Modular Deductive Verification of Multiprocessor Hardware Designs

Muralidaran Vijayaraghavan[1(✉)], Adam Chlipala[1], Arvind[1], and Nirav Dave[2]

[1] MIT, Cambridge, USA
{vmurali,adamc,arvind}@csail.mit.edu
[2] SRI International, Menlo Park, USA
ndave@csl.sri.com

Abstract. We present a new framework for modular verification of hardware designs in the style of the Bluespec language. That is, we formalize the idea of components in a hardware design, with well-defined input and output channels; and we show how to specify and verify components individually, with machine-checked proofs in the Coq proof assistant. As a demonstration, we verify a fairly realistic implementation of a multicore shared-memory system with two types of components: memory system and processor. Both components include nontrivial optimizations, with the memory system employing an arbitrary hierarchy of cache nodes that communicate with each other concurrently, and with the processor doing speculative execution of many concurrent read operations. Nonetheless, we prove that the combined system implements sequential consistency. To our knowledge, our memory-system proof is the first machine verification of a cache-coherence protocol parameterized over an arbitrary cache hierarchy, and our full-system proof is the first machine verification of sequential consistency for a multicore hardware design that includes caches and speculative processors.

1 Introduction

A modern high-performance, cache-coherent, distributed-memory hardware system is inherently complex. Such systems by their nature are highly concurrent and nondeterministic. The goal of this work is to provide a framework for full verification of complex hardware systems.

Modularity has long been understood as a key property for effective design and verification in this domain, decomposing systems into pieces that can be specified and verified independently. In our design, processors and memory systems independently employ intricate optimizations that exploit opportunities for parallelism. We are able to prove that each of these two main components still provides strong guarantees to support sequential consistency (SC) [25], and then compose those proofs into a result for the full system. Either component may be optimized further without requiring any changes to the implementation, specification, or proof of the other. Our concrete optimizations include speculation in processors and using a hierarchy of caches in memory.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part II, LNCS 9207, pp. 109–127, 2015.
DOI: 10.1007/978-3-319-21668-3_7

We thus present **the first mechanized proof of correctness of a realistic multiprocessor, shared-memory hardware system**, including **the first mechanized correctness proof of a directory-based cache-coherence protocol for arbitrary cache hierarchies**, *i.e.,* the proof is parameterized over an unknown number of processors connected to an unknown number of caches in an unknown number of levels (*e.g.,* L1, L2). Our proof has been carried out in the Coq proof assistant and is available at http://github.com/vmurali/SeqConsistency. Since our technique is based on proof assistants, the computational complexity of verification remains constant for any choice of parameters. In the process, we introduce **a methodology for modular verification of hardware designs**, based on the theory of labeled transition systems (LTSes).

LTSes as hardware descriptions are an established idea [2,17,18], for which there are compilers that convert LTSes into efficient hardware. Our work is based on the Bluespec language [3,6], whose semantics match the formalism of this paper. Bluespec specifies hardware components as atomic rules of a transition system over state elements, and its commercial compiler synthesizes these specs into circuits (*i.e.,* Verilog code) with competitive performance. The model that we verify is close to literally transliterated from real Bluespec designs that have been compiled to hardware. Our cache-coherent memory system is based on a Bluespec implementation [13] used to implement an FPGA-based simulator for a cache-coherent multiprocessor PowerPC system [23]. The hardware synthesized from that implementation is rather efficient: an 8-core system with a 2-level cache hierarchy can run 55 million instructions per second on the BEE FPGA board [10]. Within Coq we adopt a semantics style very close to Bluespec, using inductive definitions of state transition systems, where each transition rule corresponds to an atomic Bluespec rule.

Our high-level agenda here is to import to the hardware-verification domain good ideas from the worlds of programming-language semantics and formal software verification, and to demonstrate some advantages of human-guided deductive techniques over model-checking techniques that less readily support modularity and generalization over infinite families of systems, and which may provide less insight to hardware designers (*e.g.,* by not yielding human-understandable invariants about systems).

Paper Organization: We begin with a discussion of related work in Sect. 2. Section 3 introduces our flavor of the labeled transition systems formalism, including a definition of trace refinement. Section 4 shows a generic decomposition of any multiprocessor system, independently of the memory model that it implements, and discusses the store atomicity property of the memory subcomponent. Section 5 gives a simple formal model of sequential consistency. The following sections refine the two main subcomponents of our multiprocessor system. Section 7 discusses definition and verification of a speculative processor model, and Sect. 8 defines and proves our hierarchical cache-coherence protocol. Finally, Sect. 9 shows the whole-system modular proof of our complex system and ends with some conclusions in Sect. 10.

2 Related Work

Hardware verification is dominated by model checking – for example, processor verification [8,29] and (more recently) Intel's execution cluster verification [22]. Many abstraction techniques are used to reduce designs to finite state spaces, which can be explored exhaustively. There are limits to the construction of sound abstractions, so verifications of protocols such as cache-coherence have mostly treated systems with concrete topologies, involving particular finite numbers of caches and processors. For instance, explicit-state model checking tools like Murphi [15] or TLC [21,26] are able to handle only single-level cache hierarchies with fewer than ten addresses and ten CPUs, as opposed to the billions of addresses in a real system, or the ever-growing number of CPUs. Symbolic model-checking techniques have fared better: McMillan *et al.* have verified a two-level MSI protocol based on the Gigamax distributed multiprocessor using SMV [31]. Optimizations on these techniques (*e.g.,* partial-order reduction [4], symmetry reduction [5,11,12,16,19,37], compositional reasoning [20,28,30], extended-FSM [14]) also scale the approach, verifying up to two levels of cache hierarchy, but are unable to handle multi-level hierarchical protocols. In fact, related work by Zhang et al. [37] insists that parameterization should be restricted to single dimensions for the state-of-the-art tools to scale practically. In all these cases, finding invariants automatically is actually hard. Chou et al. [12] require manual insertion of extra invariants, called "non-interference lemmas", to eliminate counterexamples that violate the required property. Flow-based methodology [35] gives yet another way of manually specifying invariants. In general, we believe that the level of complexity of the manually specified invariants between those approaches and ours is similar. Moreover, we might hope to achieve higher assurance and understanding of design ideas by verifying *infinite families* of hardware designs, which resist reduction to finite-state models. Past work by Zhang et al. [37] has involved model-checking hierarchical cache-coherence protocols [38], with a restriction to *binary* trees of caches only, relying on paper-and-pencil proofs about the behavior of fractal-like systems. Those authors agree that, as a result, the protocol suffers from a serious performance handicap. Our cache protocol in this paper is chosen to support more realistic performance scaling.

