Formal Verification of an Out-of-Order Multiprocessor against an In-Order Weak-Memory ISA

JANGGUN LEE, KAIST, Republic of Korea
JEEHOON KANG, FuriosaAI, Republic of Korea

Out-of-order multiprocessor is a critical piece of modern hardware, and their verification must solve the following challenges. First, inter-core interleaving, in which the order their reads and writes reach shared memory is unrestricted. Second, intra-core out-of-order execution, in which instructions fire out of program order. The combination of the two yields weak outcomes, which no sequential execution explains, and modern ISA allows such behaviors to account for them. However, the microarchitecture even exhibits *excess out-of-order executions*, temporarily entering states forbidden by the ISA. While discarded later, such states complicate reasoning about the core in full-system verification. Prior works verify a range of processor designs, while none have performed unbounded verification for out-of-order multiprocessor exhibiting such weak outcomes.

We present the first formal verification of an out-of-order multiprocessor against an in-order, weak-memory ISA. Our key idea is a well-designed core specification, which captures the essence of excess executions in a single list of instructions. Building upon this, we decompose the proof into two steps. The first is a core refinement, proving a core implementation against this specification, abstracting away every microarchitectural state except those necessary to reason about excess executions and the core interface. The second is a system inclusion, serializing the out-of-order memory executions and inter-core interleaving into the ISA, easily removing excess executions thanks to the core specification. All of our proofs are mechanized in Rocq, heavily utilizing large language model (LLM) agents to write proofs automatically.

1 Introduction

One of the most fundamental components of modern systems, multicore processors are difficult to design and reason about due to *weak behaviors* which no sequentially consistent execution explains. Such behaviors arise due to two aspects. First, the cores are subject to *inter-core interleaving*, as the order in which their reads and writes reach the shared memory is unrestricted. Interleaving alone yields only *sequential consistency* (SC), where every outcome is explained by some in-order execution of the cores. Second, each core performs various optimizations, notably *intra-core out-of-order execution*, firing independent instructions out of program order, *e.g.*, a later load ahead of an earlier one whose address is not yet ready. Adding out-of-order execution produces observable beahviors in which no sequentially consistent execution explains.

To capture such weak behaviors, modern specifications for instruction set architectures (ISAs) and programming languages employ a *weak memory model*. A rich literature simplifies and improves the design of such weak memory models [2, 10, 17, 23, 24, 27].

Verification of processors. Existing works on weak-memory semantics, however, assume that a processor implements its ISA semantics rather than proving it. Proving that processors implement the ISA is important as **(1)** they are complex, so a violation is likely; and **(2)** a violation is costly, since an in-production processor is difficult to update, unlike patchable software.

Prior works verify a range of processor designs, but none covers a design that **(1)** comprises multiple cores; **(2)** performs intra-core out-of-order execution; **(3)** and enjoys theorems covering arbitrary traces and design parameters (Table 1). Criteria (1) and (2) are essential when proving a design against a weak-memory ISA, as only their combination yields weak outcomes, and (3) is essential for full correctness.

Authors' Contact Information: Janggun Lee, KAIST, Daejeon, Republic of Korea, janggun.lee@kaist.ac.kr; Jeehoon Kang, FuriosaAI, Seoul, Republic of Korea, jeehoon.kang@furiosa.ai.

Table 1. Coverage of the key aspects of out-of-order multiprocessor verification across closely related case studies. ✓ = covered, ✗ = not covered; *unbounded* means correctness for arbitrary traces and design parameters such as buffer size.

	Multicore	Out-of-order	Unbounded
Reid et al. [26]	✓	✓	✗
Choi et al. [8]	✓	✗	✓
Bourgeat et al. [3]	✗	✗	✓
Choi et al. [7]	✗	✗	✓
Ours	✓	✓	✓

Reid et al. [26] presented the verification flow done at Arm, which applies bounded model checking to processor designs. It scales to an out-of-order multiprocessor and reasons against an ISA with a weak memory model. As with bounded model checking in general, it targets only fixed-size designs and bounded execution traces, and does not give full correctness theorems.

Choi et al. [8] presented Kami, a Rocq framework for designing and verifying hardware in the style of the Bluespec [20] hardware description language. A Bluespec module comprises internal state and atomic rules that execute one at a time ("one-rule-at-a-time" semantics). Kami leverages this semantics to prove hardware modules one rule at a time. They modularly designed and proved a pipelined multicore system, including its cache, against SC, with unbounded theorems covering arbitrary traces and design parameters. However, their cores are in-order and exhibit no weak outcomes.

Bourgeat et al. [3] presented Fjfj, another Bluespec-inspired language and verification framework. Fjfj further simplifies Kami's semantics through the restriction that a rule calls at most one update method of each submodule. Under this semantics, Fjfj enables proving hardware modules on a per-method and per-rule basis. They modularly designed and proved a pipelined single-core system with an arbitrary number of in-flight instructions. However, their design targets SC and is an in-order single core.

Choi et al. [7] presented a verification framework for the Verilog hardware description language. They redefine Verilog semantics with least fixpoints, and prove the new semantics equivalent to the standard scheduling semantics for synthesizable designs. On top of this semantics, they verify functional correctness and liveness of a pipelined RISC-V processor, again with unbounded theorems. However, their processor consists of a single in-order core, and they do not connect it to any form of memory.

1.1 The Challenge: Excess Out-of-Order Execution

The main challenge of out-of-order multiprocessor proofs is to tame hardware's *excess out-of-order execution*, which reorders instructions where the ISA does not. In particular, microarchitecture may compute outcomes the ISA forbids, only discarding them at a later time.

Excess by incoherent out-of-order loads. Figure 1 exhibits excess execution induced by out-of-order loads. Here `x` and `y` are shared memory locations, both initially `0`, and `a0` and `a1` are registers. The writer stores `1` to `x` and then to `y`, while the reader loads `y` into `a0` and then `x` into `a1`. For simplicity, assuming that the two stores to `x` and `y` happened in-order, we analyze possible outcomes of the loads.

We consider the *distinct-address* case where $x \neq y$ and *same-address* case where $x = y$. The ISA *allows* the outcome $a0 = 1, a1 = 0$ in the distinct-address variant. Since the read to y resulted in 1, it must have executed after the two stores have finished. As distinct addresses have no coherence

Writer (core 0)		Reader (core 1)	
`sd t0, 0(s0)`	`# x := 1`	`ld a0, 0(s1)`	`# a0 = y`
`sd t1, 0(s1)`	`# y := 1`	`ld a1, 0(s0)`	`# a1 = x`

Variant	Loads target	`a0 = 1, a1 = 0`
distinct addresses	$x \neq y$	✓ allowed
same address	$x = y$	✗ forbidden (coherence)

Fig. 1. Out-of-order execution of two loads. In both variants, the hardware may fire the second load before the first, transiently obtaining `a0 = 1, a1 = 0`; only the distinct-address variant may retire it.

relation, the second load may read the stale initial value of `x`, just as if the two loads had executed in reverse order. The ISA *forbids* that outcome in the same-address variant, since coherence requires a later load to observe an equal-or-newer value than an earlier one.

The hardware, however, may execute the two loads out of order in both variants. If the address calculation of the first load stalls, the second load fires first and reads the stale value `0` into `a1`, so `a0 = 1, a1 = 0` may arise in transient microarchitectural state. In the distinct-address variant, the core may retire both loads, realizing the allowed outcome. In the same-address variant, the core must detect the coherence violation and squash the offending load before it retires, so that the forbidden outcome is never architecturally observable.

Excess by branch misspeculation. Another source of excess execution is instructions executed beyond a branch misspeculation. Branch speculation is one of the most important optimizations, in which the processor fetches instructions beyond a branch instruction by predicting the execution result. If the prediction was incorrect, the core must detect it and squash succeeding instructions before they retire.

1.2 Contributions

We present the first formal verification of an out-of-order multiprocessor against an in-order, weak-memory ISA, unbounded in execution length and design parameters. Specifically, we prove the following theorem:

Theorem 1 (Out-of-order multiprocessor correctness). $n \cdot \mathsf{Core}_i + \mathsf{ShMem} \subseteq \mathsf{ISA}$.

The theorem reads: the composition of n core implementations Core_i, along with a shared memory specification ShMem, behaviorally refines ($\subseteq$) the instruction set architecture ISA. Here, we take the register file of each core as external behavior. Core_i is our single-issue, out-of-order RISC-V core with LR/SC atomics, written as Bluespec-style modules in Fjfj. ShMem is a primitive module specifying the coherent shared memory, which we axiomatize. ISA is our instruction-set architecture, for which we take promising semantics of Pulte et al. [24][1] proven equivalent to the axiomatic RISC-V memory model. We modularly prove Theorem 1 utilizing various module refinements ($\sqsubseteq$) as follows.

The refinement decomposition through Core_s ***(§3).*** Our central contribution is a decomposition of Theorem 1 that tames the hardware's out-of-order execution in two steps, pivoting on a core specification Core_s. The main goal of Core_s is to have a simple reasoning principle on resolving

[1]We replace Pulte et al. [24]'s tree-structured syntax with a list of bytes and a program counter closer to a real ISA, but leave the memory semantics unchanged.

excess executions. Core_s achieves this by merging all instruction information into a single program-order list. With this design, an excess behavior can be resolved simply by dropping all instructions after the violating instruction. Another key aspect is that the positions of each instruction serves as a unique identifier for it, which enables simple coherence calculations to detect loads to squash. Utilizing this, the *core refinement* $\mathsf{Core}_i \sqsubseteq \mathsf{Core}_s$ eliminates every microarchitectural component except the memory-interface bookkeeping. The *system inclusion* $n \cdot \mathsf{Core}_s + \mathsf{ShMem} \subseteq \mathsf{ISA}$ then re-serializes what remains, the observable memory reordering and the inter-core interleaving.

Modular proofs of the two steps (§4). We prove both steps by further decomposition, in a transitive and compositional way. Specifically, we prove the core refinement per component and the system inclusion by chaining intermediate specifications. Refining each of the components of Core_i into a clean specification first enables the core refinement proof to use such specs for clean invariants.

Refinement with preconditions (§5). In a design as complex as an out-of-order processor, many interface methods assume they are called with inputs satisfying some *precondition*. Typical refinement frameworks, however, are unconditional, requiring an implementation to match its specification for every input. Recently, *conditional contextual refinement* (CCR) and its successors [15, 29, 30] implemented refinement with preconditions through *wrappers*, which embed separation logic pre/postconditions into the operational semantics. Instead of reimplementing the complex design of CCR, we develop a lightweight technique limited to pure predicates, which encodes preconditions into existing module specifications rather than semantics. While restrictive, the encoding leaves the framework untouched, reuses the existing metatheory, and suffices for every module specification of §4.

Proof mechanization (§6). We mechanize the end-to-end refinement in Rocq and evaluate the proof effort, with large language model (LLM) agents as the proof workhorse. The human authors designed the initial specifications, invariants, and abstractions above, while the agents wrote the large majority of the mechanized proofs. The agents completed the system inclusion nearly autonomously, consulting us only when one was unprovable or a proof attempt grew outsized.

2 Background

We review the out-of-order multiprocessor (§2.1), the promising semantics it implements (§2.2), and our proof method, modular refinement (§2.3).

2.1 Design of the Out-of-order Multiprocessor

Our multiprocessor design consists of a single-issue, out-of-order RISC-V core implementing RV32I with the LR/SC atomic instructions, inspired by RiscyOO [34], and its connection to an abstract shared memory specification ShMem. We implement the core in the style of Bluespec [20], so a module consists of submodules, rules that update the submodule state, and methods that interact with the outside world. For simplicity, we assume the instruction memory stays constant, so we do not have a separate module for it. The design exhibits load–load, store–load, and store–store reordering.[2] The first two are induced by out-of-order issue of loads, and the last is induced by the store buffer's freedom to request pending addresses in any order.

