Architecting Hierarchical Coherence Protocols for Push-button Parametric Verification

Opeoluwa Matthews[1, 2] Daniel J. Sorin[1]
Duke University[1], University of Michigan[2]
luwa.matthews/sorin@duke.edu

ABSTRACT

Recent work in formal verification theory and verification-aware design has sought to bridge the divide between the class of protocols architects want to design and the class of protocols that are verifiable with state of the art tools. Particularly, the recent Neo work in formal verification theory, for the first time, formalizes how to compose flat subprotocols with an arbitrary number of nodes into a hierarchy while maintaining correct behavior. However, it is unclear if this theory scales to realistic systems. Moreover, there is a diversity of systems architects would be interested in, to which it is not clear if the theory applies.

In this paper, we show how the abstract Neo theory can be leveraged to design a realistic hierarchical coherence protocol. As such, we present the first realistic hierarchical coherence protocol verified with fully-automated (push-button) verification tools for all scales and tree configurations. We explore the practical limitations posed by both the theory and the verification tools in designing this verifiable hierarchical protocol. We experimentally evaluate our protocol, comparing it to more complex protocols that have optimizations prohibited by the theory and verification tool. Finally, we discuss how a variety of system configurations and protocols architects might be interested in can be adapted to the Neo theory, which we hope opens up the theory to future work in verification-aware protocol design.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Hardware** → *Model checking*;

KEYWORDS

cache coherence, formal verification, multicore

ACM Reference format:
Opeoluwa Matthews and Daniel J. Sorin. 2017. Architecting Hierarchical Coherence Protocols for Push-button Parametric Verification. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123971

1 INTRODUCTION

Cache coherence protocols are used in shared-memory multicore systems to ensure proper adherence to the underlying memory consistency model [30]. Hence, it is important to verify that a coherence protocol is devoid of bugs. There are two main approaches to this verification: simulation and formal verification. Formal verification is the more desirable approach because, unlike simulation, it provides a mathematical guarantee that a protocol behaves correctly in every reachable state.

Generally, formal verification is done with either *theorem provers* or *model checkers*. Theorem provers can theoretically be used to verify any system, including multiprocessors with complex coherence protocols [35]. However, they typically require significant expertise and considerable manual effort to operate. The verification engineer must also have deep knowledge of the system to be verified and a formal understanding of why the system is correct. Altogether, these issues could lead to long and costly design and verification cycles. Model checkers, on the other hand, offer a fully-automated (push-button) approach to verification. The engineer simply expresses a model of the system to the model checker, which traverses the reachable state space of the system. The model checker can then either certify that the system is correct or provide a trace to show how an erroneous state was encountered. Whenever possible, model checkers are the preferred tools, given the simplicity of use and the push-button automation that they provide.

Unfortunately, model checkers suffer from the *state explosion problem*. The state space of systems is often too large for the model checker to explore before running out of memory. As a result, there is an overly restrictive class of protocols and system configurations that can be verified with model checkers. Given that the state space grows exponentially with the number of nodes, the Murϕ model checker [15], for example, can only verify a realistic coherence protocol that has a handful of private caches. Also, model checkers currently cannot verify non-trivial hierarchical protocols, as hierarchies introduce significantly more interactions and, consequently, larger state spaces.

There has been some prior work in designing coherence protocols specifically to be scalably verifiable in model checkers [4, 5, 36, 40, 41]. Unfortunately, all have significant constraints and limitations. Fractal Coherence [41] proposed designing coherence protocols with hierarchies that have self-similarity on each scale; Fractal Coherence was limited to binary tree configurations. Voskuilen et al. showed how to collapse the Fractal Coherence hierarchy into *flat* directory protocols [36, 37]. PVCoherence [40] showed how to design *flat* directory coherence protocols to fit a parametric model checking technique. Note that this technique involved manual, iterative refinement of an over-approximated model of the protocol [10]. Manager-client Pairing (MCP) [4, 5] proposed composing heterogeneous coherence protocols into a hierarchy using a permission-checking algorithm that maintains coherence. Unfortunately, MCP was shown to be theoretically flawed [24] and did not allow for arbitrary degrees at each node.

In this work, our goal is to design hierarchical protocols to be automatically verifiable without the overt restrictions of prior work to specific configurations. Hierarchical coherence protocols are increasingly important and have been proposed as a way to improve scalability as the number of cores in shared-memory systems increases [4, 23, 27]. In such systems, it would be desirable for architects to be able to pick any specific configuration of a cache tree hierarchy (number of nodes, arity at each node, depth, etc.) based on factors such as system scale, workloads, cache sizes, etc. Unfortunately, the coherence protocol loses its mathematical guarantee of correctness if it is instantiated in a configuration for which it was not verified [2]. Therefore, we seek to be able to verify that a hierarchical coherence protocol behaves correctly regardless of the configuration of the hierarchy, which prior work has yet to provide support for.

A recent work in formal verification theory shows promise in enabling parameterized verification of hierarchical protocols. Matthews et al. present the Neo methodology for automatically verifying hierarchical protocols in a model checker, for any number of nodes and any arity at each node [24], which was previously not possible. The key is the theory allows one to reduce the verification of an entire hierarchy into the verification of flat subprotocols. The theory provides novel formal properties that, when successfully checked in a model checker, guarantee that the flat subprotocols can be composed into any arbitrarily large hierarchy while maintaining correctness.

While the Neo theory is formally rigorous, there are multiple reasons it is quite far from being practically applicable. First, it is unclear if the theory can be used to verify realistic protocols. To illustrate their approach, Matthews et al. verify only a coherence protocol, NeoGerman, derived by composing the German protocol [13] into a hierarchy. The German protocol is a toy coherence protocol with three stable states, no transient states, no data forwarding, and only a dozen transitions. The simplicity of NeoGerman belies the actual verification scalability of the Neo methodology. Features of most realistic protocols, such as transient states and data forwarding, enable a significant number of interleavings of transitions that must be checked by a verification methodology like Neo. Second, it is not clear what performance penalties are imposed on coherence protocols that are designed to fit the Neo theory. Third, the theory is abstract and does not flesh out what system configurations and architectures it can be applied to.

As our first contribution in this paper, we show how to use the Neo theory to, for the first time, design and verify realistic hierarchical coherence protocols using fully automated tools. Initially, in using the Neo methodology to verify a simple hierarchical protocol constructed from a baseline MSI directory protocol [30], the model checker exhausts over 200GB of RAM without terminating. This observation highlights the fact that the Neo verification methodology, as is, does not scale well with protocol complexity, which forces us to modify the methodology to be more efficient. As a result, we successfully design a specific hierarchical coherence protocol, NeoMESI, such that it can be verified to be correct regardless of the number of nodes or arity at each node. NeoMESI is a hierarchical directory protocol that supports a fully-inclusive cache hierarchy and provides MESI permissions on each level, with transient states to still allow for concurrency while requests are pending. Parent nodes can communicate with children, and nodes on the same level can communicate among each other for data forwarding.

