIFOs on the outputs of some units such as the Load Store and Division units. As such, the only result that can be forwarded to the issue stage is the result that is being committed to the register file on any given cycle.

This design is capable of both exception-safe and non-exception safe behaviour. While operating in non-exception safe mode, instructions can be committed to the register file out-of-order. Every cycle, the oldest instruction that is ready to write to the register file is committed even if it is not the oldest instruction in-flight. For exception-safe behaviour, if no in-flight instruction can cause an exception, the behaviour of the non-exception safe mode is used. When an exception can occur, as is the case for load/store operations, instructions are committed in-order until the exception status of the load/store instruction(s) has been resolved. Tracking of instructions is performed by a stack structure that maintains an ordering of all IDs. On any given cycle, any active ID can be retired from any location within the stack. Additionally, on any given cycle, a store operation can also be committed in parallel with a retired instruction requiring two updates to the stack.

3) Specific Strengths and Weaknesses: As simple units such as the ALU just register their output, there is no additional LUT usage overhead for storage of results making this storage mechanism resource efficient. However, as some units utilize FIFO outputs, not all instruction results are visible at the writeback stage thus limiting forwarding options. Additionally, as units such as the ALU can only hold one result at a time, if a subsequent instruction requires the ALU it cannot be issued until the ALU's result has been accepted by the writeback stage. This type of structural hazard is similar to a fixed-pipeline limitation where any instruction that stalls in the execute or memory stage blocks the progress of subsequent instructions. Finally, while not all embedded systems will require load/store exceptions, in systems that to do, the performance penalty of exception-safe behaviour can be as high as 10% as will be shown in the following section.

B. Per-Unit ID-Buffers

A potential solution to the weakness of the *early-commit* design is demonstrated in our *per-unit ID-buffers* design shown in Figure 2 (b). Unlike *early-commit*, this design is always exception safe, can forward any completed result and removes the structural hazard of the *early-commit* design. Instead, it is constrained by the processor's limit on instructions-in-flight.

1) Storage Mechanism: As a possible solution to the structural hazards and incomplete result visibility of the *early-commit* design, the mix FIFOs and register storage is replaced with per-unit LUTRAMs that store the result indexed by the instruction's ID.

2) Forwarding Support: As seen in Figure 2(b), instruction results are stored in LUTRAMs and multiple read ports can be utilized to support both accessing results for operand forwarding and by the writeback mux for committing to the register file.

3) Other changes: As instructions are not issued if there are no available IDs, there will always be space in the per-unit

buffer to store the instruction's result. Additionally, IDs are not recycled until the result has been committed to the register file. As such, the units no longer need an acknowledgment that the instruction has been committed. This results in a simplification and reduction of control logic for the units. With this setup, all units in the processor can now accept an instruction on any cycle and thus have their ready signals tied to one. The limitation in execution resources is now the number of IDs supported, ie. the maximum number of in-flight instructions.

By increasing the storage capacity of each unit, and exposing all completed results to the forwarding logic, the opportunistic early-commit behaviour of the previous design no longer improves the IPC. As such, the ID stack is replaced in this design with two simple counters (modulo the max number of instructions-in-flight) that keep track of the issue ID and the oldest non-committed ID. Instructions are then always committed in-order and thus, always exception safe. Stores are still completed in parallel with other instructions in this design, with the retired counter incremented by two instead of one.

4) Specific Strengths and Weaknesses: Compared to *early-commit*, *per-unit ID-buffers* provides increased result storage and greater access to results for operand forwarding, both of which we expect to improve the IPC of the processor. However, LUT usage is increased due to the switch from Flip-Flop based storage to LUTRAM based storage with approximately 72 LUTs required per unit. Additionally, not all LUTs can be configured as LUTRAMs, which may increase routing delays in the design. In this paper, maximum in-flight counts of four and eight are evaluated; however, due to this design's use of LUTRAMs for storage, ID limits of up to 32 can be supported without increasing the LUTs required for the ID-buffers on Xilinx FPGAs.

C. Commit-Buffer

The *commit-buffer* design, shown in Figure 2 (c), shares much in common with the *per-unit ID-buffers* design, but differs primarily in its result storage mechanism.

