Appears in the 20th International Symposium on High Performance Computer Architecture (HPCA)
Orlando, Florida, February 17-19, 2014

PVCoherence: Designing Flat Coherence Protocols for Scalable Verification

Meng Zhang[1], Jesse D. Bingham[2], John Erickson[2], and Daniel J. Sorin[1]

[1]Department of ECE
Duke University

[2]Intel Corporation

Abstract

The goal of this work is to design cache coherence protocols with many cores that can be verified with state-of-the-art automated verification methodologies. In particular, we focus on flat (non-hierarchical) coherence protocols, and we use a mostly-automated methodology based on parametric verification (PV). We propose several design guidelines that architects should follow if they want to design protocols that can be parametrically verified. We experimentally evaluate performance, storage overhead, and scalability of a protocol verified with PV compared to a highly optimized protocol that cannot be verified with PV.

1. Introduction

There is a tension in cache coherence protocol design between performance and verification difficulty. Efforts to improve coherence protocol performance often lead to increasing amounts of parallelism and coherence state that must be maintained. However, increasing the amount of parallelism and coherence state makes the verification of protocols more difficult. Typically, performance is the primary goal, and the verification team attempts to verify the protocol as best as possible. Complete formal verification of typical high-performance protocols requires more time than is practical, and thus industry relies on some combination of incomplete formal verification (e.g., verifying a protocol with far fewer caches than in the real system) and incomplete testing of the protocol in a simulator.

In this work, our ultimate goal is to design cache coherence protocols such that they can be formally verified. There are two primary techniques for formal verification: model checking and theorem proving. Model checking uses fully automated tools to model the system and then check whether the coherence invariants hold throughout the entire reachable state space. With theorem proving, the verification team models a system mathematically and manually guides a tool towards a proof that the system is correct. Both formal techniques are considered complete methods as they mathematically prove the correctness of a design. However, neither technique is capable of verifying today's high-performance cache coherence protocols. Model checking has the infamous "state explosion" problem and does not scale as we increase the number of cores. Theorem proving requires an impractical amount of manual effort to verify non-trivial systems. Thus we still need a way to verify large scale systems.

Some recent work has indeed proposed coherence protocol designs that can be verified for arbitrary numbers of cores [31][4][30]. The key insight in these papers is that hierarchical designs can be inductively verified for arbitrarily sized systems—*but only assuming that the base case or the building blocks can themselves be verified*. To satisfy this assumption, TreeFractal [31] is limited to a binary tree organization that incurs significantly more latency and storage overhead for directory controllers than if one could verify a base case with a higher degree tree. Similarly, MCP [4] assumes that the building blocks it composes together can be verified, which is true only for small building blocks.

We seek to architect arbitrarily large flat (non-hierarchical) 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 [31][4]. To achieve this goal, we use a previously developed technique called ***parametric verification***. The key idea of parametric verification 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 value of the parameter*.

There have been several proposals for how to perform parametric verification (PV), and in this work we focus on a method that is highly automated. We believe that automation is critical for usability by non-experts. We use a method developed by Chou et al. [8] and McMillan [22] that makes heavy use of automated tools plus a relatively small amount of manual intervention. We discuss this method, which we refer to as Simple-PV, [1] in Section 2.

PV, particularly Simple-PV, has been previously used to verify specific cache coherence protocols [21][24], and prior work has focused on *how to apply PV* to given protocols. These papers are formal and tailored towards verification experts who must verify protocols given to them. Our focus in this paper differs in that we explore *how to design*

[1] The method is called CoMpositional Parameterized verification or CMP in the verification literature, but the CMP acronym is overloaded in the architecture literature.

protocols such that they can be verified with PV. While it is true that *verification experts* often have intuition into which protocol features are verifiable, we seek to provide *architects* with a practical, non-mathematical set of guidelines for designing protocols.

In this paper, we have two goals: 1) clearly lay out the guidelines that the protocol must follow, in terms that do not require expertise in formal verification and 2) show that conforming to these guidelines does not have a large performance impact.

To achieve the first goal, we explore the design space of cache coherence protocols that can be parametrically verified with Simple-PV. We seek to categorize the design options of a cache coherence protocol as either "can be verified" or "cannot be verified," so that architects know what design options they can use. We do not claim that these design options include all possible features of a cache coherence protocol, but we address many of the most important design choices. We study the fundamental limitations of PV, in general, as well as the limitations introduced by Simple-PV.

Based on these observations, we extract several design guidelines for making a cache coherence protocol amenable to PV. Today's protocols already adhere to some of these guidelines (although not necessarily because the architects were considering PV), but other guidelines are not typically met. Regardless of today's protocols, we hope to make the designers of future protocols aware of these guidelines.

We refer to any protocol that obeys all of the proposed guidelines as a ***PVCoherence protocol***. It is possible that a PVCoherence protocol, due to limitations imposed by following our guidelines, may perform worse or require more storage overhead than a traditional optimized protocol. To evaluate this possibility and thus achieve our second goal, we construct a typical high performance protocol, called OP-MOESI, that cannot be parametrically verified. Then we create a new protocol called PV-MOESI by modifying OP-MOESI to adhere to the design guidelines we propose for PVCoherence. We verify PV-MOESI using Simple-PV.

We experimentally examine the differences between OP-MOESI and PV-MOESI in performance, network traffic, scalability, and storage overhead, and we find that they are comparable. Even though the evaluation is performed on only a single PVCoherence protocol, we believe the conclusions extend to other PVCoherence protocols, because none of the changes required to convert a typical protocol to a PVCoherence protocol is likely to have a large impact on performance, network traffic, or storage costs.

We make the following contributions.

- We present a set of design guidelines for cache coherence protocols. Following these guidelines, architects can design cache coherence protocols that can be parametrically verified.
- We describe the design process of a PVCoherence protocol, called PV-MOESI, that can be verified with model checking tools and limited human intervention for any arbitrary number of nodes.
- We experimentally compare the performance, network traffic, scalability, and storage overhead of PV-MOESI and a highly optimized protocol, OP-MOESI, that cannot be verified with Simple-PV.

The rest of the paper is organized as follows. Section 2 explains parametric verification. Section 3 describes the system model we assume. Section 4 presents our guidelines on how to design protocols such that they can be parametrically verified. Section 5 illustrates how we convert a highly optimized protocol that cannot be verified into a PVCoherence protocol. Section 6 presents our experimental results. Section 7 discusses related work, and Section 8 concludes the paper.

