2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines

Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for FPGAs using the RISC-V ISA

Eric Matthews, Zavier Aguila and Lesley Shannon
School of Engineering Science
Simon Fraser University
ematthew@sfu.ca, zaguila@sfu.ca, lshannon@ensc.sfu.ca

***Abstract*—FPGA-based soft-processors have traditionally focused on fixed-pipeline designs. These designs have limited Instruction Level Parallelism (ILP) and constrain the integration of tightly-coupled accelerators, potentially limiting the speedup they can provide. Recently, it has been proposed that replacing the fixed-pipeline datapath in these soft processors with variable-latency parallel-execution functional units could facilitate the integration of custom instructions.**

In this paper, we discuss and analyze the architectural impact and requirements for decoupling the pipeline stages and supporting parallel execution units. We find that, relative to a fixed pipeline architecture, our variable-latency, parallel-execution architecture: increases resource usage by 8% LUTs and 9% FlipFlops but results in up to a 42% increase in Instruction Per Cycle (IPC), with an overall improvement of 28% MIPS/LUT. Finally, we analyze the performance tradeoffs of tightly integrating custom instructions into a fixed pipeline versus parallel execution units architecture.

I. Introduction

Until recently, most Field Programmable Gate Array soft-processor research has focused on creating resource optimized designs with high operating frequencies. These processors employ fixed-pipeline datapaths, which have limited instruction-level parallelism. There has also been work on optimizing complex out-of-order structures to FPGAs for achieving high operating frequency and performance [1][2][3]; however, these designs achieve high frequency by trading off considerable resource usage. While researchers have also studied vector processing architectures [4][5] to increase ILP, we are interested in improving ILP and instructions per cycle (IPC) for programs without single instruction, multiple data (SIMD) behaviour.

Our goal is to develop and explore the soft-processor design space focusing on resource efficient designs increasing IPC and MIPS/LUT compared to today's fixed-pipeline soft-processors without the increased resource usage of complex out-of-order structures. We use the open-source 32-bit, RISC-V Taiga processor for Intel and Xilinx FPGAs [6] as our research platform. The authors proposed that a soft processor's fixed-pipeline datapath could be replaced with variable-latency parallel-execution functional units without reducing operating frequencies or significantly increasing resource usage [6]. The Taiga paper outlined: 1) its general architecture, focusing on its pipeline structure and execution unit interface that allows easier integration of new functional units as well as its configurability; and 2) provided resource and frequency comparisons to the LEON3 [7] and Rocket [8] soft processors, two open source processors that support multicore and Linux configurations [6].

However, Matthews et al. failed to analyze the resource and operating frequency costs or IPC benefits of a variable-latency, parallel execution unit architecture relative to a fixed pipeline version of the Taiga architecture or other open-source RISC-V architectures [6]. They also did not study how the small FIFOs needed to decouple the different pipeline stages should be mapped to the FPGA fabric to improve resource usage and improve operating frequency.

Our paper investigates techniques to better map and leverage parallel execution unit pipeline designs to FPGAs focusing on improving processor performance and resource efficiency. Specifically, we present:

- An analysis of small FIFO implementations for soft-processors that improve the original Taiga's resource usage and operating frequency.
- New FPGA-optimized support for out-of-order committing with parallel execution units.
- Performance efficiency comparisons against other FPGA-based open-source RISC-V processors as a comparison of the cost of parallel execution units and early-committing.
- A case study highlighting integration advantages for tightly-coupled accelerators.

Overall, we find that while our new Taiga architecture increases resource usage by 8% LUTs and 9% FlipFlops on Xilinx and by 8% LUTs and 13% registers on Intel respectively compared to our fixed-pipeline version of Taiga , it results in up to a 42% increase in Instruction Per Cycle (IPC), for an overall improvement of 28% MIPS/LUT.

The remainder of this paper is structured as follows. Section 2 discusses related works on soft-processors, particularly those designed for FPGAs. Section 3 presents our investigation of different FIFO designs and their impact on processor resource usage and frequency. Section 4 describes our technique to increase ILP with parallel execution units. Section 5 compares the "original" Taiga [6] to our "new" Taiga (with our improvements) against our fixed-pipeline design and other open-source RISC-V processors. Section 6 presents a case study highlighting the benefits of a parallel execution unit pipeline for the integration of tightly-coupled accelerators. Finally, Section 7 concludes the paper and discusses future work.

2576-2621/18/$31.00 ©2018 IEEE
DOI 10.1109/FCCM.2018.00010

II. Background

Researchers have explored various aspects of soft-processor designs for FPGAs. Yiannacouras et al. studied various design tradeoffs for fixed-pipeline designs [9]; others have targetted minimal area and high frequency operation by leveraging DSP resources [10]. Wong et al. explored the different design considerations between FPGAs and ASICs for micro-architectural features [11]. Performance improving techniques such as runahead execution [12],support for checkpointing the register file [13], and multithreading support for soft-processors [14][15] have also been investigated.

Complex out-of-order ASIC designs [16][17] have been ported onto FPGAs, but they did not map well to FPGA fabrics, resulting in extremely large designs with low operating frequencies. Recently, there has been more work on complex, out-of-order, superscalar processors designed specifically for FPGAs; the focus has been on subcomponents such as reorder buffers [1], the instruction scheduler [2] and memory hierarchies [3]. These works have been able to achieve much higher operating frequencies compared to ASIC mapped implementations but still result in a large consumption of FPGA resources. As complex out-of-order processors consume significant FPGA resources, we are interested in design elements (such as pipeline stage decoupling and functional unit organization) that can be applied to less complex in-order, single-issue designs. To the authors knowledge, Taiga

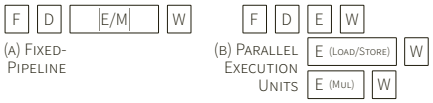

Fig. 1. Single in-order issue, (a) fixed pipeline versus (b) parallel execution unit pipeline

is the only open-source soft-processor for FPGAs with a non-fixed pipeline and *independent* (i.e. not vectorized) execution units [6]. However, the authors did not present any design space exploration. Figure 1(a) illustrates a traditional single-issue, fixed pipeline. Only one instruction is issued at a time; when multi-cycle execute phase instructions occur, the pipeline will stall until the instruction moves to the next stage. The Taiga processor [6], shown in Figure 1(b), decouples the execute phase from the decode phase. Although only one instruction is issued per cycle, it is possible for multiple instructions to be in-flight in the execute phase at the same time. We use this architectural framework, improving the original Taiga's mapping to FPGAs, to study and increase its performance efficiency.