As our second contribution, we use our process of designing NeoMESI to explore some protocol optimizations and whether or not the verification tools and the Neo theory can successfully verify them. We start from the simpler baseline hierarchical directory protocol discussed above. Then, we iteratively add features to attain a protocol beyond which the theory and model checker exhaust hardware resources or time bounds in the verification. This endeavor is important because, to guarantee correctness as hierarchies are scaled, Neo provides additional invariants that must be checked in the model checker. These additional invariants place more burdens on the model checker. Hence, our endeavor seeks to capture a snapshot of what protocols and features current formal verification theory and tools can handle.

Additionally, we evaluate the effects that the optimizations precluded by the Neo theory and verification tools have on performance by comparing the formally verified NeoMESI to protocols with these precluded optimizations. Since we are unaware of any other theory that enables push-button parametric hierarchical verification, we note that these optimized protocols are currently not amenable to push-button verification. Using the gem5 full-system simulator [7], we run benchmarks from the PARSEC suite [6] with multiple cache hierarchy organizations on each protocol. Overall, we find that NeoMESI performs statistically on-par with these optimized protocols.

As our final contribution in this paper, we describe how to model as Neo Systems multiple types of systems and protocols that do not seem like obvious fits to the theory. This is useful because architects might be interested in using Neo to design a wide variety of protocols beyond tree directory protocols like our NeoMESI. However, the Neo theory is quite abstract and, for some systems, it is not intuitive how to directly model them as Neo Systems. We hope that this contribution and, in general, our work in this paper can enable future work in verification-aware coherence protocol design based on the recent Neo theory.

2 THE NEO THEORY

Parametric verification involves verifying a protocol with the number of nodes expressed as a abstract parameter; this parameter can then be instantiated to any arbitrary number when the protocol is implemented [32]. A long-term goal of the formal verification community has been to develop and improve automated parametric verification tools. Unfortunately, until recently, automated parametric verification tools and techniques [12, 16, 32] could only verify flat (non-hierarchical) protocols. Parametric hierarchical verification, in which the arities at *all* the nodes in a tree are expressed as independent parameters, consistently led to exhaustion of memory, even for trivial systems.

The goal of the Neo theory [24] was to extend push-button parametric verification to hierarchical protocols. The Neo theory is abstract and covers different types of protocols (e.g., coherence,

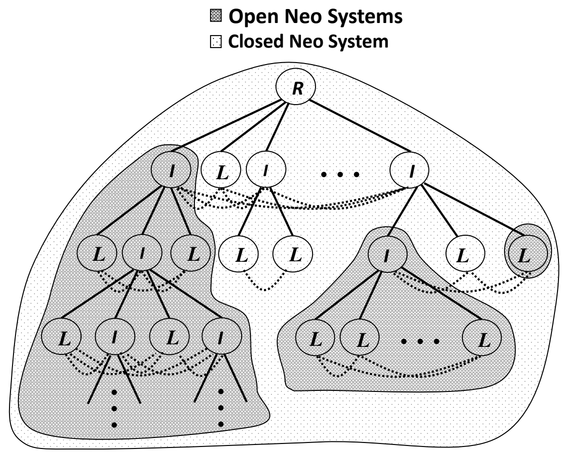

Figure 1: Illustration of a Neo Hierarchy.

power management, distributed lock management, etc.). The system organization Matthews et al. target is illustrated in Figure 1. There is one root node (R) and an arbitrary number of leaf (L) and intermediate nodes (I). These nodes could be, say, cache controllers in a coherence protocol or compute resources in a power management protocol. The hierarchy need not be balanced, as each root or intermediate node can have any number of children. Communication is permitted between parents and children (solid lines) and among siblings nodes (dotted lines).

At the core of the Neo theory is a systematic approach to breaking down a 2-dimensional hierarchical verification problem into the verification of a handful of flat subprotocols. The theory relies on a number of formal techniques to enable this approach. First, the Neo theory formulates a novel safety definition that takes a protocol's safety invariant and represents it in a hierarchical form. Second, the Neo theory provides a novel Safe Composition Invariant. This invariant helps guarantee that any hierarchy satisfying the Neo safety property can be repeatedly scaled by replacing a leaf with a subhierarchy while still satisfying the Neo safety property. Hence, we can construct hierarchies like that in Figure 1 while maintaining safety. Finally, the theory provides a mechanism for guaranteeing that the (hierarchical) Neo safety property implies safety for the particular protocol of interest.

In this section, we seek to provide the reader with an intuitive summary of the Neo theory in the context of coherence. Given that the theory is quite robust, we inevitably cannot present in this paper all the crucial details required for properly applying the theory. Hence, we refer the more interested reader to the Neo paper [24] for a more complete account.

In the following sections, we first define Neo Systems. Then, we discuss the formal techniques the theory employs to reduce the 2-dimensional verification problem to a 1-dimensional problem. Finally, we discuss how to verify Neo Systems in a fully automated model checker.

2.1 Defining Neo Systems

The Neo theory formally specifies a class of transition systems to which Matthews et al.'s parametric verification methodology can be applied. This class is called *Neo Systems*. There are 3 building blocks of Neo Systems such as the one in Figure 1:

(1) identical ***leaf*** **nodes** (*L*)
(2) multiple, possibly different, ***internal*** **nodes** (*I*)
(3) a single ***root*** **node** (*R*)

There are two kinds of Neo Systems: *Open Neo Systems* and *Closed Neo Systems*. Capped by an internal node, an Open Neo System is any strict subtree in a Neo hierarchy. An Open Neo System can communicate with the parent and siblings of the internal node. Capped by a root node, a Closed Neo System is a complete Neo hierarchy and cannot engage in any external communication. Within any Open (Closed) Neo System, the internal (root) node can communicate internally with each composed Open Neo System (illustrated with solid edges). Also, within any Neo System, all composed Open Neo Systems (siblings) can communicate internally among each other (illustrated with dotted edges).

Defined recursively, a leaf is an Open Neo System. A collection of an arbitrary number of Open Neo Systems composed with an internal node is an Open Neo System (the darker shading in Figure 1 highlights a few examples of Open Neo Systems). A Closed Neo System is constructed by composing an arbitrary number of Open Neo Systems with a root node (lighter shading in Figure 1). For nodes or Neo Systems B and D, we denote composing B and D by $B \odot D$. Then, formally, Open (Closed) Neo System $\Omega = A \odot \Omega_1 \cdots \odot \Omega_n$, where A is an internal (root) node and each component Ω_i is an Open Neo System.

