2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Composable Building Blocks to Open up Processor Design

Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, Arvind
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{szzhang, acwright, bthom, arvind}@csail.mit.edu

***Abstract*—We present a framework called *Composable Modular Design* (CMD) to facilitate the design of out-of-order (OOO) processors. In CMD, (1) The interface methods of modules provide instantaneous access and perform atomic updates to the state elements inside the module; (2) Every interface method is *guarded*, i.e., it cannot be applied unless it is ready; and (3) Modules are composed together by *atomic rules* which call interface methods of different modules. A rule either successfully updates the state of all the called modules or it does nothing. CMD designs are compiled into RTL which can be run on FPGAs or synthesized using standard ASIC design flows.**

The atomicity properties of interfaces in CMD ensures composability when selected modules are refined selectively. We show the efficacy of CMD by building a parameterized out-of-order RISC-V processor which boots Linux and runs on FPGAs at 25 MHz to 40 MHz. We also synthesized several variants of it in a 32 nm technology to run at 1 GHz to 1.1 GHz. Performance evaluation shows that our processor beats in-order processors in terms of IPC but will require more architectural work to compete with wider superscalar commercial ARM processors. Modules designed under the CMD framework (e.g., ROB, reservation stations, load store unit) can be used and refined by other implementations. We believe that this realistic framework can revolutionize architectural research and practice as the library of reusable components grows.

I. Introduction

The software community has benefited for years from open source projects and permissive licensing. This has allowed reusable libraries with proper APIs to flourish. The RISC-V ISA is an attempt to open up the processor design community in a similar way [1]. By providing a free and open ISA, unencumbered by restrictive licenses, suddenly it has become possible for the research community to develop sophisticated (and not so sophisticated) architectures which may benefit small companies as well as the architecture community at large.

To realize the full potential of RISC-V, it is not enough to have several good and free implementations of RISC-V. Such implementations can serve as the role of exemplars, but to build a real community of architects we need to provide a hardware development framework where many people can contribute. The framework must result in a variety of parts, whose numbers and replacements grow with time, and which can be modified and combined together easily to build commercially competitive systems-on-a-chip (SoCs). This paper is about providing such a framework and showing that the framework is rich enough to express modern out-of-order multicores and their memory systems efficiently.

A natural design flow for any complex piece of digital hardware is to first design each component module and then compose them together. Given the complexity of OOO processors, after the first design pass, performance tuning requires refining the implementations of various modules and even the way different modules interact with each other. There are two key challenges in this design flow: (1) How to compose all the modules together; and (2) How to ensure modules can be combined correctly during the performance tuning phase, when modules are selectively refined. An essential property for composability and modular refinement is *latency-insensitivity*, i.e., the time it takes to produce an answer for a request does not affect the correctness of the overall design. It is common knowledge that if modules can be designed to have only latency-insensitive FIFO-like interfaces, then they can be composed easily and refined individually. However, in processor design, a microarchitectural event may need to read and write states across many modules simultaneously, requiring interfaces to have the *atomicity property* which goes beyond latency insensitivity. The atomicity property of interfaces governs the concurrent execution of multiple methods and is specified via the *conflict matrix* [2] for the methods of a module.

We present a framework called *Composable Modular Design* (CMD) to address these issues. In CMD, a design is a collection of *modules*, which interact with each other according to a set of *atomic rules* such that,

- The interface methods of modules provide combinational access and perform atomic updates to the state elements inside the module;
- Every interface method is *guarded*, i.e., it cannot be applied unless it is ready; and
- Modules are composed together by *rules* or "transactions" which call interface methods of different modules. A rule either successfully updates the states of all the called modules or it does nothing.

CMD designs are compiled into RTL which can be simulated in software, run on FPGAs or synthesized using standard ASIC design flows. The resulting hardware behaves as if multiple rules are executed every clock cycle, though the resulting behavior can always be expressed as executing rules one-by-one. CMD designs are often highly parameterized – hardware synthesis and RTL generation is feasible only after each parameter has been specified. The CMD framework can

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00015

be expressed in just about any HDL by appropriately annotating wires and state elements, but we have used Bluespec System Verilog (BSV) in our work. One benefit of using BSV is that it provides type checking and enforces a guarded interface protocol and rule atomicity. Any violation of these is detected by the compiler, along with combinational cycles, and this enforcement helps considerably in preserving the correctness of the design as various refinements are undertaken.

This paper makes the following contributions:

- Composable Modular Design (CMD) flow, a new framework for implementing complex and realistic microarchitectures;
- A complete set of parameterized modules for implementing out-of-order RISC-V processors. Modules include ROB, instruction issue queue, load-store unit and a cache-coherent memory system;
- A parameterized OOO processor built using the CMD framework, which boots Linux, runs on FPGAs at up to 40 MHz and can be synthesized in a 32nm technology to run up to 1.1 GHz clock speed;
- Extensive performance evaluation using SPEC CINT2006 and PARSEC benchmark suites – some executing up to three trillion instructions – without seeing any hardware bugs. The results show that our processor easily beats in-order processors in terms of IPC but requires more work to compete with wider superscalar commercial ARM processors.

The ultimate success of the CMD flow would be determined by our ability and the ability of others in the community to refine our OOO design in future. For this reason, we have released all the designs at `https://github.com/csail-csg/riscy-OOO` under the MIT License.

Paper organization: In Section II we present some other frameworks for designing processors, especially for RISC-V. Section III introduces modules and rules for the CMD flow via simple examples. In Section IV we illustrate the need for atomic actions across modules using an important example from OOO microarchitecture. Section V introduces the top-level modular decomposition of a multicore. We will define the interfaces of the selected modules to show how these modules can be connected to each other using atomic rules. Section VI discusses the performance of SPEC benchmarks on our processor as compared to other RISC-V processors and ARM. We also discusses the synthesis results. Finally we present our conclusions in Section VII.

II. Related Work

There is a long, if not very rich, history of processors that were designed in an academic setting. Examples of early efforts include the RISC processors like MIPS [3], RISC I and RISC II [4], and SPUR [5], and dataflow machines like Monsoon [6], Sigma1 [7], EM4 [8], and TRIPS [9]. All these attempts were focused on demonstrating new architectures; there was no expectation that other people would improve or refine an existing implementation. Publication of the RISC-V ISA in 2010 has already unleashed a number of open-source processor designs [10]–[17] and probably many more are in the works which are not necessarily open source. There are also examples of SoCs that use RISC-V processors. e.g.: [18]–[22].

Most open source RISC-V designs are meant to be used by others in their own SoC designs and, to that extent, they provide a framework for generating the RTL for a variety of specific configurations of the design. We discuss several examples of such frameworks:

- *Rocket chip generator* [15], [23], [24]: Developed at UC Berkeley, it generates SoCs with RISC-V cores and accelerators. The RISC-V cores are parameterized by caches, branch predictors, degree of superscalarity, and ISA extensions such as hardware multipliers (M), atomic memory instructions (A), FPUs (F/D), and compressed instructions (C). At the SoC level, one can specify the number of cores, accelerators chosen from a fixed library, and the interconnect. Given all the parameters, the Rocket chip generator produces synthesizable Verilog RTL.
  Rocket chip generator has been used for many SoC designs [22], [25]–[29]. Some modules from the Rocket chip generator have also been used to implement Berkeley's Out-of-Order BOOM processor [16]. The Rocket chip generator is itself written in Chisel [30].
- *FabScalar* [31]: Developed at North Carolina State University, it allows one to assemble a variety of superscalar designs from a set of predefined pipeline-stage blocks, called CPSL. A template is used to instantiate the desired number of CPSL for every stage and then glue them together. For example, one can generate the RTL for a superscalar core with 4 fetch stages, 6 issue/register read/write back, and a chosen set of predictors. FabScalar have been successful in generating heterogeneous cores using the same ISA for a multicore chip [32], [33]. They report results which are comparable to commercial hand-crafted RTL. These CPSLs are not intended to be modified themselves.
- *PULP* [14]: Developed at ETH Zurich, the goal of PULP project is to make it easy to design ultra-low power IoT SoC. They focus on processing data coming from a variety of sensors, each one of which may require a different interface. The processor cores themselves are not intended to be refined within this framework. A number of SoCs have been fabricated using PULP [21].

All these frameworks are structural, that is, they guarantee correctness only if each component meets its timing assumptions and functionality. For some blocks, the timing assumptions are rigid, that is, the block takes a fixed known number of cycles to produce its output. For some others, like cache accesses, the timing assumption is latency insensitive. In putting together the whole system, or in replacing a block, if the user observes all these timing constraints, the result should be correct. However no mechanical verification is performed to ensure that the timing assumptions were not violated, and often these timing violations are not obvious due to interactions across blocks with different timing assumptions.

