PVCoherence: Designing Flat Coherence Protocols for Scalable Verification

The goal of this work is to design cache coherence protocols with many cores such that they can be verified with existing verification methodologies. In particular, the authors focus on flat (nonhierarchical) coherence protocols using a mostly automated methodology based on parametric verification. They present design guidelines that, if followed by architects, enable parametric verification of protocols with arbitrary numbers of cores.

Meng Zhang
Duke University

Jesse D. Bingham
John Erickson
Intel

Daniel J. Sorin
Duke University

Verification is one of the greatest challenges in the development of complicated systems. A company incurs significant risk in shipping a product that is not fully verified, because a latent bug can lead to an expensive recall and damage to the company's reputation. Even finding a bug before shipping a chip, if late in the product development cycle, can lead to a costly redesign of the chip and product delay.

Because of these risks, the computing industry invests more time and money on verification than on design and, despite all of this effort, companies still cannot completely verify their chips. Current chip designs are just too complicated to completely verify. The current state of the art is to use a combination of extensive simulation (to hunt for bugs) and some formal techniques to verify small or abstracted aspects of the design (for example, an abstract three-core version of the coherence protocol). Although this verification is effective in finding most bugs, it is time consuming and costly and, most importantly, incomplete.

We hypothesize that the key to solving the verification problem is through verification-aware architecture. We must design systems such that they can be formally verified with existing verification methodologies, rather than expecting arbitrary designs to be verifiable. Architecting an entire multicore processor to be verifiable is not yet feasible, so we have started with subsystems that are notoriously complicated and need to be scalable. In this article, we focus on cache coherence protocols, because it is difficult to verify that a protocol functions correctly in all possible situations, particularly for a system with numerous caches. We seek to architect arbitrarily large flat (nonhierarchical) protocols such that they can be verified using a mostly automated methodology. These flat protocols can be used either on their own or as building blocks in inductively verified hierarchical protocols.[1,2]

Published by the IEEE Computer Society

To achieve our goal, we use a previously developed technique called parametric verification (PV). The key idea here is to treat the number of nodes—in this work on coherence protocols, a node is a core plus its private cache(s)—as a parameter instead of as a concrete number. Parametric verification can prove that certain properties are true for the system regardless of the parameter's value.

Prior work on PV has focused on how to apply it to given protocols. These papers are formal and tailored toward verification experts who must verify protocols given to them. Our research differs in that we explore how to design protocols such that they can be verified with PV. Although it is true that verification experts often have intuition into which protocol features play well with verification techniques, we seek to provide architects with a practical, nonmathematical set of guidelines for designing protocols.

We provide a set of guidelines for designing protocols that can be parametrically verified with PV, and we refer to any protocol that obeys all of the proposed guidelines as a PVCoherence protocol. We show that the performance of a PVCoherence protocol does not necessarily incur significant performance overheads compared to typical protocols that are not verifiable with PV.

PV background

This explanation of PV is relevant to architects who want to design protocols that can be verified with this method.

Model checking

Model checking is a method to mathematically verify a finite-state concurrent system. The user provides a model (a description of a system) and a specification (the properties to which the system should adhere). Then a model-checking tool automatically and exhaustively searches the reachable state space and checks whether this model meets the given specification. If not, the tool provides a counterexample, which is a sequence of states that leads to a violation of the specification. Model checking is limited to small systems (typically two to five nodes) because of its use of exhaustive search.

There are various kinds of model-checking tools. Murphi is widely used, especially for the verification of cache coherence protocols.[3] The user models the cache coherence protocol in an expressive language and specifies the invariants. For the verification of cache coherence protocols, there are two primary invariants. In the *permissions invariant,* the protocol must enforce the single-writer, multiple-reader (SWMR) invariant: for each block of memory, at any given time, the block either has a single writer or zero or more readers.[4] In the *data invariant,* a read of a block must return the value of the most recent write to that block.