This classification of Open and Closed Neo Systems is important because the Neo theory ultimately relies on proving that within any Neo System, all Open Neo Systems behave indistinguishably from a leaf node. The recursive definition is useful because the Neo theory relies on formulating invariants such that they apply on each level of the hierarchy. Also, note that a hierarchy can have multiple "flavors" of Open Neo Systems to allow for different behaviors in different levels of the hierarchy. This heterogeneity can be introduced by using different internal nodes to compose with the leaves (or other Open Neo Systems). We discuss heterogeneity further in Section 6.

2.2 Neo Safety Property

The Neo theory relies on a recursive safety definition that is formulated from a given protocol's safety invariant to reflect a hierarchical structure. We discuss this Neo safety definition here.

In Neo Systems, *states* capture *all* variables that can be changed in a transition. In a coherence protocol, this could include message buffers, cache data, coherence permissions, and sharer sets.

Cache coherence is typically defined in terms of coherence permissions of caches. For example, in a typical MOESI protocol, the invariant is that if a cache has a block in E or M, then every other cache must have the block in I [30]. The Neo theory formalizes safety properties by using *sum* functions to summarize states of Neo Systems into their appropriate protocol permissions (e.g., MOESI in coherence). Defined recursively, these *sum* functions take in states of internal or root nodes and summaries of component Open Neo Systems to summarize the state of the entire subhierarchy.

For a node or Neo System B, let *states*(B) denote the set of all states of B. Let P be the set of coherence permissions, including a

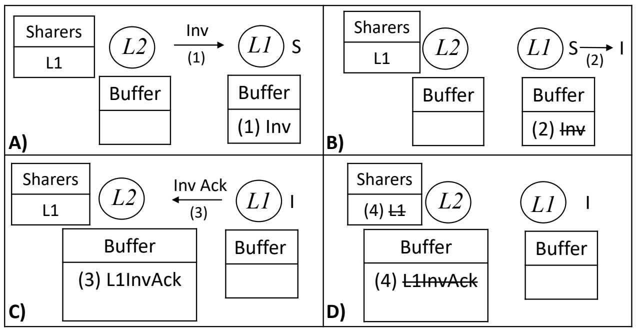

Figure 2: Illustration of Neo System Executions.

special term *bad* (e.g., $\{M, O, E, S, I, bad\}$). For any leaf L, $sum_L : states(L) \to P$ is a function that takes in a leaf state S and returns the coherence permission component of that state.

Let A be a root or internal node and $\Omega_0, \ldots, \Omega_{n-1}$ be Open Neo Systems. Then, for any (Open or Closed) Neo System $\Omega = A \odot \Omega_0 \odot \cdots \odot \Omega_{n-1}$, its summary function $sum_\Omega : states(A) \times sum_{\Omega_0}(states(\Omega_0)) \times \cdots \times sum_{\Omega_{n-1}}(states(\Omega_{n-1})) \to P$ takes in a state of the internal or root node and summaries of states of the component Open Neo Systems and returns an output from P. The theory requires the following of sum_Ω:

(1) if any of the component Open Neo Systems Ω_i summarizes to *bad*, then sum_Ω must output *bad*.
(2) if the summaries of the component Open Neo Systems constitute a safety violation, with respect to the protocol's original safety property, sum_Ω must output *bad*. For example, in a MOESI coherence protocol, if sum_{Ω_0} outputs E and sum_{Ω_1} outputs O, then sum_Ω must output *bad*.

The Neo Safety definition is as follows: A Neo System is said to be *safe* if no reachable state summarizes to *bad*. The above requirements force all safety violations anywhere within a subhierarchy to be captured by the output of the summary function of the Closed Neo System at the top of the entire hierarchy.

2.3 Safe Composition Invariant

The Neo theory formulates novel properties that enable one to guarantee that a subhierarchy (Open Neo System) can be used to scale a Neo System (by replacing a leaf) while still maintaining the Neo Safety property defined in Section 2.2. The key behind the Safe Composition Invariant is that it ensures that at "runtime," for certain "externally visible" properties of transitions, every transition of an Open Neo System Ω could be matched by a transition of a leaf node L, in which case we say Ω *implements* L.

To formalize the Safe Composition Invariant, we first define *executions*, which capture all possible behaviors that can be exhibited by a Neo System. Then, we describe how the Neo theory summarizes those executions to capture the essential properties relevant to maintaining correctness. After that, we can formalize the invariant.

2.3.1 Executions. Let Ω be a Neo system. Then, an execution e of Ω is a sequence $e = s_0, a_1, s_1, \ldots, s_{k-1}, a_k, s_k$. Each s_i is a state of Ω. s_0 is a start state (for coherence, that typically means empty message buffers and sharer sets and I permissions for cache blocks). Each tuple $t = (s_i, a_{i+1}, s_{i+1})$ forms a transition where Ω moves from state s_i to s_{i+1}, with *action* a_{i+1} on the edge of the transition. If Ω is sending a message to another node with transition t, then a_{i+1} is called an *output* action and t is an *output* transition. If it is receiving a message in transition t, a_{i+1} is an *input* action, which makes t an *input* transition. Otherwise, a_{i+1} is an *internal* action, in which case t is an *internal* transition, where Ω "silently" changes its state from s_i to s_{i+1}.

We illustrate executions in Figure 2. Here, an $L2$ cache controller is trivially composed with an $L1$ controller to form an Open Neo System $\Omega = L2 \odot L1$. Shown in Fig. 2A, L2's state includes the sharer set (initially containing only L1) and an incoming message buffer (initially empty), while L1's state includes its MOESI coherence permission (initially S) and an incoming message buffer (initially empty). At time (1), L2 sends an invalidation to L1. Also at (1), L1 makes a state transition—it changes its message buffer from empty to containing the invalidation message. At time (2) (Fig. 2B), L1 pops the invalidation message from its buffer and changes its coherence permission from S to I. At time (3) (Fig. 2C), L1 sends an invalidation ack to L2 and L2 makes a state transition to receive and buffer the message at the same time (3). At time (4) (Fig. 2D), L2 makes a final state transition to pop the invalidation ack from its buffer and remove L1 from its sharer set.