The goal of our CMD framework is more ambitious, in the sense that, in addition to parameterized designs, we want the

users to be able to incorporate new microarchitectural ideas. For example, replace a central instruction issue queue in an OOO design, with several instruction issue queues, one for each functional unit. Traditionally, making such changes requires a deep knowledge of the internal functioning of the other blocks, otherwise, the processor is unlikely to function. We want to encapsulate enough properties in the interface of each block so that it can be composed without understanding the internal details.

A recent paper [34] argues for agile development of processors along the lines of agile development of software. The methodological concerns expressed in that paper are orthogonal to the concerns expressed in this paper and the two methodologies can be used together. However, we do advocate going beyond simple structural modules advocated in that paper to achieve true modularity which is amenable to modular refinement.

III. Interface Methods and Guarded Atomic Actions in BSV

Modules in BSV are like objects in object-oriented programming languages such as C++ and Java. In these languages, an object can be manipulated only by its interface methods and the internals of a module are beyond reach except via its methods. This allows us to use alternative implementations as long as the interface does not change. We associate some extra properties with interfaces in our hardware modules. Consider the case of dequeuing from an empty FIFO. In software, this will raise an exception, but in our hardware model, we assume that there is an implicit guard (a ready signal) on every method which must be true before the method can be invoked (enabled). The methods of a module are invoked from a *rule*, aka, *guarded atomic action*, or from other methods. Another difference from a software language is that an action may call multiple methods of the same or different modules concurrently. When an action calls multiple methods then its effect is *atomic*; it is as if either all of the methods were applied successfully or none were applied. Thus, if an action has an effect, by definition the guards of all of its method calls must have been true. Generally, a rule can be applied at any time as long as its guard is true. There are some exceptions to this general firing condition which will be explained as we encounter them. We illustrate these properties using a module which generates the hardware to compute the greatest common divisor (GCD) of two numbers.

A. GCD: An Example to Illustrate Latency-insensitivity

The interface of GCD is shown in Figure 1. It has two interface methods: *start*, an *Action method* that affects the state of the module but does not return any value; and *getResult*, an *ActionValue method* which returns a value in addition to affecting the state of the module. There is another type of method called a *value method*, which does not affect the state but returns a value of a given type. This GCD interface does not have a value method, but we could have provided a different interface by replacing *getResult* by a pair of methods, i.e., a value method called *result* and an action method called *deqResult* to delete the result internally. The # sign marks a type parameter, e.g., one could have defined the GCD interface for n-bit values.

Figure 2 shows an implementation of the GCD interface called *mkGCD*. An implementation first instantiates state elements with appropriate reset values. In this example these are registers x, y, and *busy*. An implementation of GCD must provide implementations of methods in its interface: *start* and *getResult*. The guard of the *start* method is the *busy* flag which must be false before the method can be invoked, while the guard condition of the *getResult* method is that the value of register x is zero and module was busy. Method *getResult* resets the busy flag, ensuring that *start* can only be called when the result has been picked up. Such interfaces are called *latency-insensitive*, because the use of GCD does not require one to know how many iterations it takes to compute the result.

Finally the implementation must provide the rule or rules that perform the actual GCD computation. In this implementation rule *doGCD* accomplishes that. Though it is not required by the semantics, the implementation of BSV provided by the Bluespec Inc. compiler always implements each rule to execute in a single cycle, that is, the register values are read at the beginning of a clock cycle, and updated at the end. Thus, $x <= y;\ y <= x;$ swaps the values of x and y. The circuit generated by this implementation of GCD is shown in Figure 3.

```
interface GCD;
  method Action start(Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
```

Fig. 1. GCD Interface

```
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);
  rule doGCD (x != 0);
    if(x >= y) begin
      x <= x - y;
    end else begin // swap
      x <= y; y <= x;
    end
  endrule
  method Action start(Bit#(32) a, Bit#(32) b) if(!busy);
    x <= a;
    y <= (b == 0) ? a : b;
    busy <= True;
  endmethod
  method ActionValue#(Bit#(32)) getResult if(busy && x==0);
    busy <= False;
    return y;
  endmethod
endmodule
```

Fig. 2. GCD Implementation

B. High-throughput GCD

Suppose the GCD implementation in Figure 2 is "streamed", i. e., a rule initiates it repeatedly by calling *start* and another rule picks up the result by calling *getResult*. This implementation does not permit a call to *start* unless the *getResult* for the previous *start* has been executed, and therefore, at most one

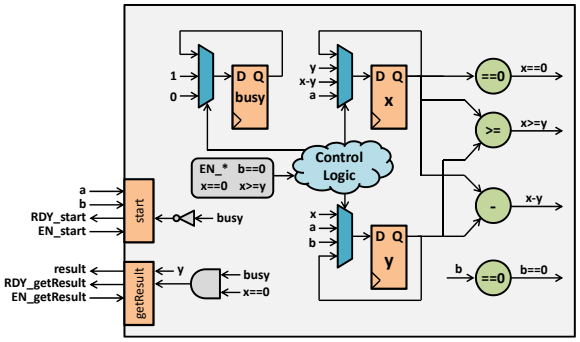

Fig. 3. Generated Hardware for GCD

GCD would compute at a time. We can provide a different implementation for the same GCD interface to give up to twice the throughput by including two *mkGCD* modules that compute concurrently. Figure 4 shows *mkTwoGCD*, a GCD implementation to achieve this.

```
module mkTwoGCD(GCD);
  GCD gcd1 <- mkGCD;
  GCD gcd2 <- mkGCD;
  Reg#(Bool) inTurn <- mkReg(True);
  Reg#(Bool) outTurn <- mkReg(True);
  method Action start(Bit#(32) a, Bit#(32) b);
    if(inTurn) begin
      gcd1.start(a,b); inTurn <= !inTurn;
    end else begin
      gcd2.start(a,b); inTurn <= !inTurn;
    end
  endmethod
  method ActionValue#(Bit#(32)) getResult;
    let y = ?;
    if(outTurn) begin
      y <- gcd1.getResult; outTurn <= !outTurn;
    end else begin
      y <- gcd2.getResult; outTurn <= !outTurn;
    end
    return y;
  endmethod
endmodule
```

Fig. 4. High-throughput GCD

This implementation uses a very simple policy where the two internal *mkGCD* modules are called in a round-robin manner. The flags at the inputs and outputs of the mkGCD modules track which unit should be used for *start* and which unit should be used for *getResult*. It is possible to generalize this implementation to improve the throughput further by using N *mkGCD* modules. Even higher throughput can be achieved if we don't insist on the first-in first-out property of the GCD module, however in that case the interface would have to change to include tags for identification in the input and output. Notice, the interface and the latency-insensitive aspects of GCD are not affected by whether a module is pipelined or not.

The latency-insensitive design paradigm illustrated by this example is useful in processors where we may want to increase or decrease the number of functional units, or inside the memory system where we may want to build non-blocking caches with different degrees of associativity.

IV. Need for Atomic Actions Across Modules

The latency-insensitive design paradigm, though very useful for composing modules, is not sufficient. We will first introduce the problem using the Instruction Issue Queue (IQ) from OOO microarchitectures and then show how our guarded interfaces extended with concurrency properties help solve the problem.

A. Instruction Issue Queue (IQ)

All OOO processors contain one or more *instruction issue queue* (IQ) like structures as shown in Figure 5. IQ contains instructions which have been renamed but have not yet been issued to execution pipelines. It holds all the source fields (SRC#) and the ROB tag for every instruction. Each source field contains the name of the physical register, and its ready bit (R#), which is also known as the *presence bit*. The physical register file (PRF) holds the values in the "renamed registers" and the bit vector RDYB holds the associated presence bit for each register. After renaming the source registers and allocating a destination register in the PRF, an instruction, along with its ROB tag, is *entered* into the ROB (not shown) and the IQ by copying the PRF presence bits from the RDYB. The presence bit in RDYB of the destination register of the instruction also gets reset. When the instruction has all its presence bits set, it can be *issued* to or equivalently pulled by an execution pipeline. After the instruction leaves the execution pipeline, it writes the result in the destination register in the PRF and sets its presence bit. Additionally, if this register is present in any IQ, then the presence bit associated with each of those locations is also updated; this process is often known as *wakeup*. It is a big challenge to design an IQ where, *enter*, *issue*, and *wakeup* are performed concurrently.