Many existing soft-processors, including the MicroBlaze [18], NIOSII [19] and the RISC-V Rocket processor [8], provide interfaces for custom hardware accelerators. However, their fixed-pipelines impose constraints on the integration of accelerators limiting the range of accelerator designs that can be considered and/or the speedup that can be realized. With variable-latency parallel execution units and our new support for early-commits, integration of tightly-coupled accelerators are constrained primarily by the scalability of the design to support additional accelerators as we explore in Section 6.

A. FPGA-based RISC-V Processors

As the RISC-V ISA has matured, the number of processors designed for FPGAs has grown including GRVI [20], ORCA [21], PicoRV32 [22], VexRISCV [23] and Taiga [6]. All of these processors, with the exception of Taiga, feature fixed pipeline designs.

TABLE I
RISC-V FPGA-Based Soft-Processor Comparisons.

		Extensions							
	Base	**M**	**A**	**C**	**Local Mem**	**Caches**	**MMU TLBs**	**Open-Source**	**Multiple Data Sources**
GRVI	32	mul only	LR/SC only	no	instr-only	no	no	no	no
PicoRV32	32	yes	no	yes	yes	no	no	yes	no
ORCA	32	yes	no	no	yes	no	no	yes	no
VexRISCV	32	yes	no	no	no	yes	yes	yes	no
Taiga	32	yes	yes	no	yes	yes	yes	yes	yes

Table I summarizes the different configuration options of existing FPGA-Based RISC-V designs. This includes the ISA bit-width (32-bits) and the extensions it supports (M: Multiply & Divide, A: Atomic, C: Compressed) as well as memory configurations (LocalMem/Scratch and Caches). It highlights the large variation in what each processor supports, without even considering the design constraints each processor is targeting. To the authors knowledge, Taiga is the only FPGA-based RISC-V processors that supports multiple different data sources (that are all memory mapped) in a single configuration (ie. System bus, tightly-coupled scratch memory and DDR through caches). In systems, such as PicoRV32, that support scratch memory and system buses separately, this means with a scratch memory configuration no system peripherals like a UART would be available to use.

The GRVI processor [20] focuses on area optimization with its design hand-mapped to Xilinx's Ultrascale FPGAs. This approach makes it a highly efficient processor for large scale replication; however, it is not open-source and does not support RISC-V ISA components such as the Control and Status Registers (CSRs) or the full Multiple/Divide extensions utilized in the experiments presented in this paper. The PicoRV32 processor [22], is an FPGA-Based size and frequency optimized design that supports both local memory and AXI bus interfaces. It supports the Multiply and divide (M) extension and bus support that is used in this paper. ORCA [21], is a cross-vendor FPGA-Based RISC-V processor that can be configured with a four or five stage pipeline. The VexRISCV [23] processor is a cache based design, written in a scala derived language with a modular design.

We have chosen ORCA and PicoRV as comparison points due to their support for local memory and large shared overlap with Taiga configurations. The paper that introduced Taiga [6], found that the Rocket processor mapped poorly to FPGA fabric, not leveraging DSPs for multiply support and a branch table design that mapped entirely to LUTs combined with a low operating frequency. As such, the Rocket processor is excluded from our comparisons.

B. Overview of the Taiga Processor

An overview of Taiga's design [6], the platform for our investigations into performance efficiency of parallel execution units, is provided in Figure 2. The parallel execution unit's

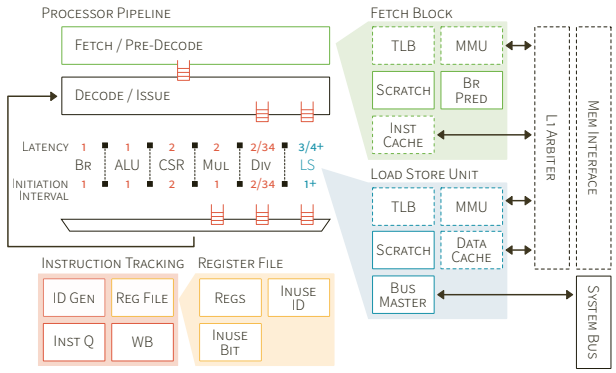

Fig. 2. Taiga Overview and Pipeline Structure. Latency and throughput information is provided for each of the execution units neglecting the effect of FIFOs on their inputs/outputs. The number above each unit is the latency and the number below is the rate at which the unit can start additional requests (the iteration interval). Where multiple numbers are presented, there can be multiple different latencies and for units with variable latencies, the numeric value of the minimum latency is given in conjunction with a plus symbol.

latency and initiation intervals are labelled above and below the units, respectively. The impact of FIFO buffering on units is not accounted for in those numbers. While Taiga supports Caches, TLBs and MMUs they are not used in this study and are marked with dotted lines in the Figure.

Our work modifies this baseline Taiga processor and investigate alternative FIFO implementations along with our replacement of the instruction tracking and writeback logic to facilitate out-of-order commits.