1) Storage Mechanism: The *commit-buffer* result storage is centralized within an ID-based writeback buffer implemented with Flip-Flops, rather than on a per-unit basis. As multiple writes could occur per cycle, we have chosen to implement this structure with Flip-Flops as opposed to LUTRAMs as there is already additional delay due to the unit select muxes compared to the *per-unit ID-buffers* design. Additionally, for each ID's storage, all units' outputs must be muxed as shown in Figure 2. As such, the number of muxes scales with the number of IDs and in depth by the number of writeback units.

2) Forwarding Support: Forwarding support for the *commit-buffer* design is the same as in the *per-unit ID-buffers* design; however, the forwarding mux and writeback mux scales based on the number of IDs as opposed to the number of execution units.

3) Specific Strengths and Weaknesses: The commit-buffer design shares the same throughput aspects as the *per-unit ID-buffers* design, differing only in how the instruction results are stored. However, this storage difference is expected to have a significant impact on the frequency and scalability of these two designs.

D. ID-Banked Register File

The last design studied in this work is an *ID-banked register file* based design and is shown in Figure 2 (d).

1) Storage Mechanism: In the *ID-banked register file* design, the storage mechanism for instruction results is the register file itself. As an instruction completes, its result is immediately written to the register file. Instead of a single register file, this design replicates the register file based on the number of IDs (instructions in-flight) supported. Tracking of the most recently updated ID-bank is handled by a Live-Value-Table (LVT) (that already existed in Taiga for ID tracking and forwarding purposes [7]). Like the *commit-buffer* design, this design centralizes the storage of results and requires muxing between all unit outputs for each register file bank.

2) Forwarding Support: As can be seen Figure 2, unlike the other designs, there is no forwarding mechanism for this design as all operands are sourced directly from the register file. While there is no forwarding, there is still an additional mux to select between the ID banks based off of the LVT.

3) Other changes: In all other designs, only a single instruction is committed to the register file in any given cycle. In this design, up to the ID limit can be written in a single cycle. Most of the existing writeback logic was able to be retained. However, the tracking of in-use/available IDs and the tracking of which registers are in-use needed to be changed. As previously mentioned regarding the writeback logic of the other designs, when an instruction completes, a flag is set for that ID to indicate that a pending commit operation is required. For the *ID-banked register file* design, instead, a bit is set when an instruction is issued. The next ID is then found by scanning this bit vector for the first unused/unset ID.

For register in-use tracking, the existing component, an xor-based 2 write-port LUTRAM, had to be replaced. For the previous designs, two write ports is sufficient as there were at most two updates per cycle, one from the issuing instruction and one from the committing instruction. For the *ID-banked register file* design there can now be multiple commits in a cycle. To support this, instead of storing in-use bits per register, the destination register for each ID is stored and compared against the issuing instruction's operands to check for conflicts. As the checks need to be made against all IDs this is implemented as a CAM structure.

4) Specific Strengths and Weaknesses: While the *per-unit ID-buffers* and *commit-buffer* designs are inherently exception-safe in their operation, the *ID-banked register file* design is not inherently exception-safe, similar to the *early-commit* design. As instructions commit as soon as they complete, its possible for an instruction to commit before the exception status of a previously issued instruction is known. To make the design exception-safe we explored a simple change. As the Load Store unit is the current source of potential exceptions, once a Load Store instruction has been issued, no issues to other units can occur until the exception status of all load/store operations

TABLE I
RESOURCE USAGE AND OPERATING FREQUENCY COMPARISON

	LUTs	FFs	Freq (MHz)
early-commit	1927	811	121.6
per-unit ID-buffers	2185 (+13%)	763 (-6%)	97.2 (-20%)
commit-buffer	2047 (+6%)	869 (+7%)	119.7 (-1.6%)
ID-banked register file	2071 (+7%)	(-3%) 785	106.2 (-13%)

has been resolved. Multiple loads and stores can be in-flight at once as, if an exception does occur, the input buffer to the load store unit can be flushed.