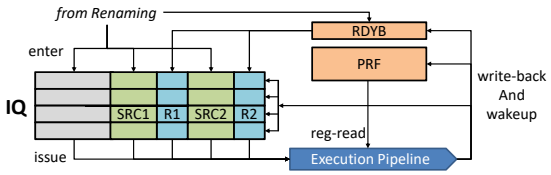

Fig. 5. Atomicity issues in reading and writing the register ready bits.

To understand the difficulty of concurrently executing these operations, consider the situation when an instruction is being entered to IQ, and one of its source registers is concurrently being woken up. Without a bypass path from the wakeup signal to the entering logic, the instruction in the IQ will have its source marked as not present, but the corresponding entry in the PRF will be marked present. Thus, this newly entered instruction will never be issued and will eventually lead to deadlock.

To avoid this deadlock but still permit the *enter* and *wakeup* operations to be performed concurrently, a bypass path is required to compare the entering source registers against the register being written. This bypass path can exist as special "glue logic" between the RDYB and the IQ, or it can reside within either the RDYB or the IQ. Concurrent actions like this often destroy the modularity of a hardware design; modifying

the IQ implementation would require you to know internal details of the RDYB implementation. For example, if the RDYB implementation is missing a bypass path, then the IQ needs to have a bypass path.

To make more sense of this concurrency issue, let's examine two different placement of the "wakeup bypass" path.

1) *Inside RDYB*: A bypass path can be implemented such that *enter* sees the present-bit values as if they have been updated by *wakeup*.
2) *Inside IQ*: A bypass path can be implemented such that *wakeup* searches and updates all entries including the newly inserted instruction by *enter*.

Both of these solutions imply an ordering between the two concurrent actions; in solution 1 *wakeup* happens before *enter*, but in solution 2 *enter* happens before *wakeup*. The crux of the correctness problem is that each action needs to be performed atomically, e.g., *wakeup* must update RDYB and IQ simultaneously before any other action can observe its effects. Similarly, *enter* must read RDYB and update IQ atomically. When viewed in this manner, then the final state must correspond to as if *wakeup* was performed before *enter*, or *enter* was performed before *wakeup*. Thus, both bypassing solutions produce a correct design as long as the ordering is consistent across modules.

B. Conflict Matrix: Extending Module Interface with Concurrency Properties

A rule can call several methods of a module concurrently. In such cases, the module interface definition must specify whether such concurrent calls are permitted, and if they are permitted then what is the resulting functionality. That is, does the state of the module after the two of its methods f1 and f2 are executed concurrently looks as if f1 executed followed by f2, or does it look as if f2 executed followed by f1. This information can be summarized in a *Conflict Matrix* (CM), which for each pair of methods f1 and f2 specifies one of the four possibilities $\{C, <, >, CF\}$, where C means f1 and f2 conflict and cannot be called concurrently. In all other cases f1 and f2 can be called concurrently, and $<$ means that the net effect would be as if f1 executed before f2, while $>$ means that the net effect would be as if f2 executed before f1. CF means that the methods are "conflict free", i.e., the method ordering does not affect the final state value.

The notion of CM extends to rules in a natural way: two rules conflict if any of their method calls conflict; $rule1 < rule2$ if every method of $rule1$ is either $<$ or CF with respect to every method of $rule2$. The BSV compiler automatically generates the CM for each module implementation and each pair of rules. It also declares a rule or a method to be illegal if conflicting methods are called concurrently [35], [36]. For example, the CM for the GCD implementation in Figure 2 would show that *start* and *getResult* conflict with each other because both methods update the *busy* flag,

A constructive procedure to transform a module implementation to achieve the desired CM was given by Rosenband [2], [37]. A detailed explanation of the transformation procedure is beyond the scope of this paper. Now we will show how CM information can be used to solve the IQ problem without knowing the detailed implementation of each module.

C. A Composable Modular Design for IQ

Figure 6 shows the structure of a composable modular design that solves the IQ concurrency problem. The interface methods for modules IQ and RDYB are given in Figure 7. Though no implementation is given, let us assume we can instantiate modules *iq* and *rdyb* corresponding to each of these interfaces, respectively. The rules that operate on these modules are shown in Figure 8. We have taken liberties with type declarations in the code to avoid clutter.

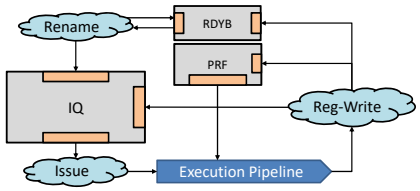

Fig. 6. CMD's view of the concurrency problem. Each module has interface methods that are called by external rules.

```
interface IQ;
  method Action enter(decodedRenamedInst, rdy1, rdy2);
  method Action wakeup(dstPhyReg);
  method ActionValue#(IssueInst) issue();
endinterface
interface RDYB;
  method Bool rdy1(r1); // check presence bit
  method Bool rdy2(r2);
  method Action setReady(dstR); // set presence bit
  method Action setNotReady(dstR); // reset presence bit
endinterface
```

Fig. 7. Interfaces for the design in Figure 6

```
rule doRename;
  // dInst has been decoded, renamed and entered into ROB
  Bool rdy1 = rdyb.rdy1(dInst.src1);
  Bool rdy2 = rdyb.rdy2(dInst.src2);
  rdyb.setNotReady(dInst.dst);
  iq.enter(dInst, rdy1, rdy2);
endrule
rule doIssue;
  let rdyInst <- iq.issue();
  exec.req(rdyInst); // go to exe pipeline
endrule
rule doRegWrite;
  let wbInst <- exec.resp(); // inst leaves exe pipeline
  iq.wakeup(wbInst.dst);
  rdyb.setReady(wbInst.dst);
  // write PRF, set ROB complete...
endrule
```

Fig. 8. Rules for the design in Figure 6

Now let us consider the rule *doRename*; it invokes three methods: *rdy1*, *rdy2*, and *setNotReady* of the RDYB module and the *enter* method of IQ concurrently. It is conceivable that IQ is full and thus the *enter* method is not ready. In which case to preserve rule atomicity, the whole *doRename* rule cannot be executed.

Consider the case where all three rules execute concurrently and affect the state in the order: $doIssue < doRegWrite <$

$doRename$. This will be feasible only if methods of various modules have certain properties.

- IQ methods must behave as if $issue < wakeup < enter$
- RDYB methods must behave as if $setReady < \{rdy1, rdy2, setNotReady\}$

It is always possible to design modules so that their methods will satisfy these properties [2]. The interesting question is what happens to the overall design if a module has slightly different properties. For example, suppose the RDYB module does not do internal bypassing, and therefore $\{rdy1, rdy2, setNotReady\} < setReady$. In this case, *doRename* and *doRegWrite* will no longer be able to execute concurrently preserving atomicity. But *doIssue* will still be able to fire concurrently with either one of them, but not both. So the design with such a RDYB module will have less overall concurrency implying less performance, but it will still be correct. This type of reasoning is the main advantage of thinking of a modular design in terms atomic actions and interface methods as opposed to just an interconnection of finite-state machines.

D. Modularity and Architectural Exploration

Now we illustrate another point where a different ordering of atomic actions can have different implications for performance and thus, can be a mechanism for microarchitectural exploration. Consider the case where all three rules execute concurrently and affect the state in the order: $doRegWrite < doIssue < doRename$. This will be feasible only if methods of various modules have the following properties.

- In IQ $wakeup < issue < enter$
- In RDYB $setReady < \{rdy1, rdy2, setNotReady\}$

This ordering implies that entries in the IQ are woken up before issuing, so an instruction can be set as ready and issued in the same cycle. This reduces a clock cycle of latency compared to the other ordering of these rules. The point is that by playing with these high-level ideas, the focus shifts from correctness to exploration and performance.

V. Composing an Out-of-Order Processor

Figure 9 shows the overall structure of the OOO core. The salient features of our OOO microarchitecture are the physical register file (PRF), reorder buffer (ROB), a set of instruction issue queues (IQ) – one for each execution pipeline (only two are shown to avoid clutter), and a load-store unit, which includes LSQ, non-blocking D cache, etc.