III. Investigating the Impact of FIFO Implementations on Processor Frequency and Resource Usage

FIFOs are an essential component of decoupled pipelines, allowing different processor stages to advance at different rates. At the issue phase, supporting buffering for operations that have longer initiation intervals reduces stalls in the issuing logic allowing for greater ILP. Additionally, as the pipeline grows more complex through the addition of caches and TLBs, FIFOs can be used to isolate complex control signals.

FIFOs are commonly used for buffering in ASIC processors; modern FPGA support for LUT RAMs and large register provisioning enable small FIFOs to be implemented efficiently for more flexible and higher performance processor designs. There are several ways to architect FIFOs, with different vendor support for different configurations. Figure 3 presents three FIFO designs that vary based on whether the input or output is mapped to a fixed location. While the FPGA vendors provide FIFO generators, their selection of configuration options for smaller, non Block RAM based FIFOs is limited. First-word-fall-through FIFOs that are 32-122 bits wide (bit widths of the FIFOs within Taiga) with 2-4 entries are investigated in this paper. The depth of the FIFOs is determined either by how many entries are needed to absorb small bursts of instructions (4 for the LS unit, 2 for the div unit), or by the requirements of control flow signals (early_full signals) as is the case for the instruction fetch buffer. For the FIFOs shown in Figure 2, the fetch/decode FIFO, and LS FIFOs have depth 4, all other FIFOs have depth 2. Additionally, there is a 4 entry FIFO inside of the Load/Store unit not shown in the Figure. Neither Xilinx nor Intel's FIFO generators support sizes of 2 nor do they support the full range of configurations tested here.

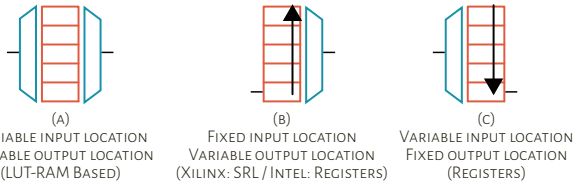

Fig. 3. FIFO Variations. Type (A) is a memory based FIFO with separate read and write indices. Type (B) is shift register based with a fixed input location and a read index. Type (C) is also shift register based, but with a fixed output location and a write index. For types (B) and (C) the arrow indicates the direction of shift register movement (for B on push, and for C on pop)

The unmodified Taiga processor utilized type A FIFOs for its FIFO implementation. We have created two new types of FIFOS, Types B and C, and made FIFO selection configurable for each FIFO within Taiga. All FIFOs share the same occupancy tracking logic for valid, full, early_full signals and differ only in their data I/O and storage mechanisms. Depending on the FPGA vendor, these structures can map differently. For Xilinx and Intel, type A FIFOs map to LUT-RAMs and are typically the most resource efficient implementation. Type B FIFOs map to Shift Register LUTs (SRLs) on Xilinx FPGAs and to registers and LUTs on Intel FPGAs. As these FIFOs have a fixed register for their input, they may result in better timing for critical paths that terminate at the FIFO. Optionally, for Xilinx, the type B FIFOs can be forced to map to registers and LUTs based on a depth threshold. Type C FIFOs map to registers and LUTs for both vendors. For type C FIFOs, the output is a fixed register which may increase frequency in designs where the output is needed early in a pipeline stage.

Tables II and III present the results of our FIFO implementation investigation on Xilinx and Intel FPGAs, respectively. During our FIFO investigations and the addition of our out-of-order commit logic (Section 4), we improved the original Taiga processor design. This includes improving resource usage when only a subset of bus/cache/scratch memory is used and trimming control logic in the decode stage. Additionally, the unit done signals were split to "special case" the control path for units that complete in a single cycle; this improves our clock frequency and our out-of-order commit support (Section 4). As such, we have also included an original Taiga entry, labelled as *unmodified Taiga*, in the tables as a reference point (FIFO depths are the same across all configurations). To obtain realistic resource usage and frequency numbers for the processor, the processor is built as part of a system with 128KB of scratch memory, and an AXI/Avalon bus connected to a UART. The configuration used for the processor includes Multiply and Divide support along with a 512 entry branch predictor and early-commit logic. The numbers reported in the tables are for the processor core only. For the Xilinx FPGA, the type B FIFOs (3), can be implemented using SRLs or Registers; as such, both variants are presented. The *Mixed* entry includes FIFOs of various types that have been selected

TABLE II
(XILINX ZYNQ X7CZ020) COMPARISON OF FIFO IMPLEMENTATION CHOICE ON PROCESSOR RESOURCE USAGE AND OPERATING FREQUENCY

Processor with FIFO Type	LUTs	FFs	Slices	BRAMs	DSPs	Freq(MHz)
unmodified Taiga[1]	1682	959	579	1	4	109
A (LUT-RAM)	1689 (0%)	770 (-20%)	568 (-2%)	1	4	117 (+7%)
B (SRLs)	1668 (-1%)	773 (-19%)	570 (-2%)	1	4	111 (+2%)
B (Registers)	1725 (+3%)	1626 (+70%)	613 (+6%)	1	4	115 (+6%)
C (Registers)	2118 (+26%)	1646 (+72%)	744 (+28%)	1	4	117 (+7%)
Mixed	1553 (-8%)	1038 (+8%)	536 (-7%)	1	4	115 (+6%)

[1] Original Taiga [6] used Type A LUT-RAM FIFOs

TABLE III
(INTEL ARRIA 10 (GX115)) COMPARISON OF FIFO IMPLEMENTATION CHOICE ON PROCESSOR RESOURCE USAGE AND OPERATING FREQUENCY

Processor with FIFO Type	LUTs	FFs	ALMs	M20Ks	DSPs	Freq(MHz)
unmodified Taiga[1]	1765	893	1443	2	3	232
A (LUT-RAM)	1614 (-9%)	866 (-3%)	1330 (-8%)	2	3	221 (-5%)
B (Registers)	1879 (+6%)	1868 (+109%)	1568 (+9%)	2	3	233 (0%)
C (Registers)	2243 (+27%)	1838 (+106%)	1687 (+17%)	2	3	223 (-4%)
Mixed	1723 (-2%)	1408 (+58%)	1419 (-2%)	2	3	236 (+2%)

[1] Original Taiga [6] used Type A LUT-RAM FIFOs

to maximize operating frequency per LUT. Finally, the original Taiga design is considered the baseline for all comparisons; and percentage increases/decreases are calculated relative to the unmodified Taiga design.