The key feature of the *ID-banked register file* is that it can commit multiple instructions per cycle. This may not have significant benefits for a single-issue processor design, but it may strongly impact a multi-issue processor design's efficiency. Additionally, compared to the *per-unit ID-buffers* and *commit-buffer* buffer designs, this design does not have to wait for instructions to be written from intermediate storage to the register file. Thus, it is less likely than the other designs described herein to run out of available IDs.

IV. DESIGN PERFORMANCE COMPARISON

In this section, we provide a performance comparison of the four different writeback mechanisms, focusing on IPC, frequency and resource usage.

For our test configuration, a high performance baseline will enable us to properly evaluate the impact of the different writeback designs. If other aspects of the processor, such as branch prediction or execution unit performance significantly limit performance, then the impact of the writeback design choice would be nominal. As such, we have configured the processor for the RV32IM ISA with the following feature set:

- 2-way 512-entry set-associative 2-bit saturating branch predictor with an 8-entry Return Address Stack (RAS)
- hardware multiply and variable latency divider [20]
- max in-flight count of 4 IDs

Additionally, our setup is configured with 64KB of local memory to support the benchmarks used in this paper.

A. Resource Usage and Frequency Results

Table I, presents resource usage and frequency results obtained with Vivado 2019.2 for a Zynq X7CZ020 FPGA (Zedboard). Percent increases in resource usage are provided relative to the *early-commit* design sourced from the Taiga repo [19]. For the *early-commit* and *ID-banked register file* designs, we found that the difference between the exception-safe and non-exception safe variants did not impact frequency results or change LUT usage by more the a couple LUTs and thus, just the non-exception-safe results are presented here. As all processor configurations require two BRAMs (for the branch predictor) and four DSPs (for the multiplier), they are excluded from the table.

We can see that all designs have additional resource overhead compared to the *early-commit* design. This is to be expected since all designs have additional muxes and storage elements compared to the *early-commit* design as discussed in the previous section. The largest increase is for the *per-unit ID-buffers* design with a 13% increase in LUT usage. For this design, the largest increase in resources comes from the storage change from registers (for units such as the ALU) to the per-ID LUTRAMS. For the ALU, this results in an increase of approximately 72 LUTs (24 LUTs per LUTRAM output).

Reducing the number of forwarding operands can thus reduce resource usage. In our analysis of the benchmarks used in this paper, we found that forwarding of both operands occurred only 2% of the time, on average. However, the forwarding logic needs the IDs of the instructions that rs1 and rs2 are waiting on. To share a read port, we must first determine if they are waiting on a result before selecting which one to use to access the LUTRAM. This additional delay in arbitrating the LUTRAM read port further lowered the clock frequency of the *per-unit ID-buffers* design. As this design already has the lowest operating frequency, we consider this change to be undesirable.

The increase in LUTs for the *commit-buffer* design over *early-commit* comes from the increase in muxes both for forwarding and for storing to the commit-buffer itself. Similarly, for the *ID-banked register file* design the increase is also primarily from the additional writeback muxes and the cost of the additional register file banks (48 LUTs per bank). However, both of these designs are only a modest increase in resources at 6% and 7% respectively over the *early-commit* design with the *commit-buffer* design being inherently exception safe.

In terms of frequency, the *early-commit* and *commit-buffer* designs are within the variability we find with the Vivado tools when making small changes in our designs. In all designs, we find that the critical path is from the source operands through the ALU to whichever storage mechanism is used. With the *per-unit ID-buffers* and *ID-banked register file* designs, we see larger decreases in operating frequency of -20% and -13% respectively. The additional costs for these two designs come from the additional delays of the LUTRAMs compared to the Flip-Flop storage.In the case of the *per-unit ID-buffers* design, rather than storing the ALU result in a register that is paired with the LUT that generated the result, it must now route to another LUT. This additional LUT may also be in a different column, as not all logic blocks support LUTRAMs in Xilinx FPGAs. Additionally, the LUTRAMs' data-access and storage times have additional delay compared to the registers' data-access and storage times. As the critical path starts and ends in LUTRAMs in these designs, this results in additional delays to what was already a critical path in the design. While the ALU datapath is typically the critical path in the processor (and also the only single cycle unit), during development the variable latency divider and branch prediction logic were sometimes reported as the critical paths in the design.