Simple-PV methodology

Chou et al. propose a simple method[5] based on McMillan's work,[6] which we call Simple-PV, to parametrically verify cache coherence protocols. The main advantage of Simple-PV, compared to other PV methods, is that it leverages automated tools where possible to minimize the required manual effort, and it is practical for realistic designs.

Consider a system with an arbitrary number of nodes, N. We illustrate the Simple-PV process for verifying this system's coherence protocol in Figure 1 and discuss each step. We start with a nonparametric model, as in a typical nonparametric verification.

Step 1: Automatically create parametric model. The first step is to create a parametric model from the nonparametric model. Consider the system with N nodes in Figure 2. Starting with two concrete nodes in the nonparametric model, we abstract the other $N-2$ nodes into a single "Other" node that we refer to as OtherNode. (The number of concrete nodes to instantiate depends on the protocol.[7]) OtherNode represents the behaviors of all $N-2$ nodes, and we must ensure that the parametric model permits all possible behaviors that the concrete nodes can do, as well as the actions those abstracted nodes can do to them.

We perform this abstraction with a fully automated tool called Abster that was developed at Intel.[8] Given a nonparametric Murphi model, Abster produces a parametric Murphi model by generating the behavior for OtherNode. The key point of abstraction is that it must preserve all the behaviors of OtherNode that could occur. Thus, Abster conservatively overapproximates the behavior

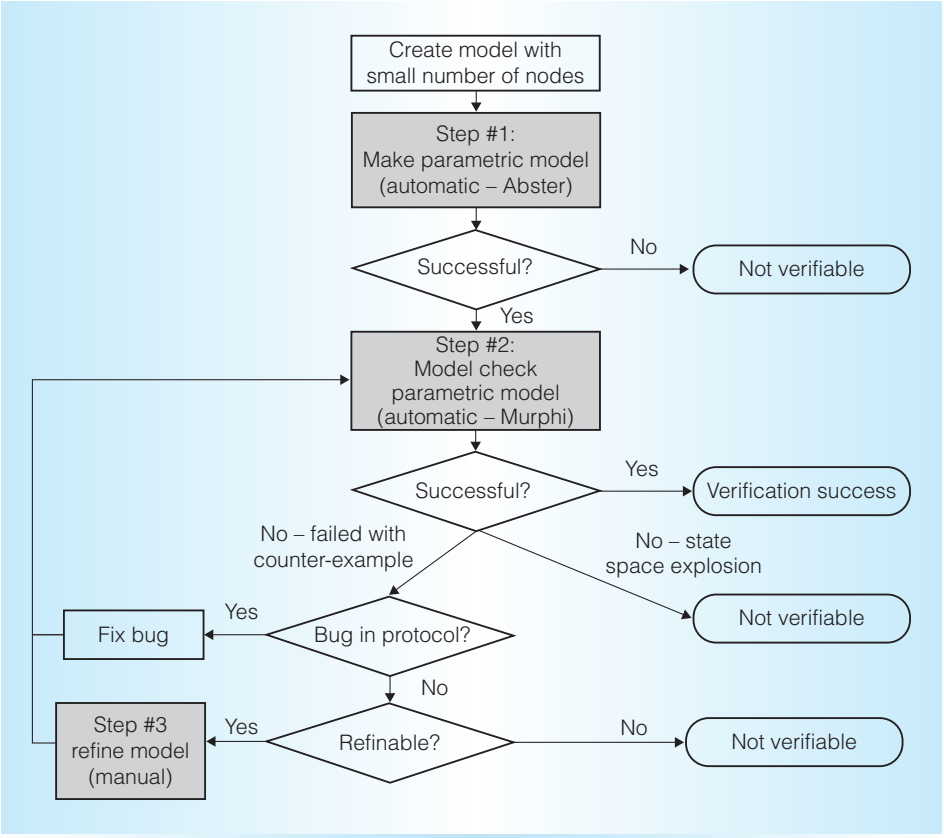

Figure 1. Simple-PV verification process. Starting with a nonparametric model, we automatically create a parametric model and then model check it. If model checking fails, we try to refine the model before model checking it again.