For both vendors, fully register-based designs (FIFO Types B-Reg and C) require between 16% and 39% more LUTs in addition to over 110% as many FlipFlops compared to the use of LUT-RAM (Type A) based FIFOs and require an additional 8-31% additional total logic blocks. We find that for the Xilinx FPGA that LUT-RAMs (Type A) and SRLs (Type B-SRL) result in a design with the same size, varying by a percent or less in LUTs and FFs, but have slightly different performance characteristics with the Type A based design clocking 5% faster. When the input data to the FIFO is part of the critical path (e.g. input to Load/Store unit), the SRL FIFOs marginally increase clock frequency, and when the output of a FIFO was part of the critical path (e.g. Fetch instruction buffer output) LUT-RAM FIFOs offered better performance.

Type C FIFOs require the largest amount of resources for both vendors. Their design has each register input multiplexed between its neighbour and the input data, based on the push and pop signals. This results in a larger design than the Type B FIFOs, which have only a single mux for the output.

For Xilinx, we found that that smaller Type B style FIFOs, of size 2 are best implemented as LUTs and registers, resulting in lower resource utilization than forcing their mapping to SRLs or using Type A LUT-RAMs (This threshold is the synthesis tool default for Xilinx). As such, we found that for Xilinx, a mixed approach leads to a further reduction in resource usage over a design implemented purely with SRLs or LUT-RAMs. In our Mixed implementation, the fetch buffer is implemented with a LUT-RAM (Type A), the input of the Load/Store unit with an SRL (Type B-SRL) and all other FIFOs (which have depths of 2) are implemented as Type B register-based FIFOs. This results in a design that has the highest operating frequency per LUT.

For the Arria 10 FPGA, the LUT-RAM outputs (Type A) that are part of the critical path were found to have a larger impact on the clock frequency. We also did not find the same cost savings for small (2 entry) Type B FIFOs as with Xilinx. As such, the only FIFO that benefited in changing from a LUT-RAM type was the instruction fetch buffer as its outputs contribute to the critical path of the processor. The Mixed design for the Arria 10 has the instruction fetch buffer as a Type B FIFO and the rest as LUT-RAMs. This configuration has approximately the same frequency/LUT ratio as the fully LUT-RAM based version, but has a slightly higher clock frequency and was thus chosen as the baseline Intel configuration for the rest of the paper.

IV. EFFICIENTLY LEVERAGING PARALLEL EXECUTION UNITS IN SOFT-PROCESSOR DESIGN

A potential advantage of parallel execution units is that shorter latency instructions may complete before longer latency instructions that were issued first. However, if the processor commits instructions purely in execution order, this extra performance potential is not utilized. Very high performance techniques, such as register renaming have been studied for FPGAs [13], but even when optimized for FPGAs they still require a large number of resources. Our goal is to target optimizations that are still efficient in terms of performance gained per LUT. Towards that goal, we were inspired by techniques developed earlier in processor design such as scoreboarding [24]. We developed our own solution that: maps efficiently to FPGA resources, handles Write-After-Write (WAW) hazards efficiently and can support precise exceptions.

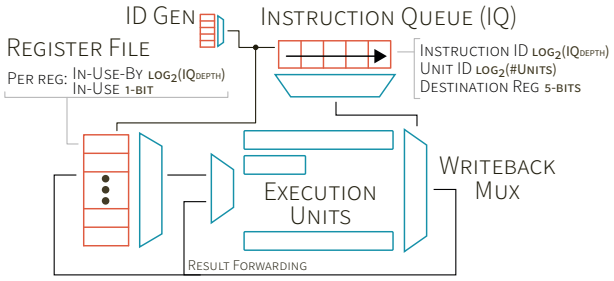

Fig. 4. Major Components of Instruction Tracking and Instruction Flow.

Figure 4, provides an overview of the major components used in issuing, tracking and committing instructions: ID generation, the Register file, the *Instruction Queue* and the writeback logic. Inside the register file, as detailed in Figure 4, every register has an in-use bit and and in-use-by field (width determined by depth of the *Instruction Queue*) that tracks whether there is an outstanding write to that register. When an instruction that writes to the register file is issued, it is assigned a unique ID (across all instructions that are in-flight that write to the register file). That unique ID is stored in the the corresponding in-use-by field and the in-use-bit is set. On completion, if an instruction's ID matches the in-use-by ID, the result is committed; otherwise, this indicates that there is a newer instruction in-flight that also writes to this register. Instructions that try to read a register with its in-use bit set will not be issued, until the bit is cleared, unlike

scoreboarding which splits the issue and operand read phases, or a forwarding path makes the data available. This approach allows multiple outstanding writes to the same register to be in-flight at any given time. The limit for outstanding writes to the register file is the same as the global limit for the number of IDs. This is a configuration parameter for the processor that is currently set to four. During testing, none of our existing benchmarks exhibit behaviour where there can be more than four outstanding register writes in-flight. However, higher depths of up to eight can be set without decreasing the processor's maximum operating frequency.