Figure 2 shows the design of the core. Blue rectangles indicate submodules, while orange round-edged boxes indicate internal rules or external methods. We describe the core's interface, sketch

[2]Notably, it does not exhibit load–store reordering, which is explicitly documented as being preserved in RiscyOO. Many modern cores do not exhibit load–store reordering as it has minimal performance benefits and complicates recovery logic. See Lee et al. [16] for details.

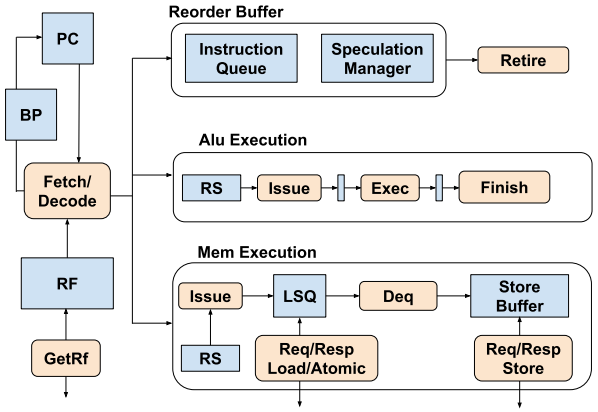


Fig. 2. Design of the out-of-order core. Instructions flow PC → fetch/decode → the arithmetic and memory units.

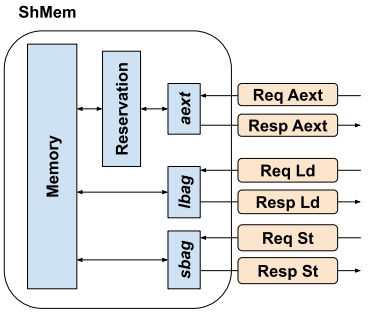


Fig. 3. Specification for shared memory, single-core case for simplicity

how an instruction traverses the core, describe each component, and compose the cores into the full system.

Interface. A single core has seven methods. The GetRf method obtains the core's register file, serving as its external behavior all the way to the ISA. Six memory methods form three request–response pairs, one each for load, store, and atomic operations. Each requests a memory operation to shared memory, waiting for a response to arrive later.

Instruction lifecycle. An instruction dispatches out of order between fetch and retire, which both proceed in program order. We follow the reader of Fig. 1 as it loads y and then x. The core fetches each load from the instruction memory at the program counter, decodes it, and allocates it an entry in the *reorder buffer* (ROB). It then reads the load's source operands, a value or a reference, and places it into the memory unit's *reservation station* (RS), which holds it until the operands are ready and then dispatches it for execution. If the first load's address is not ready, the second load dispatches first, realizing the out-of-order execution of Fig. 1. During execution, the load carries an *issue state*, recording whether it has issued to memory and received its response. The ROB retires the two loads in program order at retire. The core squashes speculatively executed instructions on a misprediction and flushes the pipeline on a load-coherence violation.

Reorder buffer and register file. The ROB holds every in-flight instruction in program order, from allocation at decode to retirement at retire, and serves as the core's dependency and speculation tracker. The register file (RF) keeps the committed architectural state. The ROB resolves each source operand to one of three forms: **(1)** a value from the register file, if no in-flight instruction produces the register; **(2)** a value from the producing in-flight instruction, if it has finished executing in the ROB; **(3)** a reference to the producing in-flight instruction, if it is still executing in the ROB. The core tracks dependencies through ROB indices, which identify each in-flight instruction. The ROB additionally tags every in-flight instruction with a *speculation mask*, which is a bitmask that records which unresolved branches precede it. Tracking speculation inside the ROB is specific to our design (§4.2), as it gives an overall cleaner specification. Instructions retire from the ROB's head, writing their register results to the register file.

Arithmetic unit. The arithmetic logic unit (ALU) executes *pure* instructions, arithmetic and branches, out of program order. It selects any reservation-station entry whose operands are ready, computes the result, and broadcasts it to dependent instructions, all in a pipelined manner. For a branch, the ALU computes the actual direction, trains the branch predictor, and compares it against the prediction made during decode. On a correct prediction, the core clears the branch from every younger instruction's speculation mask. On a misprediction, the core squashes those instructions and redirects the fetch.

Memory unit. The memory unit (MemUnit) issues loads speculatively and delays stores until after retire, using a program-order *load–store queue* (LSQ), a RS, and a *store buffer* (StB). Similar to ALU, entries are marked as ready to request in an out-of-order manner. A load either issues to memory or takes a forwarded value from an older in-flight store to the same address, either from the load–store queue or the store buffer. Loads can issue if there are no older instructions to the same address, which is the source of load–load and store–load reorderings. A store withholds its write until it retires, then moves into the StB and drains to memory. StB drains stores to distinct addresses in no fixed order, making it the source of store–store reordering. Atomic instructions are non-speculative and take effect only at the retire point. An acquire fence blocks later loads from issuing, while a release fence stalls until StB is drained.

Branch predictor. The branch predictor supplies a predicted direction at fetch, letting the core speculate past an unresolved branch and keep fetching. The ALU trains the predictor whenever a branch's result is computed.

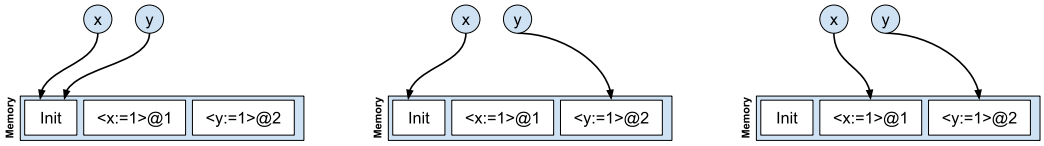
Squashing excess loads. The most important role of the LSQ is detecting excess loads and marking them as to-be squashed. We illustrate the mechanism using the example of Fig. 1 in the same-address case. At the start, the two loads will be placed inside the LSQ in program order. Suppose the second load to x has received its response, and the first load's address calculation has just finished. We scan later instructions in the LSQ, and as the second load violates coherence we mark it as to-be-squashed. The load is squashed at a later time, after it dequeues from the LSQ and moves to the head of the ROB. When such a load is detected, the ROB flags the core to flush the entire pipeline.

Squashing mispredicted branches. Mispredicted branches are eagerly squashed. If a misspeculation is detected after execution, the core records the speculation tag of the branch, and then squashes all instructions with a speculation mask containing the tag.

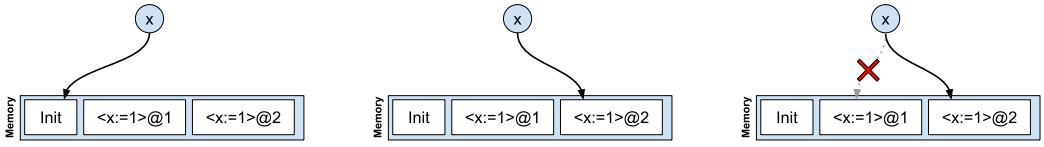
The multicore system. The full system comprises multiple cores Core_i, each connected to the coherent ShMem. Fig. 3 shows the specification of ShMem. Its methods also consists of request-response interface, which are connected to the respective counterparts of each core. The internal states consist of memory and bookkeeping states for pending memory requests. *lbag*, *sbag*, and *aext* store pending loads, stores, and atomic requests, respectively. The system also exposes the per-core GetRf method of each core as external observable state, and internally serves pending requests in a nondeterministic order.

2.2 Promising Semantics

The promising semantics [14, 24] is an operational specification of a weak-memory ISA, in which each thread executes one instruction at a time in program order. It abstracts the out-of-order execution shown in §2.1, recovering the core's out-of-order loads through a *multi-valued* memory, a growing, append-only pool of timestamped *messages*, and out-of-order stores through *promises*, writes performed ahead of program order.



(a) The views of core 1 for different address case.



(b) The views of core 1 for same address case.

Fig. 4. The promising semantics on the load–load example, showing the pool of messages and the reader's coherence view for the distinct-address (top) and same-address (bottom) variants.

Figure 4 replays the load–load example of Fig. 1 under the promising semantics. The writer's two stores append messages to the pool, each consisting of the address, value, and identifier for the writing core. `<x:=1>@1` records the write to `x` at timestamp 1 and `<y:=1>@2` the write to `y` at timestamp 2, joining the initial message `Init`.

Every core keeps a *coherence view*, a timestamp per location that restricts which writes a load may read from, initially at 0. When executing a load, it can only read from writes whose timestamp is at least the coherence view, and also raises the coherence view to the timestamp it has read from. The coherence view admits the weak outcome `a0=1, a1=0` only when `x` and `y` are distinct. When the addresses are distinct (Fig. 4a) reading `a0` from `@2` raises the view of `y` to 2, while its view of `x` is still 0, allowing `a1` to read from the initial state. When `x` and `y` are the same location (Fig. 4b), the first read raises the view to 2, so `a1` must read from a write as young as `@2`.

Stores take effect out of order through *promises*. A thread may promise a future write, making it visible to other cores and itself before it reaches that write in program order. Promises thus admit store–store and store–load weak outcomes. To avoid out-of-thin-air behaviors, a promise must be *certifiable*: the promising thread must be able to *fulfill* all promises when running in isolation.

Promises make the promising semantics *in-order* rather than compromise it. Each thread executes its instructions one at a time in program order, so no instruction takes effect before its program-order predecessors. The model relaxes only the order in which a write becomes visible, never the order in which instructions run. When a thread later reaches the promised store in program order, it fulfills exactly the message it promised.

Promising semantics additionally support acquire and release fences, and atomics which interact with views in non-trivial ways, requiring multiple kinds of additional thread-local views. We refer interested readers to Kang et al. [14], Pulte et al. [24] for the full details.

2.3 Transitive and Compositional Refinement

We explain the definition of modules and refinement between them, and its key properties, transitivity and compositionality. Fjfj [3] realizes modular refinement for Bluespec-style modules, and our development builds on it. A module is a labelled transition system over a private state space, its value methods reading the state and its action methods updating it. It is either primitive, with its transitions given directly as axiomatized specifications, or composed, built from submodules.

In Fjfj, an implementation module M_i refines the specification module M_s, written $M_i \sqsubseteq M_s$, when there exists a simulation relation between their states such that **(1)** related states are indistinguishable through the method interface; and **(2)** every method and rule of M_i preserves the relation. Refinement is transitive and composable:

$$M_1 \sqsubseteq M_2 \implies M_2 \sqsubseteq M_3 \implies M_1 \sqsubseteq M_3, \qquad M_i \sqsubseteq M_s \implies C[M_i] \sqsubseteq C[M_s],$$

where C is a module with submodule M_i. Transitivity lets us build a refinement incrementally through intermediate specifications. Compositionality permits freely replacing a submodule with its specification and vice versa.

Fjfj also provides a more traditional *trace semantics*, given as a sequence of method calls. Refinement implies inclusion between trace sets, which we write $M_i \subseteq M_s$. The trace inclusion of Fjfj guarantees only safety, leaving liveness properties, such as eventual retirement, outside our scope.

3 Proof Overview

We first develop the proof of Theorem 1 in a high-level narrative, while §4 expands each of the two steps into its bottom-up, per-module proof. We utilize transitivity and compositionality to decompose it into two steps meeting at the heart of the proof, the core specification Core_s (§3.1). The two steps are the core refinement $\mathsf{Core}_i \sqsubseteq \mathsf{Core}_s$ (§3.2); and the system inclusion $n \cdot \mathsf{Core}_s + \mathsf{ShMem} \subseteq \mathsf{ISA}$ (§3.3).