The front end has three different branch predictors (BTB, tournament direction predictor, and return address stack) and it enters instructions into ROB and IQs after renaming. We use *epochs* for identifying wrong path instructions. Instructions can be flushed because of branch mispredictions, load miss-speculations on memory dependencies, and page faults on address translation. Each instruction that may cause a flush is assigned a *speculation tag* [16], [31], [38], and the subsequent instructions that can be affected by it carry this tag. These speculation tags are managed as a finite set of bit masks

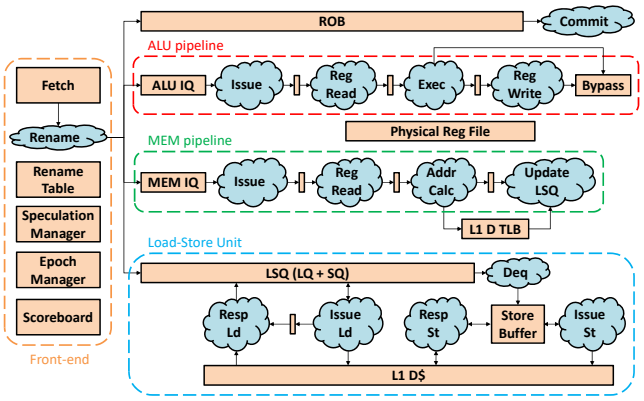

Fig. 9. Structure of the OOO core

which are set and cleared as instruction execution proceeds. When an instruction can no longer cause any flush, it releases its speculation tag, and the corresponding bit is reset in the bit masks of subsequent instructions so that the tag can be recycled. To reduce the number of mask bits, we only assign speculation tags to branch instructions, while deferring the handling of interrupts, exceptions and load speculation failures until the commit stage. Every module that keeps speculation-related instructions must keep speculation masks and provide a *correctSpec* method to clear bits from speculation masks, and a *wrongSpec* method to kill instructions. We do not repeatedly describe these two methods in the rest of this section.

We also maintain two sets of PRF presence bits to reduce latency between dependent instructions. The true presence bits are used in the Reg-Read stage to stall instructions. Another set of presence bits (Scoreboard in Figure 9) are set optimistically when it is known that the register would be set by an older instruction with small predictable latency. These optimistic bits are maintained as a scoreboard, and are used when instructions are entered in IQ and can improve throughput for instructions with back-to-back dependencies.

In Figure 9, boxes represent the major modules in the core, while clouds represent the top-level rules. A contribution of this paper is to show a set of easily-understandable interfaces for all the modules, and show some atomic rules that are used to compose the modules. The lack of space does not allow us describe all the details but in the following subsections we discuss all the salient modules and some important rules. We will also describe briefly how we connect multiple OOO cores to form a multiprocessor. The whole design has been released publicly as the *RiscyOO* processor at `https://github.com/csail-csg/riscy-OOO`. Due to lack of space, we do not discuss the details of front-end, and directly get into the execution engine and load-store unit.

A. The Execution Engine

The execution engine consists of multiple parallel execution pipelines, and instructions can be issued from the IQs in different pipelines simultaneously. The number of execution

pipelines is parameterized. Though instructions execute and write the register file out of order, the program order is always kept by ROB. The Physical Register File (PRF) is shared by all execution pipelines, and it stores a presence bit for each physical register. Unlike the Scoreboard, the presence bit can be set only when the data is written to the physical register. Each IQ is responsible for tracking read-after-write (RAW) dependencies, and issuing instructions with all source operands ready, as discussed in Section IV-C. The most important module is ROB which we discuss next.

ROB: ROB keeps in program order all in-flight instructions which have been renamed but not yet committed. Each entry has a PC, instruction type, a speculation mask, a completion bit, detected exceptions, index to LSQ and page-fault address for memory instructions, and a few more miscellaneous status bits. Instructions that manipulate system special registers (CSRs in RISC-V) overload the fault-address field as a data field. In a different design, it may be possible to reduce the width of ROB entries by keeping these data or address fields in a separate structure, without affecting the ROB interface. ROB can use a single register to hold CSR data, because we allow only one CSR instruction in flight, and another register to store the oldest faulting address. However, LSQ may need to keep virtual addresses in each of its slots, because in RISC-V, a memory access can cause exception even after address translation. In addition to *enq*, *deq*, *first*, ROB has the following methods:

- *getEnqIndex:* returns the index for the slot where the next entry will be allocated.
- *setNonMemCompleted:* marks the instruction at the specified ROB index to have completed (so that it can be committed later).
- *setAfterTranslation*: is called when a memory instruction has finished address translation. It tells the ROB whether the memory instruction can only access memory non-speculatively (so ROB will notify LSQ when the instruction reaches the commit slot), and also marks it complete in case of a normal store.
- *setAtLSQDeq:* is called when load or memory-mapped store is dequeued from LSQ. It marks exception or load speculation failure or complete.

Bypass: Instead of bypassing values in an ad-hoc manner, we have created a structure to bypass ALU execution results from the Exec and Reg-Write rules in the ALU pipeline to the Reg-Read rule of every pipeline. It provides a *set* method for each of the Exec and Reg-Write rules to pass in the ALU results, and a *get* method for each of the Reg-Read rules to check for the results passed to the *set* methods in the same cycle. These methods are implemented such that $set < get$.

B. Load-Store Unit

The load-store unit consists of an LSQ, a store buffer (SB) and a non-blocking L1 D cache. LSQ contains a load queue (LQ) and a store queue (SQ) to keep in-flight loads and stores in program order, respectively. The SB holds committed stores that have not been written into L1 D.

When a memory instruction leaves the front-end, it enters the IQ of the memory pipeline and allocates an entry in LQ or SQ. The memory pipeline computes the virtual address in the Addr-Calc stage and then sends it to L1 D TLB for address translation (see Figure 9). When the translation result is available, the Update-LSQ stage checks if there is a page fault, and if the memory instruction is accessing the normal cached memory region or the memory mapped IO (MMIO) region. It also updates the ROB, and LQ or SQ entry for this instruction accordingly. In case of a normal load, it is executed speculatively either by sending it to L1 or by getting its value from SB or SQ. However, it may have to be stalled because of fences, partially overlapped older stores, and other reasons. Thus, LQ needs internal logic that searches for ready-to-issue loads every cycle. Speculative loads that violate memory dependency are detected and marked as to-be-killed when the Update-LSQ stage updates the LSQ with a store address.

Unlike normal loads, MMIO accesses and atomic accesses (i.e., load reserve, store conditional and read-modify-write) can only access memory when the instruction has reached the commit stage.

Normal stores can be dequeued from SQ sequentially after they have been committed from ROB. In case of TSO, only the oldest store in SQ can be issued to L1 D provided that the store has been committed from ROB. However it can be dequeued from SQ only when the store hits in the cache. Though there can be only one store request in L1 D, SQ can issue as many store-prefetch requests as it wants. Currently we have not implemented this feature. In case of a weak memory model like WMM [39] or GAM [40], the dequeued store will be inserted into a store buffer (SB) without being issued to L1 D. SB can coalesce stores for the same cache line and issue stores to L1 D out of order.

Normal loads can be dequeued from LQ sequentially after they get the load values and all older stores have known addresses, or they become faulted or to-be-killed. A dequeued load marks the corresponding ROB entry as complete, exception, or to-be-killed.

LSQ: As mentioned earlier, loads and stores are kept in separate queues. In order to observe the memory dependency between loads and stores, each load in LQ keeps track of the index of the immediately preceding SQ entry. In case a load has been issued from LQ, the load needs to track whether its value will come from the cache or by forwarding from an SQ entry or a SB entry. When a load tries to issue, it may not be able to proceed because of fences or partially overlapped older stores. In such cases, the load records the source that stalls it, and retries after the source of the stall has been resolved. In case of ROB flush, if a load, which is waiting for the memory response, is killed, then this load entry is marked as waiting for a wrong path response. Because of this bit, we can reallocate this entry to a new load, but not issue it until the bit is cleared. The LSQ module has the following methods:

- *enq:* allocates a new entry in LQ or SQ for the load or store instruction, respectively, at the Rename stage.

- *update:* is called after a memory instruction has translated its address and, in case of a store, the store has computed its data. This fills the physical address (and store data) into the corresponding entry of the memory instruction. In case the memory instruction is a store, this method also searches for younger loads that violate memory dependency ordering and marks them as to-be-killed. Depending upon the memory model, more killings may have to be performed. We have implemented the killing mechanisms for TSO and WMM; it is quite straightforward to implement other weak memory models.
- *getIssueLd:* returns a load in LQ that is ready to issue, i.e., the load does not have any source of stall and is not waiting for wrong path response.
- *issueLd:* tries to issue the load at the given LQ index. This method will search older stores in SQ to check for forwarding or stall. The method also takes as input the search result on store buffer, which is combined with the search result on store queue to determine if the load is stalled or can be forwarded, or should be issued to cache. In case of stall, the source of stall will be recorded in the load queue entry.
- *respLd:* is called when the memory response or forwarding data is ready for a load. This returns if the response is at wrong path, and in case of a wrong path response, the waiting bit will be cleared.
- *wakeupBySBDeq:* is called in the WMM implementation when a store buffer entry is dequeued. This removes the corresponding sources of stall from load queue entries.
- *cacheEvict:* is called in the TSO implementation when a cache line is evicted from L1 D. This searches for loads that read stale values which violate TSO, and marks them as to-be-killed.
- *setAtCommit:* is called when the instruction has reached the commit slot of ROB (i.e., cannot be squashed). This enables MMIO or atomic accesses to start accessing memory, or enables stores to be dequeued.
- *firstLd/firstSt:* returns the oldest load/store in LQ/SQ.
- *deqLd/deqSt:* removes the oldest load/store from LQ/SQ.