2. Background on Parametric Verification

In this section, we explain parametric verification (PV) at a level that is relevant to architects who would want to design protocols that can be verified with this method. Due to space constraints and our intended audience, we intentionally do not delve deeply into the theoretical foundations of PV.

There are a number of methodologies for PV, with greatly varying levels of automation. At one extreme, any modern theorem proving system is capable of doing PV, given enough human insight and effort. At the other extreme, there are fully automated approaches [3][12]. Unfortunately automation can be costly; such methods typically suffer from extreme limitations on the protocols that can be verified and/or high computational complexity (i.e., the "automation" is of no practical use). In the middle ground we find approaches [10] that are more automated than pure theorem proving but do require some manual intervention. Simple-PV falls into this space and has seen perhaps the most success in academia and industry.

2.1. 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. The counterexample gives useful information for the user to debug the system and find design errors.

There are various kinds of model checking tools. Murphi [11] is a widely used tool, especially for the verification of cache coherence protocols. The user models the cache coherence protocol in an expressive language and specifies the invariants. When model checking coherence protocols, it is common to assume that each cache contains only a single 1-bit line. Furthermore, details such as the physical interconnection and the memory controller are not modeled, because they have no impact on the correctness of the

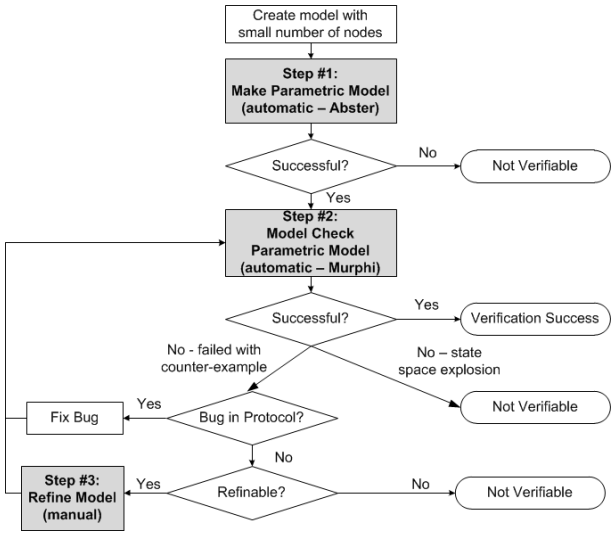

Figure 1. Simple-PV Verification Process

protocol. For the verification of cache coherence protocols, there are two primary invariants.

1. Permissions Invariant: The protocol must enforce the single-writer, multiple-reader (SWMR) invariant [27]: for each block of memory, at any given time, the block either has a single writer or zero or more readers.
2. Data Invariant: A read of a block must return the value of the most recent write to that block.

Model checking is limited to small systems (typically 2-5 nodes), because of its use of exhaustive search. Researchers have developed some techniques for reducing the size of the reachable state space—such as leveraging the "symmetry" of identical components to remove redundant states [23]—but these techniques cannot enable the verification of systems with arbitrary numbers of nodes.

2.2. Methodology of Simple-PV

Although model checking is attractive because it is fully automated, model checking is insufficient for parametric verification. Except in very restrictive situations (e.g., trivial systems), parametric verification fundamentally requires some amount of human effort [15], and we focus on the PV method that is practical and that we have found to require the least amount of human effort.

Chou et al. [8] propose a simple method, which we call Simple-PV, to parametrically verify cache coherence protocols. This method is based on McMillan's compositional reasoning theory [22]. 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 now discuss each step. We start with a non-parametric model, like in a typical non-parametric verification.

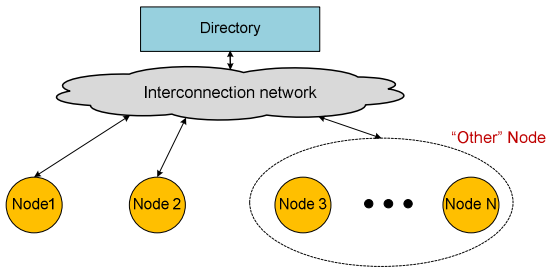

Figure 2. Parametric Model

Step #1: Automatically Create Parametric Model

The first step in Simple-PV is to create a parametric model from the non-parametric model. Consider the system with *N* nodes in Figure 2. Starting with two concrete nodes[2] in the non-parametric model, we then ***abstract*** the other *N*-2 nodes into a single "Other" node that we refer to as OtherNode. 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 process of abstraction with a fully automated tool called Abster [28] that was developed by Intel. Given a non-parametric model specified in the Murphi model checker's language, Abster produces a parametric Murphi model by generating the behavior for OtherNode. Abster's automation helps greatly in avoiding tedious and error-prone manual abstraction. The key point of abstraction is that it must preserve all the behaviors of OtherNode that could occur. Thus, Abster conservatively over-approximates the behavior 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 over-approximation leads to a challenge that we address in Step #3.

Abster may fail to generate a parametric model. This failure does not necessarily mean that a protocol cannot be verified with Simple-PV; instead, it means the protocol is not compatible with Abster. Because we would like to make the verification as automated as possible, we modify the protocol until it is compatible with Abster.

Step #2: Automatically Model Check the Model

If Abster successfully creates a parametric model, we use Murphi model to automatically model check that the model satisfies the two coherence invariants listed in Section 2.1. If Murphi succeeds, the protocol is coherent and we are done. If Murphi fails, there are four possible scenarios:

1. There are real bugs in the cache coherence protocol design. In this case, we must debug the protocol and then return to Step #2.

[2] The number of concrete nodes to instantiate depends on the protocol, for reasons explained in Section 2.3, but it tends to be two or three.

2. The state space of the parametric model exceeds the capacity of Murphi. Even with parameterization, there are systems that are too large for Murphi. In this case, we must re-design the protocol such that the abstracted parametric protocol "fits" in Murphi.
3. The over-approximation 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.
4. The protocol is incompatible with Simple-PV. In this case, no amount of fixing OtherNode leads to a protocol that can be verified with Murphi.

Step #3: Manually Refine the Model

Because Abster over-approximates 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 counter-example provided by Murphi, the verifier modifies OtherNode to restrict its behavior.