The use of in-use-by IDs and in-use bits are the primary means by which data dependency handling is performed, allowing support for variable-latency units and multiple in-flight writes to the same register. The in-use-by fields, due to having only one read and write port, map efficiently to LUT-RAMs and the in-use bits require only 32 FlipFlops. If we remove the in-use-by IDs and limit the *Instruction Queue* to a size of one, we have a configuration that behaves effectively like a 4-stage pipeline (fetch, decode, execute and writeback). Early-commit support only requires an additional 57 LUTs and 20 FFs over this configuration. Additional details relating to performance/resource efficiency are discussed in Section 5.

Instruction committing is handled by communication between the *Instruction Queue* block, the *write-back mux* and the execution units. The *Instruction Queue* entries store a unique ID for each instruction along with a corresponding unit ID. The structure is a shift register where any entries can be read or popped in any given cycle. Additionally, the destination register address is stored in a small LUT-RAM table using the instruction ID as an index. Each cycle, the contents of the *Instruction Queue* are scanned by logic in the write-back mux and the oldest instruction that will be complete in the next cycle is selected for committing, unlike scoreboarding which does not maintain an ordering of instructions. If there is an instruction within the execution units that could cause an exception, the write-back mux reverts to in-order commit ordering and will only select the oldest instruction even if other subsequent instructions are ready to commit.

Read-After-Write (RAW) hazards are handled by stalling the issue logic until either the previous result is ready or a forwarding path is available for that instruction. An example of which is the the Load/Store unit. Write-After-Read (WAR) hazards are prevented as operands are either read at issue time or are provided by forwarding paths known at issue time. Write-After-Write (WAW) hazards are avoided, not by stalling, as is the case for scoreboarding, but by allowing multiple concurrent writes to a register to be in progress, with only the most recently issued write set to update the register. Load-store loops, that occur during memory copy operations, are an example where multiple instructions can be in-flight that write to the same register. We find that supporting this type of operation improves performance by up to 6% in benchmarks such as Dhrystone. Additionally, the ID fields, which enable this support, are needed for identifying instructions that cause exceptions after being issued.

V. Efficiency Comparisons against open-source RISC-V Processors

In this section, we evaluate the performance efficiency, in terms of performance/LUT, of our instruction issue/tracking/commit logic as well as perform comparisons evaluating the overall efficiency of the processor implementation against ORCA [21] and PicoRV32 [22] two open-source FPGA-based RISC-V processors.

A. Test Configuration

For these experiments, all three processor systems were built with the base integer instruction set and the Multiply and Divide extension. The example system available for ORCA [21] for the Zedboard was used for both ORCA and PicoRV32. This system has the processor connected to an AXI bus for instruction fetch and data access, with block RAM memory (128KB), UART, and DDR connected to the bus. For our modified Taiga, we created a system with the same memory mapping, but with the scratch memory directly connected to the processor as only Taiga supports tightly-coupled local memory *and* bus support in a single configuration.

The ORCA and PicroRV32 systems used the same binary, and the Taiga build used the same compile flags and memory mapping. Comparisons of instructions executed between PicoRV32 and Taiga varied by less than 1% for all benchmarks. For ORCA, which does not include the retired instruction counter, the retired instruction count from PicroRV32 was used as they ran the same binary. As ORCA and PicoRV32 do not support tightly-coupled scratch memory and general system bus access at the same time, for our modified Taiga design we present results for multiple configurations in order to analyze the performance efficiency of more sophisticated instruction tracking behaviours. These Taiga system configurations include: 1) branch prediction and early-commits (*Taiga-Early-Commit*); 2) branch prediction and *in-order* commits (*Taiga-Inorder*: equivalent to original Taiga IPC), a "fixed-pipeline" variant with a branch predictor that only permits one unit, with one instruction, to be active at a time (*Taiga-Fixed-Pipeline*), and another "fixed-pipeline" variant but with the branch predictor removed (*Taiga-Fixed-Pipeline-NoBrPred*).

1) Benchmarks: For the various tests presented in this section, all benchmarks are compiled at the -O2 optimization level using the gcc 7.1 toolchain sourced from ORCA's github [21] repository. The Dhrystone benchmark [25] is the 2.1 version that is included with the RISC-V tool suite. AES[1] and fixed-point FFT[2] benchmarks are located online. Qsort is another RISC-V provided benchmark. The Rand benchmark uses a simple linear congruential generator to generate random numbers and loops through a non-power of two array. All benchmarks have print statements that present results of self checks at the end of the benchmark after the profiled portions. For all benchmarks, the results were in agreement across all test systems.

[1]https://github.com/kokke/tiny-AES128-C
[2]http://www.jjj.de/fft/fix_fft.tar.gz

TABLE IV
PROCESSOR RESOURCE USAGE AND OPERATING FREQUENCY COMPARISONS ON A XILINX ZYNQ X7CZ020

	LUTs	FFs	Slices	BRAMs	DSPs	Freq (MHz)
Taiga-Fixed-Pipe-noBrPred	1434	948	495	0	4	120
ORCA 4-stage	1512 (+5%)	800 (-16%)	505 (+2%)	1[1]	4	73 (-39%)
ORCA 5-stage	1625 (+13%)	934 (-1%)	527 (+6%)	1[1]	4	75 (-38%)
PicoRV32	1545 (+8%)	830 (-12%)	481 (-3%)	0	4	172 (+43%)
Taiga-Fixed-Pipe	1496 (+4%)	1018 (+7%)	523 (+6%)	1[2]	4	116 (-3%)
Taiga-Early-Commit	1553 (+8%)	1038 (+9%)	536 (+8%)	1[2]	4	115 (-4%)
unmodified Taiga [6]	1682 (+17%)	959 (+1%)	579 (+17%)	1	4	109 (-9%)