Store Buffer: The store buffer has the following methods:

- *enq:* inserts a new store into the store buffer. If the new store address matches an existing entry, then the store is coalesced with the entry; otherwise a new buffer entry is allocated.
- *issue:* returns the address of an unissued buffer entry, and marks the entry as issued.
- *deq:* removes the entry specified by the given index, and returns the contents of the entry.
- *search:* returns the content of the store buffer entry that matches the given address.

L1 D Cache: The L1 D Cache module has the following methods:

- *req:* request the cache with a load address and the corresponding load queue index, or a store address and the corresponding store buffer index.
- *respLd:* returns a load response with the load queue index.
- *respSt:* returns a store buffer index. This means that the cache has exclusive permission for the address of the indexed store buffer entry. The cache will remained locked until the *writeData* method is called to write the store data of the indexed store buffer entry into cache.
- *writeData:* writes data to cache; the data should correspond to the previously responded store buffer index.

L1 D also has the interface to connect to the L2 cache.

L1 D TLB: The L1 D TLB has FIFO-like request and response interface methods for address translation. It also has methods to be connected to the L2 TLB.

C. Connecting Modules Together

Our CMD framework uses rules to connect modules together for the OOO core. The rules call methods of the modules, and this implicitly describes the datapaths between modules. More importantly, the rules are guaranteed to fire atomically, leaving no room for concurrency bugs. The challenging part is to have rules fire in the same cycle. To achieve this, the conflict matrix of the methods of each module has to be designed so that the rules are not conflicting with each other. Once the conflict matrix a module is determined, there is a mechanical way to translate an initial implementation whose methods conflict with each other to an implementation with the desired conflict matrix. It should be noted that though the rules fire in the same cycle, they still behave as if they are firing one after another.

There are about a dozen rules at the top level. Instead of introducing all the rules, we explain two rules in detail to further illustrate the atomicity issue. Figure 10 shows the *doIssueLd* and *doRespSt* rules. The *doIssueLd* rules first gets a ready-to-issue load from the LSQ module. Then it searches the store buffer for possible forwarding or stall (due to partially overlapped entry). Next it calls the *issueLd* method of LSQ to combine the search on the store buffer and the search on the store queue in LSQ to derive whether the load can be issued or forwarded. The *respSt* rule first gets the store response from the L1 D cache. Then it dequeues the store from the store buffer and writes the store to L1 D. Finally, it wakes up loads in LSQ that has been stalled by this store earlier.

```
1  rule doIssueLd;
2    let load <- lsq.getIssueLd;
3    let sbSearchResult = storeBuffer.search(load.addr);
4    let issueResult <- lsq.issueLd(load, sbSearchResult);
5    if(issueResult matches tagged Forward .data) begin
6      // get forwarding, save forwarding result in a FIFO
        which will be processed later by the doRespLd rule
7      forwardQ.enq(tuple2(load.index, data));
8    end else if(issueResult == ToCache) begin
9      // issue to cache
10     dcache.req(Ld, load.index, load.addr);
11   end // otherwise load is stalled
12 endrule
13 rule doRespSt;
14   let sbIndex <- dcache.respSt;
15   let data, byteEn <- storeBuffer.deq(sbIndex);
16   dcache.writeData(data, byteEn);
17   lsq.wakeupBySBDeq(sbIndex);
18 endrule
```

Fig. 10. Rules for LSQ and Store Buffer

Without our CMD framework, when these two rules fire in the same cycle, concurrency bug may arise because both rules race with each other on accessing states in LSQ and the store buffer. Consider the case that the load in the *doIssueLd* rule has no older store in LSQ, but the store buffer contains a partially overlapped entry which is being dequeued in the *doRespSt* rule. In this case, the two rules race on the valid bit of the store buffer entry, and the source of stall for the load. Without CMD, if we pay no attention to the races here and just let all methods read the register states at the beginning of the cycle, then the *issueLd* method in the *doIssueLd* rule will records the store buffer entry as stalling the load, while the *wakeupBySBDeq* method in the *doRespLd* rule will fail to clear the stall source for the load. In this case, the load may be stalled forever without being waken up for retry. With our CMD framework, the methods implemented in the above way will lead the two rules to conflict with each other, i.e., they cannot fire in the same cycle. To make them fire in the same cycle, we can choose the conflict matrix of LSQ to be $issueLd < wakeupBySBDeq$, and the conflict matrix of the store buffer to be $search < deq$. In this way, the two rules can fire in the same cycle, but rule $doIssueLd$ will appear to take effect before rule $doRespSt$.

D. Multicore

We have connected the OOO cores together to form a multiprocessor as shown in Figure 11. The L1 caches communicate with the shared L2 via a cross bar, and the L2 TLBs sends uncached load requests to L2 via another cross bar to perform hardware page walk. All memory accesses, including memory requests to L1 D made by memory instructions, instruction fetches, and loads to L2 for page walk, are coherent. We implemented an MSI coherence protocol which has been formally verified by Vijayaraghavan et al. [41]. It should not be difficult to extend the MSI protocol to a MESI protocol. Our current prototype on FPGA can have at most four cores. If the number of cores becomes larger in future, we can partition L2 into multiple banks, and replace the cross bars with on-chip networks.

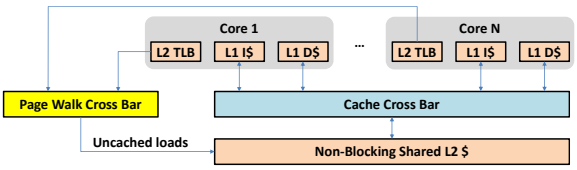

Fig. 11. Multiprocessor

VI. Evaluation

To demonstrate the effectiveness of our CMD framework, we have designed a parameterized superscalar out-of-order processor, *RiscyOO*, using CMD. To validate correctness and evaluate performance at microarchitectural level, we synthesized the processor on AWS F1 FPGA [42] with different configurations (e.g., different core counts and buffer sizes). The processor boots Linux on FPGA, and benchmarking is done under this Linux environment (i.e., there is no syscall emulation). For single-core performance, we ran SPEC CINT2006 benchmarks with ref input to completion. The instruction count of each benchmark ranges from 64 billion to 2.7 trillion. With the processor running at 40 MHz on FPGA, we are able to complete the longest benchmark in about two days. For multicore performance, we run PARSEC benchmarks [43] with the **simlarge** input to completion on a quad-core configuration at 25 MHz.

We will first give single-core performance, then multicore performance followed by ASIC synthesis results.

A. Single-Core Performance

Methodology: Figure 12 shows the basic configuration, referred to as *RiscyOO-B*, of our RiscyOO processor. Since the number of cycles needed for a memory access on FPGA is much lower than that in a real processor, we model the memory latency and bandwidth for a 2 GHz clock in our FPGA implementation.

We compare our design with the four processors shown in Figure 13: Rocket[1] (RISC-V ISA), A57 (ARM ISA), Denver (ARM ISA), and BOOM (RISC-V ISA). In Figure 13, we have also grouped these processors into three categories: Rocket is an in-order processor, A57 and Denver are both commercial ARM processors, and BOOM is the state-of-the-art academic OOO processor.

The memory latency of Rocket is configurable, and is 10 cycles by default. We use two configurations of Rocket in our evaluation, i.e., *Rocket-10* with the default 10-cycle memory latency, and *Rocket-120* with 120-cycle memory latency which matches our design. Since Rocket has small L1 caches, we instantiated a *RiscyOO-C*$^-$ configuration of our processor, which shrinks the caches in the RiscyOO-B configuration to 16KB L1 I/D and 256KB L2.

To illustrate the flexibility of CMD, we created another configuration *RiscyOO-T*$^+$, which improves the TLB microarchitecture of RiscyOO-B. In RiscyOO-B, both L1 and L2 TLBs block on misses, and a L1 D TLB miss blocks the memory execution pipeline. RiscyOO-T$^+$ supports parallel miss handling and hit-under-miss in TLBs (maximum 4 misses in L1 D TLB and 2 misses in L2 TLB). RiscyOO-T$^+$ also includes a split translation cache that caches intermediate page walk results [45]. The cache contains 24 fully associative entries for each level of page walk. We implemented all these microarchitectural optimizations using CMD in merely *two* weeks.