Restricting the behavior of OtherNode 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 behaviors that are possible and that lead to genuine violations of the coherence invariants.

The key to refinement is to both constrain the behavior of OtherNode *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 (i.e., true for a non-abstracted model). Furthermore, this added invariant is checked on the concrete nodes.[3] 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 (a) Murphi successfully verifies the parametric model, in which case we know the protocol is correct for any arbitrary number of nodes, or (b) 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. We discuss why protocols may be incompatible with Simple-PV in Section 2.4.

While Step #3 involves manual effort, prior work (on simple protocols [8]) and our work here indicates that the process is both straightforward and tends to involve only a few iterations.

[3] It appears that checking the lemma on the concrete nodes when OtherNode has been constrained is circular. However, verification literature has shown that the circularity is broken using an induction over time along with symmetry, and hence the method is sound [8][22].

2.3. Choosing the Number of Concrete Nodes

One issue in Simple-PV is choosing the number of concrete nodes in the parametric model. This number is a function of both the protocol and the invariants. A rigorous explanation of the theory behind choosing the number of concrete nodes [8] is beyond the scope of this paper, but we provide the intuition here. Essentially, we must have enough concrete nodes such that we can describe every possible situation and invariant. For example, if we have only one concrete node, then we cannot describe the invariant that two different (concrete) nodes cannot both be in M(odified) state at the same time. In this situation, at least two concrete nodes are needed. Our protocol in this paper requires two concrete nodes, but some other protocols may require three.

2.4. Limitations of Simple-PV

One limitation of Simple-PV is that, as mentioned previously and illustrated in Figure 1, some possible protocols are incompatible with Simple-PV. This limitation of Simple-PV is understandable, because there is a trade-off between the expressiveness of a logical formalism and the difficulty of its decision problem [13]. To be more specific, Simple-PV seeks to maximize the usage of automated tools and reduce the human effort. The automated tools ease the verification process, but heavy reliance on automation may somewhat limit the kinds of protocols we can verify.

From a formal and mathematical perspective, Krstic [14] describes the limitations of Simple-PV. Krstic describes the syntax of a mathematical language that can be used to describe protocols for which Simple-PV is always applicable. However, it is not obvious how to recast these mathematical formulations into a set of guidelines for realistic protocols (that are not specified mathematically).

One motivation for our paper is that the practical limitations of applying Simple-PV to today's coherence protocols are mostly unknown. Most of the systems verified with Simple-PV are old and may not be suitable for today's multicore processors. For example, the two example protocols in Chou et al.'s paper [8] assume that each core is its own chip with its own dedicated link to its own portion of the distributed memory. The exception is recent work by O'Leary et al. [24] that shows that Simple-PV can be successfully applied to a specific high-performance modern protocol. However, as with much prior work, it is unclear how to generalize from the specific protocol verified to the space of protocol features that are compatible with Simple-PV. Furthermore, even if a feature is incompatible with Simple-PV, it is unclear whether the constraints imposed to ensure compatibility are too costly in terms of performance or storage.

Our goal is to explore the limitations of Simple-PV from an *architect's* perspective and provide the designers with insights into how to design cache coherence protocols that are compatible with Simple-PV.

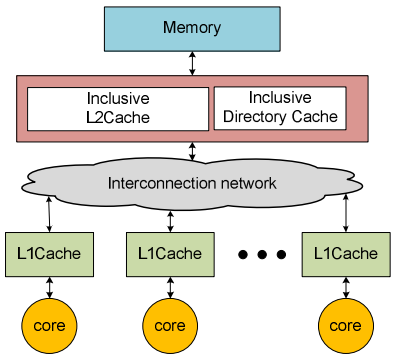

Figure 3. System Architecture

3. System Architecture

In this section, we present the system architecture based on which we design cache coherence protocols. Currently multicore processors usually employ a multi-level cache hierarchy, in which each core has one or more private caches and all cores share a last level cache. This system model includes Intel's Nehalem [26] and AMD's Barcelona [9]. We illustrate this system model in Figure 3.

Coherence Protocol: We assume a directory-like coherence protocol. We describe the operation of the cache coherence protocols in this paper using terminology common to both Sorin et al. [27] and the protocols that are distributed with the gem5 simulator [7]. The coherence requests are: "GetS" to obtain Shared (read-only) access, "GetM" to obtain Modified (read-write) access, "PutM" to writeback a Modified block, "PutO" to writeback an Owned block, and "PutE" to writeback an Exclusive block.

Cache Hierarchy: The last-level L2 cache is inclusive with respect to the L1 caches. To maintain inclusion, evicting a block from an L2 requires invalidating that block from any L1 caches that hold it. With an inclusive L2, the L2 tags can be extended to create a co-located directory cache. That is, each L2 block holds the directory state for that block. Because the L2 is inclusive, the directory cache has the state of all blocks present in one or more L1 caches, and a miss in the L2 implies that the block's state is I(nvalid) and leads to a memory access.

Interconnection Network: We make no assumptions about the interconnection network except regarding virtual channels. Directory protocols require multiple virtual channels to avoid deadlocks due to circular dependences on messages. Request messages can lead to Forwarded Request messages (including invalidations) that can lead to Response messages that can, in some protocols, lead to Completion messages. Each class of message travels on its own virtual channel. These virtual channels may or may not be ordered, depending on the protocol; we later discuss the impact of ordering on verification.

4. Guidelines for Coherence Protocol Design

We seek to design cache coherence protocols that can be parametrically verified with Simple-PV. However, not all possible protocol features are compatible with Simple-PV, due to its theoretical limitations. Our goal in this paper is to explore the fundamental limitations of Simple-PV and discover which protocol features do and do not satisfy these limitations. We have strived to include the most important protocol features in our study, but we cannot claim that the list of features is complete, because there are so many possible ways to design a protocol. Nevertheless, we believe we have explored the most important design decisions and whether they satisfy the fundamental constraints of Simple-PV. Ideally, an architect can use this study to determine if a new feature is compatible with Simple-PV.

When we discuss how compatibility with Simple-PV imposes limitations on cache coherence design, we generally focus on limitations that are due to the fundamental theory underlying Simple-PV. However there are some limitations we present that are not fundamental but are rather limitations imposed by state-of-the-art tools. A tool-based limitation may seem uninteresting, but architects must use today's tools, and there is no clear path to enhancing the tools to overcome these current limitations.