[1] BRAM is used for register file
[2] BRAM is used for dynamic branch predictor

B. Performance Evaluation

Figure 5 presents Instruction Per Cycle (IPC) results for the previously listed benchmarks. We see there is a 2.4-5x improvement in IPC for all the benchmarks (except rand) for our *Taiga-Fixed-Pipeline-NoBrPred* (i.e. our closest configuration to ORCA and PicoRV32) compared to ORCA and PicoRV32. This is largely due to the fact that both ORCA and PicoRV32 access data through the bus, where ORCA's measured bus access time was 4 cycles. This number does not include any additional latency incurred inside of the processor and it does not appear that data accesses for ORCA are pipelined. Instruction access appears to be pipelined through the bus, but would still incur additional latency on branches. For Taiga, loads to scratch memory take 3 cycles, stores 2 cycles, both are fully pipelined, and a store that uses the result of a load has an internal forwarding path inside the load/store unit that causes no stall to occur. We believe that ORCA and PicoRV32's operating with tightly-coupled memory would have similar IPC to *Taiga-Fixed-Pipeline-NoBrPred* (i.e. if they supported tightly-coupled memory in conjunction with the bus). As the impact of not supporting both at runtime is quite large, we use our *Taiga-Fixed-Pipeline-NoBrPred* configuration, which has a 4-stage fixed pipeline configuration with no branch prediction (similar to ORCA) as the baseline for the remainder of our discussions for IPC improvements with Taiga's architecture.

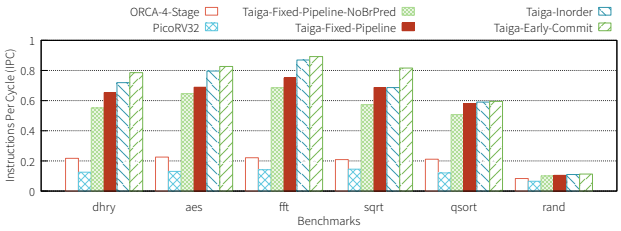

Fig. 5. Processor IPC comparison on Zynq System.

In the *rand* benchmark, the IPC results are quite similar across all processors and configurations due to the behaviour of the benchmark. In this benchmark, the majority of the runtime is spent within a tight loop generating random numbers where a remainder instruction is executed just before a load instruction dependent on the result. As this instruction has a 32+ cycle latency on all the processors, the impact of higher latency loads is greatly diminished. Additionally, the extra support in our version of Taiga for increasing ILP does not provide an advantage as there is no additional ILP in the code to leverage here (assuming no speculative execution).

TABLE V
PROCESSOR RESOURCE USAGE AND OPERATING FREQUENCY COMPARISONS ON AN ARRIA 10 (GX115)

	LUTs	FFs	ALMs	M20Ks	DSPs	Freq (MHz)
Taiga-Fixed-Pipe-noBrPred	1601	1247	1278	0	3	232
ORCA 4-stage	1483 (-7%)	869 (-30%)	998 (-22%)	2[1]	3	181 (-22%)
ORCA 5-stage	1549 (-3%)	1006 (-19%)	1038 (-19%)	2[1]	3	218 (-6%)
PicoRV32[2]	1531 (-4%)	756 (-39%)	1103 (-14%)	0	3	299 (+29%)
Taiga-Fixed-Pipe	1706 (+7%)	1386 (+11%)	1405 (+10%)	2[3]	3	233 (0%)
Taiga-Early-Commit	1723 (+8%)	1408 (+13%)	1419 (+11%)	2[3]	3	236 (+2%)
unmodified Taiga [6]	1765 (+10%)	893 (-28%)	1443 (+13%)	2	3	232 (–)

[1] BRAM is used for register file
[2] PicoRV32 numbers built with processor as top-level, not part of a system
[3] BRAM is used for dynamic branch predictor

For most benchmarks, there is a clear increase in performance with each additional performance feature enabled in Taiga; *Taiga-Fixed-Pipeline* includes the branch predictor, *Taiga-Inorder* has the branch predictor and parallel execution with in-order commit, and finally *Taiga-Early-Commit* has the branch predictor and parallel execution with out-of-order commit. For *sqrt*, there is no benefit from allowing multiple instructions in-flight without also allowing instructions to commit out-of-order (which provides a 19% improvement over inorder operation). The benefit from out-of-order commits comes from allowing ALU operations to issue and commit overlapping with a multiply instruction. For *qsort*, there is little added benefit after the branch predictor is included. This occurs as *qsort* spends significant time in code segments that load a value from memory and then perform a branch instruction on the result, incurring the full latency penalty of the load instruction (3 cycles) that can often be hidden otherwise. Overall, the increase in IPC from the *Taiga-Fixed-Pipeline-NoBrPred* to *Taiga-Early-Commit* ranges from 13-42% (including rand).

Tables IV and V provide resource usage and frequency results on a Zynq and Arria 10 FPGA respectively with all builds compared against *Taiga-Fixed-Pipeline-noBrPred* as a baseline. Systems were built with vendor defaults for synthesis and place and route options. The Arria 10 results for ORCA were generated by porting their DE2 board design to the Arria 10 FPGA. For systems built on the Zynq FPGA, the total variation in size between systems (in LUTs) is less than 14% from the largest (ORCA 5-stage) to the smallest (Taiga-Fixed-Pipeline-noBrPred). ORCA has the lowest clock frequency at 73-75MHz. This is a result of its critical path extending from its issue logic through its stall logic into the AXI data bus. When built with an Avalon bus on the Arria 10 FPGA, ORCA's critical path either starts or ends in external bus logic depending on the configuration (4 or 5 stages). For both PicoRV32 (which is built with the same system as ORCA) and Taiga the critical paths were self-contained within the processors.

