An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor

Susumu Mashimo*, Akifumi Fujita†, Reoma Matsuo‡, Seiya Akaki†, Akifumi Fukuda†, Toru Koizumi†,
Junichiro Kadomoto†, Hidetsugu Irie†, Masahiro Goshima§, Koji Inoue*, Ryota Shioya†
* Kyushu University, Japan † The University of Tokyo, Japan
‡ Nagoya University, Japan § National Institute of Informatics, Japan
Email: susumu.mashimo@cpc.ait.kyushu-u.ac.jp

***Abstract*—High-performance soft processors in field-programmable gate arrays (FPGAs) have become increasingly important as recent large FPGA systems have relied on soft processors to run many complex workloads, like a network software stack. An out-of-order (OoO) superscalar approach is a good candidate to improve performance in such cases, as evidenced from OoO hard processor studies. Recent studies have revealed, however, that conventional OoO processor components do not fit well in an FPGA, and it is thus important to carefully design such components for FPGA characteristics.**

Hence, we propose the RSD processor: a new, open-source OoO RISC-V soft processor optimized for an FPGA. The RSD supports many aggressive OoO execution features, like speculative scheduling, OoO memory instruction execution and disambiguation, a memory dependence predictor, and a non-blocking cache. While the RSD supports such aggressive features, it also leverages FPGA characteristics. Therefore, it consumes fewer FPGA resources than are consumed by existing OoO soft processors, which do not support such aggressive features well. We first introduce the end result of the RSD microarchitecture design and then describe several novel optimization techniques. The RSD achieves up to 2.5-times higher Dhrystone MIPS while using 60% fewer registers and 64% fewer lookup tables (LUTs) as compared to state-of-the-art, open-source OoO processors.

I. Introduction

General-purpose processors play important roles in systems based on field-programmable gate arrays (FPGAs). As systems get larger and more complex, processors run many control tasks, like an operating system and a network software stack. Moreover, some systems require processors to run main compute kernels, which are impractical to deploy on dedicated hardware. For instance, Microsoft uses a combination of hard processors in an application-specific integrated circuit (ASIC) and soft processors in an FPGA in a large FPGA-based system for datacenter services [22].

As an FPGA and soft processors are increasingly used in a large variety of applications, the performance of soft processors is becoming increasingly important. Even though soft processors are never more efficient than hard processors, they remain useful because of their high flexibility, reconfigurability, and low cost, as they can be integrated without an additional chip. In the Microsoft system, for example, one main compute kernel, which is too complex to deploy on dedicated hardware, is run by specialized soft processors.

To improve the performance of soft processors, several recent studies have focused on out-of-order (OoO) superscalar approaches, as evidenced from OoO hard processor studies showing significant performance improvement over in-order approaches. While early research on FPGA-synthesizable OoO processors was mainly for ASIC prototyping [6], [7], [26], [34], these recent studies have targeted a high-performance, resource-efficient OoO soft processor on an FPGA. In particular, these studies have explored or proposed FPGA-friendly microarchitectures for the reorder buffer (ROB) [23], the rename unit [3], the issue queue (IQ) [4], [12], [30], and the memory system [29], [31]. A key insight derived from these studies is that the performance and resource efficiency of OoO soft processors are highly improved by microarchitectures leveraging FPGA characteristics.

Applying this insight, we propose the *RSD*: a new open-source RISC-V OoO soft processor. For high performance, the RSD supports several advanced microarchitectural features, like speculative OoO load and store execution, a memory dependence predictor (MDP), speculative scheduling, and a non-blocking cache. In our evaluations, the RSD achieved up to 2.5-times higher Dhrystone million instructions per second (MIPS) with 60% fewer registers and 64% fewer lookup tables (LUTs) as compared to two state-of-the-art, open-source OoO processors, as summarized in Table I.

This high performance and efficiency was achieved through two novel techniques leveraging FPGA characteristics. The first technique was FPGA-friendly *speculative scheduling*. Speculative scheduling is a technique to minimize execute-to-use latency. This technique issues instructions speculatively, before the validity of their operands is determined. We observed that speculative scheduling could achieve a gain of up to 26.8% in instructions per cycle (IPC) for SPECint 2006/2017 [27], [28] on a software simulator. Even though this technique has generally been used for OoO hard processors, it is not well studied for OoO soft processors. Hence, this paper explores an FPGA-friendly speculative scheduling implementation that achieves a better tradeoff between the performance and hardware resource overhead on an FPGA.

The second technique was optimization for multiport-RAM-based components (e.g., the physical register file (PRF)) to significantly improve the resource efficiency by using an FPGA-optimized multiport RAM. In today's open-source OoO processors, these components are built naively by using flip-flops (FFs) and logic circuits, thus constituting the dominant resource overhead. Several previous works [23], [24] have pointed out this issue for components such as the ROB. There

TABLE I
EVALUATED OPEN-SOURCE OOO PROCESSORS.

	RSD	OPA [29]	BOOM [6]
ISA	**RV32IM**	RV32IM w/o DIV, CSR	RV64GC / RV32IMAC
Ld/St Exec.	**OoO Ld/St Exec. with Forwarding**	OoO Load Exec. & InO Store Exec. w/o Forwarding	OoO Ld/St Exec. with Forwarding
Mem. Dep. Predictor	**Support**	N/A	N/A
Speculative Scheduling	**Support with IQ or Replay Queue**	Support with IQ	N/A
Cache	**Non-blocking**	Blocking	Non-blocking
TLB	N/A	N/A	Support
Memory	**BRAM or DRAM**	BRAM only	BRAM or DRAM
Interconnect	**AXI4 or AHB**	N/A	AXI4 or AHB
Language	SystemVerilog	VHDL	Chisel

are open questions, however, as to (1) which components we can apply an FPGA-optimized multiport RAM to and (2) how much this technique reduces the consumption of FPGA resources in the entire OoO processor design. We thus explain which components in the RSD had this optimization applied, and we show that it saved almost half the FPGA resources.