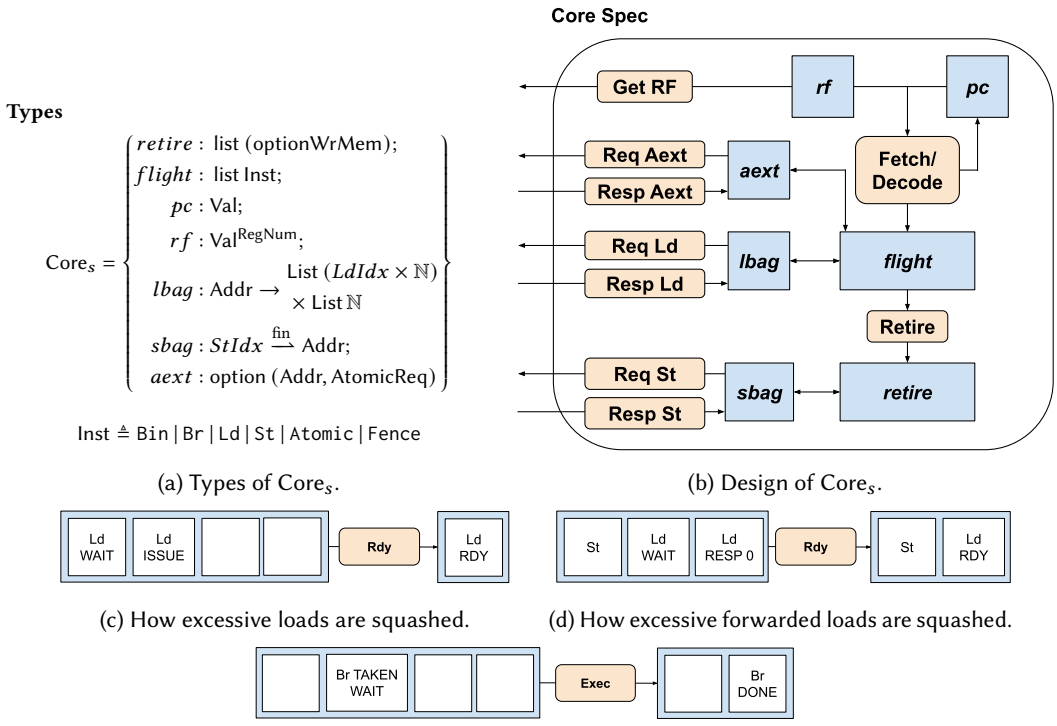
3.1 The Specification Core_s

Fig. 5 describes how Core_s specifies the core. Compared to Core_i, it abstracts the method for speculation detection, the concrete cause of out-of-order execution, and duplicate sources of instruction metadata.

State. Figure 5a describes the internal state of Core_s. At the center is the program-order sequence of instruction consisting of a $retire$ prefix and a $flight$ suffix. A key property is that we do not discard retired instructions but place them in $retire$, so for each in-flight instruction, $|retire|$ + index in $flight$ serves as a unique id, and for retired ones, their index in $retire$ serves as one. We use such an id to track the origin of forwarded stores, which allows us to efficiently compute coherence violations. Each retired instruction is either a to-be-drained store's write to memory (`WrMem`) or is `None`, and each $flight$ entry is an `Inst`. The core also has "architectural state", consisting of the program counter pc and register file rf. Finally, it has $lbag$, $sbag$, and $aext$, holding outstanding load, store, and atomic requests.

Interface. Core_s has the same seven methods as Core_i (§2.1). GetRf returns rf of the specification core. The memory methods thread request payloads through $lbag$, $sbag$, and $aext$ (Fig. 5b). $lbag$ is a per-address collection of two lists. The first contains requested loads, recording the request index and the index of the load in the instruction list. An index of `None` indicates that the corresponding load was squashed. The second contains issued but not yet requested loads (§2.1), recording just the index. We collect them in a per-address list as two load requests to the same address are responded to in the order they are made by ShMem. $sbag$ is a finite map from store request ids ($StIdx$) to the corresponding address. Unlike loads, a core can only have one request for each address, so a finite map suffices. $aext$ records the target address and type (`Lr` or `Sc`) of the core's atomic operation.

Internal transitions. An instruction advances through internal steps until it retires from the head of $flight$ to $retire$ (Fig. 5b). We illustrate the three cases causing excess behaviors, a load, a forwarded load, and a mispredicted branch, and show how the specification handles them similarly.



(a) Types of Core_s.

(b) Design of Core_s.

(c) How excessive loads are squashed.

(d) How excessive forwarded loads are squashed.

(e) How mispredicted branches are squashed.

Fig. 5. The types and design of Core_s

Fig. 5c describes the execution of Fig. 1 at the Core_s level. At the start, the two loads are stored in *flight* in program order, with other instructions following the second load, the second load already issued, and the address calculation for the first load not yet finished. When the address calculation of the first load finishes, it scans later instructions to find incoherent loads. Since the second load read from memory before the first, the younger load may have missed a newer store the older load will read from. Thus, the second load is incoherent, so it and all later instructions are squashed.

Fig. 5d describes another execution with excess loads, this time via forwarding. Initially, there is a store, a waiting load, and a finished load that forwarded from the store. Each responded load records the origin, either from memory ($\perp$) or an earlier store, which is identified by its index in the instruction list. When the address calculation of the first load finishes, since the index of the forwarded store, 0, is less than the index of the load, 1, again the younger load may have missed a newer store the older load will read from, so it is again squashed.

Fig. 5e describes the execution of a mispredicted branch at the Core_s level. At the start, the branch is not yet executed, and records the predicted direction, here predicted as `TAKEN`, followed by more instructions. When the branch executes and turns out to be `NOT_TAKEN`, we squash all later instructions excluding the branch. Unlike loads, we can keep the branch as we know its final result, hence there is no need to re-execute it.

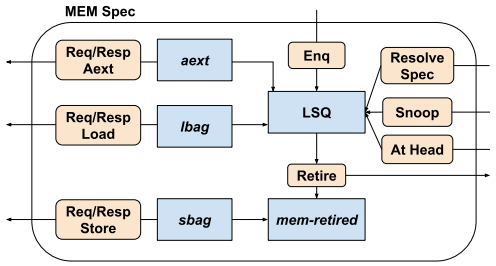


Fig. 6. Design of MemUnit

3.2 Proving $\mathrm{Core}_i \sqsubseteq \mathrm{Core}_s$

We describe the key ideas of the core refinement proof, $\mathrm{Core}_i \sqsubseteq \mathrm{Core}_s$. The proof is done by a simulation argument under a relation, where we must show that if Core_i can take a step, Core_s can take zero or more steps to reestablish the simulation relation. Such matching is almost routine, other than the mismatch of memory operation retirement in Core_i and Core_s. We first describe the internal state of Core_i and the simulation relation, then the key problems and our solutions.

Submodules of Core_i. The specifications of the core components build Core_i's internal state. ROB is a list of instruction metadata, such as instruction id, potential destination register, and whether it is retire-ready. ALU is a finite map from the instruction id of pure instructions to their execution state. Figure 6 describes the specification design of MemUnit. It closely resembles Core_s, but records only memory instructions in LSQ, and omits RF and PC. All of the above components carry a speculation mask corresponding to the instruction. The RF's state is Val^{RegNum}, PC is simply a value, and the branch predictor is of a unit type.

Requirement: forwarding origins must correspond. As the key component of coherence calculation is the origin of forwarded loads (Fig. 5d), it is essential that the origins of forwarded loads recorded in MemUnit are mapped to their respective entries in Core_s. Otherwise, the coherence calculations of MemUnit and Core_s will not match, which can lead to a load being squashed in MemUnit but not in Core_s. As in-flight and retired instructions of both MemUnit and Core_s have a unique id, it may seem trivial to create a one-to-one mapping between them that is easily preserved under instruction squash. But it is not easy to do so due to the root cause of *early retire of memory instructions*.

Problems from early retire of memory instructions. The hardest part of the core proof stems from early retirement of memory instructions. In MemUnit, when a memory instruction retires from LSQ, it is sent to *mem-retired* well-before the corresponding instruction is retired from the ROB, and hence $flight$. This creates two concrete problems, that of *squashed memory instructions* and *acquire operations*.

Squashed-memory instructions. Fig. 7 presents how MemUnit creates retired instructions that Core_s never retires. Initially, there is an uncompleted branch, and a load in both LSQ and $flight$. We retire the load from the LSQ, which places it inside *mem-retired*. The branch then turns out to be mispredicted, which squashes the later load inside $flight$. Thus, we have a memory instruction that exists in *mem-retired* but not *retire* nor $flight$, which must be filtered in the simulation relation.

To resolve this issue, the simulation uses two position maps to record squashed instructions inside *mem-retired*. The total `lsq_map` : List $\mathbb{N}$ assigns each LSQ entry to its position in $flight$,

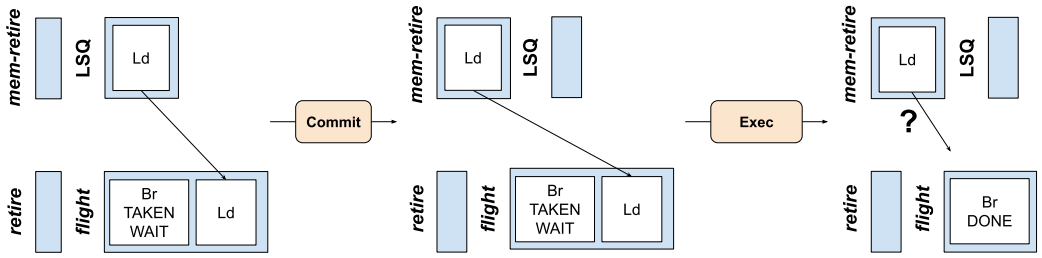
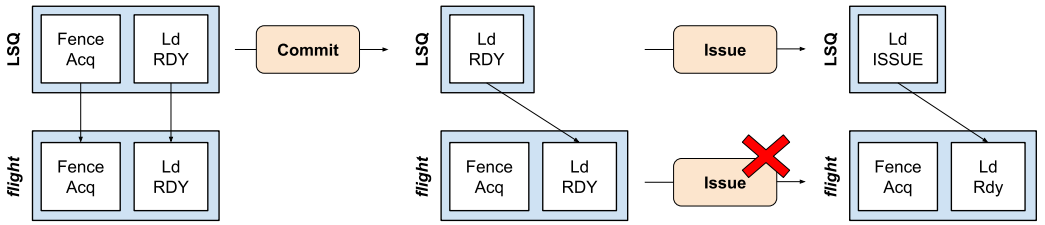
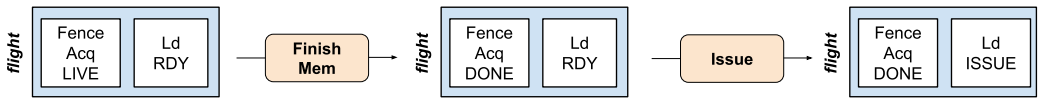


Fig. 7. Execution where an instruction is in *mem-retired* but not in *retire*. After retiring a load, an earlier branch is determined to be mispredicted, which squashes the load and causes the two lists to disagree.



(a) The acquire problem, where LSQ commits an acquire operation and issues a younger load while Core_s cannot.



(b) Core_{sf}'s `FinishMem` transitions clears the fence's live flag, which unblocks younger load's issue.