We can see comparing Tables IV and V that there is a wider spread in resource usage for the Intel FPGA. For the Intel FPGA, the ORCA 4-stage configuration is the smallest design, with both PicoRV32 and our new Taiga configurations requiring more resources. These differences are related to differences in LUT-RAM structures between the two vendors. Two copies of the register file are required on the Arria 10 device for PicoRV32 and Taiga as the Arria 10 LUT-RAMs

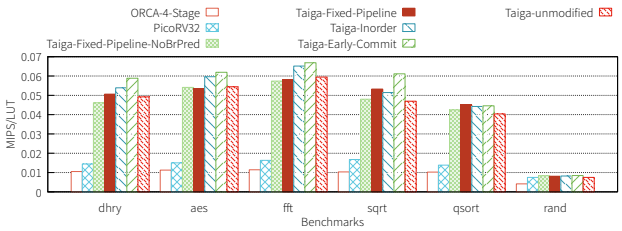

Fig. 6. Processor MIPS/LUT comparison for Zynq System. (Frequency and LUT usage is provided in Table IV)

support one-read-port, one-write-port and two read-ports are needed for the register file. The further increase in resources for Taiga comes from its additional use of LUT-RAM FIFOs. We found that as FIFO size scaled, LUT-RAMs on the Zynq device required fewer LUTs requiring less than one LUT per bit for the bitwidth of the FIFO by a factor of about 1.5, whereas for the Arria 10 device, LUT usage was always about 15% higher than the bitwidth of the FIFO. Comparing the relative increase in LUT usage across the two devices for our modified Taiga we see that percent LUT increases are the same for our Early-Commit configuration and for neither vendor do the additional architectural features significantly impact operating frequency.

Figure 6 presents the IPC results, scaled by clock frequency and LUT usage, on the Zynq FPGA on which the data was collected. For ORCA, the 4-stage configuration was selected over the 5-stage configuration as it had both higher IPC*frequency and the higher MIPS/LUT metric. Here we see that PicoRV32's higher operating frequency compared to ORCA and our modified Taiga results in it achieving a higher Millions of Instructions Per Second (MIPS)/LUT score than ORCA despite having a lower IPC. For the Taiga configurations, with the exception of the specific benchmark behaviours previously discussed, that all architectural additions over the baseline "Fixed-Pipeline" variant improve both the performance of the processor and its performance efficiency when scaled by the cost of those features. As our modified Taiga uses local memory, moving the system to the Arria 10 FPGA would not change the IPC of the processor, and as the percent increases in LUTs was the same on the Arria 10 device for the Early-Commit configuration, the general trend would be he same. From *Taiga-Fixed-Pipeline-noBrPred* to early-commit configuration we see IPC improvements of 13-42% and MIPS/LUT improvements of 1-28%. From *Taiga-Inorder* to *Taiga-Early-Commit* we find IPC improvements of 1-19% and MIPS/LUT improvements over the Taiga-unmodified inorder resource usage of 17-37%. Altogether, the combination of efficient mapping of the small FIFOs to the FPGA fabrics and support for leveraging parallel-execution units results in a processor that has higher performance and higher performance/LUT than a fixed-pipeline design.

VI. A Case Study on the Integration of Tightly-Coupled Accelerators

A significant advantage for processors with parallel execution units and support for variable-latency units emerges

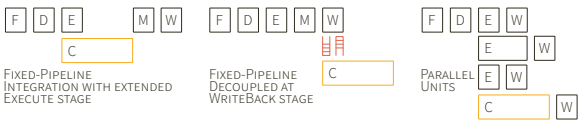

Fig. 7. Accelerator Integration Options for Processor Pipelines

when considering the integration of tightly-coupled accelerators. Figure 7 presents three examples of how tightly-coupled accelerators can be integrated into fixed-pipelines and into a pipeline with parallel execution units. When considering the performance benefit of an accelerator, the impact it has on the pipeline it is integrated into must be considered.

In the first fixed-pipeline example, the instruction is integrated into the execute stage of the pipeline. If the instruction has single cycle latency, ideally, the only additional costs are in multiplexing/and or registering the data through to write-back. However, there are considerations of whether the instruction can forward data to subsequent instructions before writing to the register file. If not, the instruction will have an additional latency penalty depending on the depth of the pipeline (in this example just 2 extra cycles/stages). In the case of a multi-cycle instruction, or further, one with variable latency, the control logic for pipeline advancement now depends on this signal as well, which could impact the clock frequency of the design.

In the second fixed-pipeline example, clock frequency concerns are mostly alleviated by isolating the additional logic with FIFOs at the expense of extra latency. Instead of stalling the pipeline, separate instruction can be used to initiate and retrieve results from the accelerator. However, for single cycle latency operations there is now additional overhead. This type of configuration is thus better suited to longer operations with larger speedups (ie. it would not be suitable for a single cycle operations that only replace a few ALU instructions).

With parallel functional units, all of the previously mentioned drawbacks are removed. In the case of single-cycle instructions operations, results can be made available on the next cycle to subsequent instructions by leveraging the write-back mux for forwarding, as is done in Taiga [6]. Longer multi-cycle operations can occur in parallel with other work as every unit has its own pipeline and results from instructions do not have to propagate through additional pipeline stages. Internal critical paths can be isolated with FIFOs without latency overhead and can be used for buffering accelerators with large initiation intervals. Additionally, with out-of-order commit support, other instructions issued after the accelerator instruction can complete if the accelerator has a long latency. The question that remains then is how is the processor's frequency impacted by adding additional units to the design.