We now present the guidelines in order from the most intuitive to what we consider to be the least intuitive.

Guideline #1: All nodes must be identical.

If, instead of identical nodes 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. Hence our notion of Simple-PV disallows such protocols.

In theory, an automated tool could abstract a system with two different concrete nodes and all other *N*-2 nodes being the same type as one of the two concrete nodes. But such a system is not practically interesting and we do not consider it.

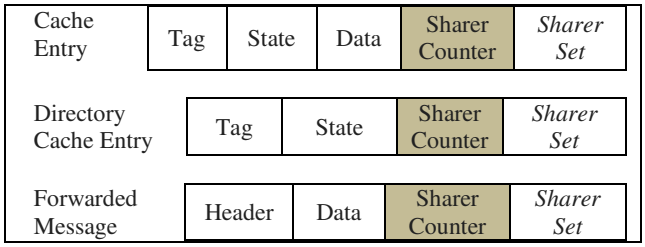

Figure 4. Components impacted by Guideline #2

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

With Simple-PV, we treat the number of nodes as a parameter rather than as 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 actual value of the parameter.

This guideline most directly impacts coherence protocol design by prohibiting the use of counters (that count the number of nodes). Typical directory protocols often use

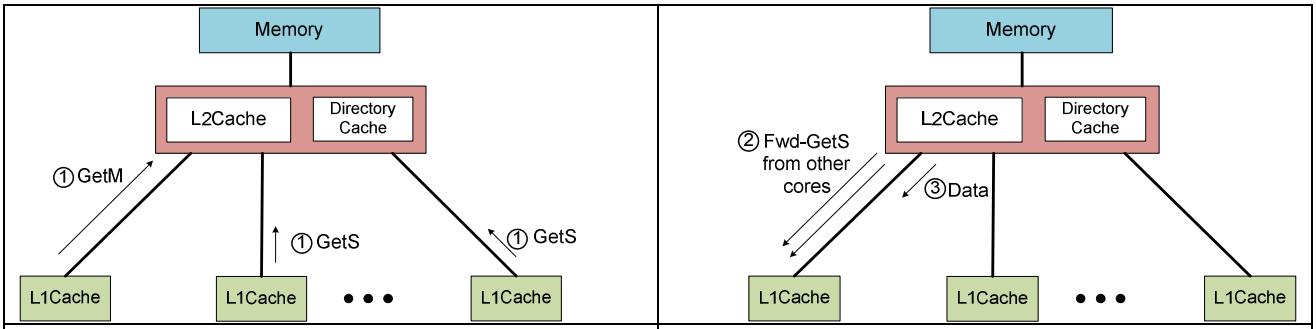

Figure 5. Scenario #2 forbidden by Guideline #3

counters to aid in collecting acknowledgments, such that a core waits to write to a block until it has received acknowledgments from some number of other cores that had been sharing the block. Token Coherence [18] uses counters to track tokens used to determine permissions.

Figure 4 shows the protocol components that may be impacted by Guideline #2. The gray entries indicate storage and message fields that we need to avoid: sharer counters. Instead of a sharer counter, we need to use a bit vector to denote the sharer set. This constraint leads to potential overhead in two ways. 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.

It is perhaps surprising that Simple-PV allows us to use a bit vector with a size equal to the number of nodes but disallows a counter with a size equal to the number of nodes, even though the bit vector is larger than the counter. However, the intuition is that using a counter is prohibited because it involves comparisons to a value that is parameterized.

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

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

This guideline has a significant impact on coherence protocol design. 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. Some queues have a depth that does not depend on the number of nodes, such as a queue of requests from one L1 cache to the L2; the depth of this queue depends on the number of outstanding requests the L1 can have, which is both small and not parameterized.

Nevertheless, there are situations in which we might want a queue with a depth that depends on the number of nodes, and we discuss two examples in order of complexity.

(1) Consider a system in which all of the L1 caches share a queue of requests to the L2. This queue has a depth that is proportional to the number of nodes. Such a queue is compatible with Simple-PV only if it is unordered.

(2) Consider a protocol that relies upon point-to-point ordering of forwarded coherence requests from the directory to each L1. Many protocols rely on this ordering to avoid races that would otherwise complicate the protocol and require additional messages to acknowledge message reception. However, ordering of 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, in many protocols, this situation is possible. For example, if Core C1 has requested Modified permissions for block B, the directory could forward subsequent GetS requests for B to C1 from every other core before C1 receives the data for B and can start responding to the GetS requests that have filled its queue. We illustrate this scenario in Figure 5. Most protocols do not rely on ordering of the forwarded GetS requests in this example [4]; 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 #3 certainly constrains protocol design, but this constraint is necessary for compatibility with Simple-PV.

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 have seen), arrays are used to 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 (e.g., 4) or as 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 explained in the discussion of

[4] Ordering is more useful for races involving writebacks.

Guideline #3, 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 we want to specify an array that is parameterized in more than one dimension. The consequence of this constraint is that it affects a common protocol design option. Namely, it precludes us from letting a core that issues a GetM collect all of the acknowledgment messages from cores that were invalidated by the GetM. In this scenario, Core C1 issues a GetM to the L2 and the L2 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 two-dimensional 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 somewhat less efficient than having the requesting L1 collect the acknowledgments, because it requires an extra message on the critical path for completing the transaction.

Guideline #4 is not as fundamental as the others; we could violate this guideline and still have a protocol that is compatible with Simple-PV. Nevertheless, there are two reasons we follow this guideline. First and foremost, parameterizing in multiple dimensions requires much more concrete state to be maintained in the parameterized model (compared to a model with a one-dimensional parameterization), and this extra state may well exceed the capacity of the model checker. A secondary reason to follow this guideline is that multiple dimensional parameterization requires a more sophisticated abstraction tool, which may not be available. Abster, as one example, does not support parameterization in multiple dimensions. This tool-specific reason for following Guideline #4 is a practical but not fundamental limitation of Simple-PV.

Observations about Specific Optimizations

The above four guidelines are basic rules for protocol design. However, even if a protocol follows all of these guidelines, the subtle details of the protocol can affect the protocol's compatibility with Simple-PV. We explore the design space to determine which design choices are compatible with Simple-PV and which are not.