We also instantiated a *RiscyOO-T*$^+$*R*$^+$ configuration which extends the ROB size of RiscyOO-T$^+$ to 80, in order to match BOOM's ROB size and compare with BOOM.

Figure 14 has summarized all the variants of the RiscyOO-B configuration, i.e., RiscyOO-C$^-$, RiscyOO-T$^+$ and RiscyOO-T$^+$R$^+$.

The evaluation uses all SPEC CINT2006 benchmarks except perlbench which we were not able to cross-compile to RISC-V.

[1]The prototype on AWS is said to have an L2 [44], but we have confirmed with the authors that there is actually no L2 in this publicly released version.

We ran all benchmarks with the **ref** input to completion on all processors except BOOM, whose performance results are taken directly from [46]. We did not run BOOM ourselves because there is no publicly released FPGA image of BOOM. Since the processors have different ISAs and use different fabrication technology, we measure performance in terms of one over the number of cycles needed to complete each benchmark (i.e., 1 / cycle count). Given so many different factors across the processors, this performance evaluation is informative but not rigorous. The goal here is to show that the OOO processor designed with CMD can achieve reasonable performance.

Front-end	2-wide superscalar fetch/decode/renam 256-entry direct-mapped BTB tournament branch predictor as in Alpha 21264 [47] 8-entry return address stack
Execution Engine	64-entry ROB with 2-way insert/commit Total 4 pipelines: 2 ALU, 1 MEM, 1 FP/MUL/DIV 16-entry IQ per pipeline
Ld-St Unit	24-entry LQ, 14-entry SQ, 4-entry SB (each 64B wide)
TLBs	Both L1 I and D are 32-entry, fully associative L2 is 2048-entry, 4-way associative
L1 Caches	Both I and D are 32KB, 8-way associative, max 8 requests
L2 Cache	1MB, 16-way, max 16 requests, coherent with I and D
Memory	120-cycle latency, max 24 req (12.8GB/s for 2GHz clock)

Fig. 12. RiscyOO-B configuration of our RISC-V OOO uniprocessor

Name	Description	Catergory
Rocket	Prototype on AWS F1 FPGA for FireSim Demo v1.0 [48]. RISC-V ISA, in-order core, 16KB L1 I/D, no L2, 10-cycle or 120-cycle memory latency.	In-order
A57	Cortex-A57 core on Nvidia Jetson Tx2. ARM ISA, 3-wide superscalar. OOO core, 48KB L1 I, 32KB L1 D, 2MB L2.	Commercial ARM
Denver	Denver core [49] on Nvidia Jetson Tx2. ARM ISA, 7-wide superscalar. 128KB L1 I, 64KB L1 D, 2MB L2.	
BOOM	Performance results taken from [46]. RISC-V ISA, 2-wide superscalar. OOO core, 80-entry ROB, 32KB L1 I/D, 1MB L2, 23-cycle L2 latency, 80-cycle memory latency.	Academic OOO

Fig. 13. Processors to compare against

Variant	Difference	Specifications
RiscyOO-C$^-$	Smaller Caches	16 KB L1 I/D, 256 KB L2
RiscyOO-T$^+$	Improved TLB	Non-blocking TLBs, page table walk cache
RiscyOO-T$^+$R$^+$	Larger ROB	RiscyOO-T$^+$ with 80-entry ROB

Fig. 14. Variants of the RiscyOO-B configuration

Effects of TLB microarchitectural optimizations: Before comparing with other processors, we first evaluate the effects of the TLB microarchitectural optimizations employed in RiscyOO-T$^+$. Figure 15 shows the performance of RiscyOO-T$^+$, which has been normalized to that of RiscyOO-B, for each benchmark. Higher values imply better performance. The last column is the geometric mean across all benchmarks. The TLB optimizations in RiscyOO-T$^+$ turn out to be very effective: on average, RiscyOO-T$^+$ outperforms RiscyOO-B by 29% and it doubles the performance of benchmark astar.

To better understand the performance differences, we show the number of L1 D TLB misses, L2 TLB misses, branch mispredictions, L1 D cache misses and L2 cache misses per thousand instructions of RiscyOO-T$^+$ in Figure 16. Benchmarks mcf, astar and omnetpp all have very high TLB miss rates. Although RiscyOO-B has a very large L2 TLB, the blocking nature of L1 and L2 TLBs still makes TLB misses incur a huge penalty. The non-blocking TLB designs and translation caches in RiscyOO-T$^+$ mitigate the TLB miss penalty and result in a substantial performance gain.

This evaluation shows that microarchitectural optimizations can bring significant performance benefits. It is because of CMD that we can implement and evaluate these optimizations in a short time. Since RiscyOO-T$^+$ always outperforms RiscyOO-B, we will use RiscyOO-T$^+$ instead of RiscyOO-B to compare with other processors.

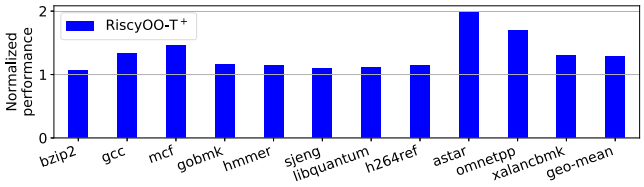

Fig. 15. Performance of RiscyOO-T$^+$ normalized to RiscyOO-B. Higher is better.

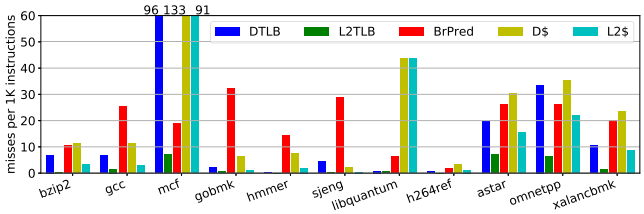

Fig. 16. Number of L1 D TLB misses, L2 TLB misses, branch mispredictions, L1 D misses and L2 misses per thousand instructions of RiscyOO-T$^+$

Comparison with the in-order Rocket processor: Figure 17 shows the performance of RiscyOO-C$^-$, Rocket-10, and Rocket-120 for each benchmark. The performance has been normalized to that of RiscyOO-T$^+$. We do not have libquantum data for Rocket-120 because each of our three attempts to run this benchmark ended with an AWS server crash after around two days of execution.

As we can see, Rocket-120 is much slower than RiscyOO-T$^+$ and RiscyOO-C$^-$ on every benchmark, probably because its in-order pipeline cannot hide memory latency. On average, RiscyOO-T$^+$ and RiscyOO-C$^-$ outperform Rocket-120 by 319% and 196%, respectively. Although Rocket-10 has only 10-cycle memory latency, RiscyOO-T$^+$ still outperforms Rocket-10 in every benchmark, and even RiscyOO-C$^-$ can outperform or tie with Rocket-10 in many benchmarks. On average, RiscyOO-T$^+$ and RiscyOO-C$^-$ outperforms Rocket-10 by 53% and 8%, respectively. This comparison shows that our OOO processor can easily outperform in-order processors.

Comparison with commercial ARM processors: Figure 18 shows the performance of ARM-based processors, A57 and Denver, for each benchmark. The performance has been

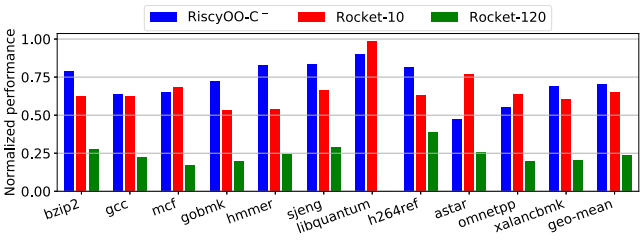

Fig. 17. Performance of RiscyOO-C⁻, Rocket-10, and Rocket-120 normalized to RiscyOO-T⁺. Higher is better.

normalized to that of RiscyOO-T⁺. A57 and Denver are generally faster than RiscyOO-T⁺, except for benchmarks mcf, astar and omnetpp. On average, A57 outperforms RiscyOO-T⁺ by 34%, and Denver outperforms RiscyOO-T⁺ by 45%.