Theorem provers have also been used to verify microprocessors, *e.g.,* HOL to verify an academic microprocessor AVI-1 [36]. Cache-coherence proofs have also used mechanized theorem provers, though all previous work has verified only single-level hierarchies. Examples include using ACL2 for verifying a bus-based snoop protocol [32], using a combination of model-checking and PVS [33] to verify the FLASH protocol [24], and using PVS to mechanize some portions of a paper-and-pencil proof verifying that the Cachet cache-coherence protocol [34] does not violate the CRF memory model. The first two of these works do not provide insights that can be used to design and verify other protocols. The last falls short of proving a "full functional correctness" property of a memory system. In this paper, we verify that property for a complex cache protocol, based on human-meaningful invariants that generalize to related protocols.

3 Labeled Transition Systems

We make extensive use of the general theory of labeled transition systems, a semantics approach especially relevant to communicating concurrent systems. As we are formalizing processors for Turing-complete machine languages, it is challenging to prove that a system preserves almost any aspect of processor behavior from a model such as SC. To focus our theorems, we pick the time-honored property of *termination*. An optimized system should terminate or diverge iff the reference system could also terminate or diverge, respectively. All sorts of other interesting program properties are reducible to this one, in the style of computability theory. Our basic definitions of transition systems build in special treatment of halting, so that we need not mention it explicitly in most of the following contexts.

Definition 1. *A **labeled transition system (LTS)** is a ternary relation, over $\mathcal{S}^H \times \mathcal{L}^\epsilon \times \mathcal{S}^H$, for some sets $\mathcal{S}$ of states and $\mathcal{L}$ of labels. We usually do not mention these sets explicitly, as they tend to be clear from context. We write X^ϵ for lifting of a set X to have an extra "empty" element ϵ (like an* `option` *type in ML). We write X^H for lifting of a set X to have an extra "halt" element H. We also implicitly consider each LTS to be associated with an* initial *state in $\mathcal{S}$.*

For LTS A, we write $(s) \xrightarrow[\mathrm{A}]{\ell} (s')$ as shorthand for $(s, \ell, s') \in A$, and we write A_0 for A's initial state. The intuition is that A is one process within a concurrent system. The label ℓ from set $\mathcal{L}$ of labels is produced when A participates in some IO exchange with another process; otherwise it is an empty or "silent" label ϵ. For brevity, we may omit labels for ϵ steps.

3.1 Basic Constructions on LTSes

From an LTS representing single-step system evolution, we can build an LTS capturing arbitrary-length evolutions.

Definition 2. *The **transitive-reflexive closure** of A, written A^*, is a derived LTS. Where A's states and labels are $\mathcal{S}$ and $\mathcal{L}$, the states of A^* are $\mathcal{S}$, and the labels are $\mathcal{L}^*$, or* sequences *of labels from the original system. A^* steps from s to s' when there exist zero or more transitions in A that move from s to s'. The label of this transition is the* concatenation *of all labels generated in A, where the empty or "silent" label ϵ is treated as an identity element for concatenation.*

We also want to compose n copies of an LTS together, with no explicit communication between them. We apply this construction later to lift a single-CPU system to a multi-CPU system.

Definition 3. *The n-**repetition** of A, written A^n, is a derived LTS. Where A's states and labels are $\mathcal{S}$ and $\mathcal{L}$, the states of A^n are $\mathcal{S}^n$, and the labels are $[1, n] \times \mathcal{L}$, or pairs that* tag *labels with which component system generated them. These labels are generated only when the component system generates a label. The whole system halts whenever one of the components halts.*

Eventually, we need processes to be able to communicate with each other, which we formalize via the + composition operator that connects same-label transitions in the two systems, treating the label as a cooperative communication event that may now be hidden from the outside world, as an ϵ label.

Definition 4. *Where A and B are two LTSes sharing labels set $\mathcal{L}$, and with state sets $\mathcal{S}_A$ and $\mathcal{S}_B$ respectively, the* ***communicating composition*** *$A + B$ is a new LTS with states $\mathcal{S}_A \times \mathcal{S}_B$ and an empty label set, defined as follows:*

$$A\,\frac{(a) \xrightarrow[A]{} (a') \qquad a' \neq H}{(a,b) \xrightarrow[A+B]{} (a',b)} \qquad B\,\frac{(b) \xrightarrow[B]{} (b') \qquad b' \neq H}{(a,b) \xrightarrow[A+B]{} (a,b')} \qquad H_A\,\frac{(a) \xrightarrow[A]{} (H)}{(a,b) \xrightarrow[A+B]{} (H)}$$

$$H_B\,\frac{(b) \xrightarrow[B]{} (H)}{(a,b) \xrightarrow[A+B]{} (H)} \qquad \mathit{Join}\,\frac{(a) \xrightarrow[A]{\ell} (a') \qquad (b) \xrightarrow[B]{\ell} (b') \qquad a', b' \neq H}{(a,b) \xrightarrow[A+B]{} (a',b')}$$

3.2 Refinement Between LTSes

We need a notion of when one LTS "implements" another. Intuitively, transition labels and halting are all that the outside world can observe. Two systems that produce identical labels and termination behavior under all circumstances can be considered as safe substitutes for one another. We need only an asymmetrical notion of compatibility:

Definition 5. *For some label domain $\mathcal{L}$, let $f : \mathcal{L} \to \mathcal{L}^\epsilon$ be a function that is able to replace labels with alternative labels, or erase them altogether. Let LTSes A and B have the same label set $\mathcal{L}$. We say that A* ***trace-refines*** *B* ***w.r.t.*** *f, or $A \sqsubseteq_f B$, if:*

$$\forall s_A, \eta.\ (A_0) \xrightarrow[A^*]{\eta} (s_A) \Rightarrow \exists s_B.\ (B_0) \xrightarrow[B^*]{f(\eta)} (s_B)\ \wedge (s_A = H \Leftrightarrow s_B = H)$$

Each label in the trace is replaced by the mapping of f on it, and labels mapped to ϵ by f are dropped. f is overloaded to denote the multilabel version when applied to η.

For brevity, we write $A \sqsubseteq B$ for $A \sqsubseteq_{\mathsf{id}} B$, for identity function id, forcing traces in the two systems to match exactly. Under this notion of identical traces, we say that A is sound w.r.t. B. That case matches traditional notions of trace refinement, often proved with simulation arguments, which we also adopt.

3.3 A Few Useful Lemmas

We need the following theorems in our proof.