B. Performance Results

To evaluate the performance of the different writeback mechanisms, we have selected the Embench [21] benchmark suite; a relatively new benchmark suite designed for benchmarking embedded processors. All benchmarks are built

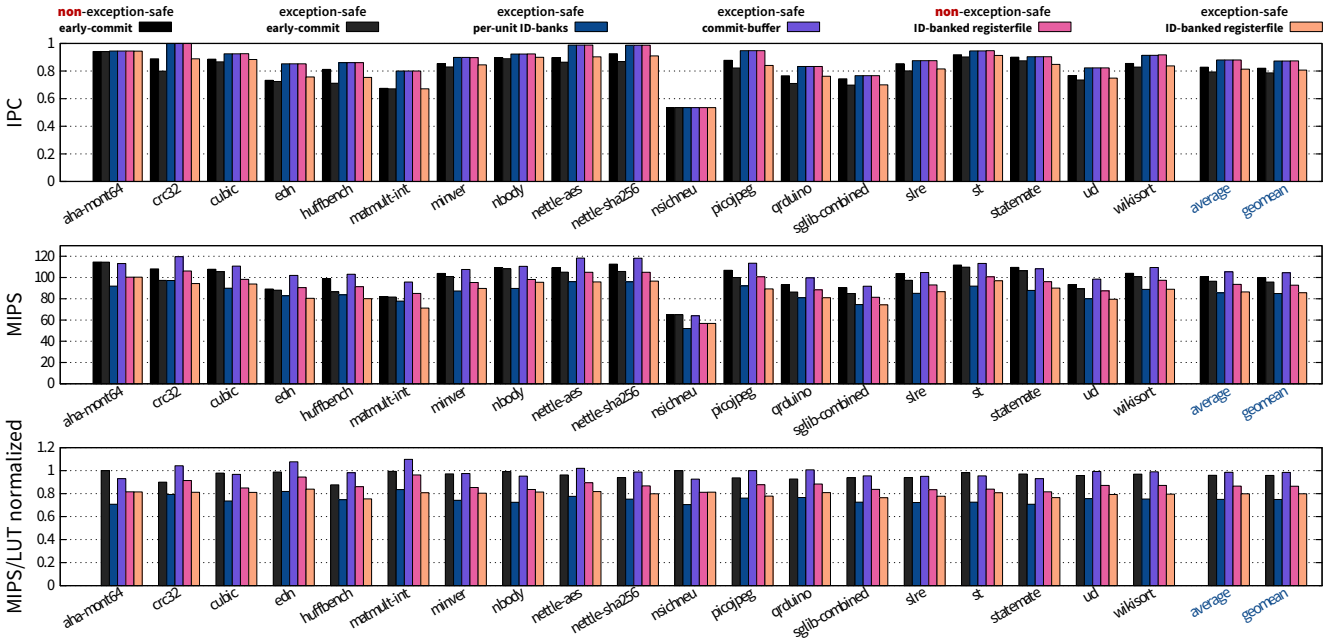

Fig. 4. Embench benchmarks: IPC, MIPS and MIPS per LUT (normalized to non-exception-safe *early-commit*) per writeback mechanism