The remainder of this paper is structured as follows. Section II discusses related work on OoO soft processors. Section III presents the RSD microarchitecture. Section IV describes the speculative scheduling mechanisms that we explored for the RSD. Section V explains the RSD components for which we applied FPGA-optimized multiport RAM. Finally, section VI presents our evaluation results, before a brief conclusion.

II. RELATED WORK

There are several open-source, FPGA-synthesizable OoO processors, but most of them use an FPGA solely for ASIC prototyping or a research/education environment, rather than designing an OoO processor targeting an FPGA [6], [7], [10], [34]. For example, the RISC-V BOOM processor runs on an FPGA, but it is aimed at an ASIC implementation and thus not optimized for an FPGA [6].

We are currently aware of only one open-source OoO processor targeting an FPGA: the open processor architecture (OPA) [29]. The main component optimized for an FPGA is the store queue (STQ). As a conventional STQ is a content-addressable memory (CAM), which is very expensive on an FPGA, the OPA eliminates the STQ completely. Although this optimization may reduce FPGA resources and improve operating frequency, it hurts performance because a store instruction can be executed only when it becomes the oldest instruction in the processor, and a load instruction cannot forward data from a preceding store instruction.

We compare the RSD with the BOOM and the OPA, the two open-source OoO processors mentioned above, which are optimized for an ASIC and an FPGA, respectively. Table I summarizes the supported features of these processors.

III. THE RSD MICROARCHITECTURE

This section introduces the RSD microarchitecture and provides background on the proposed techniques described in sections IV and V. Fig. 1 shows a block diagram of the RSD. The microarchitecture consists of three blocks: a *front-end block*, a *scheduling block*, and an *execution block*.

The front-end block fetches and decodes instructions from the L1 instruction cache (L1IC) in program order. The current implementation uses the gshare branch predictor [17]. The following subsections describe the scheduling block and the execution block.

A. Scheduling Block

The scheduling block extracts instruction-level parallelism (ILP) for instructions sent from the front-end block, and it issues instructions to the execution block out of program order. The scheduling block mainly consists of the *rename unit*, the *dispatch unit*, the *issue queue* (IQ), and the *reorder buffer*.

1) Rename Unit: To remove false dependencies (write after write and write after read) between instructions, the rename unit renames the operand logical registers of an instruction. Specifically, it renames the logical registers of destination operands to physical registers obtained from the *PRF free list*, and it then registers the mapped physical registers in a *register map table* (RMT). The logical registers of source operands are renamed to the physical registers most recently renamed to those logical registers by using the RMT.

2) Dispatch Unit: The dispatch unit allocates an entry for a renamed instruction in several components, such as the ROB, the IQ, the *load queue* (LDQ), and the *store queue* (STQ), depending on the instruction type. The dispatch unit stalls when any of these structures is full. The following subsections describe the details of these components and the submodules of the dispatch unit.

3) Issue Queue: The IQ is responsible for issuing instructions to the execution block when all of an instruction's source operands are available. It consists of three submodules: the *wakeup logic*, the *select logic*, and the *instruction payload RAM*. The wakeup logic keeps track of the readiness of each uncompleted instruction, and the select logic selects ready instructions and issue them to the execution block.

The instruction payload RAM holds a payload (e.g., the instruction type) required for executing an instruction. When an instruction is stored in the IQ, its payload is thus stored in the RAM.

The RSD supports a speculative scheduling mechanism. The *instruction replay logic* (IRL) is a unit to support this mechanism, as described in detail in section IV.

We use a matrix-based wakeup logic with a random select logic in the current implementation [11], [12], [25]. Fig. 2 shows a block diagram of the matrix-based wakeup logic in the four-entry IQ. In this example, the rename unit renames one instruction, and the IQ issues one instruction per cycle.

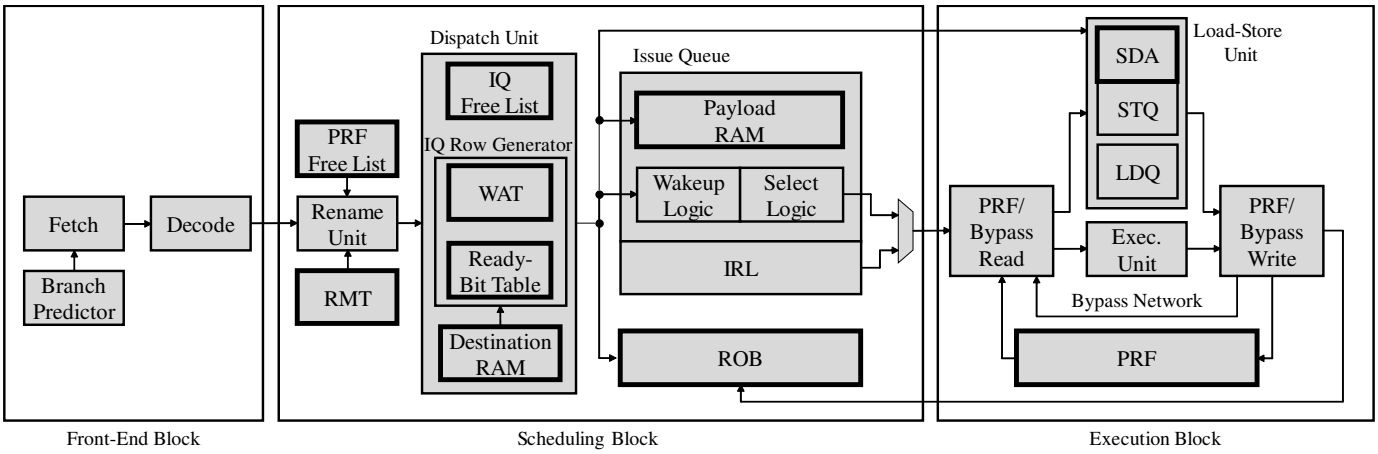

Fig. 1. Block diagram of the RSD, with multiport-RAM-based components highlighted in bold.