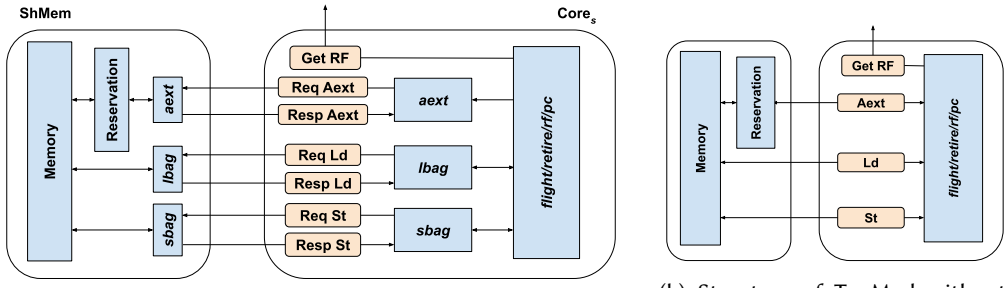
Fig. 8. The acquire problem and how Core_{sf} handles it.

and the partial `ret_map` : List (option $\mathbb{N}$) assigns each *mem-retired* entry to its position in $flight$ or *retire*, with `None` representing squashed memory instructions that Core_s never retired.

Acquire operations problem. Figure 8a describes the acquire operations problem, where a load can issue from Core_i while Core_s forbids it. Initially, both LSQ and $flight$ hold an acquire fence and a load whose address calculation is finished. Recall from §2.1 that an in-LSQ acquire operation blocks younger loads from issuing, and the same goes for Core_s as well. We then dequeue the fence from the LSQ, while it has yet to retire from $flight$. This creates a scenario where we can issue the load inside LSQ, but not the load inside $flight$, as it is still blocked by an acquire fence.

Figure 8b shows how a new intermediate specification Core_{sf}, the *s*pec with unblocking *f*ences, handles the acquire problem. Core_{sf} extends Core_s with a live flag on fence operations, and updates the semantics of acquire fences to block younger loads only while the fence is `LIVE`. The example now proceeds as follows. When we retire the fence from the LSQ, we flip the `LIVE` flag for the corresponding entry in $flight$ to `DONE`, unblocking the younger load. Then, the load in $flight$ can be issued in-sync with the load inside LSQ.

Composing into core refinement. The core refinement is decomposed into two refinements with Core_{sf} in between. The *folding refinement* $\text{Core}_i \sqsubseteq \text{Core}_{sf}$ folds the submodules of Core_i into the



(a) Implementation of Core_s + ShMem.

(b) Structure of TopMod without request–response.

Fig. 9. The goal of $n \cdot \text{Core}_s + \text{ShMem} \sqsubseteq \text{TopMod}$, removing the request–response interface.

instruction list, utilizing the two-map structure and unblocking fences to solve the early retirement gap. The *eager-retirement refinement* $\text{Core}_{sf} \sqsubseteq \text{Core}_s$, restores the core specification Core_s. The crux of this proof is to design an invariant such that when a load issues from Core_{sf}, it can also issue from Core_s. As the name hints, the key idea is to retire instructions from Core_s as eagerly as possible. For example, in the execution of Fig. 8b, Core_s will retire the acquire fence when its live flag flips to `DONE`. This is possible as a fence's live flag only flips when it is at the head of $flight$, so it can be retired as well. A later retire step of Core_{sf} will simply be a no-op in Core_s.

3.3 Proving $n \cdot \text{Core}_s + \text{ShMem} \subseteq \text{ISA}$

We lift the per-core specification to the whole system, connecting n copies of Core_s to the shared-memory specification ShMem and showing the result included in the in-order ISA. The proof is decomposed into three steps as follows:

$$n \cdot \text{Core}_s + \text{ShMem} \;\sqsubseteq\; \text{TopMod} \;\subseteq\; \text{ISA}_{CF} \;\subseteq\; \text{ISA}.$$

TopMod is the top-level implementation module composing the n cores and ShMem, and ISA_{CF} is the certification-free machine of Pulte et al. [24]. In ISA_{CF}, a thread may promise without certifying, and must instead fulfill every promise before the run ends. We take ISA_{CF} as an intermediate step as it's lack of certification makes it easier to work with, and to re-use the proofs of Pulte et al. [24]. §4.8 details each step.

The merging refinement. Fig. 9 shows the *merging refinement* $n \cdot \text{Core}_s + \text{ShMem} \sqsubseteq \text{TopMod}$, whose goal is to remove the bookkeeping for the request–response interface. Fig. 9a shows the implementation of the whole system for one core, which connects the request/response interface of the core to the matching interface of ShMem. A core files a memory request in one transition, and ShMem picks a pending request, applies its effect to memory, and returns the response in another. Fig. 9b shows the design of TopMod, the first specification for the whole system. TopMod keeps each core's $flight$, $retire$, pc, and rf together with the shared memory, leaves every core-internal transition unchanged, and merges each request and its response into one atomic memory transition.

The threading inclusion. The *threading inclusion* $\text{TopMod} \subseteq \text{ISA}_{CF}$ continues to the certification-free machine, whose threads promise writes without certification and must fulfill every promise by the end of the run. We prove trace inclusion directly instead of refinement, since choosing which write to promise requires knowing the rest of the execution, which a step-local simulation cannot see. The proof threads each finished TopMod execution into ISA_{CF} in program order, resolving

Types

$$\mathsf{SIList}\ Idx\ T\ S \triangleq \{list : \mathsf{List}(S \times T);\ idxs : Idx \xrightarrow{\text{fin}} \mathbb{N};\ wf : \mathsf{SIListWF}\ list\ idxs\}$$

Methods

(SIList-Enq)
$\{\mathsf{TLRel}(st.(list), ss)\}$
$\quad \mathsf{enq}(ss, e)$
$\{idx.\ \mathsf{enqueue}(idx, ss, e, st, st')\}$

$$\mathsf{enqueue}(idx, ss, e, st, st',) \triangleq idx \notin st.(idxs)\ \wedge st'.(list) = st.(list) +\!\!+ [(ss, e)] \wedge st'.(idxs) = st.(idxs)[idx \mapsto \mathsf{length}\ st.(list)]$$

(SIList-Deq)
$\{True\}$
$\quad \mathsf{deq}()$
$\{(hd, idx).\ \mathsf{dequeue}(hd, idx, st, st')\}$

$$\mathsf{dequeue}\ hd\ idx\ st\ st' \triangleq \exists\ tl.\ st.(list) = hd :: tl \wedge st.(idxs)[idx] = 0\ \wedge\ st'.(list) = tl \wedge st'.(idxs) = \mathsf{map}(\lambda x.\ x - 1, \mathsf{delete}(idx, st.(idxs)))$$

(SIList-KillIdx)
$\{\mathsf{True}\}\ \mathtt{kill\_idx}(sid)\ \{st' = \mathsf{killIdx}(sid, st)\}$

(SIList-Clear)
$\{\mathsf{True}\}\ \mathtt{clear}()\ \{st' = \langle []; \emptyset; \_\rangle\}$

Fig. 10. Specification of the sorted indexed list SIList.

the inter-core interleaving and the residual out-of-order execution. The main burden is designing sufficient invariants to satisfy coherence, which is not difficult but cumbersome.

The certification inclusion. The *certification inclusion* $\mathsf{ISA}_{CF} \subseteq \mathsf{ISA}$ restores the certification requirement, showing every ISA_{CF} execution is reproduced by a certifying one. We reuse the same proof done by Pulte et al. [24], re-proving the theorem for our notion of ISA and trace refinement, following their structure as closely as possible.

4 Detailed Proofs of the Multiprocessor System

We specify and verify each module bottom-up, each proof giving a clean-slate specification for the implementation. As most modules implement some form of a queue or map, their specification may look routine. However, hardware methods are heavily pipelined, so a method call may only take effect after several internal transitions. Pipelining is the source of proof difficulty, while hardware's verbose interface is merely a source of length.

We proceed through the shared building blocks (§4.1); ROB (§4.2); ALU (§4.3); LSQ (§4.4); StB (§4.5); MemUnit (§4.6); the core refinement (§4.7); and the system inclusion (§4.8). Throughout, st and st' are implicitly quantified, denoting each module's pre- and post-state.

Method specification is given a Hoare triple $\{P(st, a)\}\ e(a)\ \{ret.Q(st, a, st', ret)\}$, which states that if $e(a)$ is called while $P(st, a)$ is satisfied, e returns ret and updates the state to st' satisfying $Q(st, a, st', ret)$. We make the encoding precise in §5.

4.1 Common Building Blocks

The ROB, ALU, and MemUnit rest on three shared modules, the sorted indexed list (SIList), the speculation-tag manager (SMan), and RS, we thus specify these modules first. All three are themselves verified modules, each with its own implementation and refinement proofs which we omit. ROB uses SIList and SMan, ALU uses RS, and MemUnit uses SIList and RS. While SMan is only used in one of them, the speculation mechanism itself is used by ALU and MemUnit.

Types

$$st \in \mathsf{SManS} \triangleq \{l : \mathsf{List}\ \mathit{SId};\ \mathit{ND} : \mathsf{NoDup}\ l\}$$

Methods

(ST-Current)
$\{\mathsf{True}\}\ \mathtt{current}()\ \{s.\ s = \mathsf{toBitSet}\ st.l\}$

(ST-Claim)
$\{\mathsf{True}\}\ \mathtt{claim}()\ \{\mathit{sid}.\ st'.l = \mathit{sid} :: st.l\}$

(ST-Correct)
$\{\mathsf{True}\}$
$\quad \mathtt{correct}(\mathit{sidx})$
$\{st'.l = \mathsf{remove}\ \mathit{sidx}\ st.l\}$

(ST-Wrong)
$$\{\mathsf{True}\}\ \mathtt{wrong}(k)\ \left\{st'.l = \begin{cases}[\,] & k = \mathsf{All} \\ \mathsf{killFrom}\ \mathit{sidx}\ st.l & k = \mathsf{One}\ \mathit{sidx}\end{cases}\right\}$$

Fig. 11. Specification of the speculation-tag manager SMan.

Sorted indexed list. Figure 10 specifies the sorted indexed list SIList $\mathit{Idx}\ T\ S$, a *list* of type T where each entry additionally carries an abstract tag of an ordered type S. SIList also lets a client access elements at an arbitrary index of type Idx. It additionally carries a finite map *idxs* mapping the stable index to an index of the list. The list is sorted with respect to the type S. S records the element's speculation state, namely the branch instructions it depends on. Because the list is sorted by this speculation tag, the tags grow monotonically along program order, so SIList-KillIdx removes a suffix of the list rather than scattered entries. We use a generic type as different lists need to record different information about speculation state. LSQ stores only each entry's branch dependence, while the ROB also marks whether the entry is itself a branch. The well-formedness predicate wf relates *list* and *idxs* and enforces the sort order.

The methods cover a typical queue lifecycle (enq and deq), a speculative kill, and a reset. SIList-Enq adds a new entry and returns a fresh index *idx*, given that the new tag keeps the list sorted (the TLRel premise). SIList-Deq removes and returns the first element of the list, along with the removed index. SIList-KillIdx applies *killIdx*, which finds the first element whose tag includes *sid* and removes it and all later elements. SIList-Clear resets the list to a fresh state.

Speculation-tag manager. Figure 11 specifies SMan as a duplicate-free list of speculation tags with the newest at the head, where *SId* is the tag type. ST-Current returns the current status of SMan as a speculation mask, a bitset of *SId*s. ST-Claim allocates a new speculation tag and prepends it to the list. ST-Correct resolves a correct prediction by removing its tag *sid* from the list. ST-Wrong either clears SMan entirely (k = All) or drops every tag from the head down to *sid* (k = One *sid*).