Theorem 1. *$\sqsubseteq$ is reflexive and transitive.*

Theorem 2. *If $A \sqsubseteq_f B$, then $A^n \sqsubseteq_{f^n} B^n$, where f^n is f lifted appropriately to deal with indices ($f^n(i,\ell) = (i,\ell')$ when $f(\ell) = \ell'$, and $f^n(i,\ell) = \epsilon$ when $f(\ell) = \epsilon$).*

Theorem 3. *If $A \sqsubseteq_f A'$ and $B \sqsubseteq_f B'$, then $A + B \sqsubseteq_{\mathsf{id}} A' + B'$.*

All these theorems can be proved using standard techniques.

4 Decomposing a Shared-Memory Multiprocessor System

Any conventional multiprocessor system can be divided logically into three components, as shown in Fig. 1. The top-level system design is shown in the middle, while the details of its components, the memory system and the processor (P_i), are shown in the magnified boxes. The processor component P_i can be implemented in a variety of ways, from one executing instructions one-by-one in program order, to a complex one speculatively executing many instructions concurrently to exploit parallelism. The memory component is normally implemented using a hierarchy of caches, in order to increase the performance of the overall system, because the latency of accessing memory directly is large compared to that of accessing a much smaller cache. Between each processor and the global memory subsystem appears some local buffer, LB_i, each specific to processor P_i.

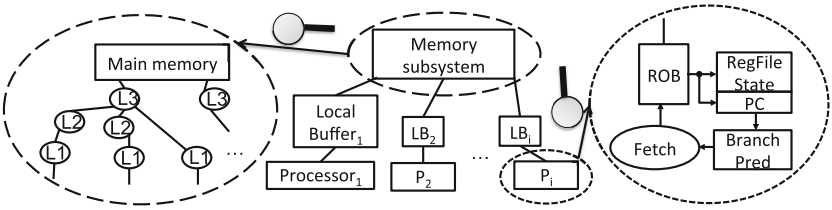

Fig. 1. Components of a multiprocessor system

Popular ISAs, such as Intel x86, ARM, and PowerPC, do not guarantee sequential consistency. However, we want to emphasize that, in every weak-memory system we are aware of, *the main memory still exposes atomic loads and stores*! Weaker semantics in a core P_i arise only because of (1) reordering of memory instructions by the core and/or (2) the properties of the local buffers LB_i connected to P_i.

Consequently, we focus on this opportunity to simplify proof decomposition. We prove that our main memory component satisfies an intuitive *store atomicity* property – which is an appropriate specification of the memory component even for implementations of weaker memory models. Store atomicity can be understood via the operational semantics of Fig. 2, describing an LTS that receives load and store requests (Ld and St) from processors and sends back load responses (LdRp). The transfer happens via input buffers $ins(p)$ from processor p and output buffers $outs(p)$ to processor p. Note that this model allows the memory system to answer pending memory requests in any order (as indicated by the bag union operator $\uplus$), even potentially reordering requests from a single processor, so long as, whenever it does process a request, that action appears to take place *atomically*.

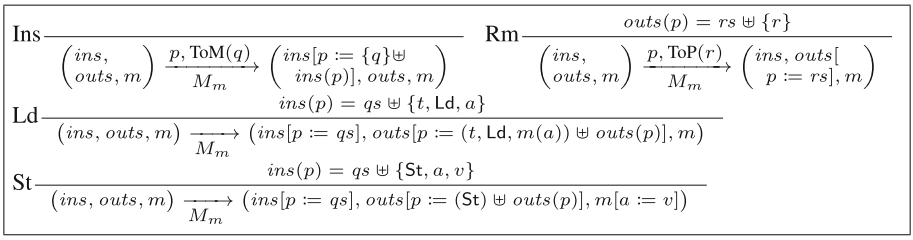

Fig. 2. LTS for a simple memory

Figure 2 provides our first example of a hardware component specified as an LTS via a set of inference rules. Such notation may seem far from the domain of synthesizable hardware, but it is actually extremely close to Bluespec notation, and the Bluespec compiler translates automatically to hardware circuits in Verilog [1].

The memory component is composed of a *hierarchy of caches*, with cache nodes labeled like "L1," "L2," etc., to avoid the latency of round trips with main memory. Therefore, it is the responsibility of the hierarchy of caches (which forms the memory subcomponent) to implement the store atomicity property. In fact, as we prove in Sect. 8, the purpose of the cache-coherence protocol is to establish this invariant for the memory subcomponent. Concretely, we have verified a *directory-based* protocol for coordinating an arbitrary tree of caches, where each node stores a conservative approximation of its children's states.

As an instance of the above decomposition, we prove that a multiprocessor system with no local buffering in between the processor and the memory components indeed implements SC. We implement a highly speculative processor that executes instructions and issues loads out of order, but commits instructions (once some "verification" is done) in order.

The processor itself can be decomposed into several components. In the zoomed-in version of Fig. 1, we show a highly speculative out-of-order-issue processor. We have the normal architectural state, such as values of registers. Our proofs are generic over *a family of instruction set architectures*, with parameters for opcode sets and functions for executing opcodes and decoding them from memory. Other key components are a *branch predictor*, which guesses at the control-flow path that a processor will follow, to facilitate speculation; and a *reorder buffer (ROB)*, which decides which instructions along that path to try executing ahead of schedule. Our proofs apply to an arbitrary branch predictor and any reorder buffer satisfying a simple semantic condition.

Our framework establishes theorems of the form "if system A has a run with some particular observable behavior, then system B also has a run with the same behavior." In this sense, we say that A correctly implements B. Other important properties, such as *deadlock freedom* for A (which might get stuck without producing any useful behavior), are left for future work.

5 Specifying Sequential Consistency

Our final theorem in this paper establishes that a particular complex hardware system implements sequential consistency (SC) properly. We state the theorem in terms of the trace refinement relation $\sqsubseteq$ developed in Sect. 3. Therefore, we need to define an LTS embodying SC. The simpler this system, the better. We need not worry about its performance, since we prove that an optimized system remains faithful to it.