We start with a simple protocol with three stable states: M, S, and I. The protocol follows Guidelines #1-#4 and is conservative (has little concurrency). Basically, only one transaction is allowed at a time. Before a transaction can complete, the requesting L1 cache must send a "completion" message to the L2 cache. Before this completion message arrives at the L2, the L2 blocks subsequent requests from other cores. This simple protocol can be abstracted by Abster and verified by Murphi with the manual addition of only one lemma. Informally, this lemma constrains the behavior of a node such that a node with a block in state M has the most current data for that block. Without specifying this lemma, the block's data value that is generated by Abster is arbitrary, which can lead to violations of the Data Invariant.

Although this protocol is compatible with Simple-PV and requires minimal manual effort to verify, this protocol is overly simplistic and would not perform well. Hence, we optimize this simple protocol by adding more states (both stable and transient states) and transactions, which increases concurrency. We considered the following list of optimizations, adding them in this order:

1. We add the stable state E(xclusive).
2. We add the stable state O(wned).
3. We add an Upgrade request for increasing coherence permission from read-only (Shared) to read-write (Modified). The response to an Upgrade request does not require a large data message. Without an Upgrade request, a core with a Shared block must issue a GetM and receive data even though it already has valid data.
4. We add silent eviction for Shared blocks. An L1 cache can evict a Shared block without notifying the L2 cache.
5. We remove the completion messages for GetS transactions when the data response comes from the L2 (and not another L1). An L1 cache that sends a GetS request to the L2 does not have to notify the L2 once it has received the data from the L2, and the L2 no longer blocks while waiting for completion messages.
6. We remove the completion messages for GetM transactions. An L1 cache that sends a GetM request to the L2 does not have to notify the L2 once it has received the data and the (aggregated) acknowledgment from the L2, and the L2 no longer blocks while waiting for completion messages.

The impact of these optimizations on Simple-PV varies. Adding the "E" state (Optimization 1) has zero impact. Because the protocol is still conservative in that it allows only one transaction at a time, we can verify it without adding more lemmas. Adding the "O" state (Optimization 2) requires two lemmas; for example, one lemma constrains OtherNode's behavior based on whether the L2's coherence state indicates that a concrete L1 is in state S or not.

Optimizations 3-5 require a few extra lemmas during the iterative verification process, but the protocols are still verifiable with Simple-PV. For example, one lemma says that when an L1 cache is waiting for a data reply from the L2 cache, there cannot be other L1 caches sending data to the requesting L1. This lemma constrains the behavior of OtherNode, preventing its abstracted caches from sending data to the concrete caches when they are not supposed to.

Optimization 6 is not compatible with Simple-PV. Assume that one of the *N*-2 abstracted L1 caches is the Modified owner of a block. L1 cache C0 has the block in state I and issues a GetM to the L2 and transitions to transient state IM

(in I, waiting to go to M). The L2 forwards C0's GetM to OtherNode and immediately changes the directory state to indicate that C0 is the owner.[5] Before C0 receives data from the owner (in OtherNode), another abstracted L1 cache issues a GetM to the L2. The L2 forwards this request to C0 and changes the directory state to indicate that the owner is OtherNode. C0 still does not have data, and it changes its block state to the transient state IMI (in I, waiting to go to M, will do one store when it gets the data, and then will go back to I). At this time, cache C1 goes through the same process that C0 has just gone through and also ends up in state IMI.

Now the problem for Simple-PV emerges. OtherNode replies with data but cannot determine whether to send to C0 or C1. C0 and C1 are in the same state and both are eligible to receive data. The L2 cannot maintain the ordering in which the GetM requests arrived, because that would require ordering a list that has a size proportional to the number of nodes (and Guideline #3 explains why ordering such a list is incompatible with Simple-PV). Thus the races caused by this particular optimization make the protocol with this optimization incompatible with Simple-PV. One could imagine protocols that allow the L2 to send data to either C0 or C1 without regard to the order in which their requests arrive, but such protocols would suffer from fairness and potential starvation problems.

Conclusions: Designing a coherence protocol that can be verified with Simple-PV requires adhering to several guidelines. The list of features in this section may not be exhaustive, but we believe that we have included all of the fundamental reasons for why features may be incompatible.

5. Design of a PVCoherence Protocol

Following the above guidelines, we can design PVCoherence protocols. The common feature of all PVCoherence protocols is that they can be formally verified using Simple-PV. Although all PVCoherence protocols obey the design guidelines presented in Section 4, there can still be considerable variation between different PVCoherence protocols. In this section, we show the design process of one PVCoherence protocol, called PV-MOESI. We have also experimented with other PVCoherence protocols, but PV-MOESI covers more of the interesting features discussed in Section 4 and thus we present it here.

PV-MOESI is based on the system architecture in Figure 3. 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, and it provides high performance but it cannot be verified using Simple-PV. In the design of PV-MOESI, we try to keep it as similar to OP-MOESI as possible, only modifying it when necessary to satisfy the constraints of Simple-PV.

[5]This immediate transition differs from a protocol with a completion message; with a completion message, the directory state would not change until the completion arrives from C0.

5.1. Optimized Baseline Protocol: OP-MOESI

The OP-MOESI coherence protocol is a fairly standard directory protocol that is optimized for performance. OP-MOESI is similar to other prevalent protocols [9][26]. OP-MOESI has five stable L1 cache coherence states (MOESI) and more than 30 transient states to improve performance. An L2 block can be in a similar set of states, except that an L2 block cannot be in E (there is no use for it) and it can be in one of two "stale" states: M(s) and O(s). These stale states denote when there is an L1 that has the block in M or O, respectively, and that L1 potentially has a more up-to-date value of the data than the L2.

The directory state, which is co-located with the L2 tag/state, includes a full-map bit vector that denotes which L1 caches are currently caching each block. We denote L2 states in the form *X:Y*, where *X* is the state of the L2 block itself and *Y* is the directory state. For example, an L2 state of M:I denotes that the L2 holds an M copy of the block and no L1 caches have a copy of the block.

The protocol relies on having three virtual channels in the system; there are virtual channels for requests, forwarded requests, and responses. All of these virtual channels enforce point-to-point ordering for OP-MOESI.