Let the states of $L2$ be represented in the form $(SharerSet, MessageBuffer)$ and the states of $L1$ in the form $(CoherencePermission, MessageBuffer)$. Let λ represent *internal* transitions of $L2$ and $L1$. Then, the execution of Ω illustrated in Figure 2 is:

$e_\Omega = ((\{L1\}, []), (S, [])), Inv,$
$((\{L1\}, []), (S, [Inv])), \lambda,$
$((\{L1\}, []), (I, [])), InvAck,$
$((\{L1\}, [L1InvAck]), (I, [])), \lambda,$
$((\{\}, []), (I, [])).$

2.3.2 Execution Summaries. To help define the composition invariant, the Neo theory uses *sum* functions to summarize executions as follows. Let Ω be any Open Neo System (i.e., a leaf or composite Open Neo System) and let $e = s_0, a_1, s_1, \ldots, a_k, s_k$ be an execution of Ω. Then, $sum(e)$ is a sequence that is generated as follows[1]:

- substitute each s_i with $sum_\Omega(s_i)$
- substitute each *internal* a_i with the symbol λ.

Observe that $sum(e)$ now contains precisely the information about the execution that is pertinent to safety and interactions with systems Ω is composed with.

2.3.3 Implementation Relation. Now we are ready to formally state the Safe Composition Invariant, which is as follows. For every execution e_Ω of Open Neo System Ω, there must exist an execution e_L of leaf L such that $sum(e_L) = sum(e_\Omega)$. When this property holds, Ω is said to *implement* L.

2.4 Mapping Neo Safety to Coherence

The Neo definition of safety is quite abstract and captures a wide range of safety invariants. However, we are specifically interested

[1]Matthews et al. present a more relaxed definition of *sum* in the Neo theory. However, they use the stricter definition we present here for their case study because it is more amenable to model checking.

in coherence invariants. Hence, we must choose *sum* functions such that they output *bad* whenever there is a coherence violation anywhere in a subhierarchy. In their case study, Matthews et al. use a simple summary function which we believe would cover most coherence protocols, including the NeoMESI protocol that we design and verify (Section 3).

Let Open Neo System $\Omega = A \odot \Omega_0 \odot \cdots \odot \Omega_{n-1}$, where A is an internal node and each Ω_i is an Open Neo system. Keep, as part of the state of A, a variable, *Permission*, that ranges over the permission set P.

Then define the coherence summary function $sum_c(\Omega) \equiv Permission$, where the sum_C function always outputs the value of *Permission*. For a leaf L, $sum_c(L)$ simply returns its coherence permission.

Additional requirements must be added to guarantee that sum_C catches all coherence violations (i.e. $Permission \neq bad$ implies the subhierarchy below Ω is coherent). Given the partial ordering $<$ of coherence permissions—$I < S, O < M, E < bad$—these properties are as follows:

(1) *Permission* of Ω must always be greater than or equal to the summary of each Ω_i.
(2) for two distinct Ω_i and Ω_j, if Ω_i summarizes to E or M and Ω_j does not summarize to I, *Permission* of Ω must return *bad*.

2.5 Neo Verification Methodology

The goal of the Neo theory is to verify that every Neo System satisfies safety. The Neo safety definition and Safe Composition Invariant above are not much help if we still have to pass full Neo systems into model checkers to verify them, as model checkers cannot handle hierarchical protocols. However, the Neo theory was able to prove that one need only pass *flat* Neo Systems into a model checker to verify properties that guarantee safety of any arbitrary hierarchical Neo System.

Matthews et al. prove that any arbitrary Neo System one constructs satisfies Neo safety, given the following two antecedents:

Antecedent 1: Let A be an *internal* or *root* node and L be a leaf node. Then, flat Neo System $\Omega = A \odot L \odot \cdots \odot L$ satisfies Neo safety.

Antecedent 2: Let A be an *internal* node, L be a leaf node and flat Open Neo System $\Omega = A \odot L \odot \cdots \odot L$. Then, Ω *implements* L.

If the above two antecedents hold, then one can construct any Neo system like Figure 1 and it would satisfy Neo safety. Coupled with proving specific properties that connect Neo safety to a protocol's safety invariant (Section 2.4), that would imply that any Neo System satisfies the protocol's safety invariant.

The Neo theory presents a methodology to prove in an automated parametric model checker that the flat Neo Systems satisfy the two antecedents above and the additional coherence-specific properties for the sum_C summary function in Section 2.4. Parametrically verifying Neo Systems involves two tasks: verifying that Neo Safety holds and verifying that the Safe Composition Invariant holds. We discuss these tasks below.

2.5.1 *Verification of Neo Safety.*

To verify Neo safety of the flat Neo Systems, one must first model them in a parametric model checker. It is crucial that the models of the flat Open Neo Systems

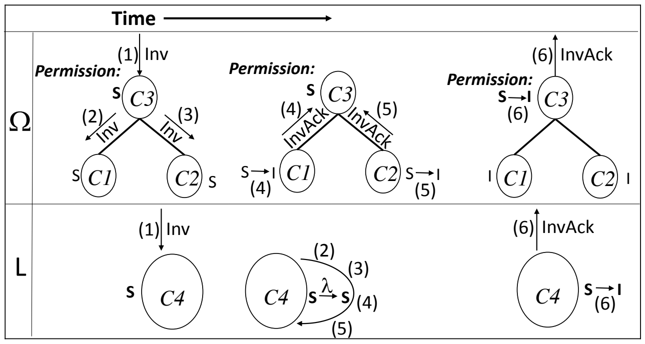

Figure 3: Illustration of Proof of Implementation Relation.

also contain *all* input and output transitions, which are used for communication with other subsystems they are composed with.

Every transition in the model checker must have a unique name, which corresponds to the *actions*. One must then have the model checker verify that, in every transition, *Permission* (the output of sum_C) never evaluates to *bad*. One must also have the model checker verify the coherence-specific properties for the sum_C summary function (specified in Section 2.4).

2.5.2 *Verification of Safe Composition Invariant.*

The second antecedent in Section 2.5 requires that we prove that a flat Open Neo System *implements* a leaf. Intuitively, this proof is done by modeling a leaf and each Open Neo system in parallel and showing that, for every transition of the Open Neo system Ω, there exists a matching leaf L transition. In Section 4.1, we describe Matthews et al.'s specific approach to this proof and our optimizations to it. Here, we provide some intuition.

In Figure 3, we illustrate this proof by showing how L matches Ω's transitions in satisfying an invalidation request. Ω is a flat Open Neo System with internal node (or directory) C3 composed with leaf nodes (cache controllers) C1 and C2. The leaf L that Ω must implement is simply a cache controller C4. At time (1), Ω makes a transition on an *input* action *Inv* and L matches the transition. From times (2) through (5), C3 collects invalidations from its children, which all constitute actions internal to Ω. L matches these transitions by simply stuttering on internal actions (λ) and keeping its coherence permission the same. Finally, at time (6), Ω sends an *output* action *InvAck* via C3 and changes *Permission* from S to I, which is matched by L.