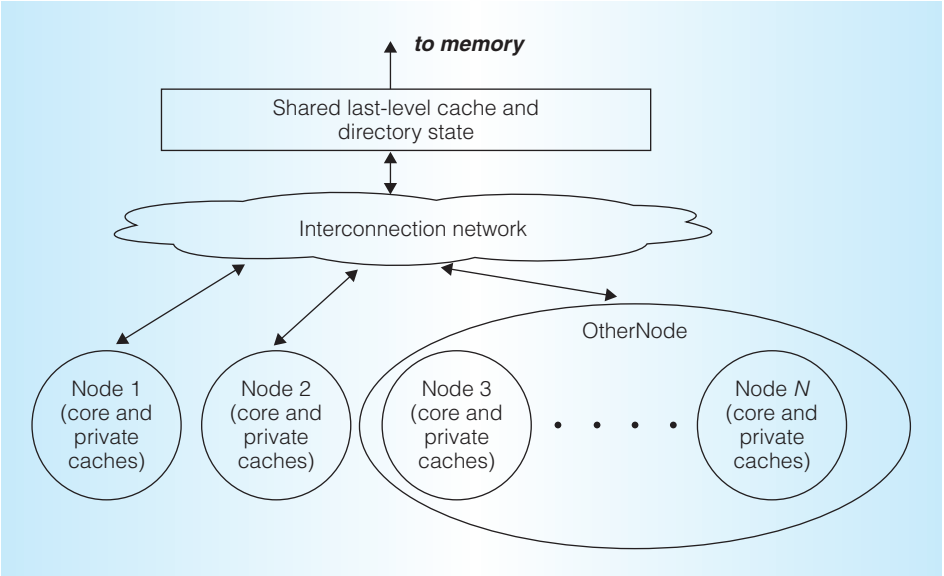

Figure 2. Parametric model. After abstraction, there are two concrete nodes and a single "OtherNode" that represents a superset of the behaviors of all $N-2$ other nodes.

of the $N - 2$ nodes that it abstracts; that is, the automatically generated OtherNode is likely to exhibit behaviors that would not be possible had we instead instantiated $N - 2$ concrete nodes. This overapproximation leads to a challenge that we address in Step 3.

Step 2: Automatically model-check the model. We use Murphi to automatically model-check that the parametric model satisfies the two coherence invariants. If Murphi succeeds, the protocol is coherent and we are done. If Murphi fails, there are four possible scenarios:

- There are real bugs in the cache coherence protocol design. In this case, we must debug the protocol and then return to Step 2.
- The parametric model's state space exceeds Murphi's capacity. Even with parameterization, some systems are too large for Murphi. In this case, we must redesign the protocol such that the abstracted parametric protocol "fits" in Murphi.
- The overapproximation in Step 1 enables OtherNode to behave in a way that causes spurious violations of the coherence invariants. In this case, we proceed to Step 3 (to fix OtherNode) and then return to Step 2.
- The protocol is incompatible with Simple-PV. In this case, no amount of fixing OtherNode will lead to a protocol that can be verified with Murphi.

If Murphi fails in Step 2 because of overapproximation, then we proceed to Step 3.

Step 3: Manually refine the model. Because Abster overapproximates when it abstracts the $N - 2$ nodes into OtherNode, it is possible that, in Step 2, Murphi discovers spurious violations of invariants. When this happens, the verifier must manually intervene and refine the parametric model by modifying OtherNode. Based on the counterexample provided by Murphi, the verifier modifies OtherNode to restrict its behavior.

Restricting OtherNode's behavior may seem to introduce the possibility of "defining away the problem." If we arbitrarily remove behaviors from OtherNode, we could fool ourselves into a false verification in which we remove possible behaviors that lead to genuine violations of the coherence invariants.