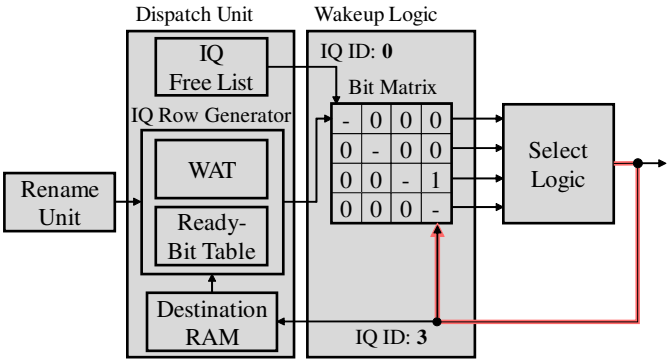

Fig. 2. Block diagram of the matrix-based wakeup logic in the four-entry, one-issue IQ. The rename unit renames one instruction per cycle in this example. The select-wakeup critical loop is highlighted in red.

The wakeup logic tracks the dependencies of instructions in the IQ by using a bit matrix of dependency bits. Each combination of a row and column corresponds to an instruction in the IQ, so that the other bits in a row indicate other instructions in the IQ. An asserted bit in a row thus means that the instruction corresponding to the row depends on the instruction corresponding to the column. For example, in Fig. 2, the asserted rightmost bit in the third row means that an instruction in the third IQ entry depends on an instruction in the fourth IQ entry. Bits in a column are cleared by the select signal of the corresponding instruction when it is selected and issued. An instruction is ready for selection when all bits in the corresponding row are deasserted.

The matrix-based wakeup logic uses IQ IDs instead of register IDs to schedule instructions; therefore, an instruction allocating an IQ entry must know the IQ IDs of the depending instructions from its source operand register IDs. For this purpose, the *wakeup allocation table* (WAT), a RAM structure, is used to convert a logical register ID to the IQ ID of the youngest older instruction that updates the logical register.

The *ready-bit table* is used for an instruction allocating an IQ entry to know the readiness of its source operands. It is updated by an issued instruction using its destination physical register ID. The *destination RAM* is used to convert an IQ ID to the corresponding destination physical register ID, because the wakeup logic knows only the IQ ID of an issued instruction.

4) Reorder Buffer: The ROB keeps the state of dispatched instructions and commits an instruction once it is executed correctly and becomes the oldest instruction in the processor. When an instruction is executed incorrectly, the instruction and the following instructions are flushed or replayed.

B. Execution Block

The execution block executes the instructions sent from the scheduling block.

1) Physical Register File and Bypass Network: The PRF holds operand physical register data. When an instruction comes to this block, it first receives its source operand data from the PRF or a preceding instruction via a *bypass network*. The corresponding execution unit then executes the instruction and writes the result data to the PRF and the bypass network.

2) Load-Store Unit: The *load-store unit* (LSU) manages loads and stores and communicates with the memory system. When a load or a store misses in the L1 data cache (L1DC), the LSU sends a cache fill request to the memory system.

The LSU involves a major design decision that affects its performance and design complexity: whether it allows loads and stores to be executed OoO *speculatively*. Loads and stores have special dependencies called *memory dependencies*, in which a load depends on an older store that writes to the same memory location. Because the memory dependency is disambiguated only at execution time, only a few loads and stores can be reordered precisely. Today's high-performance OoO processors overcome this limitation by executing memory instructions speculatively out of program order.

The RSD supports such speculative OoO memory instruction execution. To achieve this, the LSU guarantees the correctness of memory instructions by performing dynamic memory disambiguation using the LDQ and the STQ, which are CAM structures populated by loads and stores, respectively. The LSU performs dynamic memory disambiguation in the following manner.

- The *store-data array* (SDA) holds data for all stores that are not yet committed to make them appear to the memory system in program order nonspeculatively.
- A load searches older address-generated stores in the STQ and receives data from the youngest older store with the same address (store-load forwarding).

- A store searches younger completed loads in the LDQ with the same address and performs instruction replays for all matching loads and their dependents (store-load ordering violation).

In this manner, any memory access ordering violations are detected and recovered. To reduce the number of instruction replays due to store-load ordering violations, the RSD uses a *memory dependence predictor* (MDP), as described in the following section.

C. Memory Dependence Predictor

The RSD has an MDP [8], [19] to improve performance by reducing store-load ordering violations. The MDP implementation in the RSD predicts whether a load depends on any older in-flight store, instead of a store that a load depends on. When the MDP predicts a dependency, the predicted load waits in the IQ until all older stores have been issued.

This implementation may lose some opportunities to execute a load earlier, but it can be implemented with a very modest hardware overhead. The MDP consists of a 1-bit vector. When a load causes a store-load ordering violation, it accesses the bit vector by using the load's PC value, and it asserts the indexed bit. The MDP predicts a renamed load to have a memory dependency if the bit indexed by the load's PC is set. The predicted dependency is then resolved in the same way as a register dependency in the matrix-based wakeup logic.

IV. Speculative Scheduling and Replay

This section describes one key technique to improve the performance of the RSD: speculative scheduling and replay. This technique is widely used in hard processors [21], but there are few studies on its use in soft processors. Therefore, FPGA-friendly implementation of speculative scheduling and replay is an open research question. First, we give an overview of speculative scheduling and classify the existing microarchitectural options into two types. Then, we characterize implementations of each type in an FPGA.