3 NeoMESI: DESIGNING A NEO COHERENCE PROTOCOL

In designing verification-aware coherence protocols, the role of the architect is to adhere to the requirements about transitions, composition, communication, organization of nodes, etc., specified by the Neo theory. The architect must also bake into the design Neo properties such as the Safe Composition Invariant and the recursive Neo Safety property. This then produces a protocol amenable to fully automated hierarchical parametric verification.

There are many classes of coherence protocols one could design with the Neo theory. As a contribution of this work, we illustrate the design process by picking one concrete class of protocols—tree directory protocols—and designing and verifying a realistic protocol from that class. We call our protocol NeoMESI.

There are several limitations that are imposed by both the theory and the model checker that could prohibit certain features from a Neo protocol. Specifically, it was not clear how the Neo methodology scales with protocol complexity. In fact, we found that using the Neo verification methodology as is, we could not verify a baseline MSI tree directory protocol. Hence, as part of our contribution, we modify the Neo verification methodology to be more scalable, which we discuss in Section 4.1. Additionally, we employ a systematic approach to designing NeoMESI to get some insight into what features current formal verification theory and tools can handle. We started from the baseline MSI tree directory protocol and iteratively added optimizations until the theory or tools rendered the protocol unverifiable with the Neo methodology. NeoMESI was the last verifiable version in this process. We defer the discussion of this process to Section 4. In this section, we describe the final NeoMESI protocol.

NeoMESI is an inclusive hierarchical directory cache coherence protocol where each subtree is rooted by a directory subprotocol that provides MESI permissions for its children. Each directory may or may not be collocated with a cache (i.e., L2, L3, etc), depending on the architects' preferences. NeoMESI allows nodes sharing the same directory to forward data to each other. Note that the protocol does not assume symmetry or balance in the tree hierarchy as it is verified to be correct in all configurations of the tree.

As we designed NeoMESI, we ensured there was a direct mapping between each NeoMESI node and nodes in the Neo generic transition system illustrated in Figure 1. Each private cache controller is a leaf *L*, each intermediate directory controller is an internal node *I* and each root directory controller is a root node *R*.

We need to design two subprotocols with which to scale our hierarchy: a Closed Neo System and an Open Neo System. Note that for heterogeneous protocols, we would need to design as many Open Neo Systems as needed. We discuss heterogeneity in Section 6.1.

3.1 Closed Neo System

For our Closed Neo System, we use a 1-level MESI directory protocol that has a number of L1 caches composed with a root directory. The L1 caches can forward data to each other.

3.2 Open Neo System

To get an Open Neo System, we keep the L1 caches the same from the Closed Neo System but make a number of modifications to the root directory to make it into a Neo internal node (or internal directory). We modify the transitions of the root directory such that, to satisfy certain requests, the intermediate directory is able to communicate with a parent directory indistinguishable from how an L1 cache communicates with a directory. The intermediate directory is also able to forward data to its siblings (L1 cache controllers or other intermediate directories).

We instantiate in the intermediate directory the variable *Permission* that keeps track of what coherence permission appears to be held by the entire subhierarchy below the intermediate directory. As discussed in Section 2.4, we enforce the simple principle that no child of an intermediate directory can have higher coherence permissions than that of the variable *Permission* according to the partial ordering: $I < S, O < E, M$. We will refer to this as the *permission principle*.

Before a block is allocated by the intermediate directory, *Permission* is initialized to *Invalid*. Upon receiving a request, the directory may satisfy it within the subhierarchy if *Permission* is high enough. Otherwise, it must relay the request to its parent directory to avoid violating the permission principle.

Figure 4 helps illustrate how requests are satisfied in NeoMESI. The illustration starts with L1 cache controllers C1 through C4 in coherence states I, I, I, and M, respectively. Intermediate controller C5 has *Permission* in I and intermediate controller C6 has *Permission* in M. Events such as sending messages and changing states are numbered to indicate the order in which they occur.

At (1), C1 sends a GetShared (GetS) request to its parent directory C5. Because C5 has *Permission* in I, it must relay the request to its parent directory C7 (2). Note that C5 and C7 become blocked to all requests at (1) and (3), respectively, because NeoMESI assumes an interconnection network that does not support point-to-point ordering, as discussed in Section 4 below. The request is forwarded down the hierarchy until it gets to C4 at (4). At (5), C4 goes from state M to S and sends the data to its parent controller C6. Since sibling-sibling communication is allowed, C6 can send the data to C5 at (6) and C5 can change its variable *Permission* to S at (7). C5 can finally send the data to C1. At (9) and (10), respectively, C1 and C5 finally send Unblock messages to their parents, which both unblock the parents and update them with the valid data.

4 THE PROCESS OF DERIVING NeoMESI

Upon embarking on this work, it was not clear how the Neo methodology applies to more realistic protocols than the toy NeoGerman that Matthews et al. verified. There are two factors that constrain what features are possible with Neo protocols: the theory itself and the scalability of verifying Neo protocols in a model checker. Note that the latter factor depends both on the model checking tool and the Neo verification methodology. In the course of this work, we indeed uncover scalability issues with the Neo verification methodology. Specifically, in attempting to use the Neo methodology to verify a baseline tree directory protocol that composes a simple MSI directory protocol [30] into a hierarchy, we find that the model checker hangs after exhausting 200GB of memory. As part of our contribution in this work, we propose modifications to the verification methodology to improve its scalability (Section 4.1). Furthermore, we iteratively add features to the tree MSI directory protocol to see how they affect verifiability (Section 4.2). This helps provide a snapshot on whether or not current formal verification theory and tools can handle certain desirable protocol features.

As our parametric model checker, we use Cubicle [11], which represents the state of the art. We run Cubicle on a 2.4GHz Intel® Xeon® processor, setting a time bound of 2 days and a memory bound of 50GB.

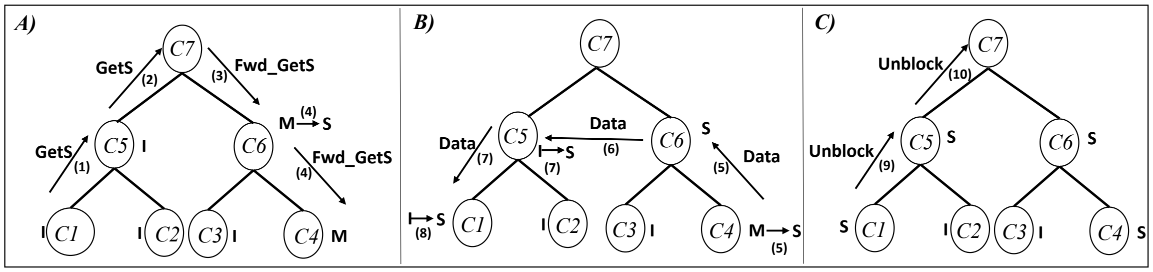

Figure 4: Illustration of NeoMESI satisfying a coherence request.