To better understand the performance differences, we revisit the miss rates of RiscyOO-T⁺ in Figure 16. Because of the high TLB miss rates in benchmarks mcf, astar and omnetpp, the TLB optimizations enable RiscyOO-T⁺ to catch up with or outperform A57 and Denver in these benchmarks. Commercial processors have significantly better performance in benchmarks hmmer, h264ref, and libquantum. Benchmarks hmmer and h264ref both have very low miss rates in TLBs, caches and branch prediction, so the higher performance in A57 and Denver may be caused by their wider pipelines (our design is 2-wide superscalar while A57 is 3-wide and Denver is 7-wide). Benchmark libquantum has very high cache miss rates, and perhaps commercial processors have employed memory prefetchers to reduce cache misses.

Since we do not know the details of commercial processors, we cannot be certain about our reasons for the performance differences. In spite of this, the comparison still shows that the performance of our OOO design is not out of norm. However, we do believe that to go beyond 2-wide superscalar our processor will require more architectural changes especially in the front-end.

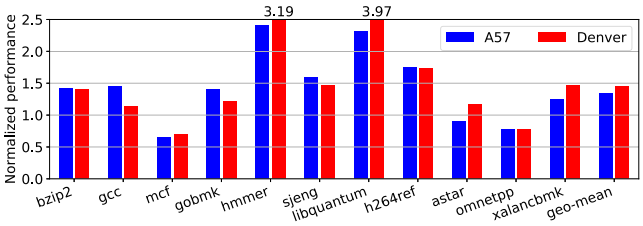

Fig. 18. Performance of A57 and Denver normalized to RiscyOO-T⁺. Higher is better.

Comparison with the academic OOO processor BOOM: Figure 19 shows the IPCs of BOOM and our design RiscyOO-T⁺R⁺[2]. We have tried our best to make the comparison fair between RiscyOO-T⁺R⁺ and BOOM. RiscyOO-T⁺R⁺ matches BOOM in the sizes of ROB and caches, and the influence of the longer L2 latency in BOOM can be partially offset by the longer memory latency in RiscyOO-T⁺R⁺. BOOM did not report IPCs on benchmarks gobmk, hmmer and libquantum [46], so we only show the IPCs of remaining benchmarks. The last column shows the harmonic mean of IPCs over all benchmarks.

On average, RiscyOO-T⁺R⁺ and BOOM have similar performance, but they outperform each other in different benchmarks. For example, in benchmark mcf, RiscyOO-T⁺R⁺ (IPC=0.16) outperforms BOOM (IPC=0.1), perhaps because of the TLB optimizations. In benchmark sjeng, BOOM (IPC=1.05) outperforms RiscyOO-T⁺R⁺ (IPC=0.73). This is partially because RiscyOO-T⁺R⁺ suffers from 29 branch mispredictions per thousand instructions while BOOM has about 20 [46].

This comparison shows that our OOO processor designed using CMD has matching performance with the state-of-the-art academic processors.

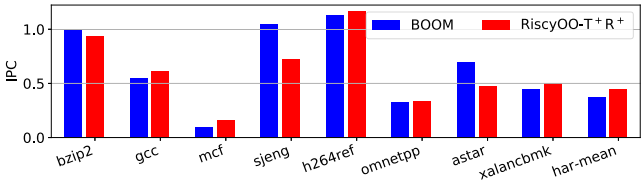

Fig. 19. IPCs of BOOM and RiscyOO-T⁺R⁺ (*BOOM results are taken from [46]*)

Summary: Our parameterized OOO processor designed using CMD is able to complete benchmarks of trillions of instructions without errors. It can easily outperform in-order processors (e.g., Rocket) and matches state-of-the-art academic OOO processors (e.g., BOOM), though not as good as highly optimized commercial processors.

B. Multicore Performance

Methodology: We implemented a quad-core multiprocessor on FPGA. Each core in the processor is derived from our OOO single-core designed using CMD to support proper memory models. Following the CMD methodology, we have derived two versions of the OOO core which supports TSO and WMM respectively. The major changes are in the LSQ module and surrounding rules.

To fit the design onto FPGA, each OOO core is instantiated with 48-entry ROB, and other buffer sizes are also reduced accordingly. Note that the OOO core is still 2-wide fetch/decode/rename/commit and it has four pipelines.

We run PARSEC benchmarks on the TSO and WMM quad-core multiprocessors. Among the 13 PARSEC benchmarks, we could not cross-compile raytrace, vips and dedup to RISC-V. Though we have managed to compile bodytrack, x264 and canneal, they cannot run even on the RISC-V ISA simulator [50], which is the golden model for RISC-V implementations. Therefore, we only use the remaining seven benchmarks for evaluation. We run each benchmark with the **simlarge** input to completion with one, two and four threads, respectively. We measure the execution time of the

[2]For each benchmark, we ran all the ref inputs (sometimes there are more than one), and computed the IPC using the aggregate instruction counts and cycles. This makes our instruction counts close to those reported by BOOM.

parallel phase (i.e., the region marked by parsec_roi_begin and parsec_roi_end).

Results: Figure 20 shows the performance of each benchmark running on TSO and WMM multicores with different number of threads. For each benchmark, the performance has been normalized to that of TSO with 1 thread. The last column is the geometric mean across all benchmarks. Higher values imply better performance. As we can see, there is no discernible difference between the performance of TSO and WMM. This is because speculative loads in TSO that get killed by cache eviction are extremely rare: maximum 0.25 kills per thousand instructions (in benchmark streamcluster). One interesting phenomenon is that benchmark freqmine has super-linear speedup. We are still investigating the reasons, but one possible explanation is that when running with one core, the data set processed by this core becomes larger, and miss rate in TLB and caches will become higher.

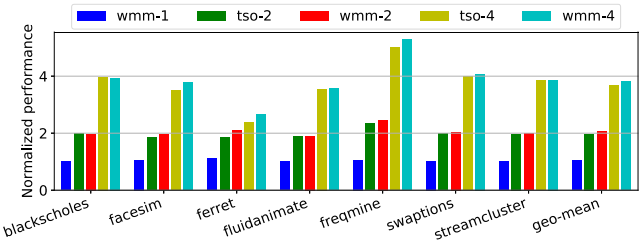

Fig. 20. Performance of TSO and WMM multicores with different number of threads (normalized to TSO with 1 thread). Legend tso-n or wmm-n means TSO or WMM with n threads. Higher is better.

C. ASIC Synthesis

To evaluate the quality of the produced designs, we synthesized a single core (processor pipeline and L1 caches) of the RiscyOO-T$^+$ and RiscyOO-T$^+$R$^+$ processor configurations for ASIC. Our synthesis flow used a 32 nm SOI technology and SRAM black-boxes using timing information from CACTI 6.5 [51]. We performed topographical synthesis using Synopsys's Design Compiler, i.e., we performed a timing-driven synthesis which performs placement heuristics and includes resistive and capacitive wire delays in the timing model. This approach significantly reduces the gap between post-synthesis results and post-placement and routing results. We produced a maximum frequency for each configuration by reporting the fastest clock frequency which was successfully synthesized. We produced a NAND2-equivalent gate count by taking the total cell area and dividing by the area of a default-width NAND2 standard cell in our library. As a result, our NAND2-equivalent gate count is logic-only and does not include SRAMs.

Core Configuration	RiscyOO-T$^+$	RiscyOO-T$^+$R$^+$
Max Frequency	1.1 GHz	1.0 GHz
NAND2-Equivalent Gates	1.78 M	1.89 M

Fig. 21. ASIC synthesis results

Results: The synthesis results are shown in Figure 21. Both processors can operate at 1.0 GHz or above. The area of the RiscyOO-T$^+$R$^+$ configuration is only 6.2% more than that of the RiscyOO-T$^+$ configuration, because RiscyOO-T$^+$R$^+$ increases only the ROB size and the number of speculation tags over RiscyOO-T$^+$. The NAND2-equivalent gate counts of the processors are significantly affected by the size of the branch predictors. This can be reduced by either reducing the size of the tournament branch predictor and/or utilizing SRAM for part of the predictor.

VII. Conclusion

To fully benefit from the openness of RISC-V, the architecture community needs a framework where many different people can cooperate to try out new ideas in real hardware. Although existing chip generators can connect parameterized building blocks together, they do not allow frequent changes to the building blocks. We have also shown that latency-insensitive interfaces alone are insufficient in processor designs. In this paper, we have proposed the CMD framework in which modules have guarded interface methods, and are composed together using atomic actions. With the atomicity guarantee in CMD, modules can be refined selectively relying only on the interface details, including Conflict Matrix, of other modules. We have shown the efficacy of CMD by designing an OOO processor which has fairly complex architectural features. Both the synthesis results and the performance results are very encouraging, and with sufficient effort by the community, it should be possible to deliver commercial grade OOO processors in not too distant a future.