A. Motivation and Overview

Recent OoO hard processors speculatively issue instructions that depend on instructions with variable latency. Typical instructions with such variable latency are load instructions, because their latency varies with cache hits or misses. If a consumer instruction waits until its producer load's latency is determined, then the load-to-use latency is prolonged. Fig. 3 illustrates this by showing pipeline diagrams of a load followed by a dependent instruction (a) without and (b) with speculative scheduling in the RSD. Without speculative scheduling, the decision to schedule the consumers of a load is delayed until the load is executed and a cache hit or miss is determined. On the other hand, with speculative scheduling, a load is assumed to hit in the L1DC, and consumers are issued so that the instructions are executed in the following cycle of the producer load execution on an L1DC hit. As a result, speculative scheduling saves two cycles of load-to-use latency.

This load-to-use latency reduction significantly improves performance. We evaluated the performance impact of speculative scheduling on a software simulator modeling the

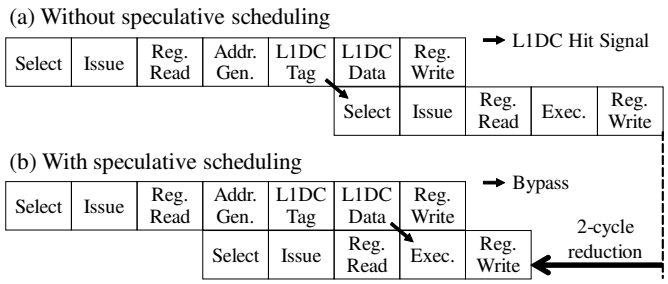

Fig. 3. Pipeline diagram of a load followed by a consumer instruction. (a) The consumer is issued once the load is known to hit in the L1DC. (b) The consumer is issued speculatively by assuming the load will hit in the L1DC.

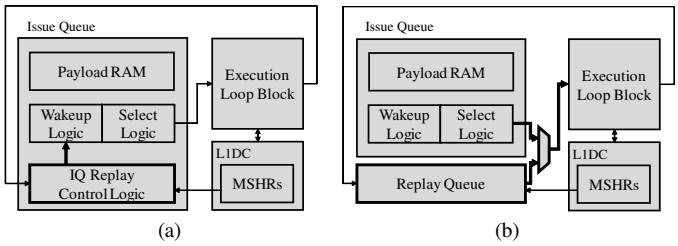

Fig. 4. Two design options for a speculative scheduling mechanism: (a) IQ-based and (b) RQ-based. The main differences are highlighted in bold.

RSD. The results showed that speculative scheduling achieved speedup of up to 26.8% (8.9% on average) over a processor without speculative scheduling for SPECint 2006/2017.

Speculative scheduling can cause misprediction when the predicted execution latency of a producer is wrong, so it requires a mechanism to aid in such misprediction. A commonly used approach is to replay (reissue) the mispredicted instruction and all speculatively scheduled consumers by using an instruction replay logic (IRL). For example, if a producer load in Fig. 3(b) misses in the L1DC, then no consumers of the load can be executed, and they are thus replayed by the IRL when the load receives data.

B. Detailed Designs

Existing design options for speculative scheduling are divided into two categories: mechanisms based on an IQ [9], [13], [18] and on a replay queue (RQ) [1], [2], [18]. We implemented both mechanisms on the RSD to explore designs that better suit to an FPGA.

1) IQ-based Design: The IQ-based design uses the IQ to keep speculatively issued instructions and re-issue them on misprediction [9], [13], [18]. We use an IQ-based design based on the *delayed selective replay* method [13].

Fig. 4 (a) shows the block diagram of the implemented design. The IQ replay control logic (RCL) interacts with the wakeup logic to replay instructions if needed. This design handles misprediction as follows:

1) Mispredicted instructions are invalidated by using *valid bits* added to the data path for each register value in the PRF. When a load misses in the L1DC, it resets a valid bit for its destination register (i.e., it invalidates its destination register). When an instruction is executed using such invalid data, it also invalidates its destination register. To avoid wastefully issuing many invalid in-

structions, when misprediction is detected, the IQ stops issuing instructions until all instructions in the execution block are drained.

2) Components speculatively updated are recovered. When an instruction is invalidated by misprediction, the IRL receives the instruction's information and recovers the instruction's dependency-bit matrix column in the wakeup logic. For these recovery actions, each dependency-bit matrix column has a shadow copy. In the same cycle, the IRL also resets the ready-bit table entry of the invalidated instruction.
3) Instructions are replayed by the IQ without any special handling, except for instructions that caused misprediction (e.g., cache-missed load instructions). Such an instruction is replayed after the cause of its execution failure is resolved (e.g., a cache line fill). For the replay, the RCL makes the corresponding IQ entry visible to the select logic once it receives data from the memory system. Once an instruction is executed without invalidation, it is removed from the IQ.

2) RQ-based Design: This mechanism replays instructions by using a structure apart from the IQ: the *replay queue (RQ)* [1], [2], [18]. When an instruction is issued from the IQ, it deallocates its IQ entry *immediately*. Then, it is pushed into the RQ for future replay and reissued or removed when it becomes ready at the head of the queue.

Fig. 4(b) shows the block diagram of the implemented design. Every cycle, instructions can be issued from either the IQ or the RQ to the execution block. When both the IQ and the RQ have instructions to be issued, the RQ issues preferentially. After the instructions are issued, they are removed from the IQ or RQ and then sent to the execution block. After flowing to the execution block, an instruction enters the RQ if it is invalidated. Similarly to the IQ-based design, an invalidated instruction invalidates dependent instructions by sending data with a valid bit reset to the PRF.