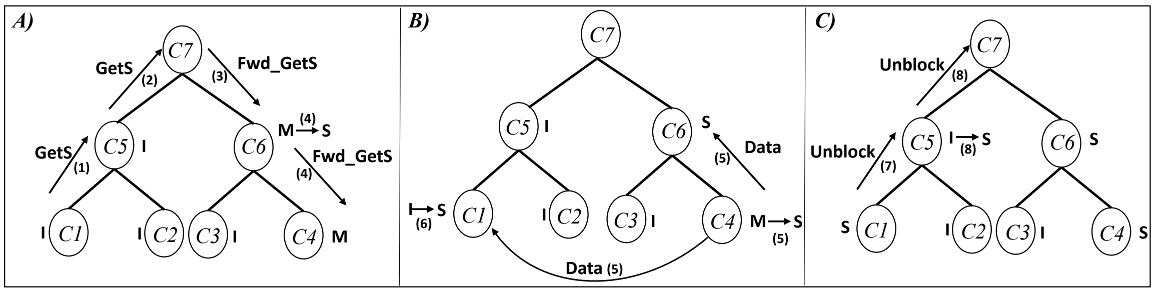

Figure 5: Illustration of NS-MESI satisfying a coherence request.

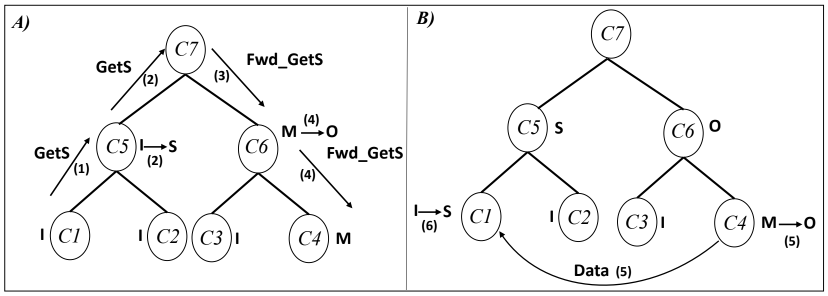

Figure 6: Illustration of NS-MOESI satisfying a coherence request.

4.1 Modifying the Neo Verification Methodology

First, we describe in more detail the Safe Composition Invariant verification methodology. Then, we discuss its scalability problems. Finally, we propose modifications to it.

4.1.1 Initial Approach. Recall that in order to verify the Safe Composition Invariant (Section 2.5.2), Matthews et al.'s verification approach requires us to model the leaf L and Open Neo System Ω side by side to ensure that L can match each Ω transition. Hence, like them, we forced the model checker to strictly alternate between an L transition and an Ω transition, starting with an Ω transition. After each Ω transition (L transition), we updated a variable *Ω_action* (*L_action*), which represents the action of the transition; the variable is set to λ if the transition's action is internal. Also, after each Ω transition, a variable *Ω_just_ran* is set. The model checker is made to verify that the following expression is invariant:

$$\Omega\_just\_ran \rightarrow (\Omega\_action = L\_action \wedge \Omega\_Permission = L\_Permission) \quad (1)$$

where *Ω_Permission* represents the *Permission* variable of Ω and *L_Permission* represents the coherence permission of L.

Matthews et al. observe that the model checker could terminate after reaching an Ω transition after which no L transition can fire (i.e., all L transitions have their guards evaluate to false). While this would imply that the Safe Composition Invariant failed, the model checker would falsely report that *all* properties passed. To avoid this situation, they also require one to check that at least one L transition can fire after an Ω transition. They do this by asking the model checker to verify as invariant the logical disjunction of the guards of *all* L transitions, given that an Ω transition has just run. Where *Ω_just_ran* represents a variable that is set after each Ω transition and L_guard_i is the guard expression of L transition i, the invariant would be expressed as follows:

$$\Omega\_just\_ran \rightarrow (L\_guard_1 \lor L\_guard_2 \lor \cdots \lor L\_guard_n) \quad (2)$$

4.1.2 Inefficiency of Initial Approach. For several reasons, the above methodology was not scalable beyond a toy protocol like NeoGerman. First, forcing the model checker to alternate between Ω and *L* transitions leads to a much larger reachable state space than that of the original subprotocol Ω. Consequently, the memory consumption required to keep track of visited states is significantly increased. Having more transitions to explore (the *L* transitions) also increases the runtime. Second, observe that the formulation in (2) will typically be a complex logical expression. Each L_guard_i is already a complex expression that determines when a transition should fire, and there could be dozens of leaf transitions, unlike the handful in NeoGerman. For every state to be explored, the model checker must check if it satisfies the invariants, which is a time consuming problem for an expression like (2).

4.1.3 Modifications to the Methodology. We observe that, to enable the model checker to pick the *right L* transitions at runtime, the Neo methodology already requires one to statically identify which *L* transition should match each Ω transition. We grant that we cannot statically determine if the *L* transition guards can evaluate to true in order to match a given Ω transition and we cannot statically determine what the values of *Permission* variables will be after each transition. Nonetheless, we can leverage the static matching to simplify the invariant (2) and reduce the number of transitions.

First, instead of alternating between Ω and *L* transitions, we embed in the body of each Ω transition the corresponding *L* transition. The state updates of the *L* transition are made if the guard expression of the *L* transition evaluates to true. Finally, a variable *L_could_fire* is updated to the value of the guard expression of *L*. These instrumentations are done automatically with a script. Our approach has similarities with Park et al.'s [26], but we verify different types of implementation relations, our proofs are in a parametric setting, and we cannot update *L*'s state with functions, as Cubicle does not provide that support. Our setup is illustrated in the psuedocode below.

```
transition Ω_action (<Ω guard expression>) {
  Ω transition body{
    <Ω state updates>
    if (<L guard expression>) {
      <L state updates>
    }
    L_could_fire:=<L guard expression>
  }
}
```

Second, instead of the complex expression in (2), we ask the model checker to prove the following is invariant:

$$L\_could\_fire = true \quad (3)$$

To compare our modifications with Matthews et al.'s original approach, observe that after every Ω transition, we effectively reduce (2) to $\Omega\_just\_ran \rightarrow L\_guard_i$ (involving only one *L* guard), which is, in fact, logically stricter than the original expression. Intuitively, our scalability advantage lies in the fact that, before model checking, we already know which L_guard_i would allow the appropriate *L* transition to fire (if one exists). So, we can use a bolder, but simpler, logical expression for the model checker to verify at each step. Also, observe that our modification to reduce the number of transitions updates the states of *L* in the same way as they would be updated in a separate transition. *L* states are updated only if the guard conditions evaluate to true.