To measure the impact on operating frequency for our modified version of Taiga we developed a simple bit-operations unit that performs a Count-Leading-Zeros (CLZ) operation or a byte swap operation. For our modified version of Taiga, the worst-case unit to add is a single cycle unit as it will have tighter placement/routing considerations to the write-back mux than a unit with registered or FIFO outputs. Additionally, units that can complete in a single cycle are checked after all

TABLE VI
RESOURCE AND OPERATING FREQUENCY IMPACT OF ADDING ADDITIONAL UNITS TO THE TAIGA-EARLY-COMMIT CONFIGURATION ON A XILINX ZYNQ X7CZ020

	LUTs	FFs	Slices	BRAMs	DSPs	Freq(MHz)
Accelerator Unit	51	1	-	0	0	-
Baseline	1553	1038	536	1	4	115
1 Accelerator	1642 (+89)	1073 (+35)	595 (+59)	1	4	114 (-1)
2 Accelerators	1747 (+105)	1108 (+35)	646 (+51)	1	4	115 +1
3 Accelerators	1794 (+47)	1148 (+40)	634 (-20)	1	4	115 (–)
4 Accelerators	1855 (+61)	1183 (+35)	621 (-13)	1	4	111 (-4)

TABLE VII
RESOURCE AND OPERATING FREQUENCY IMPACT OF ADDING ADDITIONAL UNITS TO THE TAIGA-EARLY-COMMIT CONFIGURATION ON AN ARRIA 10 GX115

	LUTs	FFs	ALMs	M20Ks	DSPs	Freq(MHz)
Accelerator Unit	69	1	-	0	0	-
Baseline	1723	1408	1419	2	3	236
1 Accelerator	1828 (+105)	1449 (+41)	1497 (+78)	2	3	221 (-15)
2 Accelerator	1812 (-16)	1496 (+47)	1490 (-7)	2	3	216 (-5)
3 Accelerator	1916 (+104)	1553 (+57)	1552 (+62)	2	3	214 (-2)
4 Accelerator	2004 (+88)	1600 (+47)	1603 (+51)	2	3	213 (-1)

possible contents in the Instruction Queue so they add the most delay to the pipeline control logic.

Tables VI and VII, present the impact on clock frequency and resource usage as the bitop unit is added and replicated. In addition to the resource cost of the unit itself, presented on the top line of the tables, there are a minimum of 34 additional registers required for input registering for the unit that is integrated with the decode logic. On the Zynq FPGA, operating frequency is not impacted until at least 4 accelerators have been added to the system, at which point there is still only a 3% decline in operating frequency. For the Arria 10 FPGA there is an initial 6% drop, with only a small decrease as additional units are added. Including the existing execution units, with 4 accelerators the processor has 10 execution units in total. Looking at the tables we can also see that the register increase is fairly constant (the min increase is 35). For 3 accelerators, there is a larger increase for both Xilinx and Intel FPGAs. At this point there are now 9 units and the unit ID width will have increased by one bit resulting in more resource usage inside of the Instruction Queue. LUT usage varies more widely with each additional accelerator, the additional LUTs coming from the additional decode logic, muxing of units outputs and the selection of which unit's result to commit. From these results we can see that Taiga provides both an infrastructure to fully leverage tightly-coupled accelerators and the ability to integrate multiple accelerators without significantly impacting clock frequency.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have shown that parallel-execution unit soft-processor designs can offer better MIPS/LUT than fixed-pipeline designs by 1-28% while also providing higher performance (IPC) by 13-42%. Our case study highlighted that this increased performance efficiency can by achieved while offering increased flexibility in accelerator integration with better ability to leverage performance from accelerators. We've shown that FIFO implementation choices are important in minimizing resources, that different types of FIFOs are best suited to different vendors FPGAs and to their size and usage within the pipeline. Future work will look into other architectural features including: more sophisticated branch prediction, increasing issue and commit width, additional forwarding paths and multi-core configurations.

VIII. ACKNOWLEDGMENTS

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) (NTGP 485577-15 and RGPN 341516), Xilinx Inc. and Microsemi corp. for supplying resources and/or funding for this project.

REFERENCES

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

[2] 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.

[3] ——, "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.

[4] P. Yiannacouras, J. G. Steffan, and J. Rose, "Vespa: Portable, scalable, and flexible fpga-based vector processors," in *CASES*. New York, NY, USA: ACM, 2008, pp. 61–70.

[5] A. Severance and G. Lemieux, "Venice: A compact vector processor for fpga applications," in *FPT*, Dec 2012, pp. 261–268.

[6] 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.

[7] *GRLIB IP Core User's Manual*, Cobham Gaisler AB. [Online]. Available: gaisler.com/products/grlib/grip.pdf

[8] K. Asanovi *et al.*, "The rocket chip generator," EECS Depart., University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[9] P. Yiannacouras *et al.*, "The microarchitecture of fpga-based soft processors," in *CASES*, 2005, pp. 202–212.

[10] H. Y. Cheah *et al.*, "idea: A dsp block based fpga soft processor," in *FPT*, Dec 2012, pp. 151–158.

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

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

[13] ——, "Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming," in *FPL*, Aug 2009, pp. 79–85.

[14] 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.

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

[16] G. Schelle *et al.*, "Intel nehalem processor core made fpga synthesizable," in *FPGA*, 2010, pp. 3–12.

[17] F. J. Mesa-Martinez *et al.*, "Scoore santa cruz out-of-order risc engine, fpga design issues," in *(WARP), held in conjunction with ISCA-33*, 2006, pp. 61–70.

[18] *MicroBlaze Processor Reference Guide*, Xilinx Inc. [Online]. Available: xilinx.com/support/documentation /sw_manuals/xilinx2016_4/ug984-vivado-microblaze-ref.pdf

[19] *Nios II Gen2 Processor Reference Guide*, Intel Corp. [Online]. Available: altera.com/en_US/pdfs/literature /hb/nios2/n2cpu-nii5v1gen2.pdf

[20] J. Gray, "Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator," in *FCCM*, May 2016, pp. 17–20.

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

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

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

[24] J. E. Thornton, "Parallel operation in the control data 6600," in *Proc. of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, 1965, pp. 33–40.

[25] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, pp. 1013–1030, October 1984.