Figure 3 defines an LTS for an n-processor system that is sequentially consistent, parameterized over details of the ISA. In particular, the ISA gives us some domains of architectural states s (e.g., register files) and of program counters pc. A function $\mathsf{dec}(s, pc)$ figures out which instruction pc references in the current state, returning the instruction's "decoded" form. A companion function $\mathsf{exec}(s, pc, d)$ actually executes the instruction, returning a new state s' and the next program counter pc'.

$$\text{Halt}\ \frac{\theta(i) = (s, pc) \quad \mathsf{dec}(s, pc) = H}{(\theta, m) \xrightarrow[\text{SC}]{} (H)} \qquad \text{NonMem}\ \frac{\theta(i) = (s, pc) \quad \mathsf{dec}(s, pc) = (\mathsf{Nm}, x) \quad \mathsf{exec}(s, pc, (\mathsf{Nm}, x)) = (s', pc')}{(\theta, m) \xrightarrow[\text{SC}]{} (\theta[i := (s', pc')], m)}$$

$$\text{Load}\ \frac{\theta(i) = (s, pc) \quad \mathsf{dec}(s, pc) = (\mathsf{Ld}, x, a) \quad \mathsf{exec}(s, pc, (\mathsf{Ld}, x, m(a))) = (s', pc')}{(\theta, m) \xrightarrow[\text{SC}]{} (\theta[i := (s', pc')], m)}$$

$$\text{Store}\ \frac{\theta(i) = (s, pc) \quad \mathsf{dec}(s, pc) = (\mathsf{St}, a, v) \quad \mathsf{exec}(s, pc, (\mathsf{St})) = (s', pc')}{(\theta, m) \xrightarrow[\text{SC}]{} (\theta[i := (s', pc')], m[a := v])}$$

Fig. 3. LTS for SC with n simple processors

The legal instruction forms, which are outputs of dec, are (Nm, x), for an operation not accessing memory; (Ld, x, a), for a memory load from address a; (St, a, v), for a memory store of value v to address a; and H, for a "halt" instruction that moves the LTS to state H. The parameter x above represents the rest of the instruction, including the opcode, registers, constants, *etc.*

The legal inputs to exec encode both a decoded instruction and any relevant responses from the memory system. These inputs are (Nm, x) and St, which need no extra input from the memory; and (Ld, x, v), where v gives the contents of the requested memory cell.

We define the initial state of SC as (θ_0, m_0), where m_0 is some initial memory fixed throughout our development, mapping every address to value v_0; and θ_0 maps every processor ID to (s_0, pc_0), using architecture-specific default values s_0 and pc_0.

This LTS encodes Lamport's notion of SC, where processors take turns executing nondeterministically in a simple interleaving. Note that, in this setting, an operational specification such as the LTS for SC is precisely the proper characterization of *full functional correctness* for a hardware design, much as a precondition-postcondition pair does that in a partial-correctness Hoare logic.

Our SC LTS fully constrains observable behavior of a system to remain consistent with simple interleaving. Similar operational models are possible as top-level specifications for systems following weaker memory models, by giving the LTS for the *local buffer* component and composing the three components simultaneously.

Our final, optimized system is parameterized over an ISA in the same way as SC is. In the course of the rest of this paper, we define an optimized system O and prove $O \sqsubseteq \mathrm{SC}$. To support a modular proof decomposition, however, we need to introduce a few intermediate systems first.

$$\textsf{Halt}\ \frac{\mathsf{dec}(s, pc) = H}{(s, pc, \bot) \xrightarrow[\mathrm{P_{ref}}]{} (H)} \qquad \textsf{NM}\ \frac{\mathsf{dec}(s, pc) = (\mathsf{Nm}, x) \quad \mathsf{exec}(s, pc, (\mathsf{Nm}, x)) = (s', pc')}{(s, pc, \bot) \xrightarrow[\mathrm{P_{ref}}]{} (s', pc', \bot)}$$

$$\textsf{LdRq}\ \frac{\mathsf{dec}(s, pc) = (\mathsf{Ld}, x, a)}{(s, pc, \bot) \xrightarrow[\mathrm{P_{ref}}]{\mathrm{ToM}(\epsilon, \mathsf{Ld}, a)} (s, pc, \top)} \qquad \textsf{StRq}\ \frac{\mathsf{dec}(s, pc) = (\mathsf{St}, a, v)}{(s, pc, \bot) \xrightarrow[\mathrm{P_{ref}}]{\mathrm{ToM}(\mathsf{St}, a, v)} (s, pc, \top)}$$

$$\textsf{LdRp}\ \frac{\mathsf{dec}(s, pc) = (\mathsf{Ld}, x, a) \quad \mathsf{exec}(s, pc, (\mathsf{Ld}, x, v)) = (s', pc')}{(s, pc, \top) \xrightarrow[\mathrm{P_{ref}}]{\mathrm{ToP}(\epsilon, \mathsf{Ld}, v)} (s', pc', \bot)} \qquad \textsf{StRp}\ \frac{\mathsf{dec}(s, pc) = (\mathsf{St}, a, v) \quad \mathsf{exec}(s, pc, (\mathsf{St})) = (s', pc')}{(s, pc, \top) \xrightarrow[\mathrm{P_{ref}}]{\mathrm{ToP}(\mathsf{St})} (s', pc', \bot)}$$

Fig. 4. LTS for a simple decoupled processor ($\mathrm{P_{ref}}$)

6 Respecifying Sequential Consistency with Communication

Realistic hardware systems do not implement the monolithic SC of Fig. 3 directly. Instead, there is usually a split between processors and memory. Here we formalize that split using LTSes that compose to produce a system refining the SC model.

Figure 4 defines an LTS for a simple *decoupled* processor ($\mathrm{P_{ref}}$). Memory does not appear within a processor's state. Instead, to load from or store to an address, *requests* are sent to the memory system and *responses* are received. Both kinds of messages are encoded as labels: ToM for requests to memory and ToP for responses from memory back to the processor.

A state of $\mathrm{P_{ref}}$ is a triple (s, pc, wait), giving the current architectural state s and program counter pc, as well as a Boolean flag *wait* indicating whether the processor is blocked waiting for a response from the memory system. As in the SC model, the state of the processor is changed to H whenever dec returns H.

As initial state for system P_{ref}, we use $(s_0, pc_0, \bot)$.

The simple memory defined earlier in Fig. 2 is meant to be composed with $\mathrm{P_{ref}}$ processors. A request to memory like (t, Ld, a) asks for the value of memory cell a, associating a *tag* t that the processor can use to match responses to requests. Those responses take the form (t, Ld, v), giving the value v of the requested memory address.

A memory state is a triple $(ins, outs, m)$, giving not just the memory m itself, but also buffers *ins* and *outs* for receiving processor requests and batching up responses to processors, respectively. We define the initial state of the M_m LTS as $(\emptyset, \emptyset, m_0)$, with empty queues.

Now we can compose these LTSes to produce an implementation of SC.

For a system of n processors, our decoupled SC implementation is $\mathrm{P}^n_{\mathrm{ref}} + M_m$.