The RQ keeps and issues invalidated instructions ordered by their time of issue; therefore, the correctness of program execution is maintained. Each RQ entry stores all the instructions issued in the same cycle with *replay bits* and a latency counter. We call a set of instructions issued in the same cycle an *instruction wave*. At the end of the execution block, if any instructions in an instruction wave are invalidated, the instruction wave is pushed into the RQ. Each replay bit is set when the corresponding instruction is invalidated. The latency counter provides the mean issue latency between the pushed RQ entry and the preceding RQ entry. For example, the latency counter is set to 3 if the corresponding RQ entry was pushed 3 cycles after the preceding RQ entry. In this way, the RQ entries are ordered by time of issue, meaning that the relative issue cycles among instruction waves are maintained.

The RQ issues an instruction wave in the following manner. When the instruction wave reaches the head of the RQ, it decrements its latency counter every cycle. When the latency counter becomes zero, the invalidated instructions in the instruction wave are issued.

C. Design Comparison

The two designs described above add hardware resource overhead on different components. The IQ-based design increases the hardware resources used for the dependency-bit matrix and the ready-bit table to enable recovery from misprediction of speculative scheduling. Specifically, the shadow copy of the bit matrix increases the number of flip-flops (FFs), and the recovery logic for the bit matrix and the ready-bit table increases the number of logic circuits. In contrast, the RQ-based design consumes hardware resources on the RQ, which mainly consists of a simple FIFO (1-read, 1-write) and is mapped to a RAM.

The RQ-based design is better suited to an FPGA than the IQ-based design because of FPGA's characteristics: (1) a logic circuit is expensively emulated by using LUTs and (2) an FPGA usually has a lot of dual-port RAM primitives. In Xilinx 7-series FPGAs, for example, a part of LUTs can be used as a 6-input logic or a 1-bit, 32-entry RAM, therefore their costs on such LUTs are the same [33]; as a result, the RQ-based design requires adding only a few LUTs for the RQ, while the IQ-based design requires many LUTs to emulate its logic circuits. For example, the RQ-based design consumes up to 2.2 times fewer LUTs than the IQ-based design does in our evaluation shown in section VI-A. In an ASIC, however, a 6-input logic requires tens of transistors, while a 1-bit, 32-entry RAM requires hundreds, so it is unclear which design is more resource efficient.

D. Existing Design: the OPA

To the best of our knowledge, the OPA is the only open-source OoO soft processor that applies speculative scheduling. The OPA integrates the IQ into the ROB and uses an IQ-based speculative scheduling mechanism. The IQ keeps all renamed instructions until they are committed, and it issues the oldest ready instructions. It issues instructions depending on a load instruction speculatively before the load accesses the L1DC. When a load misses in the L1DC, it is marked as "not issued," and all dependent instructions are invalidated.

This design has some drawbacks. First, the ROB and the IQ cannot be sized separately. This may be wasteful of the IQ capacity, because the IQ should theoretically keep only uncompleted instructions, so it only needs to be 30-50% of the ROB size with almost no loss in IPC [32]. Second, the IQ-based design can consume more FPGA resources than the RQ-based design does. The evaluation described in section VI-C showed that the OPA consumes more FPGA resources than the RSD does, because the RSD uses the RQ-based design and decouples the ROB and IQ.

V. FPGA-Friendly Multiport RAM Optimization

This section describes a technique to reduce the FPGA resources by using FPGA-optimized multiport RAMs.

A. Motivation

An OoO processor has many RAM-based components (e.g., the ROB). These components consist of heavily multiport RAMs to process multiple instructions simultaneously within

TABLE II
MULTIPORT-RAM-BASED COMPONENTS IN THE RSD.

	Component	RAM type
Rename Unit	PRF Free List	Banked RAM
	RMT	I-LVT
Dispatch Unit	IQ Free List	Banked RAM
	Ready Bit Table	I-LVT
	WAT	I-LVT
	Destination RAM	I-LVT
Issue Queue	Payload RAM	I-LVT
Reorder Buffer	ROB Meta Data	Banked RAM
	ROB Exec. State	I-LVT
Physical Register	PRF	I-LVT
Load-Store Unit	SDA	I-LVT

a single cycle. They are traditionally mapped to FFs and logic circuits, because commercial FPGAs usually do not support multiport RAM primitives that have two write ports or more.

We found that existing open-source OoO soft processors, including the OPA, implement most of these components by using FFs and logic circuits, thus presenting a great opportunity to improve resource efficiency for those components.

B. RAM Construction Methods

Several studies have focused on an FPGA-optimized multiport RAM design using two-port RAM primitives on an FPGA (e.g., BRAMs or distributed RAMs on Xilinx FPGAs) [5], [14]–[16]. These techniques consume far fewer FPGA resources than an FF- and logic-based implementation does. Unfortunately, no current FPGA synthesis tool supports automatic detection of multiport-RAM-based modules and mapping onto RAM primitives by using these techniques.

Banking is another powerful technique that provides multiport RAM functionality [16]. In a banked RAM, multiple requests are distributed to multiple banks. The bank accessed from each port is usually determined by a modulo operation on an address. A banked RAM behaves the same as a true multiport RAM as long as accessed banks do not conflict. Queues or FIFOs that push or pop multiple elements simultaneously are suitable for a banked RAM. This is because accesses in such queues are always consecutive; thus, the address modulo always differs, and a bank conflict never occurs.

C. Implementation

We carefully designed the RSD by applying FPGA-friendly multiport RAM and a banked RAM in as many components as possible. We used the I-LVT [5], a state-of-the-art multiport RAM design, in the RSD. Table II lists the OoO processor components in the RSD that were implemented using the I-LVT and a banked RAM.

We implemented three components by using a banked RAM instead of the I-LVT: the PRF free list, the IQ free list, and the ROB metadata. These components are queue structures, so a bank conflict never occurs. The two free lists are queues to provide an available index of an attached component (i.e., the PRF or the IQ) for an incoming instruction.

The ROB is allocated by renamed instructions and deallocated by committed instructions in program order, so most parts of the ROB can be implemented as a queue. The execution status (e.g., completed with/without fault) part, however, is updated out of program order in the nature of OoO execution, which causes bank conflict in a banked RAM.