Acknowledgment

We thank all the anonymous reviewers for their helpful feedbacks on improving this paper. We have also benefited from the help from Jamey Hicks and Muralidaran Vijayaraghavan. We would like to particularly thank Bluespec, Inc. for providing free tool licenses. This research is supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-16-2-0004 (program BRASS) and Contract No. HR001118C0018 (program SSITH). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[2] D. L. Rosenband, "The ephemeral history register: flexible scheduling for rule-based designs," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 2004.

[3] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "MIPS: A microprocessor architecture," in *ACM SIGMICRO Newsletter*, vol. 13, no. 4. IEEE Press, 1982.

[4] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News*, vol. 8, Oct. 1980. [Online]. Available: http://doi.acm.org/10.1145/641914.641917

[5] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. D. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong *et al.*, *SPUR: a VLSI multiprocessor workstation*. University of California, 1985.

[6] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990.

[7] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba, "The SIGMA-1 dataflow supercomputer: A challenge for new generation supercomputing systems," *Journal of Information Processing*, vol. 10, 1987.

[8] S. Sakai, K. Hiraki, Y. Kodama, T. Yuba *et al.*, "An architecture of a dataflow single chip processor," in *ACM SIGARCH Computer Architecture News*, vol. 17, no. 3. ACM, 1989.

[9] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003.

[10] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, "SHAKTI processors: An open-source hardware initiative," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, Jan 2016. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7434907&newsearch=true&queryText=risc-v

[11] "SHAKTI," https://bitbucket.org/casl/shakti_public/. [Online]. Available: https://bitbucket.org/casl/shakti_public/

[12] "Picorv32," https://github.com/cliffordwolf/picorv32. [Online]. Available: https://github.com/cliffordwolf/picorv32

[13] "Scr1," https://github.com/syntacore/scr1. [Online]. Available: https://github.com/syntacore/scr1

[14] "PULP platform," https://github.com/pulp-platform. [Online]. Available: https://github.com/pulp-platform

[15] "Rocket core," https://github.com/ucb-bar/rocket, accessed: 2015-04-07. [Online]. Available: https://github.com/ucb-bar/rocket

[16] "The Berkeley out-of-order RISC-V processor," https://github.com/ucb-bar/riscv-boom, accessed: 2015-04-07. [Online]. Available: https://github.com/ucb-bar/riscv-boom

[17] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017.

[18] U. Banerjee, C. Juvekar, A. Wright, Arvind, and A. P. Chandrakasan, "An energy-efficient reconfigurable DTLS cryptographic engine for end-to-end security in IoT applications," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, Feb 2018.

[19] C. Duran, D. L. Rueda, G. Castillo, A. Agudelo, C. Rojas, L. Chaparro, H. Hurtado, J. Romero, W. Ramirez, H. Gomez, J. Ardila, L. Rueda, H. Hernandez, J. Amaya, and E. Roa, "A 32-bit RISC-V AXI4-lite bus-based microcontroller with 10-bit SAR ADC," in *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, Feb 2016.

[20] J. Gray, "Designing a simple FPGA-optimized RISC CPU and system-on-a-chip," in *2000. [Online]. Available: citeseer.ist.psu.edu/article/gray00designing.html*, 2000.

[21] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, Sep 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1070-9

[22] Y. Lee, B. Zimmer, A. Waterman, A. Puggelli, J. Kwak, R. Jevtic, B. Keller, S. Bailey, M. Blagojevic, P. F. Chiu, H. Cook, R. Avizienis, B. Richards, E. Alon, B. Nikolic, and K. Asanovic, "Raven: A 28nm RISC-V vector processor with integrated switched-capacitor DC-DC converters and adaptive clocking," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015.

[23] "Rocket chip generator," https://github.com/ucb-bar/rocket-chip, accessed: 2015-04-07. [Online]. Available: https://github.com/ucb-bar/rocket-chip

[24] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanović, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE, 2014. [Online]. Available: http://www.cs.berkeley.edu/~yunsup/papers/riscv-esscirc2014.pdf

[25] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Lotfi, J. Puscar, A. Rao, A. Rovinski, L. Salem, N. Sun, C. Torng, L. Vega, B. Veluri, X. Wang, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, I. Galton, R. K. Gupta, P. P. Mercier, M. Srivastava, M. B. Taylor, and Z. Zhang, "Celerity: An open source RISC-V tiered accelerator fabric," in *Symposium on High Performance Chips (Hot Chips)*, ser. Hot Chips 29. IEEE, August 2017.

[26] B. Zimmer, P. F. Chiu, B. Nikoli, and K. Asanovi, "Reprogrammable redundancy for cache Vmin reduction in a 28nm RISC-V processor," in *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, Nov 2016.

[27] B. Keller, M. Cochet, B. Zimmer, Y. Lee, M. Blagojevic, J. Kwak, A. Puggelli, S. Bailey, P. F. Chiu, P. Dabbelt, C. Schmidt, E. Alon, K. Asanovi, and B. Nikoli, "Sub-microsecond adaptive voltage scaling in a 28nm FD-SOI processor SoC," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, Sept 2016.

[28] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevti, B. Keller, S. Bailey, M. Blagojevi, P. F. Chiu, H. P. Le, P. H. Chen, N. Sutardja, R. Avizienis, A. Waterman, B. Richards, P. Flatresse, E. Alon, K. Asanovi, and B. Nikoli, "A RISC-V vector processor with simultaneous-switching switched-capacitor DC converters in 28 nm FDSOI," *IEEE Journal of Solid-State Circuits*, vol. 51, April 2016.

[29] Y. Wang, M. Wen, C. Zhang, and J. Lin, "RVNet: A fast and high energy efficiency network packet processing system on RISC-V," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017.

[30] "Chisel 3," https://github.com/freechipsproject/chisel3. [Online]. Available: https://github.com/freechipsproject/chisel3

[31] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg, "FabScalar: Automating superscalar core design," *IEEE Micro*, vol. 32, May 2012.

[32] V. Srinivasan, R. B. R. Chowdhury, E. Forbes, R. Widialaksono, Z. Zhang, J. Schabel, S. Ku, S. Lipa, E. Rotenberg, W. R. Davis, and P. D. Franzon, "H3 (Heterogeneity in 3D): A logic-on-logic 3D-stacked heterogeneous multi-core processor," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017.

[33] R. B. R. Chowdhury, A. K. Kannepalli, S. Ku, and E. Rotenberg, "AnyCore: A synthesizable RTL model for exploring and fabricating adaptive superscalar cores," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016.

[34] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanović, "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. PP, 2016. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7436635&newsearch=true&queryText=risc-v

[35] N. Dave, Arvind, and M. Pellauer, "Scheduling as rule composition," in *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE) 2007, May 30 - June 1st, Nice, France*, 2007. [Online]. Available: https://doi.org/10.1109/MEMCOD.2007.371249

[36] M. Vijayaraghavan, N. Dave, and Arvind, "Modular compilation of guarded atomic actions," in *11th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2013, Portland, OR, USA, October 18-20, 2013*, 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6670957/

[37] D. L. Rosenband, "A performance driven approach for hardware synthesis of guarded atomic actions," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2005. [Online]. Available: http://hdl.handle.net/1721.1/34473

[38] K. C. Yeager, "The mips r10000 superscalar microprocessor," *Micro, IEEE*, vol. 16, 1996.

[39] S. Zhang, M. Vijayaraghavan, and Arvind, "Weak memory models: Balancing definitional simplicity and implementation flexibility," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, 2017. [Online]. Available: https://doi.org/10.1109/PACT.2017.29

[40] S. Zhang, M. Vijayaraghavan, A. Wright, M. Alipour, and Arvind, "Constructing a weak memory model," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018.

[41] M. Vijayaraghavan, A. Chlipala, N. Dave *et al.*, "Modular deductive verification of multiprocessor hardware designs," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2015.

[42] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/.

[43] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[44] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovi, "FireSim: Cycle-accurate rack-scale system simulation using FPGAs in the public cloud," *7th RISC-V Workshop*, 2017.

[45] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970
[46] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanovic, "Evaluation of RISC-V RTL with FPGA-accelerated simulation," *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.
[47] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE micro*, vol. 19, 1999.
[48] "FireSim demo v1.0 on Amazon EC2 F1," https://fires.im/2017/08/29/firesim-demo-v1.0.html.
[49] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: Nvidia's first 64-bit ARM processor," *IEEE Micro*, vol. 35, 2015.
[50] "Spike, a RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.
[51] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Technical Report HPL-2009-85, 2009.