Theorem 4. $P^n_{ref} + M_m \sqsubseteq SC$

Proof. By induction on traces of the decoupled system, relating them to those of the SC reference (similar to the technique in WEB refinement [27]). We need to choose an abstraction function f from states of the complex system to states of the simple system. This function must be inductive in the appropriate sense: a step from s to s' on the left of the simulation relation must be matched by sequences of steps on the right from $f(s)$ to $f(s')$. We choose f that just preserves state components in states with no pending memory-to-processor responses. When such responses exist, f first executes them on the appropriate processors. □

7 Speculative Out-of-Order Processor

We implement a *speculative* processor, which may create many simultaneous outstanding requests to the memory – as an optimization to increase parallelism. Our processor proof is in some sense generic over correct speculation strategies. We parameterize over two key components of a processor design: a *branch predictor* (which makes guesses about processor-local control flow in advance of resolving conditional jumps) and a *reorder buffer* (which decides what speculative instructions – such as memory loads – are worth issuing at which moments, in effect *reordering* later instructions to happen before earlier instructions have finished).

The branch predictor is the simpler of the two components, whose state is indicated with metavariable *bp*. The operations on such state are $\mathsf{curPpc}(bp)$ (to extract the current program-counter prediction); $\mathsf{nextPpc}(bp)$ (to advance to predicting the next instruction); and $\mathsf{setNextPpc}(bp, pc)$ (to reset prediction to begin at a known-accurate position *pc*). We need not impose any explicit correctness criterion on branch predictors; the processor uses predictions only as hints, and it always resets the predictor using setNextPpc after detecting an inaccurate hint.

The interface and formal contract of a reorder buffer are more involved. We write *rob* as a metavariable for reorder-buffer state, and ϕ denotes the state of an empty buffer. The operations associated with *rob* are:

- $\mathsf{insert}(pc, rob)$, which appends the program instruction at location *pc* to the list of instructions that the buffer is allowed to consider executing.
- $\mathsf{compute}(rob)$, which models a step of computation inside the buffer, returning both an updated state and an optional speculative load to issue. For instance,

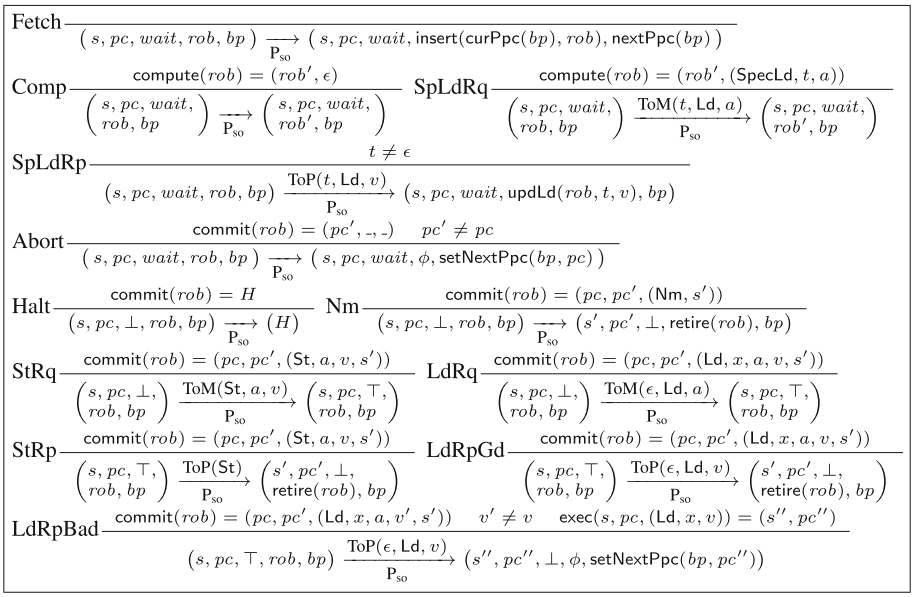

Fig. 5. Speculating, out-of-order issue processor

it invokes the dec and exec functions (as defined for SC) internally to obtain the next program counter, state, *etc.* (but the actual states are not updated).
- updLd(rob, t, v), which informs the buffer that the memory has returned result value v for the speculative load with tag $t \neq \epsilon$.
- commit(rob), which returns the next instruction in serial program order, if we have accumulated enough memory responses to execute it accurately, or returns ϵ otherwise. When commit returns an instruction, it also returns the associated program counter plus the next program counter to which it would advance afterward. Furthermore, the instruction is extended with any relevant response from memory (used only for load instructions, obtained through updLd) and with the new architectural state (*e.g.,* register file) after execution.
- retire(rob), which informs the buffer that its commit instruction was executed successfully, so it is time to move on to the next instruction.

Figure 5 defines the speculative processor LTS $\mathrm{P_{so}}$. This processor may issue arbitrary speculative loads, but it *commits* only the instruction that comes next in serial program order. The processor will issue two kinds of loads, a speculative load (whose tag is not ϵ) and a commit or real load (whose tag is ϵ). To maintain SC, every speculative load must have a matching verification load later on, and we maintain the illusion that the program depends only on the results of verification loads, which, along with stores, *must be issued in serial program order*.

When committing a previously issued speculative load instruction, the associated speculative memory load response is verified against the new commit load

response. If the resulting values do not match, the processor terminates all past uncommitted speculation, by emptying the reorder buffer and resetting the next predicted program counter in the branch predictor to the correct next value. In common cases, performance of executing loads twice is good, because it is likely that the verification load finds the address already in a local cache – thanks to the recent processing of the speculative load. Moreover, 60 % to 90 % of verification loads can be avoided by tracking speculative loads [9]; in the future we will extend our proofs to include such optimizations.

A full processor state is $(s, pc, wait, rob, bp)$, comprising architectural state, the program counter, a Boolean flag indicating whether the processor is waiting for a memory response about an instruction being committed, and the reorder-buffer and branch-predictor states. Its initial state is given by $(s_0, pc_0, \bot, \phi, bp_0)$. The interface of this processor with memory (*i.e.,* communication labels with $\mathsf{ToM}, \mathsf{ToP}$) is identical to that of the reference processor.

Finally, we impose a general correctness condition on reorder buffers (Fig. 6). Intuitively, whenever the buffer claims (via a commit output) that a particular instruction is next to execute (thus causing certain state changes), that instruction must really be next in line according to how the program runs in the SC system, and its execution must really cause those state changes.