4.2 Iteratively Adding Features

After making these modifications to the Neo verification methodology, we were able to successfully verify the tree directory protocol composed from an MSI directory protocol [30] on each level (discussed above). This was the first protocol in our iterative design/verification process. This initial protocol does not support non-blocking directories or a fully inclusive cache hierarchy (e.g., it does not support explicit eviction notifications). We then iteratively added features and attempted to verify the protocol with our modified methodology at each step. In the sections below, we outline our findings for each feature we sought to add.

4.2.1 Related to Theory. In a hierarchical protocol, a feature that architects could be interested in is having, say, caches that do not share a parent directory directly forward data to each other [38]. This direct data forwarding could reduce the latency of satisfying requests and network bandwidth. Unfortunately, all direct non-sibling communication is explicitly prohibited by the Neo theory. Such communication significantly complicates the theory and verification methodology, so Matthews et al. deferred enabling non-sibling communication to future work.

4.2.2 Related to Model Checking Scalability. We considered three features that could affect whether or not the verification of a protocol can be completed within reasonable bounds of time and hardware resources: non-blocking directories, explicit eviction notification to parent directories, and coherence permissions (specifically, presence of the E and O states). Note that these features are compatible with the Neo theory itself.

Non-blocking Directories: When a directory has a number of requests buffered, it is preferable for the directory to process the requests back-to-back by sending data or forwarding the request, instead of blocking to wait for completion messages from the recipients between requests. It is challenging to design deadlock-free protocols with non-blocking directories if the underlying interconnection network does not provide the guarantee of point-to-point ordering [30].

Unfortunately, modeling ordered buffers requires more complex data structures than what Cubicle supports. This restriction is deliberate by the developers because it would otherwise make the model checking intractable by significantly increasing memory consumption and sabotaging the optimizations in the underlying model checking algorithm. Such restrictions on complex data structures are typical among parametric model checkers [16, 32, 40].

Support for Inclusive Cache Hierarchy: To obviate the need to explicitly represent the coherence state of all blocks in memory, many protocol designs opt for an inclusive cache hierarchy [23, 30]. Two common optimizations help improve the performance of inclusive protocols. First, before evicting a block, a cache controller first evicts the block in all its children that have it cached. Second, upon evicting a block, cache controllers send explicit eviction notifications to their parent directories [14, 23]. These optimizations add additional transitions to the model checker, which both increase memory consumption and verification time.

Fortunately, we found that we were able to have a successful verification even with this feature and optimizations present.

Presence of the E and O States: Many modern coherence protocols have the E(xclusive) and O(wned) states. Unfortunately, these states add more transitions to the protocol, which increases the memory consumption and verification time.

While we were able to add the E state and get a successful verification, we found that the model checker could not handle the O state.

4.2.3 Summarizing Findings. In summary, we found that, to keep our protocol verifiable within the time and memory bounds, were able to add only two optimizations: the E state and support for fully inclusive hierarchies. Two features the model checker could not handle were the O state and non-blocking directories. We were also unable to add non-sibling communication because it is explicitly prohibited by the Neo theory.

5 EVALUATION

In this section, we experimentally evaluate our formally verified NeoMESI protocol. While performance is not the focus of this paper, we still want to determine the effects that the optimizations prohibited by the Neo theory and verification tools have on performance. Hence, we designed two protocols that feature these optimizations so that we can compare them against our verifiable NeoMESI. Note that these two protocols are unverifiable with current model checking tools, given that we are unaware of any other theory that enables push-button parametric verification of hierarchical protocols. We describe these protocols in Section 5.1, our experimental methodology in Section 5.2 and our results in Section 5.3.

5.1 Comparison Protocols

In Section 4, we discussed optimizations that we could not add to NeoMESI because they either violated the Neo theory or made the model checker exhaust time and memory bounds. In this section, we describe NS-MESI and NS-MOESI—two protocols we designed by adding these prohibited optimizations to NeoMESI.

5.1.1 NS-MESI: Adding Non-sibling Communication. As discussed earlier, the Neo theory explicitly prohibits non-sibling communication, which precludes data forwarding between cache controllers that do not share a parent. We make modifications to NeoMESI that violate this principle and call the new version NS-MESI.

Figure 5 illustrates how requests are satisfied in NS-MESI and we compare against NeoMESI (Figure 4). Events at times (1) through (4) are identical for NeoMESI and NS-MESI. However, in NS-MESI, upon receiving the forwarded GetS, C4 directly sends the data both to its parent (the new owner) and C1. This saves a hop. As with NeoMESI, Unblock messages are sent after C1 receives the Data, updating C5 and C7 with the valid data. However, observe that C5 now changes *Permission* from I to S after an Unblock message from C1 at (8), instead of after a Data reply from C6.

Table 1: Simulation System Configurations

Processor and OS	
Cores and ISA	32 in-order x86 cores
Frequency	2GHz
OS	Linux
Memory Hierarchy	
Inclusivity	Fully Inclusive Hierarchy
Cache Block Size	64 Bytes
L1 I&D Caches	32KB, 2-way, 2-cycle
L2 Cache	4MB, 8-way, 6-cycle, Unbanked
L3 Cache	64MB, 16-way, 16-cycle, Unbanked
DRAM	2GB, 160-cycle
Network	
Link Bandwidth	32GB/s
Link Latency	1-cycle

5.1.2 NS-MOESI: Adding the O State and Eliminating Blocking. Next, we add to NS-MESI the optimizations that we could not verify NeoMESI with within reasonable time and memory bounds. Those optimizations are the O state and eliminating blocking in directories. We call this version NS-MOESI.

Figure 6 illustrates how requests are satisfied in NS-MOESI. Events from time (1) through (4) are identical between NS-MESI and NS-MOESI, except for the fact that, upon receiving the forwarded GetS at time (4), C6 can immediately change *Permission* to the newly added O state, without blocking. Upon receiving the forwarded GetS, C4 can transition to O and directly forward the data to C1. Observe that C4 need not send the data to its parent C6, since C4 remains the owner of the data.