Reservation station. Figure 12 specifies RS, which holds each pending instruction's source operands until they are ready. RS Idx is a finite map from an index type Idx to the reservation station entry, RSE. Each RSE holds two source operands (r_1, r_2) and the entry's speculation mask (*ss*), a set of *SId*. RsSrc either waits for the instruction with index *idx*, or holds a finished value *val*.

The methods insert and issue entries, snoop results into waiting operands, and resolves speculations. Rs-Insert adds a new entry to the reservation station. Rs-Issue removes and returns an entry whose source operands are both ready. Rs-Snoop finds the source operands waiting on *idx* and sets them ready with *data*. Rs-CorrectSpec drops a correctly predicted branch's tag *sid* from every entry's *ss*. Rs-WrongSpec clears the entire reservation station (k = All), or removes entries dependent on a mispredicted branch (k = One *sid*).

Types

$$st \in Idx \xrightarrow{\text{fin}} \text{RSE } Idx \quad \text{RSE } Idx \triangleq \{r_1 : \text{RsSrc } Idx,\ r_2 : \text{RsSrc } Idx,\ ss : \text{SSet}\}$$

$$\text{RsSrc } Idx \triangleq \text{Wait } (idx : Idx) \mid \text{Data } (val : Val)$$

Methods

(Rs-Insert)
$\{dst \notin \text{dom}(st)\}$
`insert`(dst, e)
$\{st' = st[dst \mapsto e]\}$

(Rs-Issue)
$$\{\text{True}\}\ \texttt{issue}()\left\{(idx, d_1, d_2).\ \begin{array}{l} st[idx] = \langle \text{Data } d_1, \text{Data } d_2, \_\rangle \wedge \\ st' = \text{delete}(idx, st) \end{array}\right\}$$

(Rs-Snoop)
$\{\text{True}\}$
`snoop`$(idx, data)$
$\{st' = \text{map } (\lambda e.\ \text{snoop } idx\ data\ e)\ st\}$

(Rs-CorrectSpec)
$\{\text{True}\}$
`correct_spec`$(sidx)$
$\{st' = \text{map } (\lambda\langle r1, r2, ss\rangle.\ \langle r1, r2, ss \setminus \{[sidx]\}\rangle)\ st\}$

(Rs-WrongSpec)
$$\{\text{True}\}\ \texttt{wrong\_spec}(k)\left\{st' = \begin{cases} \emptyset & k = \text{All} \\ \text{filter}(\lambda rse.\ sidx \notin rse.ss, st) & k = \text{One } sidx \end{cases}\right\}$$

Fig. 12. Specification of the reservation station RS.

Types

$$st \in \text{SIList } Idx \text{ RobE RobSS} \quad \text{RobE} \triangleq \{dst : \text{option Reg};\ ex : \text{option RobSt}\}$$

$$\text{RobSS} \triangleq \{ss : \text{SSet};\ osid : \text{option SId}\} \quad \text{RobSt} \triangleq \text{Done } (redir\ val : \text{option Val}) \mid \text{IsStore}$$

Selected Methods

(Rob-Enq)
$$\{\text{True}\}\ \text{enq}(b, e)\left\{(idx, ss, osid).\ \begin{array}{c} b \Leftrightarrow osid \neq \text{None} \wedge \\ st'.(list) = append(idx, \langle ss, osid\rangle, e, st) \wedge \\ (st.(list) = [\,] \rightarrow ss = \emptyset) \end{array}\right\}$$

(Rob-ReadReg)
$\{\text{True}\}$
`read_reg`(reg)
$\{ret.\ ret = \text{lastWrite}(reg, st)\}$

(Rob-Deq)
$\{\forall e.\ \text{head}(st.(list)) = e \rightarrow e.2.(ex) \neq \text{None} \rightarrow e.1.ss = \emptyset\}$
`deq`$()$
$\{(idx, e).\ e.2.(ex) \neq \text{None} \wedge \text{dequeue}(e, idx, st, st')\}$

(Rob-CorrectSpec)
$\{\text{True}\}$
`correct_spec`(sid)
$\{st' = \text{correctSpec } sid\ st\}$

(Rob-WrongSpec)
$$\{\text{True}\}\ \texttt{wrong\_spec}(k)\left\{st' = \begin{cases} \langle [\,];\ \emptyset;\ \_\rangle & k = \text{All} \\ \text{filter}(\lambda rob_e.\ sidx \notin rob_e.2.ss, st) & k = \text{One } sidx \end{cases}\right\}$$

(Rob-Update)
$\{\exists tag, st.\ (idxs)[idx] = tag\}\ \texttt{set\_exec}(idx, f)\ \{st'.(list) = \text{alter}(f, tag, st.(list))\}$

Fig. 13. Specification of the reorder buffer ROB.

4.2 The Reorder Buffer

The ROB allocates a slot for each fetched instruction, dequeues the instructions in program order, and in our design also manages speculation. Unlike typical designs that keep SMan as a separate module, we embed it inside the ROB, so the ROB's own methods allocate speculation tags. The main benefit is that a client interacts with a single allocator module rather than juggling multiple. A client also need not keep the internal states of SMan and the buffer in sync, because the ROB refinement establishes it instead.

Specification. Figure 13 specifies the ROB as a SIList *Idx* RobE RobSS. RobE records the destination register *dst* (if any) and an optional execution state *ex*, `None` until the instruction executes. Execution state is either *Done*, carrying the register write (if any) and any pc redirect, or *IsStore*. RobSS records two pieces of speculation information for an entry, the speculation mask it depends on (*ss*) and, for a branch entry, its own tag (*osid*). The sort order precisely captures the speculation state: For adjacent entries e_1, e_2, if e_1 is a branch with tag *sid* then $e_2.ss = e_1.ss \cup \{sid\}$; otherwise $e_2.ss = e_1.ss$.

The methods layer speculation bookkeeping over the list operations. Rob-Enq adds a new instruction. Like SIList-Enq it allocates an index *idx*, and it additionally assigns the speculation mask *ss* and, for a branch (b = `true`), a fresh tag *sid*. As a special case, it ensures that if the ROB is empty, *ss* is empty as well. Rob-Deq removes the head entry from the ROB, given that an executed head has an empty *ss*, and guarantees the dequeued entry is executed. The empty-*ss* condition holds because every branch before a committing instruction is already resolved. Rob-ReadReg performs a "register read" on the ROB, trying to obtain the data for the latest entry with destination register *reg*. Rob-Update updates an entry at index *idx* according to the function f. Rob-CorrectSpec removes tag *sid* from every entry and from the *osid* of the matching entry. Rob-WrongSpec either clears or filters the ROB, similar to Rs-WrongSpec.

Implementation. The implementation composes SIList *Idx* RobE SSet (the tag type is SSet, not RobSS) with SMan, and each method forwards to the matching submodule methods.

Proof. The main invariant is that a tag lies in the implementation's SMan exactly when it appears in *osid* of some entry of the specification ROB, while maintaining monotonicity. Rob-Enq is the only hard case. Its use of SIList-Enq needs the ST-Current postcondition together with the fact that every tag in SMan already appears in the ROB. The other cases are routine.

4.3 The Arithmetic Unit

The arithmetic unit executes binary (`Bin`) and branch (`Br`) instructions out of order.

Specification. Figure 14 specifies the arithmetic unit, whose methods mirror the reservation station's up to result computation. The ALU's state pairs an operation type with a reservation-station entry (Fig. 12). Only Alu-Exec differs, additionally computing the result, including branch speculation result, snooping it to the remaining entries, and deleting the issued entry.

Implementation. The ALU is a standard pipelined design, a reservation station feeding a pipelined execution queue. Once the station issues an operation, the queue holds it while the result is computed, until the ALU removes it.

Proof. The proof removes the implementation's internal pipeline rules. We relate the specification ALU state to a folded view of the implementation's RS and execution queue. The key step is Alu-Exec, which essentially applies Rs-Issue and the result computation in one move.

Types

$$st \in Idx \xrightarrow{\text{fin}} (\mathsf{AluOp} \times \mathsf{RSE}\ Idx)$$

Methods

(Alu-Insert)
$\{idx \notin \mathrm{dom}(st)\}$
$\quad \mathsf{add}(idx, e)$
$\{st' = st[idx \mapsto e]\}$

(Alu-Exec)
$$\{\mathsf{True}\}\ \mathsf{exec}()\ \left\{ idx,\ res.\ \begin{array}{l} st(idx) = (op, \langle \mathsf{Data}\, d_1, \mathsf{Data}\, d_2, \_\rangle) \wedge \\ res = \mathsf{eval}(op, d_1, d_2) \wedge \\ st' = \mathsf{delete}(idx, (\mathsf{snoop}(idx, res, st))) \end{array} \right\}$$

(Alu-Snoop)
$\{\mathsf{True}\}$
$\quad \mathsf{snoop}(idx, data)$
$\{st' = \mathrm{map}(\lambda e.\, snoop(idx, data, e), st)\}$

(Alu-CorrectSpec)
$\{\mathsf{True}\}$
$\quad \mathtt{correct\_spec}(sidx)$
$\{st' = \mathrm{map}(\lambda\langle r1, r2, ss\rangle.\, \langle r1, r2, ss \setminus \{[sidx]\}\rangle, st)\}$

(Alu-WrongSpec)
$$\{\mathsf{True}\}\ \mathtt{wrong\_spec}(k)\ \left\{ st' = \begin{cases} \emptyset & k = \mathsf{All} \\ \mathsf{filter}(\lambda rse.\, sidx \notin rse.ss, st) & k = \mathsf{One}\, sidx \end{cases} \right\}$$

Fig. 14. Specification of the arithmetic unit ALU.

4.4 The Load–Store Queue

The LSQ is the core's most intricate module, with the most verbose interface and the most heavily pipelined internal state, which its specification reflects.

Specification. Figure 15 and Fig. 16 specify the LSQ. It is similar to the specifications of Core_s and MemUnit, but does not have a full *sbag*. It consists of a SIList *in* of in-flight memory operations, a per-address load bag *lbag*, which keeps per address an issued list and a requested list, and a list *retire* of retired writes. Each entry, an LSE, carries an at-commit flag and an operation, one of a load, store, load-reserved, store-conditional, or fence.

We describe the methods of the LSQ, omitting those shared with SIList, first following the life cycle of a load instruction (Fig. 16), then other auxiliary methods (Fig. 15). LSQ-RdyLd marks a load's address as computed. It then kills every younger load to the same address that violates coherence, namely one that **(1)** has been issued, **(2)** has received a response from memory, or **(3)** has been forwarded from a store older than the just-readied load. The store, load-reserve, and store-conditional resolutions are analogous and omitted. LSQ-IssueLd issues a ready load, recording the request in *lbag*, when it cannot be forwarded from the store buffer or an in-flight store in the LSQ. LSQ-IssueLd-Fwd instead forwards from an in-flight store, completing the store in one step. LSQ-ReqLd carries the issued load to memory, moving its bag entry from the issued list to the requested list. LSQ-RespLd-Live brings the response back, updating the load to its response state. The case of a response to a squashed load, which the queue discards, is omitted. LSQ-SetComm marks an entry as committed, *i.e.*, at the ROB head. Committing matters for fences and atomic instructions, which can take effect only at the ROB head. LSQ-DeqSt retires a ready store into *retire*, but only if it is guaranteed the *ss* is empty. LSQ-DeqRetire pops the first committed write off *retire*. LSQ-DeqNonSt dequeues the committed head entry when it is not a store and *retire* is empty.