ROB-invariant: If P_{so} reaches a state $(s, pc, wait, rob, bp)$,

$$\begin{cases} \mathsf{commit}(rob) = (pc, pc', (\mathsf{Nm}, s')) & \Rightarrow \exists x.\ \mathsf{dec}(s, pc) = (\mathsf{Nm}, x) \wedge \mathsf{exec}(s, pc, (\mathsf{Nm}, x)) = (s', pc') \\ \mathsf{commit}(rob) = (pc, pc', (\mathsf{Ld}, x, a, v, s')) & \Rightarrow \mathsf{dec}(s, pc) = (\mathsf{Ld}, x, a) \wedge \mathsf{exec}(s, pc, (\mathsf{Ld}, x, v)) = (s', pc') \\ \mathsf{commit}(rob) = (pc, pc', (\mathsf{St}, a, v, s')) & \Rightarrow \mathsf{dec}(s, pc) = (\mathsf{St}, a, v) \wedge \mathsf{exec}(s, pc, (\mathsf{St})) = (s', pc') \\ \mathsf{commit}(rob) = H & \Rightarrow \mathsf{dec}(s, pc) = H \end{cases}$$

Fig. 6. Correctness of reorder buffer

When this condition holds, we may conclude the correctness theorem for out-of-order processors. We use a trace-transformation function noSpec that drops all speculative-load requests and responses (*i.e.,* those load requests and responses whose tags are not ϵ). See Definition 5 for a review of how we use such functions in framing trace refinement. Intuitively, we prove that any behavior by the speculating processor can be matched by the simple processor, with speculative messages erased.

Theorem 5. $P_{so} \sqsubseteq_{\mathsf{noSpec}} P_{ref}$

Proof. By induction on P_{so} traces, using an abstraction function that drops the speculative messages and the *rob* and *bp* states to relate the two systems. The reorder-buffer correctness condition is crucial to relate its behavior with the simple in-order execution of P_{ref}. □

Corollary 1. $P_{so}^n \sqsubseteq_{\mathsf{noSpec}^n} P_{ref}^n$

Proof. Direct consequence of Theorems 5 and 2 (the latter is about n-repetition). □

8 Cache-Based Memory System

We now turn our attention to a more efficient implementation of memory. With the cache hierarchy of Fig. 1, we have concurrent interaction of many processors with many caches, and the relationship with the original M_m system is far from direct. However, this intricate concurrent execution is crucial to hiding the latency of main-memory access. Figure 7 formalizes as an LTS M_c the algorithm we implemented (based on a published implementation [13]) for providing the memory abstraction on top of a cache hierarchy. We have what is called an *invalidating directory-based hierarchical cache-coherence protocol*.

We describe a state of the system using fields d, ch, cs, dir, w, $dirw$, ins, $outs$. The *ins* and *outs* sets are the interfaces to the processors and are exactly the same as in M_m (Fig. 2). We use $parent(c, p)$ to denote that p is the parent of c.

A coherence state is M, S, or I, broadly representing permissions to modify, read, or do nothing with an address, respectively, the decreasing permissions denoted by $M > S > I$. More precisely, if a node n is in coherence state M or S for some address, then there might be some node in n's subtree that has write or read permissions, respectively, for that address. Coherence state of cache c for address a is denoted by $cs(c, a)$. $d(c, a)$ represents the data in cache c for address a.

$w(c, a)$ stores the permission an address a in cache c is waiting for, if any. That is, cache c has decided to *upgrade* its coherence state for address a to a more permissive value, but it is waiting for acknowledgment from its parent before upgrading.

$dir(p, c, a)$ represents the parent p's notion of the coherence state of the child c for address a. We later prove that this notion is always conservative, *i.e.,* if the parent assumes that a child does not have a particular permission, then it is guaranteed in this system that the child will not have that permission. $dirw(p, c, a)$ denotes whether the parent p is waiting for any downgrade response from its child c for address a, and if so, the coherence state that the child must downgrade to as well.

There are three types of communication channels in the system: (i) $ch(p, c, \mathsf{RR})$ (which carries both downgrade request and upgrade response messages from parent p to its child c), (ii) $ch(c, p, \mathsf{Rq})$ (which carries upgrade request messages from child c to its parent p) and (iii) $ch(c, p, \mathsf{Rp})$ (which carries downgrade response messages from child c to its parent p). While the $ch(c, p, \mathsf{Rp})$ and $ch(p, c, \mathsf{RR})$ channels deliver messages between the same pair of nodes in the same order in which the messages were injected (*i.e.,* they obey the FIFO property, indicated by the use of :: in Fig. 7), $ch(c, p, \mathsf{Rq})$ need not obey such a property (indicated by the use of $\uplus$ for unordered bags in Fig. 7). This asymmetry arises because only one downgrade request can be outstanding for one parent-child pair for an address.

Here is an intuition on how the transitions work in the common case. A cache can spontaneously decide to upgrade its coherence state, in which case it sends an upgrade request to its parent. The parent then makes a local decision on whether to send a response to the requesting child or not, based on its directory