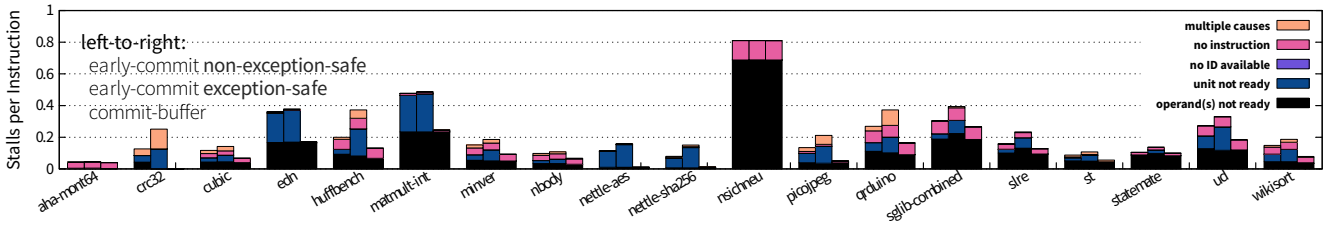

Fig. 5. Breakdown of issue stall sources per benchmark and writeback mechanism

with GCC 9.1.0 and compiled with -O2 optimization. Data collection is performed with Verilator [22] to facilitate the collection of additional trace information for evaluating our designs. For the performance results, both the exception-safe and non-exception-safe variants of the *early-commit* and *ID-banked register file* designs are included as they impact the throughput of the designs.

As a measure of how well each writeback mechanism is able to leverage the parallel execution-units, we present IPC results for designs in the upper section of Figure 4. While *commit-buffer* and *per-unit ID-buffers* design results have identical operation, they are both included in the graph for consistency with the other result graphs. We can see that even the baseline system, *early-commit*, has a high average IPC of over 0.8 with the one exception of the *nsichneu* benchmark that will be explained in the following section. As this is a single-issue processor, an IPC of one is the upper performance bound. In the case of the *crc32* benchmark, the *commit-buffer*, *per-unit ID-buffers*, and non-exception-safe *ID-banked register file* are all able to achieve an IPC of effectively one (0.997). These three systems achieve a 6% increase in IPC over the *early-commit* non-exception-safe design and an 11% increase over the exception-safe mode on average. While the *commit-buffer* and *per-unit ID-buffers* designs are inherently exception-safe, we see that the exception-safe variants of *early-commit* and *ID-banked register file*, are most often slower than their non-exception-safe counterparts with *early-commit* being impacting by over 4% on average and *ID-banked register file* by over 8% on average. If more exception sources are added to the system, we expect these numbers to grow as neither of these designs allow exceptions from multiple sources to be possible at any given time. For example, if a new unit was added that could cause an exception, instructions would not be able to be issued to the Load Store unit while an exception was possible in the new unit and vice versa.

As was discussed when the designs were presented, a potential advantage of the *ID-banked register file* design is that it can commit multiple instructions per cycle. As such, it is less likely to run out of IDs. However, in practice we find that in the best case on the *wikisort* benchmark, the performance improvement is only 0.3% over the *commit-buffer* design.

While IPC provides a measure of how well the processor's execution resources are leveraged, it does not provide a full performance comparison as it does not reflect differences in

operating frequency. For this comparison, we have scaled the IPC by the operating frequencies of each design to report performance in Millions of Instructions Per Second (MIPS) as shown in the middle section of Figure 4. In this figure, we can see that while *per-unit ID-buffers* and *ID-banked register file* have the same IPC as the *commit-buffer* design, their overall performance is lower due to their lower clock frequency. Overall, the *commit-buffer* buffer design is found to have the highest throughput with an average MIPS of 105 for a zedboard based design, a 5% increase over *early-commit* non-exception-safe and 9% over the exception-safe variant.

1) Tracing Infrastructure: To better understand the performance bottlenecks in the processor, we added tracing infrastructure to capture the causes for stalls in the issue stage of the processor. Specifically, we looked at capturing when there was an issue stall exclusively due to: missing operands, the required unit not being ready to accept a new request, having no ID available for issue, having no instruction at the issue stage (branch flush or delay from a miss-predict), or a combination of these conditions. Additionally, we recorded how often the forwarding logic was needed for each design along with branch prediction accuracy.

Figure 5 presents the breakdown of stall sources for exception-safe and non-exception safe *early-commit* and for the *commit-buffer* design normalized to the number of instructions executed in each benchmark. *Per-unit ID-buffers* is not included as it has identical performance characteristics as the *commit-buffer* design, and *ID-banked register file* is not included as it is only different in IPC from the *commit-buffer* by less than a tenth of a percent. From this figure, we can see that the largest improvement in performance from the *commit-buffer* design comes from reducing stalls due to units not being ready to accept new requests. For many benchmarks, these stalls account for over half of the total issue stalls. While in many cases the operand stalls do not decrease between the systems, it is possible that by removing the unit stalls more operand stalls could occur. However, in some benchmarks like *crc32* and *huffbench* we do see a further reduction in operand stalls. The *no ID available* stall is not visible in this plot as its at most 1% of the stalls (*st*) and, on average, less than a tenth of a percent. The *no instruction* stalls are a result of the delays incurred by branch miss-predicts. The multiple source category disappears from the *commit-buffer* based design as the only overlaps that are now possible involve overlaps with no ID's being available, which themselves were already found to be a negligible percentage of the stalls.