Implementation. The implementation combines a RS, an SIList $LdIdx$ $((\mathsf{LSE}\ Idx) \times Idx)$ SSet, and a finite request map from $LdIdx$ to bool that records the issued load indexes. RS plays the same role as it did in ALU, storing the source operands of memory operations and dispatching them once

Types

$st \in \mathsf{LSQ}\ Idx\ LdIdx\ StIdx$

$\mathsf{WrMem} \triangleq \{addr : \mathsf{Addr};\ val : \mathsf{Val};\}$

$\mathsf{LSE}\ Idx \triangleq \{at\_comm : \mathsf{bool};\ entry : \mathsf{LSSt}\ Idx\}$

$$\mathsf{LSQ}\ Idx \triangleq \left\{ \begin{array}{l} in : \mathsf{SIList}\ Idx\ (\mathsf{LSE}\ Idx)\ \mathsf{SSet}; \\ lbag : \mathsf{Addr} \to \left(\begin{array}{l} \mathsf{List}\ (LdIdx \times \mathsf{option}\ \mathbb{N}) \\ \times\ \mathsf{List}\ \mathbb{N} \end{array} \right); \\ retire : \mathsf{List\ WrMem}; \end{array} \right\}$$

$$\begin{array}{l} \mathsf{LdSt}\ Idx \triangleq \mathsf{Wait}(r : \mathsf{RsSrc}\ Idx) \mid \mathsf{Rdy}\ (a : \mathsf{Addr}) \\ \qquad\qquad \mid \mathsf{Issue}\ (k : \mathsf{bool})\ (a : \mathsf{Addr}) \\ \qquad\qquad \mid \mathsf{Resp}\ (k : \mathsf{bool})\ (a : \mathsf{Addr})\ (d : \mathsf{Val})\ (o : \mathsf{option}\ \mathbb{N}) \end{array}$$

$$\begin{array}{l} \mathsf{LSSt}\ Idx \triangleq \mathsf{Ld}\ (dst : \mathsf{Reg})\ (ld : \mathsf{LdSt}\ Idx) \\ \qquad\qquad \mid \mathsf{St}\ (st : \mathsf{StSt}\ Idx) \\ \qquad\qquad \mid \mathsf{Lr}\ (dst : \mathsf{Reg})\ (lr : \mathsf{LrSt}\ Idx) \\ \qquad\qquad \mid \mathsf{Sc}\ (dst : \mathsf{Reg})\ (sc : \mathsf{ScSt}\ Idx) \\ \qquad\qquad \mid \mathsf{Fence}\ (acq\ rel : \mathsf{bool}) \end{array}$$

Methods

(LSQ-DeqSt)

$$\left\{ \begin{array}{l} \exists addr, data.\ st.(in).(idxs)[idx] = 0 \land \\ st.(in).(list)[0].1.(entry) = \mathsf{St}\ (\mathsf{Rdy}\ addr\ data) \\ st.(in).(list).0 = \emptyset \end{array} \right\} \mathtt{deq\_st}(idx) \left\{ \begin{array}{l} \mathsf{dequeue}(\_, idx, st.(in), st'.(in)) \land \\ st'.(retire) = st.(retire) +\!\!+ [\langle addr, data \rangle] \end{array} \right\}$$

(LSQ-DeqRetire)

$$\{\mathsf{True}\}\ \mathtt{deq\_retire}()\ \left\{ hd. \begin{array}{l} \exists tl.\ st.(retire) = hd :: tl \\ \land\ st'.(retire) = tl \end{array} \right\}$$

(LSQ-DeqNonSt)

$$\{\mathsf{True}\}\ \mathtt{deq\_non\_st}()\ \left\{ (e, idx). \begin{array}{l} e \text{ is not a store } \land\ st.(retire) = [] \\ \mathsf{dequeue}(\langle \mathsf{true}, e \rangle, idx, st.(in), st'.(in)) \end{array} \right\}$$

(LSQ-SetComm)

$$\{\exists tag, st.\ (in).(idxs)[idx] = tag\}\ \mathtt{set\_comm}(idx)\ \{st'.(in).(list) = st.(in).(list)[tag \mapsto \langle \mathsf{true}, st.(in).(list)[tag].(entry) \rangle]\}$$

Fig. 15. Specification of the load–store queue LSQ (state and commit methods).

ready, updating the corresponding LSQ entries. The implementation LSQ is indexed by its own *LdIdx* rather than the ROB's *Idx*, and each entry records the *Idx* of its ROB entry. Reusing *Idx* would force the LSQ to allocate a slot for each non-memory instruction and would tie an entry's lifetime to the ROB, which would be wasteful for LSQ that only needs to track memory operations.

Reusing a load index while its response is outstanding is an instance of the classic *ABA problem* [32]. Concretely, if a load requested with id *lidx* is squashed before its response arrives and its index is re-allocated, LSQ-RespLd-Live would deliver the squashed load's stale response to the new entry, which is clearly incorrect. The request map therefore reserves a load index from issue until its response drains, so a squashed load's index is never re-allocated while its response is outstanding. The map marks *lidx* on issue and forbids allocating a marked index until the response clears it.

The implementation keeps no separate structure for the specification's *retire* list. A committed store instead stays in the LSQ as a ready-to-retire entry, so *retire* is a contiguous prefix of the LSQ itself. LSQ-DeqSt marks the store ready in place rather than dequeuing it, and LSQ-DeqRetire later pops the segment's oldest entry, which the memory unit passes on to the store buffer (§4.6).

Methods (cont.)

(LSQ-RdyLd)

$$\{\mathsf{True}\}\ \mathtt{rdy}()\ \left\{ idx.\ \begin{array}{l} \exists tag, dst, a.\ st.(in).(list)[tag] = \langle \_, \mathsf{Ld}\ dst\ (\mathsf{Wait}\ (\mathsf{Data}\ a))\rangle \\ st.(in).(idxs)[idx] = \mathsf{Some}\ tag \\ st'.(in).(list) = \mathsf{take}\ tag\ st.(in).(list) +\!\!+ [\langle \_, \mathsf{Ld}\ dst\ (\mathsf{Rdy}\ a)\rangle] \\ \qquad\qquad +\!\!+\ \mathsf{map}\ (\mathsf{KillLd}\ a\ tag)\ (\mathsf{drop}\ (tag+1)\ st.(in).(list)) \end{array} \right\}$$

(LSQ-IssueLd)

$$\{\mathsf{True}\}\ \mathtt{issue}(stb\_iss)\ \left\{ \begin{array}{l} \exists tag, dst, a.\ st.(in).(list)[tag] = \langle \_, \mathsf{Ld}\ dst\ (\mathsf{Rdy}\ a)\rangle \\ \mathsf{FwdLSQ}(st, tag, stb\_iss) = \mathsf{None} \\ st'.(in).(list) = st.(in).(list)[tag \mapsto \langle \_, \mathsf{Ld}\ dst\ (\mathsf{Issue}\ \mathsf{false}\ a)\rangle] \\ (st'.(in).(lbag)\ a).2 = (st.(in).(lbag)\ a).2 +\!\!+ [\mathsf{Some}\ tag] \end{array} \right\}$$

(LSQ-IssueLd-Fwd)

$$\{\mathsf{True}\}\ \mathtt{issue}(stb\_iss)\ \left\{ \begin{array}{l} \exists tag, dst, a.\ st.(in).(list)[tag] = \langle \_, \mathsf{Ld}\ dst\ (\mathsf{Rdy}\ a)\rangle \\ \exists d.\ \mathsf{FwdLSQ}(st, tag, stb\_iss) = \mathsf{Some}\ d \\ st'.(in).(list) = st.(in).(list)[tag \mapsto \langle \_, \mathsf{Ld}\ dst\ (\mathsf{Resp}\ \mathsf{false}\ a\ d)\rangle] \end{array} \right\}$$

(LSQ-ReqLd)

$$\{\mathsf{True}\}\ \mathtt{req\_ld}()\ \left\{ (lid, addr).\ \begin{array}{l} \exists oi, tl.\ (st.(in).(lbag)\ addr).2 = oi :: tl\ \wedge\ lid\ \mathsf{fresh}\ \wedge \\ st'.(in).(lbag)\ addr = \langle (st.(in).(lbag)\ addr).1 +\!\!+ [\langle lid, oi\rangle],\ tl\rangle \end{array} \right\}$$

(LSQ-RespLd-Live)

$$\{\exists addr, tag, tl.\ (st.(in).(lbag)\ addr).1 = \langle lid, \mathsf{Some}\ tag\rangle :: tl\}$$

$$\mathtt{resp\_ld}(lid, data)$$

$$\left\{ \begin{array}{l} \exists dst, k.\ st.(in).(list)[tag] = \langle \_,\ \mathsf{Ld}\ dst\ (\mathsf{Issue}\ k\ addr)\rangle\ \wedge \\ st'.(in).(list) = st.(in).(list)[tag \mapsto \langle \_,\ \mathsf{Ld}\ dst\ (\mathsf{Resp}\ k\ addr\ data\ \mathsf{None})\rangle]\ \wedge \\ st'.(in).(lbag)\ addr = \langle tl,\ (st.(in).(lbag)\ addr).2\rangle \end{array} \right\}$$

Fig. 16. Load methods of the load–store queue LSQ.

Proof. The difficult part of the proof is on how LSQ-DeqSt keeps a committed store in the implementation LSQ while the specification dequeues it into *retire*. The simulation relation mainly consists of the following three facts: **(1)** the folded view of the reservation station and the implementation LSQ equals the specification LSQ; **(2)** the ready-to-retire prefix in the implementation LSQ, folded to its write effects, equals the specification's *retire* and has an empty *ss*; and **(3)** the map from *LdIdx* to *Idx* induced by the implementation SIList is injective. The second invariant ensures that such ready-to-retire stores are not squashed by a mispredicted branch (One). For a squashed load, the implementation simply does not squash such ready-to-retire entries. Together, these ensure that ready-to-retire stores of the implementation are not squashed, which allows us to abstract them and combine them into the StB (§4.6).

4.5 The Store Buffer

The store buffer (StB) collects retired stores, sends store requests to memory, and responds to them by coalescing the buffered writes to one address into a single memory write.

Types

$$st \in \mathsf{StsBag}\ StIdx \quad \mathsf{StsBag}\ StIdx \triangleq \{retire : \mathsf{List}(\mathsf{option}\ \mathsf{WrMem});\ sreq : StIdx \xrightarrow{\text{fin}} \mathsf{Addr}\}$$

Methods

(Stb-Fwd)
$\{\mathsf{True}\}$
`fwd`$(addr)$
$\{r.\ r = \mathsf{fwd}\ addr,\ st.(retire)\}$

(Stb-EnqSt)
$\{\mathsf{True}\}$
`enq_st`$(addr, data)$
$\{st'.(retire) = st.(retire) +\!\!+ [\mathsf{Some}\ \langle addr, data\rangle]\}$

(Stb-EnqEmpty)
$\{\mathsf{True}\}$
`enq_empty`$()$
$\{st'.(retire) = st.(retire) +\!\!+ [\mathsf{None}]\}$

(Stb-ReqSt)
$$\{\mathsf{True}\}\ \texttt{req\_st}()\left\{(sid, addr).\ \begin{array}{r} addr \text{ is in } st.(retire) \\ sid, addr \text{ fresh in } st.(sreq) \wedge \\ st'.(sreq) = st.(sreq)[sid \mapsto addr] \end{array}\right\}$$

(Stb-RespSt)
$$\{sid \in \mathsf{dom}\ st.(sreq)\}\ \texttt{resp\_st}(sid)\left\{data.\ \begin{array}{l} \exists addr.\ st.(sreq)[sid] = addr \wedge \\ \mathsf{fwd}\ addr\ st.(retire) = \mathsf{Some}\ data \wedge \\ st'.(sreq) = \mathsf{delete}\ sid\ st.(sreq) \\ st'.(retire) = st.(retire) \text{ with matching entries flipped to None} \end{array}\right\}$$

Fig. 17. Specification of the store buffer SB.