We specify OP-MOESI at a high level in Table 1. This specification omits all of the complexity of transient states, but it provides the big picture of how the protocol works.

5.2. PV-MOESI

Although highly optimized, OP-MOESI cannot be verified with Simple-PV. Abster fails to generate an abstracted model for OP-MOESI and thus we cannot run it through Murphi. In this section, we create PV-MOESI by modifying OP-MOESI to satisfy the guidelines in Section 4. The specification of PV-MOESI is alongside the specification of OP-MOESI in Table 1, with PV-MOESI's differences highlighted in **bold**.

1. 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).
2. For GetM transactions, we have the L2, instead of the L1 requestor, collect the acknowledgments from L1 caches that are invalidated. After collecting all L1 acknowledgements, the L2 sends a single acknowledgement to the L1 requestor. This modification adds one more network hop per GetM.
3. 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. The races happen when an L1 receives a forwarded request or an invalidation while in a transient state. PV-MOESI handles these races in the usual fashion (with extra

Table 1. High-level specifications of OP-MOESI and PV-MOESI. Ignores transient states. Differences between OP-MOESI and PV-MOESI are in bold font in PV-MOESI specification.

	OP-MOESI	PV-MOESI
Structures		
L1 cache entry	64B data, tag, state={M,O,E,S,I}	
L2 cache entry	64B data, tag/bit vector to track L1 caches, directory state={I:I, S:S, O:S, M:I, O(s):O, M(s):M}	
Core C1 has load miss on block B in its L1, sends GetS to L2		
L2 = I:I	L2 gets block from memory and sends it to C1; L2 adds C1 to bit vector; L2 →M(s):M; C1's L1 → E	*same as OP-MOESI*
L2 = S:S or O:S	L2 sends data to C1; L2 adds C1 to bit vector; C1's L1→S	*same as OP-MOESI*
L2 = M:I	L2 sends data to C1; L2→M(s):M; C1's L1→E	L2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→E
L2 = O(s):O C2 is the owner	L2 forwards GetS to C2; L2 adds C1 to bit vector; C2 sends data to C1; C1's L1→S	L2 forwards GetS to C2; L2 adds C1 to bit vector; C2 sends data to C1; **C1 sends Completion to L2;** C1's L1→S
L2 = M(s):M C2 is the owner	L2 forwards GetS to C2; L2 adds C1 to bit vector; L2→O (s):O; C1's L1→S	L2 forwards GetS to C2; L2 adds C1 to bit vector; **C1 sends Completion to L2;** L2→O (s):O; C1's L1→S
Core C1 has store miss on block B in its L1, sends GetM to L2		
L2 = I:I	L2 gets block from memory and sends it to C1; L2 →M(s):M; C1's L1→M	L2 gets block from memory and sends it to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→M
L2 = S:S or O:S	L2 sends data to C1 with number of sharers and sends invalidations to sharers; sharers send acks to C1; L2 →M(s):M; C1's L1→M	**L2 sends invalidations to sharers; sharers send acks to L2; L2 sends data to C1 (without number of sharers); C1 sends Completion to L2;** L2 →M(s):M; C1's L1 → M
L2 = M:I	L2 sends data to C1; L2→M(s):M; C1's L1→M	L2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→M
L2 = O(s):O L1 owner is C2	L2 forwards GetM to C2 with number of sharers and sends invalidation to sharers; C2 sends data to C1; sharers send acks to C1; L2 →M(s):M; C1's L1→M; C2's L1→I	L2 forwards GetM to C2 **(without number of sharers)** and sends invalidations to sharers; C2 sends data to C1; **sharers send acks to L2; L2 sends ack to C1**; **C1 sends Completion to L2;** L2 →M(s):M; C1's L1→ M; C2's L1→I
L2 = M(s):M L1 owner is C2	L2 forwards GetM to C2; C2 sends data to C1; L2 →M(s):M; C1's L1 → M; C2's L1→I	L2 forwards GetM to C2; C2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1 → M; C2's L1→I
Core C1 has store miss on block B in its L1, but it has the data, sends Upgrade to L2		
L2 = S:S or O:S or O(s):O	L2 sends invalidations to sharers except C1; L2 sends ack to C1 with number of sharers; sharers send acks to C1; L2 →M(s):M; C1's L1 → M	L2 sends invalidations to sharers except C1; **sharers send acks to L2; L2 sends ack to C1; C1 sends Completion to L2;** L2 →M(s):M; C1's L1 → M
Core C1 wants to evict block B from its L1		
C1's L1=S	C1 immediately evicts block; C1's L1→I	
C1's L1=E	C1 sends PutE to L2 without data, waits for ack	
C1's L1=O or M	C1 sends PutO or PutM with data to L2, waits for ack	
L2 wants to evict block B		
L2 = I:I	L2 immediately evicts block	
L2 = S:S or O:S	L2 sends invalidations to L1 sharers, waits for acks, then evicts	
L2 = M:I	L2 writes data back to memory, waits for ack from memory, then evicts	
L2 = O(s):O	L2 sends GetM to L1 owner, sends invalidations to L1 sharers, waits for data and acks, then evicts	
L2 = M(s):M	L2 sends GetM to L1 owner, waits for data, then evicts	

messages and extra transient states) but without ever 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. Handling the races introduces some complexity but is manageable.

After the above modifications, we find that the model can be abstracted by Abster. However, the abstracted model still cannot be verified by Murphi, regardless of how we try to refine it. This problem—which arises due to multiple in-flight GetM requests—was discussed at the end of Section 4, and we handle it by modifying how the protocol handles GetM requests. When the L2 receives a GetM it forwards the GetM and/or 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 and/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 7 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 completed in under one hour and used several gigabytes of memory.

6. Evaluation

Creating PV-MOESI from OP-MOESI revealed several issues which could potentially cause PV-MOESI to be worse than OP-MOESI with respect to performance, storage, and network traffic. Therefore, we performed a series of experiments to compare PV-MOESI and OP-MOESI.

6.1. Methodology

We evaluate OP-MOESI and PV-MOESI using the gem5 full-system simulator [7]. For both protocols, we keep the common architectural parameters the same: processor configuration, L1/L2 cache size, memory size, link latency, link bandwidth, etc. We calculated the access latency of storage structures using Cacti [25]. The system parameters are shown in Table 2.