TABLE III
MICROARCHITECTURAL PARAMETERS.

	RSD	OPA	BOOM
Fetch/Decode/ Rename Width	2		
ROB	64		
PRF	64	98	64
IQ	16	64	16 Int/Mul/Div, 16 Ld/St
LDQ	16	64	16
STQ	16	-	16
Issue Port	2 Int, 1 Ld, 1 St, 1 Mul/Div	2 Int, 2 Ld/St/Mul	3 Int/Mul/Div, 1 Ld/St
Branch Predictor	Gshare + RAS	Loop Pred. + RAS	Gshare + RAS

M. Rosiere et al. proposed a banking-based ROB architecture with a small queue to handle bank conflict by delaying conflicting execution status updates in the queue [23]. A "queue-full" event stalls a processor to drain update requests, so the processor performance decreases with this ROB design. The authors reported up to 10% IPC degradation in comparison to a processor with a true multiport-RAM-based ROB.

To avoid such performance loss, we developed a hybrid ROB design that combines a banked RAM and a true multiport RAM. Specifically, we implemented the execution status part by using a true multiport RAM and the rest using a banked RAM. This optimization enables a large part of the ROB to consist of a banked RAM, thus reducing the use of FPGA resources without performance degradation.

VI. EXPERIMENTAL RESULTS

In this section, we demonstrate the high performance and high resource efficiency of the RSD by evaluating it in the following three ways: (1) evaluating the operating frequency, resource consumption, and performance of different implementations of the speculative scheduling mechanism described in section IV; (2) measuring the impact of the FPGA-friendly multiport RAM optimization described in section V; and (3) comparing the RSD with several open-source OoO processors.

All evaluated implementations were synthesized by using Synopsys Synplify Premier M-2017.03-SP1 and targeting the Xilinx Zynq XC7Z020-CLG484-1. The left column in Table III lists the microarchitectural parameters of the RSD. We did not include the FPGA resources for the memory system, including the L1IC and L1DC, in the resource evaluations.

A. Speculative Scheduling Mechanisms on FPGA

1) Experimental Setup: We built circuits for the speculative scheduling mechanisms described in section IV (i.e., the IQ- and RQ-based mechanisms). We first synthesized the speculative scheduling circuits alone and observed the FPGA resources and operating frequency. Then, we measured the performance of each configuration by synthesizing the entire RSD design and running the Dhrystone 2.1 benchmark. In both evaluations, we swept the IQ entries from 16 to 32. We fixed the RQ size to 32, because the RQ was mapped to a distributed RAM primitive in the target FPGA, and the minimum number of entries for the RAM primitive was 32.

TABLE IV
OPERATING FREQUENCIES OF THE IQ- AND RQ-BASED DESIGNS IN MHZ.

	IQ=16	IQ=24	IQ=32
IQ-based	128.3	120.3	107.4
RQ-based	124.2	122.5	105.8

TABLE V
DHRYSTONE MIPS OF THE RSD WITH DIFFERENT SPECULATIVE SCHEDULING MECHANISMS.

	IQ=16	IQ=24	IQ=32
IQ-based	181.7	187.4	186.9
RQ-based	188.6	188.5	188.1

2) FPGA resources: Fig. 5 shows the FPGA resource utilization of the IQ- and RQ-based speculative scheduling mechanisms. The results show that the RQ-based design consumes fewer FPGA resources in all configurations of IQ size. The largest contributor is the wakeup logic, whose main component is the dependency-bit matrix. Because of the shadow copy of the bit matrix for misspeculation recovery, the FPGA resources used for the bit matrix in the IQ-based design are significantly larger than in the RQ-based design.

Although the RQ-based design consumes FPGA resources for the replay queue, the resource overhead is much lower than that of the wakeup logic. This is because the main component of the replay queue is a simple 1-read, 1-write FIFO, which can be efficiently mapped to distributed RAM primitives.

3) Operating Frequency: Table IV lists the operating frequencies, with only a minor change between the IQ and RQ-based designs. This is because the critical-path logic circuits for all configurations are the same as the select-wakeup critical loop shown in Fig. 2, which is not related to the logic for the speculative scheduling mechanism.

4) Performance: Table V lists the Dhrystone MIPS (DMIPS) results for the RSD with the IQ- and RQ-based designs. Note that the operating frequencies of all six RSD configurations were almost the same, at around 95 MHz, because the critical-path logic is in the branch predictor. The results show that the RQ-based design achieved higher DMIPS than the IQ-based design did in all IQ-size configurations. In particular, when the IQ size was small, the performance gap was as large as +3.7%. This is because an instruction can free its IQ entry as soon as it is issued in the RQ-based design, whereas it must be kept in the IQ until it checks its execution status in the IQ-based design. This advantage of the RQ-based design reduces the number of instructions populated in the IQ,

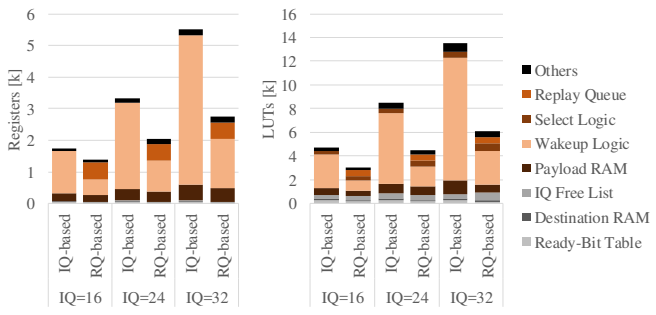

Fig. 5. FPGA resources used by the IQ- and RQ-based speculative scheduling mechanisms.