Specification. Figure 17 specifies StB as a list *retire* of committed writes and a finite map *sreq* of requested addresses. *retire*'s length never decreases and its positions are never reused, which is what makes a position a stable identifier that can record a load's forwarding origin.

The methods of the StB add new entries, send requests and responses, and check for forwarded values. Stb-Fwd scans *retire* and returns the data for the last entry with matching address, if any. Stb-EnqSt appends a store's write to *retire*. Stb-EnqEmpty instead appends an empty entry when a non-store instruction retires, keeping *retire*'s positions aligned with retirement order for the forwarding-origin bookkeeping. Stb-ReqSt requests an address in *retire* that is not yet in *sreq*, extending *sreq* with a fresh *StIdx* for it. Stb-RespSt takes an in-flight request's id, returns the address's last matching entry in *retire*, deletes the request from *sreq*, and blanks the drained entries in *retire* to None.

Implementation. The implementation builds upon a single map module, where the key type is *StIdx*, and each element tracks address, data, and issued status. Only `enq_st` is nontrivial, updating the entry that matches the store's address and allocating one otherwise.

Proof. The proof is relatively simple, relying on the following relation: **(1)** the specification's *sreq* map is the implementation map with its non-issued entries dropped; and **(2)** the value that *retire* forwards for an address equals the data the implementation map stores for it

4.6 The Memory Unit

MemUnit ties together the LSQ and StB, along with a register storing the state of atomic requests. We thus present a brief description and omit method specifications.

Types

$$\text{MemUnit } Idx\ LdIdx\ StIdx \triangleq \left\{ \begin{array}{l} lsq : \text{SIList } Idx\ (\text{LSE } Idx)\ \text{SSet}; \quad lbag : \text{Addr} \to \begin{pmatrix} \text{List } (LdIdx \times \text{option } \mathbb{N}) \\ \times\ \text{List } \mathbb{N} \end{pmatrix}; \\ retire : \text{List (option WrMem)}; \quad sreq : StIdx \xrightarrow{\text{fin}} \text{Addr}; \\ aext : \text{option (Addr} \times \text{AtomicReq)}; \end{array} \right\}$$

Fig. 18. Type of the memory unit MemUnit.

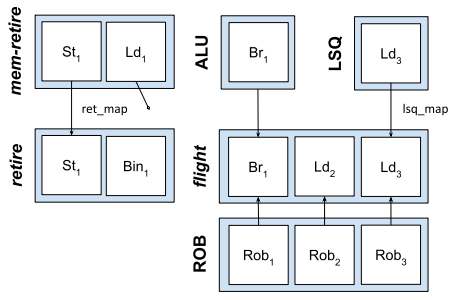


Fig. 19. The simulation relating Core_i to Core_{sf}.

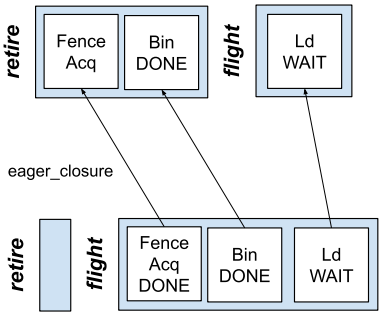


Fig. 20. The simulation relating Core_{sf} to Core_s.

Specification. Figure 18 presents the type of MemUnit, a combination of the LSQ and StB components with the two *retire* lists merged into one, *mem-retired*. It adds a *pending* slot for the outstanding atomic request.

Implementation. Most operations from the LSQ and StB are lifted verbatim. The only meaningful connections are **(1)** threading the result of the StB's `fwd` into the LSQ's `issue`; and **(2)** passing the entry that the LSQ's `deq_retire` returns to the SB's `enq_st`.

Proof. The non-trivial invariant states that the specification's *retire* equals the StB's appended with the LSQ's. The invariant lets the proof hide the move of a retired store from the LSQ to the StB as an internal pipeline step.

4.7 The Core Refinement

Recall from §3.2 the two problems from early memory retirement, the squashed-memory instructions and the early-acquire, and how we suggested two position maps and an intermediate core specification Core_{sf} as a solution. This subsection supplies the detailed proofs. We first explain the components of the folding refinement ($\text{Core}_i \sqsubseteq \text{Core}_{sf}$) and then those of the eager-retirement refinement ($\text{Core}_{sf} \sqsubseteq \text{Core}_s$).

The simulation for folding refinement. Fig. 19 describes the simulation relation for the folding refinement. ROB and $flight$ have a one-to-one correspondence. In particular, one can retire if and only if the other can. Every entry of ALU has a corresponding instruction in $flight.aext$, *sbag*, and *lbag* of MemUnit match routinely to Core_{sf} and is omitted.

The intricate part of the simulation is the mapping from *mem-retired* and LSQ, for which we use the two maps `lsq_map` and `ret_map`. Using them, we define a translation function τ translating

instruction index of MemUnit to that of Core_s:

$$\tau(t) := \begin{cases} \texttt{ret\_map}[t] & \text{if } t < |\texttt{ret\_map}|, \\ (\mathsf{map}(\lambda x.x + |\mathit{retire}|, \texttt{lsq\_map})[t - |\texttt{ret\_map}|] & \text{otherwise.} \end{cases}$$

An index inside *mem-retired* looks up `ret_map`, and a position past it indexes the LSQ, looking up `lsq_map` and landing in flight behind the $|\mathit{retire}|$ entries that Core_s already retired. We then relate the origins of forwarded loads for corresponding MemUnit and Core_s entries via τ, which ensures that coherence calculation results in the same outcome. We maintain the following invariant to guarantee the consistency of τ under squashes. For a forwarded store, the index must be strictly smaller then the load; which ensures that squashes keeps a surviving load's origin intact

The simulation for eager retirement. Fig. 20 describes the simulation relation for the eager-retirement refinement. Its main machinery is the `eager_closure` function, which computes the retireable prefix of Core_{sf}'s flight, moving register writes into Core_s's rf and stores into retire, removing the live flags along the way. For the situation in Fig. 20, it computes that the fence and binary instructions are done, and maps them to the retire state of Core_s, leaving the waiting load untouched.

To prove that we can always retire a fence in Core_s when we mark it as `DONE` in Core_{sf}, we maintain the following invariant on Core_{sf}'s state:

$$\forall p, \mathit{acq}, \mathit{rel}.\ \mathit{flight}[p] = \mathsf{Fence}(\mathit{acq}, \mathit{rel}, \texttt{DONE}) \longrightarrow p = 0 \wedge (\mathit{rel} \longrightarrow \forall e \in \mathit{retire}.\ e = \mathsf{None}).$$

A `DONE` fence sits at the head of *flight*, and if it is a release fence, it leaves no write pending in retire. The head conjunct, $p = 0$, makes every `DONE` fence retire under the closure, so Core_s does not block the loads that Core_{sf} already unblocked. The drain conjunct ensures that even release fences at the head of Core_{sf} can retire early in Core_s. A release fence's retirement condition demands a fully drained retire, so the drain clause lets the closure safely retire them in Core_s.

4.8 Proving $n \cdot \mathrm{Core}_s + \mathrm{ShMem} \subseteq \mathrm{ISA}$

The system inclusion follows the three steps of §3.3 in order, the merging refinement, the threading inclusion, and the certification inclusion. The threading and certification inclusion proceeds as the overview describes and closes the chain to Theorem 1. We thus detail only the merging refinement.

The merging refinement eliminates the request transitions and the bookkeeping they carry. Most importantly, the simulation ensures that the conditions under which the core made a request still hold at response time. For example, it ensures that a requested load satisfies coherence. The simulation also keeps flight, retire, pc, and rf of every equal, and mirrors each core's *lbag*, *sbag*, and *pending* to ShMem's own bookkeeping of the requests it has yet to serve. In terms of actual memory, the implementation coalesces the buffered writes to an address into a single memory write, whereas TopMod performs one write per store, so TopMod's memory has more writes. The two memories thus agree observably, in that every load reads the same value and a store-conditional succeeds in the implementation exactly when it succeeds in TopMod.

5 Modular Refinement with Preconditions

We design a lightweight encoding for extending a refinement framework with preconditions and realize it in Fjfj. Our key idea is lifting the specification state to `option`. We review the per-method proof obligation (§5.1), show why it admits no assumption on arguments (§5.2), and present our encoding, comparing it with CCR (§5.3).

5.1 Background: Per-Method Proof Obligations

Fjfj reduces a refinement proof (§2.3) to one proof obligation per method and rule. Rules need no precondition because they have no caller who could establish one, whereas methods do. We therefore focus on the obligations arising during proofs of methods. A method's specification is a relation $spec(arg, st, st', ret)$ over the argument, the pre- and post-states, and the return value. For a simulation relation ϕ, the proof obligation for a method is:

$$\forall arg, ret, st_i, st_i', st_s.\ \phi(st_i, st_s) \to spec_i(arg, st_i, st_i', ret) \to$$
$$\exists st_s'.\ spec_s(arg, st_s, st_s', ret) \wedge \phi(st_i', st_s')$$

We need to show that if the initial states are related by ϕ, and the implementation can take a step satisfying $spec_i$, the specification can take a corresponding step satisfying $spec_s$ and preserves ϕ.

5.2 Problem: Unconstrained Arguments

The proof obligation leaves no way to assume anything about a method's arguments. Consider a module whose state is a finite map st, with a method `update`(idx, f) that applies f to the value at idx (*e.g.*, `set_exec` of Rob-Update) and returns unit. An ideal specification for `update` would be:

$$spec_{\texttt{update}}((idx, f), st, st', \_) := \exists v.\ st[idx] = v \wedge st' = st[idx \mapsto f(v)].$$

However, the simulation proof requires this specification even for an idx outside the map's domain, where no such v exists.

The fix is to assume the missing precondition in the simulation proof, so that the obligation carries it as a hypothesis:

$$\phi(st_i, st_s) \to spec_i((idx, f), st_i, st_i') \to (\exists v.\ st_s[idx] = v) \to$$
$$\exists st_s'.\ spec_{\texttt{update}}((idx, f), st_s, st_s', \_) \wedge \phi(st_i', st_s')$$

A caller of `update` then owes a proof that the condition holds at its call site.

5.3 Solution: Lifting State to `option`

Our encoding provides a limited form of the precondition mechanism of CCR and its successors [15, 29, 30] by a lightweight encoding, leaving the underlying semantics untouched. CCR realizes conditions as wrappers, where a violated precondition triggers undefined behavior in the semantics. Our encoding instead makes the specification state vacuous, by dropping it to `None`. Their conditions own separation-logic resources and transfer them between caller and callee, while ours is limited to a pure predicate, which is what makes this simple encoding sufficient.

Guarded specifications. Our encoding separates a method specification into a precondition and a postcondition. A *guarded specification* pairs the two, over an argument type A, a state type ST, and a return type R.

$$\text{GSpec}\ A\ ST\ R := \{\ pre : A \to ST \to \text{Prop};\quad post : A \to ST \to ST \to R \to \text{Prop}\ \}.$$

The guarded specification for `update` is given as follows:

$$gspec_{\texttt{update}} := \{\ pre((idx, f), st) := \exists v.\ st[idx] = v;\quad post := spec_{\texttt{update}}\ \}.$$

The figures of §4 render a guarded specification as a Hoare triple, *pre* as the premise and *post* as the postcondition.