The key to refinement is to both constrain OtherNode's behavior and also check that these constraints are valid. Thus, for each constraint we add to OtherNode's behavior, we add an invariant that Murphi checks, and this invariant is that the constraint is justified (that is, true for a nonabstracted model). Furthermore, this added invariant is checked on the concrete nodes. In the PV literature, such an invariant is called a *lemma,* and we adopt this terminology here.

Steps 2 and 3 represent an iterative process of identifying spurious violations in Murphi and refining the model accordingly. The process ends when either

- Murphi successfully verifies the parametric model, in which case we know the protocol is correct for any arbitrary number of nodes, or
- the iterative refinement process does not eventually result in a model that Murphi can verify, in which case we consider the protocol to be incompatible with Simple-PV.

Although Step 3 involves manual effort, prior work (on simple protocols[5]) and our work here indicate that the process is straightforward and tends to involve only a few iterations.

System model

We design coherence protocols based on the multicore system architecture. Current multicore processors usually employ a multilevel cache hierarchy, in which each core has one or more private caches and all cores share a last-level cache.

We assume a directory-like coherence protocol, wherein the directory state is collocated with the last-level cache. We make no assumptions about the interconnection network except regarding virtual channels. These virtual channels may or may not be ordered, depending on the protocol; later we will discuss the impact of ordering on verification.

Coherence protocol design guidelines

We seek to design cache coherence protocols that can be parametrically verified with

Simple-PV. We now present design guidelines in order, from the most to the least intuitive.

Guideline 1: All nodes must be identical.

If we have a variety of node types, then we must have multiple "flavors" of OtherNode, one for each variety of node. This complicates the PV abstraction and refinement process, and it also makes state space explosion much more likely. Abster, for example, does not support abstraction of heterogeneous nodes.

Guideline 2: The protocol cannot use any variable that depends on the number of nodes.

With Simple-PV, the number of nodes is a parameter rather than a concrete number. We cannot perform any math function, such as addition or comparison, on the parameter. Therefore, the protocol cannot use any variable that depends on the parameter's value.

This guideline most directly affects coherence protocol design by prohibiting the use of counters (that count the number of nodes). Directory protocols often use counters to aid in collecting acknowledgments.

To avoid using counters that track the number of sharers of a block, we use a bit vector to denote the sharer set. This constraint leads to two potential overheads. One overhead is storage, because a bit vector consumes more storage than a counter. The other overhead is network traffic, because a message containing a sharer set is larger than an equivalent message containing a sharer counter.

Guideline 3: We cannot have ordering over a list or queue whose size depends on the number of nodes.

Ordering of nodes implies that nodes are being treated nonidentically. If we need to maintain ordering, we must explicitly represent each node, which precludes representing all $N - 2$ abstracted nodes with an OtherNode.

Adhering to this guideline prohibits us from designing protocols in which we enforce point-to-point ordering for a virtual channel that has a queue depth that depends on the number of nodes. This guideline constrains certain design options. For example, consider a system in which all the L1 caches share a queue of requests to the L2. This queue's depth is proportional to the number of nodes. Such a queue is compatible with Simple-PV only if it is unordered.

As another example, consider a protocol that relies on point-to-point ordering of forwarded coherence requests from the directory to each L1. Many protocols rely on this ordering to avoid races involving writebacks. However, ordering this queue is not compatible with Simple-PV if the number of forwarded messages that can be in this queue is a function of the number of nodes. Unfortunately, this situation is possible. For example, if Core C1 has requested read-write permissions (with what we refer to as a GetModified or GetM message) for block B, the directory could forward subsequent read-only requests (GetShared or GetS messages) for B to C1 from every other core before C1 receives the data for B and can start responding to the GetS messages that have filled its queue. Most protocols do not rely on ordering of the forwarded GetS messages in this example; nevertheless, the *possibility* of having a number of messages in the queue that depends on the number of nodes precludes ordering *any messages* in this queue.