For benchmarks, we use the PARSEC benchmark suite [6], except for two benchmarks, streamcluster and fluidanimate, that are not compatible with gem5. We run each experiment multiple times to accommodate the natural variability in simulation runtimes [2]; error bars in graphs indicate plus/minus one standard deviation from the mean.

6.2. Performance

The primary goal of our experimental evaluation is to determine the performance difference between the unverifiable OP-MOESI and the verifiable PV-MOESI. There are several reasons why PV-MOESI's performance could potentially be less than that of OP-MOESI, including

Table 2. 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 I&D caches	32 KB, 2-way, 2 cycle hit
L2 cache	inclusive with respect to L1s; 8MB split into 16 banks – each bank 512 KB, 8-way, 12-cycle hit
Memory	2GB, 160-cycle hit
Interconnection Network Parameters	
Topology	2D mesh
Link bandwidth	32 GB/s
Link latency	1 cycle

PV-MOESI's extra Completion messages and requiring the L2 to collect invalidation acknowledgments. The question is whether, in practice, these potential performance degradations occur. In Figure 6, we plot the runtimes for both OP-MOESI and PV-MOESI, normalized to the runtime of OP-MOESI, for 32-core systems. While there are some differences in the runtimes, they are "within the noise." On some benchmarks, PV-MOESI even has a marginally shorter runtime than OP-MOESI, but these differences are also within the noise and are not meaningful speedups.

To better understand why PV-MOESI's performance is effectively the same as that of OP-MOESI, we evaluated two issues: the impact of PV-MOESI's Completion messages and PV-MOESI's additional network usage.

Completion Messages: PV-MOESI's use of Completion messages can potentially hinder performance. While waiting for a Completion message on block *B*, the L2 stalls requests for block *B*. To evaluate the performance impact of this L2 stalling, we inspected the fraction of requests that arrive at the L2 and must stall while waiting for a Completion. For all benchmarks, this fraction was well less than 1%, i.e., the use of Completions messages causes few stalls and has little impact on performance.

Network Overhead: PV-MOESI uses more interconnection network bandwidth than OP-MOESI. This extra bandwidth is mainly due to the extra messages caused by Completions. Intuitively, this bandwidth overhead should be small, but we experimentally evaluated it to confirm this expectation. In Figure 7, we plot the total traffic consumed by PV-MOESI, normalized to the traffic consumed by OP-MOESI. For most benchmarks, the overhead is less than 5%, but it is as high as13.8% for canneal. In our system model, the performance impact of this extra network traffic is minimal, but it could be greater in systems with more limited network bandwidth.

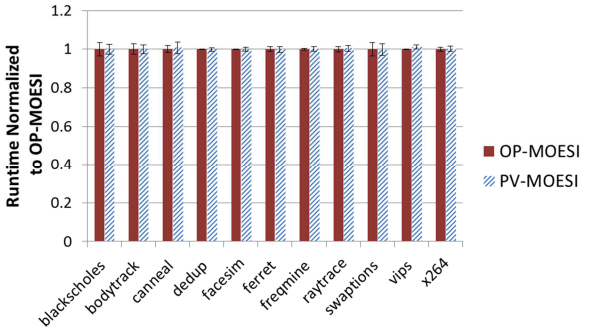

Figure 6. Runtime comparison: OP-MOESI vs PV-MOESI

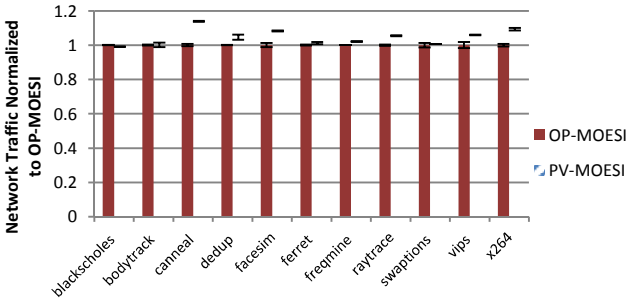

Figure 7. Network traffic overhead of PV-MOESI

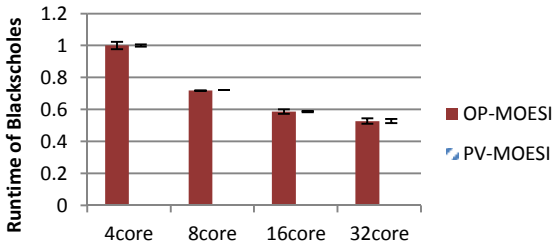

Figure 8. Performance Scalability

6.3. Scalability

Because our goal is to create protocols that are verifiable even as they scale to larger numbers of cores, we are interested in studying PV-MOESI's performance scalability. We focus on one representative benchmark, blackscholes, and we show how its performance scales from 4-32 cores. In Figure 8, we compare the runtimes for OP-MOESI and PV-MOESI, normalized to OP-MOESI's 4-core runtime, as a function of the number of cores. We observe that PV-MOESI tracks OP-MOESI's performance for all core counts and is thus just as scalable—both up and down—as OP-MOESI. We also note that speedups are less than linear with core count, which is a function of the benchmark more than that of the protocol.

6.4. Storage Overhead

We evaluate the storage overhead of PV-MOESI by looking at the L2 cache and L1 cache separately.

In the L2 cache, PV-MOESI requires a sharer set in the directory to record all L1 sharers. This is also true for OP-MOESI. Therefore, PV-MOESI adds no storage overhead compared to a protocol with a full-map directory. Those optimization techniques for reducing the storage cost of the directory, such as coarse directory, limited pointer directory [1], etc., can also be employed in PV-MOESI as long as they do not involve sharer counters.

In the L1 cache, PV-MOESI has no storage overhead, either. One could, however, imagine a PVCoherence protocol that had L1 storage overhead if the L1 maintained a sharer set. Such protocols are rare, but it is possible that a protocol would have the L1's MSHR entries track outstanding acknowledgments, in which case PVCoherence would require a sharer set instead of a less costly counter. Even in this scenario, the storage overhead is tiny compared to the overall size of the L1 cache.