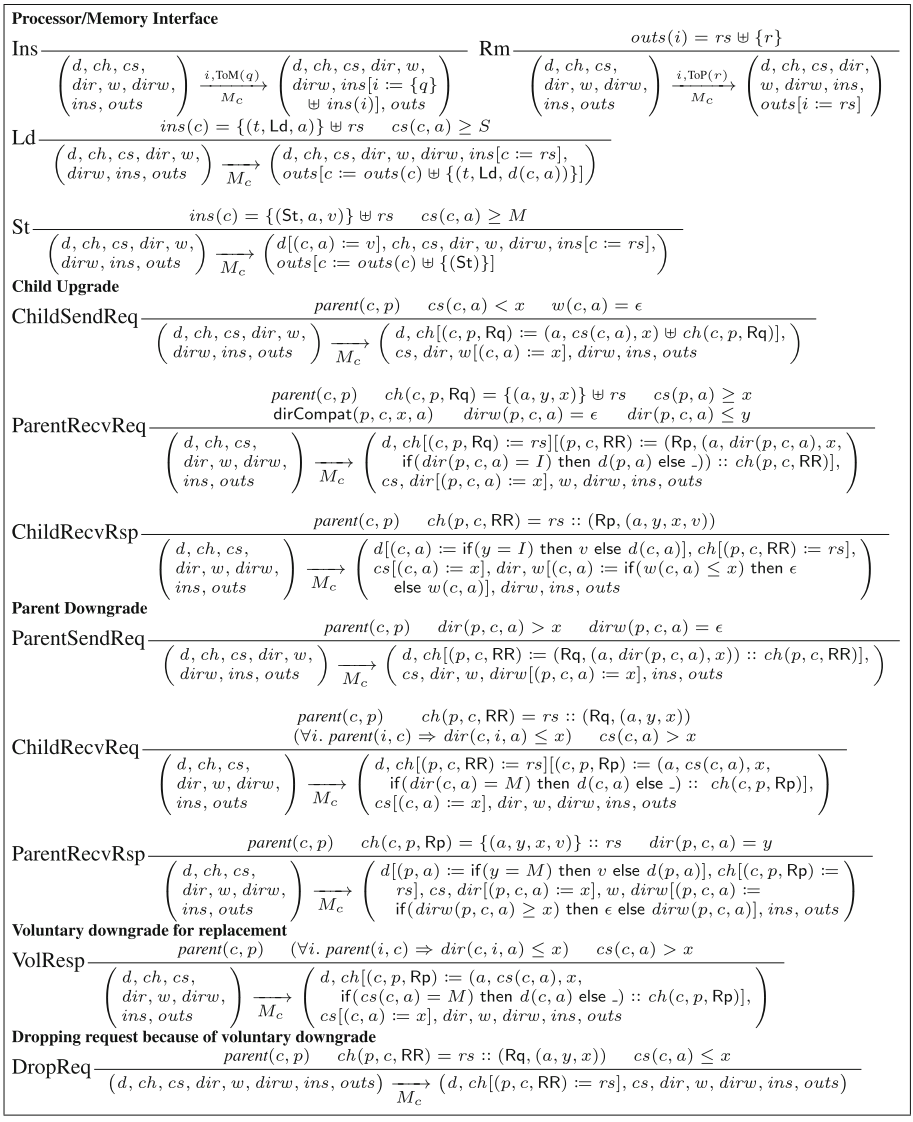

Fig. 7. LTS for cache-coherent shared-memory system

approximation and its own coherence state *cs*. If *cs* is lower than the requested upgrade, then it cannot handle the request, and instead must decide to upgrade *cs*. Once the parent's *cs* is not lower than the requested upgrade, it makes sure that the rest of its children are "compatible" with the requested upgrade (given by the dirCompat definition below). If not, the parent must send requests to the incompatible children to downgrade. Finally, when the *cs*'s upgrade and children's downgrade responses are all received, the original request can be

responded to. A request in *ins* can be processed by an L1 cache only if it is in the appropriate state, otherwise it has to request an upgrade for that address.

Definition 6. $\mathsf{dirCompat}(p,c,x,a) = \begin{cases} x = M \Rightarrow \forall c' \neq c.\ dir(p,c',a) = I \\ x = S \ \Rightarrow \forall c' \neq c.\ dir(p,c',a) \leq S \end{cases}$

A complication arises because a cache can voluntarily decide to downgrade its state. This transition is used to model invalidation of cache lines to make room for a different location. As a result, the parent's *dir* and the corresponding *cs* of the child may go out of sync, leading to the parent requesting a child to downgrade when it already has. To handle this situation, the child has to drop the downgrade request when it has already downgraded to the required state (Rule DropReq in Fig. 7), to avoid deadlocks by not dequeuing the request.

8.1 Proving M_c is Store Atomic

We must prove the following theorem, *i.e.,* the cache-based system is sound with respect to the simple memory.

Theorem 6. $M_c \sqsubseteq M_m$

We present the key theorem needed for this proof below. Throughout this section, we say *time* to denote the number of transitions that occurred before reaching the specified state.

Theorem 7. *A is store atomic, i.e.,* $A \sqsubseteq M_m$ *and* $M_m \sqsubseteq A$ *iff for any load request* $\mathsf{ToM}(t, \mathsf{Ld}, a)$ *received, the response* $\mathsf{ToP}(t, \mathsf{Ld}, v)$ *sent at time* T *is such that*

1. $v = v_0$ *(the initial value of any memory address) and no store request* $\mathsf{ToM}(\mathsf{St}, a, v')$ *has been processed at any time* T' *such that* $T' < T$ *or*
2. *There is a store request* $\mathsf{ToM}(\mathsf{St}, a, v)$ *that was processed at time* T_q *such that* $T_q < T$ *and no other store request* $\mathsf{ToM}(\mathsf{St}, a, v')$ *was processed at any time* T' *such that* $T_q < T' < T$.

The proof that M_c obeys the properties in Theorem 7 is involved enough that we state only key lemmas that we used.

Lemma 1. *At any time* T*, if address* a *in cache* c *obeys* $cs(c,a) \geq S$ *and* $\forall i.\ dir(c,i,a) \leq S$*, then* a *will have the* ***latest value****, i.e.,*

1. $d(c,a) = v_0$ *and no store request* $\mathsf{ToM}(\mathsf{St}, a, v)$ *has been processed at any time* T' *such that* $T' < T$ *or*
2. *There is a store request* $\mathsf{ToM}(\mathsf{St}, a, v)$ *that was processed at time* T_q *such that* $T_q < T \wedge d(c,a) = v$ *and no other store request* $\mathsf{ToM}(\mathsf{St}, a, v')$ *was processed at any time* T' *such that* $T_q < T' < T$.

It is relatively straightforward to prove the properties of Theorem 7, given Lemma 1. To prove Lemma 1, it has to be decomposed further into the following, each of which holds at any time.

Lemma 2. *If some response m for an address a is in transit (i.e., we are considering any time T such that $T_s \leq T \leq T_r$ where T_s is the time of sending m and T_r the time of receiving m), then no cache can process store requests for a, and m must be sent from a cache c where $cs(c, a) \geq S$ and $\forall i.\ dir(c, i, a) \leq S$.*

Lemma 3. *At any time,* $\forall p, \forall c, \forall a.\ parent(c, p) \Rightarrow$
$cs(c, a) \leq dir(p, c, a) \wedge \mathsf{dirCompat}(p, c, dir(p, c, a), a) \wedge dir(p, c, a) \leq cs(p, a)$

The same proof structure can be used to prove other invalidation-based protocols with inclusive caches (where any address present in a cache will also be present in its parent) like MESI, MOSI, and MOESI; we omit the discussion of extending this proof to these for space reasons. The MSI proof is about 12,000 lines of Coq code, of which 80 % can be reused as-is for the other protocols.

9 The Final Result

With our two main results about optimized processors and memories, we can complete the correctness proof of the composed optimized system.

First, we need to know that, whenever the simple memory can generate some trace of messages, it could also generate the same trace with all speculative messages removed. We need this property to justify the introduction of speculation, during our final series of refinements from the optimized system to SC.

Theorem 8. $M_m \sqsubseteq_{\mathsf{noSpec}^n} M_m$

Proof. By induction on traces, with an identity abstraction function. □

That theorem turns out to be the crucial ingredient to justify placing a speculative processor in-context with simple memory.

Theorem 9. $P^n_{so} + M_m \sqsubseteq P^n_{ref} + M_m$

Proof. Follows from Theorem 3 (our result about $+$), Corollary 1, and Theorem 8. □

The last theorem kept the memory the same while refining the processor. The next one does the opposite, switching out memory.