Guideline 4: We should not parameterize buffers or queues in more than one dimension.

In our protocol models (and in all model-checking work we've seen), arrays represent channels, and messages are entries in these arrays. For example, we specify the buffer of requests from Core C1's L1 cache to the L2 cache as `buffer_L1_C1_to_L2[SIZE]`, where `SIZE` is the number of entries in the buffer. It is common to designate `SIZE` as a concrete value (for example, 4) or a variable that is equal to the number of cores. The latter situation can occur, for example, in a queue of forwarded requests from the directory to a given L1 cache; as we explained earlier, such a queue could hold forwarded GetS requests from all other cores. Although the buffer depth in this example is a function of the parameterized number of nodes, the protocol is still compatible with Simple-PV, because the array is parameterized in only one dimension.

The problem for Simple-PV appears only when specifying an array that is parameterized

in more than one dimension. Consequently, this constraint precludes us from letting a core that issues a GetM collect all the acknowledgment messages from cores that were invalidated by the GetM. In this scenario, Core C1 issues a GetM to the L2, which sends an invalidation to all cores with shared copies of the block. In most protocols, the invalidated cores send acknowledgments to C1. However, that implies that we have buffers from each core to each other core. Because the number of nodes is parameterized, we thus have a 2D parameterization with a structure like `AcknowledgmentBuffers[N][N]`.

Therefore, to follow Guideline 4, a protocol must collect acknowledgments at the L2 instead of at the requesting L1. The L2 then sends a single, aggregated acknowledgment to the requesting L1. This design option is less efficient than having the requesting L1 collect the acknowledgments, because it requires an extra message.

Design of a PVCoherence protocol

The common feature of all PVCoherence protocols is that they can be formally verified using Simple-PV. Although all PVCoherence protocols obey our design guidelines, there still can be considerable variation between different PVCoherence protocols. Here, we show the design process of one PVCoherence protocol, called PV-MOESI, which is based on the system architecture described earlier.

To highlight the ramifications of designing a protocol to be compatible with Simple-PV, we compare and contrast the design of PV-MOESI with a protocol we call OP-MOESI. OP-MOESI is similar to typical multicore protocols; it provides high performance but cannot be verified using Simple-PV.

Optimized baseline protocol: OP-MOESI

The OP-MOESI coherence protocol is a typical directory protocol similar to other prevalent protocols.[9,10] OP-MOESI has five stable L1 cache coherence states (MOESI stands for modified owner exclusive shared invalid) and more than 30 transient states to improve performance. The directory state, which is colocated with the L2 tag and state, includes a full-map bit vector that denotes which L1 caches are currently caching each block. A complete specification of OP-MOESI is provided elsewhere.[7]

The protocol relies on having three virtual channels in the system for requests, forwarded requests, and responses. These channels enforce point-to-point ordering for OP-MOESI.

OP-MOESI is highly optimized, but we cannot verify it with Simple-PV. Abster fails to generate an abstracted model for OP-MOESI, so we cannot run it through Murphi.

PV-MOESI

To create PV-MOESI, we modify OP-MOESI to satisfy our guidelines (a complete specification of PV-MOESI is available elsewhere[7]):

- For GetM transactions, we remove the counter in the response message from the L2 to the requesting L1. We replace it with a sharer set that is, unfortunately, larger than the counter (C bits compared to $\log_2 C$ bits).
- For GetM transactions, we have the L2 (instead of the L1 requestor) collect the acknowledgments from the invalidated L1 caches. After collecting all L1 acknowledgments, the L2 sends a single acknowledgment to the L1 requestor. This modification adds one more network hop per GetM.
- We remove the point-to-point ordering in all virtual channels. This is the most significant change in the protocol because it leads to more races. PV-MOESI handles these races in the usual fashion—with extra messages and extra transient states—but without blocking. These races are not unique to PV-MOESI but rather a well-known issue for protocols that cannot rely on point-to-point ordering.