The *nsichneu* benchmark is the largest outlier in this set of benchmarks. To understand its behaviour we further broke-down what type of instruction (alu, branch, etc) were stalled waiting for their operands. In doing so, we found that the stalls were due to branches waiting for their operands. Looking at the dissasembly of this program, we found short instruction sequences with loads followed immediately by a branch instruction, thus the branch must wait 2 cycles for its operand. The only way to significantly improve the performance of this benchmark, from a hardware perspective, would be to either reduce the latency of load operations or allow speculative execution of branches. As designs such as the *commit-buffer* and *per-unit ID-buffers* buffer results before writing to the register file, both of these designs would be perfect candidates to explore limited speculative execution. For the remainder of the benchmarks, we find that the operand stalls were roughly split between ALU operations waiting for their operands and branch instructions waiting for their operands.

Finally, we report MIPS per LUT in the lower section of Figure 4 normalized to non-exception-safe *early-commit*. Here, we can see that while all designs developed in this paper improve IPC, only the *commit-buffer* design does so while retaining performance per LUT efficiency with only a 1% difference between it and the *early-commit* design.

In summary, we find that all of the new designs are able to achieve higher IPC than the baseline Taiga *early-commit* design by avoiding contention for execution units and providing improved forwarding of results. Of the new designs explored, *commit-buffer* is found to provide a 6% increase in IPC over non-exception-safe *early-commit* while being exception-safe itself and providing equivalent performance per LUT. When comparing against exception-safe *early-commit*, this improvement rises to an 11% IPC improvement.

V. Scalability of Designs

In the previous section, we analyzed the performance of the different writeback mechanisms based on their performance on a embedded benchmark suite. In this section, we explore how the designs scale with the addition of extra execution units and support for additional in-flight instructions.

A. Scaling Support for Additional Execution-Units

To begin with we explore how increasing the number of execution units impacts the resource usage and operating frequencies of the designs. To ensure that the impact is mostly from the differing writeback implementations and not due to the impact of the unit itself, a reference-unit was developed that has a minimal resource footprint while still fully utilizing the register file and writeback stage. As such, the reference-unit accesses two register inputs, XOR's them with a constant and registers them based on the issue signal for the reference unit. On the second cycle, the two results are XORed together. It has a baseline cost of 64 LUTs and 70 FFs.

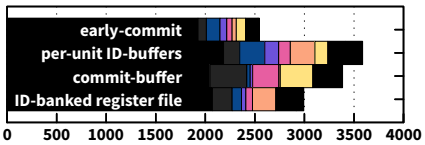

Fig. 6. LUT usage scaling with increasing numbers of reference-units

Table II presents a resource and frequency comparison of the systems with eight additional units, with percentage change relative to each system's baseline from Table I. Figure 6 presents the LUT scaling per additional reference-unit for each of the designs. We find that the *per-unit ID-buffers* design has

TABLE II
DESIGN SCALING WITH 8 ADDITIONAL REFERENCE-UNITS

	LUTs	FFs	Freq (MHz)
early-commit	2547 (+32%)	1641 (+102%)	114.7 (-5.7%)
per-unit ID-buffers	3587 (+64%)	1608 (+53%)	90.2 (-7.2%)
commit-buffer	3306 (+62%)	1702 (+96%)	113.5 (-5.2%)
ID-banked register file	2987 (+44%)	1658 (+111%)	106.0 (-0.2%)

TABLE III
RESOURCE USAGE OF 8-IDS AND INCREASE OVER 4-IDS

	LUTs	FFs	Freq (MHz)
early-commit	2093 (+8.6%)	859 (+5.9%)	116.6 (-4.1%)
per-unit ID-buffers	2246 (+2.8%)	771 (+1.1%)	98.4 (+1.2%)
commit-buffer	3684 (+80.0%)	1923 (+121.3%)	107.7 (-10%)
ID-banked register file	4003 (+93.3%)	1690 (+115.2%)	88.8 (-16.4%)

both the largest increase in LUT usage and the largest usage overall. As the *per-unit ID-buffers* design stores its results in LUTRAMs, every additional unit in this design will have a higher overhead than the other designs.