7. Related Work

The most related work consists of hierarchical protocols that were designed for verification. Zhang et al. [31] design coherence protocols in a fractal, hierarchical way, which ensures self-similarity at each scale, to enable inductive verification. The base case of the proof is the verification of a minimum-scale system. Voskuilen and Vijaykumar [30] greatly improve upon the performance of fractal coherence protocols by creating protocols that are provably equivalent to fractal protocols yet do not have some of the performance pathologies of Zhang et al's protocol. Matthews et al. [20] apply the fractal approach for a dynamic power management protocol. Beu et al. [4] leverage a coherence design framework called MCP [5] for composing heterogeneous protocols in a hierarchical fashion. Beu et al. show that, if each of the building block protocols is verified correct then the hierarchical protocol is also correct by induction. Our work in this paper complements this prior work on verifiable hierarchical protocols, because it enables the verification of larger "minimum systems" in fractal protocols and larger "building blocks" in MCP. Current automated tools can verify protocols with only 2-5 caches, which is not ideal for either fractal base cases or MCP building blocks. It is important to be able to verify larger flat protocols, both for stand-alone purposes and for use in hierarchical protocols.

Other work considers verification or design complexity when designing protocols. HCC [16] is organized hierarchically as a tree of caches. This tree organization facilitates verification of liveness and consistency. HCC is verified manually, unlike the largely automated verification in our work. Vantrease et al. [29] propose an atomic coherence protocol that avoids races and is thus simpler; we expect it would be easier to verify than a typical non-atomic protocol, but verification is not discussed in the paper.

Some prior work has compared the verification effort required for different coherence protocols. Martin [17] compared snooping and directory protocols. Marty et al. [19] discussed the formal verification efforts of different protocol

designs and further showed their proposed protocol is more amenable to formal verification. Our work differs from this work by considering verification at design time instead of analyzing verification effort for given designs.

8. Conclusions

We have shown that, with awareness of certain issues that affect parameterization, we can design protocols that are compatible with parametric verification. Furthermore, our experimental results show that we can develop a protocol that is both compatible with PV and achieves performance comparable to today's typical multicore coherence protocols.

Acknowledgments

We thank Murali Talupur of Intel for supporting us in using his Abster tool. This material is based upon work supported by the National Science Foundation under grant CCF-0811290.

References

[1] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence," in *15th Annual Int'l Symposium on Computer Architecture*, 1988.

[2] A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," in *Proc. of 9th Int'l Symp. on High-Performance Computer Architecture*, 2003.

[3] K. Baukus, Y. Lakhnech, and K. Stahl, "Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness," in *Verification, Model Checking, and Abstract Interpretation*, vol. 2294, A. Cortesi, Ed. Springer Berlin Heidelberg, 2002.

[4] J. G. Beu et al., "High-Speed Formal Verification of Heterogeneous Coherence Hierarchies," in *19th Int'l Symp. on High Performance Computer Architecture*, 2013.

[5] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies," in *Proc. 44th Annual Int'l Symposium on Microarchitecture*, 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, 2008.

[7] N. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, Aug. 2011.

[8] C.-T. Chou, P. Mannava, and S. Park, "A Simple Method for Parameterized Verification of Cache Coherence Protocols," in *Formal Methods in Computer-Aided Design*, 2004.

[9] P. Conway et al., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro,* vol. 30, no. 2, 2010.

[10] S. Das, D. L. Dill, and S. Park, "Experience with Predicate Abstraction," in *Computer Aided Verification*, 1999.

[11] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," in *IEEE Int'l Conf. on Computer Design: VLSI in Computers and Processors, 1992*.

[12] E. A. Emerson and V. Kahlon, "Exact and Efficient Verification of Parameterized Cache Coherence Protocols," in *Correct Hardware Design and Verification Methods*, 2003.

[13] J. Harrison, "Theorem Proving for Verification (Invited Tutorial)," in *CAV*, 2008, pp. 11–18.

[14] S. Krstic, "Parametrized System Verification with Guard Strengthening and Parameter Abstraction," *Automated Verification of Infinite State Systems*, 2005.

[15] R. Krzysztof and D. Kozen, "Limits for Automatic Verification of Finite-State Concurrent Systems," *Information Processing Letters*, vol. 22, pp. 307–309, 1986.

[16] E. Ladan-Mozes and C. E. Leiserson, "A Consistency Architecture for Hierarchical Shared Caches," in *20th Annual Symp. on Parallelism in Algorithms and Architectures*, 2008.

[17] M. M. K. Martin, "Formal Verification and its Impact on the Snooping versus Directory Protocol Debate," in *Proc. of the Int'l Conference on Computer Design*, 2005.

[18] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token Coherence: Decoupling Performance and Correctness," in *30th Annual Int'l Symposium on Computer Architecture*, 2003.

[19] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood, "Improving Multiple-CMP Systems Using Token Coherence," in *11th Int'l Symposium on High-Performance Computer Architecture, 2005.*

[20] O. Matthews, M. Zhang, and D. J. Sorin, "Scalably Verifiable Dynamic Power Management," in *20th Int'l Symposium on High Performance Computer Architecture*, 2014.

[21] K. L. Mcmillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," in *CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*, 2001.

[22] K. L. McMillan, "Verification of Infinite State Systems by Compositional Model Checking," in *Proc. of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1999.

[23] C. Norris IP and D. Dill, "Better Verification Through Symmetry," *Formal Methods in System Design*, v. 9, no. 1–2, 1996.

[24] J. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol Verification Using Flows: An Industrial Experience," in *Formal Methods in Computer-Aided Design,* 2009.

[25] T. Shyamkumar, M. Naveen, and A. Ho, *P., JN: Cacti 5.1.* Technical Report HPL-2008-20, HP Labs.

[26] R. Singhal, "Inside Intel Next Generation Nehalem Microarchitecture," in *Hot Chips*, 2008, vol. 20.

[27] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[28] M. Talupur and M. R. Tuttle, "Going With the Flow: Parameterized Verification Using Message Flows," in *Int'l Conf. on Formal Methods in Computer-Aided Design*, 2008.

[29] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols," in *Proc 17th Int'l Symposium on High-Performance Computer Architecture*, 2011.

[30] G. Voskuilen and T. N. Vijaykumar, "High-Performance Fractal Coherence," in *Proc. 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[31] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal Coherence: Scalably Verifiable Cache Coherence," in *Proc. of the 43rd Annual Int'l Symposium on Microarchitecture*, 2010.