After we've made these modifications, we can abstract the model using Abster. However, we still cannot verify the abstracted model using Murphi, regardless of how we try to refine it. This problem (discussed in detail elsewhere[7]) arises from multiple in-flight GetM requests; we handle it by modifying how the protocol handles GetM requests. When the L2 receives a GetM, it

Table 1. Simulation configurations.

Processor core parameters	
Cores	32 in-order x86 cores
Clock frequency	2 GHz
Cache and memory parameters	
Cache line size	64 bytes
L1 instruction and data caches	32-Kbyte, 2-way, 2-cycle hit
L2 cache	Inclusive with respect to L1s; 8 Mbytes split into 16 banks; each bank has a 512-Kbyte, 8-way, 12-cycle hit
Memory	2-Gbyte, 160-cycle hit
Interconnection network parameters	
Topology	2D mesh
Link bandwidth	32 Gbytes per second
Link latency	1 cycle

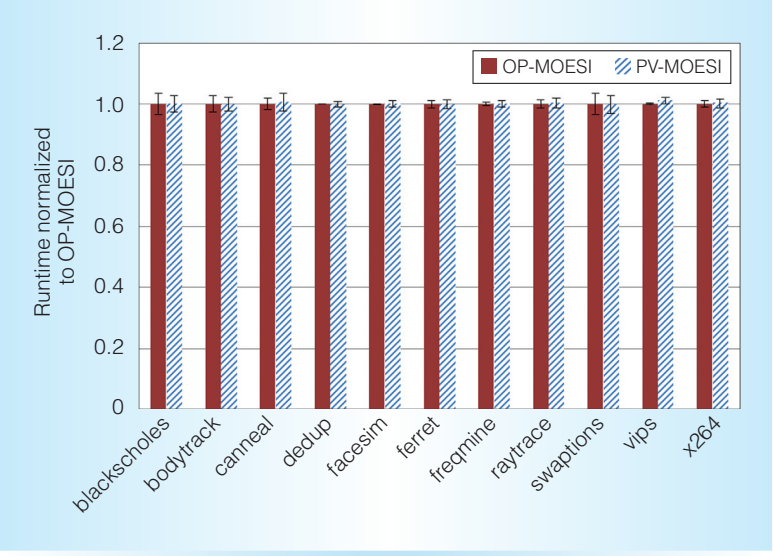

Figure 3. Runtime comparison: OP-MOESI versus PV-MOESI. Runtimes are normalized to the runtime of OP-MOESI for 32-core systems.

forwards the GetM and/or any invalidations (as in OP-MOESI) but then blocks subsequent requests until it receives a Completion message from the L1 that requested the GetM. The L1 sends the Completion once it has received data or the acknowledgment from the L2. This protocol modification potentially impacts performance due to blocking at the L2.

We formally verify PV-MOESI with Simple-PV. We find that we need to manually add only seven lemmas during refinement to enable verification. Adding these lemmas is not trivial, but neither is it terribly complicated. All other verification work is automatic with Abster and Murphi. The Murphi model checking finished in less than one hour and used several gigabytes of memory.

Performance evaluation

We experimentally evaluated the performance of OP-MOESI and PV-MOESI using the gem5 full-system simulator[11] and benchmarks from the Parsec benchmark suite.[12] Table 1 presents the system parameters. We ran each experiment multiple times to accommodate the natural variability in simulation runtimes; error bars in graphs indicate plus or minus one standard deviation from the mean.

The performance of the verifiable PV-MOESI could potentially be less than that of the unverifiable OP-MOESI. In Figure 3, we plot the runtimes for both OP-MOESI and PV-MOESI, normalized to the runtime of OP-MOESI, for 32-core systems. Although there are some differences in the runtimes, they are "within the noise."