We can see from Figure 6 that as units are added to the designs some incremental increases are much larger than others, particularly for the *commit-buffer* design. These points correspond with writeback unit muxes crossing boundaries that require additional LUTs or levels of logic to implement. In terms of frequency, we can see that all designs scale acceptably, with the *ID-banked register file* design seeing no impact after adding eight units (-0.2%) and *early-commit* and *commit-buffer* with degradations of only approximately 5%.

Since all designs show reasonable frequency scaling, as a final metric we can look at how much overhead is added, on average, (across the additional 8 units) when subtracting out the unit cost. In this way, we have a rough measure of how much overhead each additional unit requires. For *early-commit*: 14 LUTs, for *per-unit ID-buffers*: 111 LUTs, for *commit-buffer*: 93 LUTs and for *ID-banked register file*: 50 LUTs. For any non-trivial accelerator, most of these overheads are likely to be small with the exception of the *per-unit ID-buffers* and *commit-buffer* buffer designs.

B. Scaling Support for Additional IDs

In the previous section, we found that with a limit of 4-IDs, ID related stalls were less than a tenth of a percent of all issue stalls. As such, there is currently no benefit in increasing the number of IDs, however, this may not hold for all possible accelerator designs.

If we were to simply look at the maximum occupancy that all execution units can hold (excluding input FIFOs) we would end up with nine possible in-flight instructions for the processor configuration used in this paper. However, as the processor is single-issue in-order-issue, the upper bound will be much lower. Additionally, instructions such as branches do not count towards the ID limit despite accounting for 15% of the instructions, on average, in our benchmarks. In fact, the limiting factor is much more likely to be the inherent Instruction-Level-Parallelism (ILP) of the software itself, as average parallelism in systems with register renaming and large execution windows is still less than 4 in many cases [23].

For deeply pipelined units, that have high internal ILP, the existing designs that use the register in-use tracking (ie. all designs other than *ID-banked register file*) can bypass the ID limit. In these designs, if only a single unit is active at a time there is no limit to number of in-flight instructions as register dependencies are tracked on a per-register basis instead of on a per-ID basis as they are for the *ID-banked register file* design.

Table III presents the resource usage for 8-IDs for all designs along with the increase relative to their baseline of 4-IDs. For all designs other than *early-commit*, we find that the ALU is still the critical path. For *early-commit*, the critical path is now entirely within control logic ending in the update logic for its ID ordering stack. Additionally, while *early-commit* scales reasonably well up to 8 IDs, it also has the most stalls due to resource contention which may limit its ability to leverage additional IDs in systems with more execution units.

The best scaling design is the *per-unit ID-buffers* design, which sees a LUT increase of only 2.8%. For this design, the increases are purely within the ID-tracking infrastructure. As was discussed in the design section, the *per-unit ID-buffers'* LUTRAM-based storage allows it to scale up to 32 entries/IDs before requiring additional LUTs for result storage. We find that the *commit-buffer* and *ID-banked register file* designs have the worst scaling as their storage requirements are proportional to the number of IDs. To summarize, if your design truly needs a large number of in-flight instructions, and those instructions are distributed across multiple execution units, then the *per-unit ID-buffers* design provides the best scalability.