TABLE VI
FPGA RESOURCES AND OPERATING FREQUENCY OF THE RSD WITH DIFFERENT MULTIPORT RAM IMPLEMENTATIONS.

	FF+Logic	I-LVT+Bank
LUTs	28166	15379 (54.6%)
Registers	13511	8584 (63.5%)
Freq. [MHz]	89.8	95.3 (106.1%)

thus improving the performance when the IQ size is small.

5) Conclusion: We conclude that the RQ-based speculative scheduling mechanism better suits an FPGA in terms of FPGA resources and performance than the IQ-based mechanism does. The main reason for the higher resource efficiency is that the RQ is a FIFO structure that can be efficiently mapped to FPGA RAM primitives, whereas the IQ is a complex structure built on FFs and logic circuits. We thus used the RQ-based implementation in the remaining evaluations.

B. Multiport-RAM-Based Optimization

1) Experimental Setup: We built two RSD designs, in which all the components listed in Table II were implemented with (1) FFs and logic circuits (*FF+Logic*) and (2) I-LVTs and banked RAMs (*I-LVT+Bank*). We synthesized the complete RSD designs and observed the required FPGA resources and the operating frequency.

2) FPGA Resources and Operation Frequency: Table VI lists the numbers of LUTs and registers used in each configuration. The percentages in the table give the relative amount of FPGA resources normalized to the *FF+Logic* configuration. The results show that using FPGA-friendly multiport RAM saves FPGA resources significantly, requiring 45% fewer LUTs and 36% fewer registers.

As listed in the bottom row of Table VI, the *I-LVT+Bank* configuration improved the operating frequency by 6% over that of *FF+Logic*. Note that the critical path of each design is not related to the multiport-RAM-based components listed in Table II. Therefore, the frequency improvement could have been caused by the decrease in FPGA resources and the complexity of the synthesis task.

C. Core-Level Comparison

1) Experimental Setup: We compared the RSD with two open-source RISC-V OoO soft processors: the BOOM [6] and the OPA [29]. The BOOM is one of the most popular open-source OoO processors[1]. While the BOOM can be configured as either a 32- or 64-bit processor, we used the 32-bit configuration for fair comparison. As for the OPA, to the best of our knowledge, it is the only open-source OoO processor optimized for an FPGA.

Table III summarizes the microarchitectural parameters of the evaluated processors. Because of the microarchitectural limitations of each processor, we could not use exactly the same parameters for all processors. For example, the IQ size and LDQ size of the OPA must be the same as the ROB size. Therefore, we used the same parameters for the fetch/decode/rename width and the ROB size, which dictate the front-end bandwidth and the maximum number of in-flight

[1]As measured by the number of stars in GitHub.

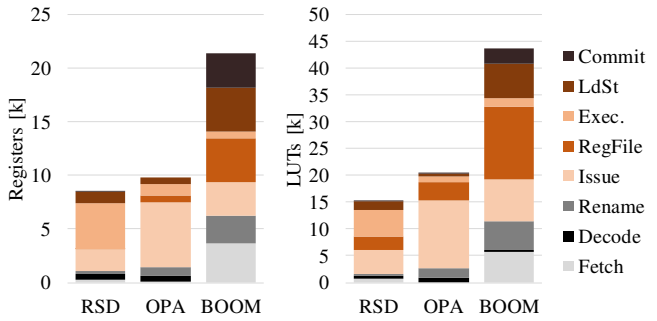

Fig. 6. FPGA resources used by the evaluated OoO processors.

instructions, respectively, and we otherwise used the same parameters for as many components as possible. All processors were evaluated with the 8KB L1IC, the 8KB L1DC and a one-cycle latency main memory.

2) FPGA Resources: Fig. 6 shows the FPGA resource utilization of the evaluated processors. The RSD consumes the fewest FPGA resources. One of the largest contributors to its high resource efficiency is the FPGA-friendly multiport RAM optimization, as discussed in the previous section. Note that the OPA and the BOOM implement most multiport-RAM-based components by using FFs and logic circuits. As a result, the RSD consumes 88%/40% of registers and 76%/36% of LUTs in comparison to the OPA/BOOM, respectively.

The RSD and the BOOM use few block RAMs (BRAMs) out of the 140 BRAMs on the target FPGA. The RSD uses six BRAMs: two BRAMs for the MDP, and four for the branch predictor. The BOOM uses three BRAMs for the branch predictor. The OPA does not use a BRAM.

3) Operation Frequency: The first row in Table VII lists the operating frequencies of the three processors. *hdiv* and *sdiv* indicate the cases with and without a hardware divide instruction, respectively. No hdiv result for the OPA is listed because of the lack of hardware divider support. The frequency of the RSD was lower than that of the OPA and higher than that of the BOOM.

The frequency gap between the RSD and OPA is mainly due to the STQ and the branch predictor, which are the critical-path logic circuits of the RSD. The RSD has a CAM-based STQ and a complex global branch predictor. On the other hand, the OPA eliminates the STQ, as described in section III-B2, and it uses a very simple branch predictor. Therefore, the OPA has fewer timing-critical components and achieves a higher operating frequency. In the next subsection, however, we show that such a design can degrade the performance significantly.

The critical path of the BOOM, like the RSD, is in the branch predictor. The BOOM has as expensive global branch predictor; in addition, it is not optimized for an FPGA.

TABLE VII
OPERATING FREQUENCY AND DHRYSTONE MIPS (DMIPS).

	RSD		OPA		BOOM	
	hdiv	sdiv	hdiv	sdiv	hdiv	sdiv
Freq. [MHz]		**95.3**		134		71.7
DMIPS	**194**	177	N/A	77.5	76.1	74.5
DMIPS/MHz	**2.04**	1.86	N/A	0.577	1.06	1.03

4) Performance: The bottom two rows in Table VII list the DMIPS and DMIPS/MHz results for the three processors. The RSD achieved significantly higher DMIPS and DMIPS/MHz than either the OPA or the BOOM did. While the OPA achieved a 1.4-times higher operating frequency, its DMIPS/MHz was 30% of that for the RSD without a hardware divide instruction. As a result, the RSD achieved a 2.2-times higher DMIPS than the OPA did.