Although this article is not the first to address the issue of verification-aware architecture, it presents the first framework for scalably verifiable flat coherence protocols, and it evaluates a concrete instantiation of this framework. Prior work such as Fractal Coherence[1] and MCP,[2] although exciting and novel, developed fairly unusual protocols that are useful as proofs of concept but unlikely to be adopted in practice. Given our positive results for a protocol that resembles existing protocols, we hope that we can change the way in which architects think about verifiability when designing protocols.

More generally, we hope that our work inspires other researchers and industrial product teams to adopt verification-aware design techniques, such as PVCoherence. Adoption of verification-aware architecture would radically change the way that design and verification are done, and it could lead to provably correct chips, or at least significant portions of chips.

MICRO

Acknowledgments

This material is based on work supported by the NSF under grant CCF-142-1167.

References

1. M. Zhang, A.R. Lebeck, and D.J. Sorin, "Fractal Coherence: Scalably Verifiable Cache Coherence," *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2010, pp. 471–482.
2. J.G. Beu et al., "High-Speed Formal Verification of Heterogeneous Coherence Hierarchies," *Proc. 19th IEEE Int'l Symp. High Performance Computer Architecture* (HPCA 13), 2013, pp. 566–577.
3. D.L. Dill et al., "Protocol Verification as a Hardware Design Aid," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 92), 1992, pp. 522–525.
4. D.J. Sorin, M.D. Hill, and D.A. Wood, *A Primer on Memory Consistency and Cache Coherence*, Morgan & Claypool Publishers, 2011.
5. C.-T. Chou, P. Mannava, and S. Park, "A Simple Method for Parameterized Verification of Cache Coherence Protocols," *Formal Methods in Computer-Aided Design*, vol. 3312, A. Hu and A. Martin, eds., Springer, 2004, pp. 382–398.
6. K.L. McMillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," *CHARME 01: IFIP Working Conf. Correct Hardware Design and Verification Methods*, LNCS 2144, 2001, pp. 179–195.
7. M. Zhang et al., "PVCoherence: Designing Flat Coherence Protocols for Scalable Verification," *Proc. 20th Int'l Symp. High Performance Computer Architecture* (HPCA 14), 2014, pp. 392–403.
8. M. Talupur and M.R. Tuttle, "Going with the Flow: Parameterized Verification Using Message Flows," *Proc. Int'l Conf. Formal Methods in Computer-Aided Design* (FMCAD 08), 2008, pp. 10:1–10:8.
9. P. Conway et al., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, 2010, pp. 16–29.
10. R. Singhal, "Inside Intel Next Generation Nehalem Microarchitecture," Hot Chips, vol. 20, 2008.
11. N. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011, pp. 1–7.
12. C. Bienia et al., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 08), 2008, pp. 72–81.

Meng Zhang is a software engineer at Oracle, where she's working on storage performance. Her research interests include memory system design and verification. Zhang has a PhD in computer architecture from Duke University, where she completed the work for this article. Contact her at zhangmeng916@gmail.com.

Jesse D. Bingham is a formal verification engineer at Intel, where he has applied, developed, supported, and researched formal hardware verification techniques. His research interests include arithmetic formal verification, protocol verification, theorem proving, and computer architecture. Bingham has a PhD in computer science from the University of British Columbia. Contact him at jesse.d.bingham@intel.com.

John Erickson is a formal verification engineer at Intel, where he applies formal validation to many-integrated-core processor designs. His research interests include formal verification, hardware verification, and automated theorem proving. Erickson has a PhD in computer science, with a focus on automated theorem proving, from the University of Texas at Austin. Contact him at john.erickson @intel.com.

Daniel J. Sorin is the W.H. Gardner Jr. Associate Professor of Electrical and Computer Engineering at Duke University. His research interests include computer architecture, reliability, and verification. Sorin has a PhD in electrical and computer engineering from the University of Wisconsin–Madison. Contact him at sorin@ee.duke.edu.