VI. CONCLUSIONS AND FUTURE WORK

In this work we have found that exception-safe writeback mechanisms, specifically a *commit-buffer* based design, can provide a 6% increase in IPC over a baseline non-exception-safe Taiga *early-commit* design. This performance increases further to 11% when compared to the exception-safe variant. When evaluating based on performance/LUT, the same *commit-buffer* design achieves the same efficiency as *early-commit* while providing higher throughput. All designs were found to scale well in terms of frequency for up to eight additional execution units. When increasing the support for the maximum number of instructions in-flight, the *per-unit ID-buffers* design was found to have exceptional scaling compared to the *ID-banked register file* and *commit-buffer* buffer designs which scale poorly with additional IDs. Future work can consider addressing the bottlenecks found in our stall analysis by exploring changes to support speculative execution or support for multi-issue.

ACKNOWLEDGMENTS

This work is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) COHESA project (NETGP485577-15), the CWSE PDF (470957), and RGPIN341516, along with in-kind support from Xilinx and Intel.

REFERENCES

[1] A. Waterman, "Design of the risc-v instruction set architecture," Ph.D. dissertation, EECS Depart., University of California, Berkeley, Jan 2016.

[2] "ORCA: RISC-V by VectorBlox," VectorBlox. [Online]. Available: github.com/VectorBlox/orca

[3] E. Matthews and L. Shannon, "Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features," in *FPL*, Sept 2017, pp. 1–4.

[4] C. Papon, "Vexriscv." [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[5] C. Wolf, "Picorv32 - a size-optimized risc-v cpu." [Online]. Available: https://github.com/cliffordwolf/picorv32

[6] H. Wong, V. Betz, and J. Rose, "Comparing fpga vs. custom cmos and the impact on processor microarchitecture," in *FPGA*, 2011, pp. 5–14.

[7] E. Matthews, Z. Aguila, and L. Shannon, "Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for fpgas using the risc-v isa," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 1–8.

[8] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 202–212. [Online]. Available: https://doi.org/10.1145/1086297.1086325

[9] C. Hui Yan, S. Fahmy, and N. Kapre, "On data forwarding in deeply pipelined soft processors," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 181–189. [Online]. Available: https://doi.org/10.1145/2684746.2689067

[10] D. Wu and A. Moshovos, "Advanced branch predictors for soft processors," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6.

[11] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for sopc area reduction," in *FCCM*, April 2006, pp. 131–142.

[12] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *CASES '07*. ACM, 2007, pp. 155–159.

[13] K. Aasaraai and A. Moshovos, "Sprex: A soft processor with runahead execution," in *ReConFig*, Dec 2012, pp. 1–7.

[14] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, and et al., "Intel nehalem processor core made fpga synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 3–12. [Online]. Available: https://doi.org/10.1145/1723112.1723116

[15] F. J. Mesa-Martinez, A. Sharma, A. W. Hill, C. A. Cabrera, C. Bazeghi, H. Kolakaleti, J. Nayfach, K. Singh, K. S. Halle, M. D. Fischler, M. Nunez, S. Nair, S. N. Kurapati, W. A. Asmawi, and J. Renau, "Scoore santa cruz out-of-order risc engine, fpga design issues," in *(WARP), held in conjunction with ISCA-33*, 2006, pp. 61–70.

[16] M. Rosière, J. l. Desbarbieux, N. Drach, and F. Wajsbürt, "An out-of-order superscalar processor on fpga: The reorder buffer design," in *DATE*, March 2012, pp. 1549–1554.

[17] H. Wong, V. Betz, and J. Rose, "High performance instruction scheduling circuits for out-of-order soft processors," in *FCCM*, May 2016, pp. 9–16.

[18] ——, "Microarchitecture and circuits for a 200 mhz out-of-order soft processor memory system," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 1, pp. 7:1–7:22, Dec. 2016.

[19] E. Matthews and L. Shannon, "Taiga," https://gitlab.com/sfu-rcl/Taiga, 2019.

[20] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking integer divider design for fpga-based soft-processors," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 289–297.

[21] Embench™ Task Group, "Embench™: Open benchmarks for embedded platforms," https://github.com/embench/embench-iot, 2019.

[22] W. Snyder, "Verilator 4.008," 2018. [Online]. Available: https://www.veripool.org/ftp/verilator_doc.pdf

[23] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 176–188. [Online]. Available: http://doi.acm.org/10.1145/106972.106991