***Lifting to* `option`.** We lift both the guarded specification and the simulation relation to `option`, so a method call violating the precondition can simply update the internal state to `None`. A GSpec becomes a method specification over `option` ST as follows:

$$\begin{aligned}\mathsf{lift}(gspec)(arg, ost, ost', ret) :=& \forall st.\ ost = \mathsf{Some}\ st \to pre(arg, st) \to \\ & \exists st'.\ ost' = \mathsf{Some}\ st' \wedge post(arg, st, st', ret)\end{aligned}$$

The simulation relation ϕ lifts alongside:

$$\mathsf{lift}(\phi) := \lambda\, st_i\ ost_s.\ \forall st_s.\ ost_s = \mathsf{Some}\ st_s \to \phi(st_i, st_s).$$

Both lifted forms hold vacuously at `None`. A value method's specification lifts the same way, its postcondition relating the argument, the state, and the return value only.

Obtaining the precondition. We explain how we can obtain the precondition utilizing the guarded specification. The obligation for `update` now runs over the lifted state:

$$\begin{aligned}&\forall (idx, f), st_i, st_i', ost_s.\ \ impl((idx, f), st_i, st_i') \to \mathsf{lift}(\phi)(st_i, ost_s) \to \\ &\exists ost_s'.\ \mathsf{lift}(gspec_{\texttt{update}})((idx, f), ost_s, ost_s') \wedge \mathsf{lift}(\phi)(st_i', ost_s')\end{aligned}$$

We perform case analysis on ost_s and then on the validity of pre. If $ost_s = \mathsf{None}$, choosing $ost_s' = \mathsf{None}$ trivially solves the obligation. If $ost_s = \mathsf{Some}\ st_s$ but $pre((idx, f), st_s)$ does not hold, the proof again chooses $ost_s' = \mathsf{None}$, which allows us to obtain the fact that $pre((idx, f), st_s)$ does hold from $lift(gspec_{\texttt{update}})$, a contradiction. The remaining case is when the precondition holds, and the goal simplifies to the ideal obligation of §5.2.

Soundness for free. We obtain preconditions at zero additional metatheory cost. The encoding only wraps the specification state in `option`, so the existing soundness theorems apply unchanged.

6 Proof Mechanization

We report on the mechanization of the end-to-end refinement (§6.1) and our experience of proving it with LLM agents (§6.2).

6.1 The Rocq Development

The end-to-end refinement of Theorem 1, from the multiprocessor implementation to ISA, is fully mechanized in 64k LOC in Rocq. The implementation is unbounded in the sense of Table 1, and Theorem 1 holds for all instantiations and for arbitrary execution lengths.

For productivity at this scale, we re-implemented Fjfj based on interaction trees [33] rather than the original deeply embedded DSL. This lets us use plain Rocq functions and types in our hardware design. The trusted computing base consists of the Rocq kernel, the semantics of our Fjfj implementation, and primitive modules (register and higher-order vector module), and the definitions involved in Theorem 1.

6.2 LLM Agents as the Proof Workhorse

Automation record. LLM agents wrote the large majority of the mechanized proofs. The agents proved the core refinement (§4.7, one week), the system inclusion (§4.8, three weeks), and most module refinements (rest of §4). Most module proofs finish in under a day, with the exception of large modules such as the load–store queue, which took three days including planning. The resulting proof scripts are much longer than those typically written by humans, which inflates the LOC count.

Workflow. Humans conduct the proof design while agents conduct the proofs. We first draft the implementation, specification, and invariant ideas in a planning document, and refine them with an agent until the design is settled. The agent then writes the actual proof scripts, under the rule that it never changes the implementation or specification on its own but consults us first. When a goal is unprovable, the agent reports a concrete counterexample, which helps localize the flawed invariant, specification, or implementation, for which the human provides concrete fixes or guidance on how to proceed.

Hands-off system inclusion. The system inclusion pushed the delegation to near autonomy. We gave the desired statement $n \cdot \mathsf{Core}_s + \mathsf{ShMem} \subseteq \mathsf{ISA}$, a few principled invariant designs, and the proof scripts of Pulte et al. [24] as reference. The agent concretized the invariants into their full form and proofs on its own, only coming back a few times with counterexamples due to specification or invariant bugs.

Trusting the kernel for generated proofs. The Rocq kernel spares us the usual worry about AI-generated code, that it cannot be trusted without review. Every proof the agent produces must pass the kernel, so an accepted proof is correct regardless of how it was found. The same trust extends to the agent's own delegation, spawning helper agents whose proofs the main agent accepts without reading.

7 Related and Future Work

We already described the closest works in §1. Here, we discuss the remaining related works.

Model checking processors. Modern hardware verification is dominated by model checking, which fundamentally bounds their guarantees to fixed parameters. Burch and Dill [5] verified processor control by flushing the pipeline into an architectural state. Successors reached realistic designs, through compositional model checking of Tomasulo's algorithm and richer microarchitectures [12, 19]. QED [25] scales furthest, reducing compliance to pairwise reorderings with single intervening events, and verifies the load–store queue of the out-of-order BOOM core against the RISC-V memory model for any program length and core count. The guarantee still covers one queue of a fixed size and checks ordering predicates rather than functional refinement.

Foramlly verified processors. Existing verification for processors are limited to in-order processors or SC semantics. Hosabettu et al. [11], Sawada and Hunt [28] verified an out-of-order design with speculative execution, exceptions, and self-modifying code, but is limited to a single core. Vijayaraghavan et al. [31] mechanized a out-of-order multiprocessor against an in-order one in Rocq. However, the verified semantics performs in-order loads, which ultimately results in SC semantics. Kôika [4] is a DSL modeling Bluespec's scheduling semantics, and proves that the semantics preserves one-rule-at-a-time semantics, along a proof of a in-order single core. PipeProof [18] performs unbounded verification formulated as SAT instance, often requiring manual invariants. Their approach does not scale well with instruction length, and only considers in-order processors.

Weak-memory semantics and its clients. Our work provides a new lower bound for existing literature on weak memory semantics. On hardware memory models, Alglave et al. [1] comprehensively tested the processor behavior against axiomatic models, and is part of official ARM. Pulte et al. [23] recast ARMv8's memory model in an abstract microarchitectural style, simplifying the interaction between individual cores and shared memory. Our core specification Core_s shares that philosophy but is much simpler and more presice, as **(1)** they model instructions as trees, while ours is a single list and is much easier for usage in verifications; **(2)** we can compute coherence violation in a thread-local manner by comparing indexes, while theirs requires global anlaysis; and **(3)** ours has a

request–response interface, which requiring well-thought designs of required microarchitectural states. Going more above, compiler correctness proofs [6, 16] bridge until software weak memory models, and program logics [9, 22], and verification on top of them [13, 21], go to user-level libraries. Putting our work below them, we have taken a solid step in creating a verification stack from user libraries down to RTL for realistic efficient weak-memory multiprocessor.

Future work. We plan to extend our design to include features commonly found in out-of-order processors, including register renaming, superscalar execution, and more microoptimizations found in the RiscyOO design. While these optimizations do not increase the behavior the cores exhibit, they are crucial for performance. We additionally plan to synthesize our design, and evaluate its performance against unverified cores. Finally, it will be interesting to add more features to the core, such as exceptions and virtualization, which will also require extending the ISA to include them.

References

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems* 36, 2 (2014), 7:1–7:74. doi:10.1145/2627752

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/1926385.1926394

[3] Thomas Bourgeat, Jiazheng Liu, Adam Chlipala, and Arvind. 2025. Making Concurrent Hardware Verification Sequential. *Proc. ACM Program. Lang.* 9, PLDI, Article 228 (June 2025), 25 pages. doi:10.1145/3729331

[4] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. doi:10.1145/3385412.3385965

[5] Jerry R. Burch and David L. Dill. 1994. Automatic Verification of Pipelined Microprocessor Control. In *Computer Aided Verification, 6th International Conference, CAV '94, Proceedings*. Springer, 68–80. doi:10.1007/3-540-58179-0_44

[6] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 213–228. doi:10.1145/3519939.3523718

[7] Joonwon Choi, Jaewoo Kim, and Jeehoon Kang. 2025. Revamping Verilog Semantics for Foundational Verification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 306 (Oct. 2025), 28 pages. doi:10.1145/3763084

[8] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. doi:10.1145/3110268

[9] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. doi:10.1145/3519939.3523451

[10] Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *Principles of Distributed Systems*, Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–328.

[11] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam K. Srivas. 2003. Formal Verification of a Complex Pipelined Processor. *Formal Methods in System Design* 23, 2 (2003), 171–213. doi:10.1023/A:1024716316140

[12] Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture Verification by Compositional Model Checking. In *Computer Aided Verification, 13th International Conference, CAV 2001, Proceedings*. Springer, 396–410. doi:10.1007/3-540-44585-4_40

[13] Jaehwang Jung, Sunho Park, Janggun Lee, Jeho Yeon, and Jeehoon Kang. 2025. Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI, Article 147 (June 2025), 25 pages. doi:10.1145/3729246

[14] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (jan 2017), 175–189. doi:10.1145/3093333.3009850

[15] Yonghee Kim, Taeyoung Yoon, Sanghyun Yi, Jaehyung Lee, Soonwon Moon, Yeji Han, Seonho Lee, Taeyoung Rhee, Yujin Im, Donghyun Nam, Jieung Kim, and Chung-Kil Hur. 2026. CRIS: The Power of Imagination in Hybrid Verification.

Proc. ACM Program. Lang. 10, PLDI, Article 239 (June 2026), 24 pages. doi:10.1145/3808317

[16] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proc. ACM Program. Lang.* 7, PLDI, Article 183 (June 2023), 24 pages. doi:10.1145/3591297

[17] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. doi:10.1145/3385412.3386010

[18] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. 2018. PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018*. IEEE Computer Society, 788–801. doi:10.1109/MICRO.2018.00069

[19] Kenneth L. McMillan. 1998. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Proceedings*. Springer, 110–121. doi:10.1007/BFb0028738

[20] Rishiyur S. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* (2004), 69–70. https://api.semanticscholar.org/CorpusID:5196158

[21] Sunho Park, Jaehwang Jung, Janggun Lee, and Jeehoon Kang. 2025. Verifying Lock-Free Traversals in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI, Article 149 (June 2025), 25 pages. doi:10.1145/3729248

[22] Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 8, PLDI, Article 154 (June 2024), 24 pages. doi:10.1145/3656384

[23] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. doi:10.1145/3158107

[24] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3314221.3314624

[25] Gokulan Ravi, Xiaokang Qiu, Mithuna Thottethodi, and T. N. Vijaykumar. 2026. QED: Scalable Consistency Verification of Memory Instruction Reordering in Hardware. In *Proceedings of the 53rd Annual International Symposium on Computer Architecture*.

[26] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 42–58.

[27] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. *SIGPLAN Not.* 46, 6 (June 2011), 175–186. doi:10.1145/1993316.1993520

[28] Jun Sawada and Warren A. Hunt, Jr. 2002. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in System Design* 20, 2 (2002), 187–222. doi:10.1023/A:1014122630277

[29] Youngju Song and Minki Cho. 2025. CCR 2.0: High-level Reasoning for Conditional Refinements. arXiv:2507.04298 [cs.PL] https://arxiv.org/abs/2507.04298

[30] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (Jan. 2023), 31 pages. doi:10.1145/3571232

[31] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II*. Springer, 109–127. doi:10.1007/978-3-319-21668-3_7

[32] Wikipedia contributors. 2023. ABA problem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ABA_problem&oldid=1166964304. [Online; accessed 4-August-2023].

[33] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119

[34] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. 2018. Composable building blocks to open up processor design. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 68–81. doi:10.1109/MICRO.2018.00015