The significant performance gap between the RSD and the OPA is mainly caused by two design differences. One is the design of a branch predictor. The OPA has only an extremely simple branch predictor, which does not include a branch direction predictor, and does not perform branch prediction except for return instructions and simple loops. Therefore the OPA has large branch misprediction penalties on many branch instructions. The other difference is the STQ. The STQ-less design of the OPA can lose performance significantly, because a store instruction stalls instruction commit by taking several execution cycles when it becomes the oldest instruction in the ROB. The commit delay frequently makes the ROB full, thus stalling the front end. We evaluated the performance impact of the lack of OoO store execution ability by using the Onikiri software simulator [20]. Note that the simulator modeled a conventional OoO processor rather than the OPA. The evaluation showed that the STQ-less optimization degraded the IPC by up to 40% (16% on average) for SPECint 2017 and SPECint 2006 [27], [28]. This result implies that we must simultaneously optimize the design for both the operating frequency and the IPC.

In comparison to the BOOM, the RSD achieved 2.5-times higher DMIPS and 1.9-times higher DMIPS/MHz. One cause of the lower DMIPS/MHz for the BOOM is that a one-cycle bubble is always inserted whenever a branch is predicted as taken. In contrast, the RSD can fetch instructions in every cycle. In addition, the RSD can take advantage of speculative scheduling, which the BOOM does not support.

VII. CONCLUSION

This paper has introduced the RSD: a high-performance, resource-efficient RISC-V OoO soft processor. To improve its performance, we explored FPGA-friendly implementations of speculative scheduling and an instruction replay mechanism, which contribute to both performance and resource efficiency. Our experimental results showed that FPGA resources can be significantly saved by carefully implementing multiport-RAM-based components using state-of-the-art FPGA-friendly multiport RAM. As a result, the RSD achieved up to 2.5-times higher DMIPS while consuming fewer FPGA resources than two state-of-the-art, open-source OoO soft processors did. The source code of the RSD is here: https://github.com/rsd-devel.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP16H05855, JP19H01105, JP19H04077. Also, this research was supported by VLSI Design and Education Center(VDEC), the University of Tokyo with the collaboration with Synopsys Corporation and Mentor Graphics Corporation.

REFERENCES

[1] D. B. A. Merchant and D. Sager, "Processor with a replay system that includes a replay queue for improved throughput," Patent, US Patent 7,200,737, 2007.

[2] D. B. A. Merchant, D. Sager and M. Upton, "Computer processor with a replay system having a plurality of checkers," Patent, US Patent 6,094,717, 2000.

[3] K. Aasaraai and A. Moshovos, "Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming," in *International Conference on Field Programmable Logic and Applications*, 2009.

[4] ——, "Design space exploration of instruction schedulers for out-of-order soft processors," in *International Conference on Field-Programmable Technology*, 2010.

[5] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported sram-based memories," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2014.

[6] C. Celio, D. A. Patterson, and K. Asanovi, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," EECS Department, University of California, Berkeley, Tech. Rep., 2015.

[7] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template," in *38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[8] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Annual International Symposium on Computer Architecture*, 1998.

[9] C. Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[10] J. Glueck and A. Wisner, "The lizard core," https://github.com/cornell-brg/lizard, 2019.

[11] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[12] A. Johri, "Implementation of instruction scheduler on fpga," 2011.

[13] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *10th International Symposium on High Performance Computer Architecture*, 2004.

[14] C. E. Laforest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported memories for fpgas via xor," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012.

[15] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010.

[16] B. C. Lai and J. Lin, "Efficient designs of multiported memory on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[17] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[18] E. Morancho, J. M. Llaberia, and A. Olive, "Recovery mechanism for latency misprediction," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[19] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *24th Annual International Symposium on Computer Architecture*, 1997.

[20] onikiri, "Onikiri2: a cycle-accurate processor simulator," https://github.com/onikiri/onikiri2, 2019.

[21] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten, "Cost-effective speculative scheduling in high performance processors," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[22] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[23] M. Rosiere, J. Desbarbieux, N. Drach, and F. Wajsbrt, "An out-of-order superscalar processor on fpga: The reorder buffer design," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.

[24] M. Rosiere, J.-L. Desbarbieux, N. Drach, and F. Wajsburt, "Morpheo: A high-performance processor generator for a fpga implementation," in *Proceedings of the 2011 Conference on Design & Architectures for Signal & Image Processing (DASIP)*, 2011.

[25] P. G. Sassone, J. Rupley, II, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[26] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel nehalem processor core made fpga synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010.

[27] Standard Performance Evaluation Corporation, "SPEC CPU 2006," https://www.spec.org/cpu2006/, 2019.

[28] ——, "SPEC CPU 2017," https://www.spec.org/cpu2017/, 2019.

[29] W. W. Terpstra, "Opa: Out-of-order superscalar soft cpu," in *An open source digital design conference (ORCONF)*, 2015.

[30] H. Wong, V. Betz, and J. Rose, "High performance instruction scheduling circuits for out-of-order soft processors," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[31] H. Wong, V. Betz, and J. Rose, "Microarchitecture and circuits for a 200 mhz out-of-order soft processor memory system," *ACM Trans. Reconfigurable Technol. Syst.*, 2016.

[32] H. T.-H. Wong, "A superscalar out-of-order x86 soft processor for fpga," Ph.D. dissertation, University of Toronto (Canada), 2017.

[33] Xilinx, "7 series fpgas configurable logic block," https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf, 2019.

[34] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind, "Composable building blocks to open up processor design," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.