Theorem 10. $P^n_{so} + M_c \sqsubseteq P^n_{so} + M_m$

Proof. Follows from Theorems 6 and 3 plus reflexivity of $\sqsubseteq$ (Theorem 1). □

Theorem 11. $P^n_{so} + M_c \sqsubseteq SC$

Proof. We twice apply $\sqsubseteq$ transitivity (Theorem 1) to connect Theorems 10, 9, and 4 □

10 Conclusions and Future Work

In this paper, we developed a mechanized modular proof of a parametric hierarchical cache-coherence protocol in Coq and use this proof modularly for a verification of sequential consistency for a complete system containing out-of-order processors. Our proof modularization corresponds naturally to the modularization seen in hardware implementations, allowing verification to be carried out in tandem with the design. Our overall goal is to enable design of formally verified hardware systems. To this end, we have been working on a DSL in Coq for translating to and from Bluespec, and we are developing appropriate libraries and proof automation, extending the work of Braibant *et al.* [7] with support for modular specification and verification, systematizing some elements of this paper's Coq development that are specialized to our particular proof.

While we provide a clean interface for an SC system, we are also working on encompassing relaxed memory models commonly used in modern processors.

Acknowledgments. This work was supported in part by NSF grant CCF-1253229 and in part by the Defense Advanced Research Projects Agency (DARPA) and the United States Air Force, under Contract No. FA8750-11-C-0249. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of Defense or the U.S. Government.

References

1. Arvind, Nikhil, R.S., Rosenband, D.L., Dave, N.: High-level synthesis: an essential ingredient for designing complex ASICs. In: Proceedings of ICCAD 2004, San Jose, CA (2004)
2. Arvind, Shen, X.: Using term rewriting systems to design and verify processors. Micro, IEEE **19**(3), 36–46 (1999)
3. Augustsson, L., Schwarz, J., Nikhil, R.S.: Bluespec Language definition, Sandburst Corp (2001)
4. Bhattacharya, R., German, S.M., Gopalakrishnan, G.C.: Symbolic partial order reduction for rule based transition systems. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 332–335. Springer, Heidelberg (2005)
5. Bhattacharya, R., German, S.M., Gopalakrishnan, G.C.: Exploiting symmetry and transactions for partial order reduction of rule based specifications. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 252–270. Springer, Heidelberg (2006)
6. Bluespec Inc, Waltham, M.A.: Bluespec SystemVerilog Version 3.8 Reference Guide, November 2004
7. Braibant, T., Chlipala, A.: Formal verification of hardware synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 213–228. Springer, Heidelberg (2013)
8. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) Computer Aided Verification. LNCS, pp. 68–80. Springer, Heidelberg (1994)

9. Cain, H.W., Lipasti, M.H.: Memory ordering: a value-based approach. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004, pp. 90–101, June 2004
10. Chang, C., Wawrzynek, J., Brodersen, R.W.: Bee2: a high-end reconfigurable computing system. Des. Test Comput. IEEE **22**(2), 114–125 (2005)
11. Xiaofang Chen, Y., Yang, G.G., Chou, C.-T.: Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. Form. Methods Syst. Des. **36**(1), 37–64 (2010)
12. Chou, C.-T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Formal Methods in Computer Aided Design, pp. 382–398. Springer (2004)
13. Dave, N., Ng, M.C., Arvind.: Automatic synthesis of cache-coherence protocol processors using bluespec. In: Proceedings of Formal Methods and Models for Codesign, MEMOCODE, Verona, Italy (2005)
14. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
15. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proceedings of the IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors, ICCD 1992, pp. 522–525, October 1992
16. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
17. Hoe, J.C., Arvind.: Synthesis of operation-centric hardware descriptions. In: Proceedings of ICCAD 2000, pp. 511–518, San Jose, CA (2000)
18. Hoe, J.C., Arvind.: Operation-centric hardware description and synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **23**(9), 1277–1288 (2004)
19. Norris Ip, C., Dill, D.L., Mitchell, J.C.: State reduction methods for automatic formal verification (1996)
20. Jhala, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 396–410. Springer, Heidelberg (2001)
21. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M.R., Yuan, Y.: Checking cache-coherence protocols with TLA$^+$. Formal Methods Syst. Des. **22**(2), 125–131 (2003)
22. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., et al.: Replacing testing with formal verification in Intel® Core^{tm} i7 processor execution engine validation. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)
23. Khan, A., Vijayaraghavan, M., Boyd-Wickizer, S., Arvind: Fast and cycle-accurate modeling of a multicore processor. In: 2012 IEEE International Symposium on Performance Analysis of Systems & Software, pp. 178–187, New Brunswick, NJ, USA, April 1–3, 2012
24. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M.A., Gupta, A.M., Rosenblum, M., Hennessy, J.: The stanford FLASH multiprocessor. In: Proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 302–313, April 1994

25. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **100**(9), 690–691 (1979)
26. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
27. Manolios, P., Srinivasan, S.K.: Automatic verification of safety and liveness for pipelined machines using WEB refinement. ACM Trans. Des. Autom. Electron. Syst. 45:1–45:19 (2008)
28. McMillan, K.L.: Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 179–195. Springer, Heidelberg (2001)
29. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, pp. 110–121. Springer, Heidelberg (1998)
30. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999)
31. McMillan, K.L., Schwalbe, J.: Formal verification of the Gigamax cache consistency protocol. In: Proceedings of the International Symposium on Shared Memory Multiprocessing, pp. 111–134 (1992)
32. Moore, J.S.: An ACL2 proof of write invalidate cache coherence. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification. LNCS, vol. 1427, pp. 29–38. Springer, Heidelberg (1998)
33. Park, S., Dill, D.L.: Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 288–296. ACM Press (1996)
34. Shen, X., Arvind, Rudolph, L.: Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In: Proceedings of the 26th annual international symposium on Computer architecture, pp. 150–161. IEEE Computer Society (1999)
35. Talupur, M., Tuttle, M.R.: Going with the flow: parameterized verification using message flows. In: Formal Methods in Computer-Aided Design, FMCAD 2008, pp. 1–8, November 2008
36. Windley, P.J.: Formal modeling and verification of microprocessors. IEEE Trans. Comput. **44**(1), 54–72 (1995)
37. Zhang, M., Bingham, J.D., Erickson, J., Sorin, D.J.: Pvcoherence: designing flat coherence protocols for scalable verification. In: 20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, pp. 392–403. IEEE Computer Society, Orlando, FL, USA, February 15–19 (2014)
38. Zhang, M., Lebeck, A.R., Sorin. D.J.: Fractal coherence: scalably verifiable cache coherence. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pp. 471–482. IEEE Computer Society, Washington, DC, USA (2010)