5.2 Experimental Methodology

Given that NeoMESI is verified for all tree configurations, we sought to evaluate all the protocols across three cache hierarchy organizations to see if the organizations expose some trends. Across all organizations, illustrated in Figure 7, there are 3 levels of caches: L1, L2, and L3. The L2 and L3 caches are collocated with directories that maintain tags for their children. The *Skewed* organization (Fig. 7A) features an *asymmetric* hierarchy, where there is a unified L3 cache, 16 cores have private L1 and L2 caches and the other 16 cores share an L2 cache. The *2 Cores per L2* organization (Fig. 7B) has a unified L3 cache and 16 L2 caches, each shared by 2 cores with private L1 caches. The *8 Cores per L2* (Fig. 7C) organization has a unified L3 cache and 4 L2 caches, each shared by 8 cores.

We implemented NeoMESI, NS-MESI, and NS-MOESI in the gem5 full-system simulator [7]. Other than the cache hierarchy organization, we maintained the same system configurations for all experiments, summarized in Table 1. We ran 7 benchmarks from the PARSEC suite [6] with each protocol and each cache organization, for a total of 63 experiments. Each experiment was run multiple times to account for the expected variability in multithreaded

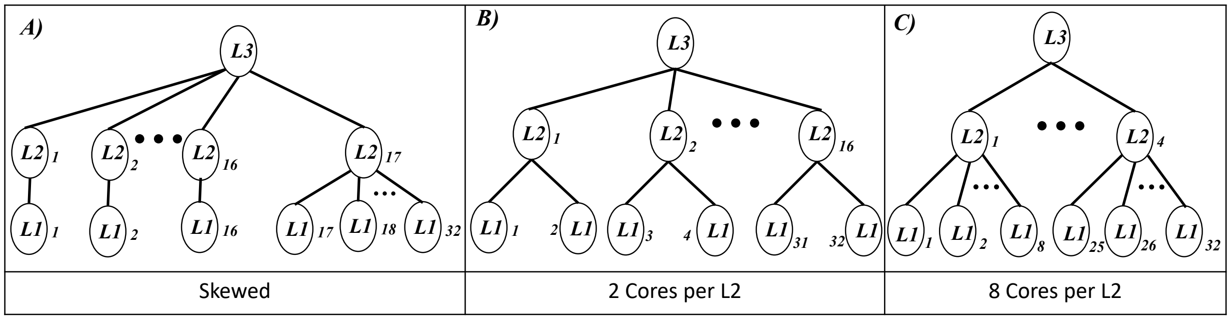

Figure 7: Cache organizations.

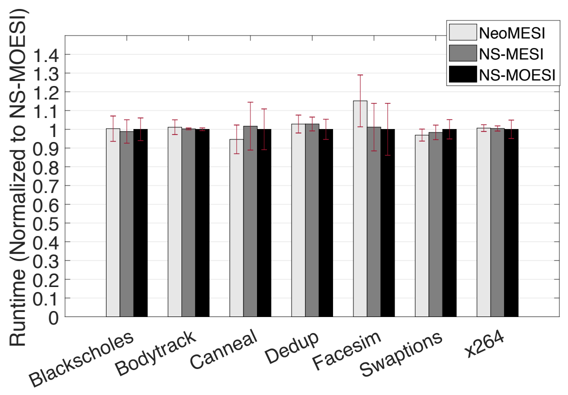

Figure 8: Runtime with 2 Cores per L2 Organization.

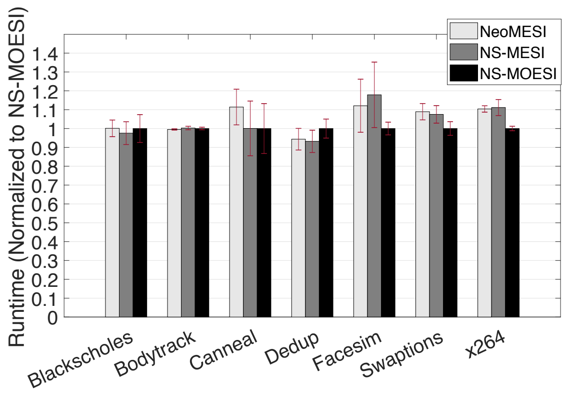

Figure 9: Runtime with 8 Cores per L2 Organization.

workloads [1]. The data we present represents the average of each experiment, with error bars showing plus or minus a standard deviation.

5.3 Results

The runtimes of the experiments are grouped by cache hierarchy organization and shown in Figures 8, 9, and 10. Comparing the protocols while holding benchmark and cache organization constant, we find that NeoMESI generally appears to be edged out by

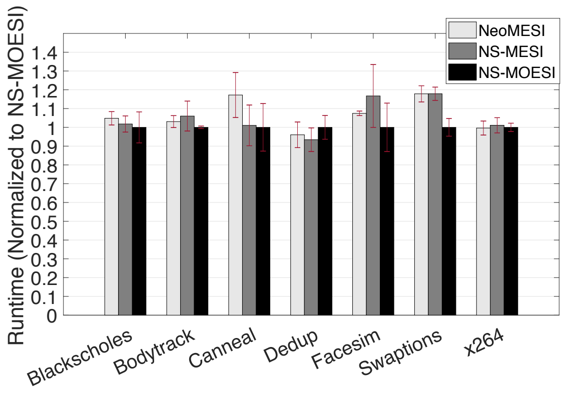

Figure 10: Runtime with Skewed Organization.

both NS-MESI and NS-MOESI. However, these advantages are not statistically significant. In fact, for some benchmarks and cache organizations, NeoMESI outperforms NS-MOESI (e.g., Canneal and Swaptions with 2 Cores per L2 in Figure 8) and outperforms NS-MESI (e.g., Bodytrack and Facesim with Skewed in Figure 10). However, in these situations, the advantages are also not statistically significant.

To investigate why the optimizations in NS-MESI and NS-MOESI seem to have little effect on the performance, we study how frequently these optimizations are utilized across all the benchmarks. We would expect non-sibling communication to reduce the latency of satisfying requests. However, we find that, for NS-MESI, the fraction of requests that missed in the L1 and are satisfied using non-sibling communication is only 1.5%. Presumably, because the O state enables L1 caches to retain ownership longer, that fraction increases in NS-MOESI, but to only 2%.

Additionally, we would expect the relaxation of blocking in the L2 and L3 to improve the performance of NS-MOESI over both NeoMESI and NS-MESI. However, we find that, among requests arriving at the L2, the fraction of blocked requests is only 0.4% and, for those arriving at the L3, that fraction is 0.7%.

We note that we can discuss only results with respect to the specific benchmarks and system configurations with which we experimented. However, we recognize that with different benchmarks and systems, the prohibited optimizations could potentially incur more significant performance penalties.

6 BROADER APPLICABILITY OF NEO

The Neo theory focused on solving the challenging problem of automated verification of hierarchical protocols. To leverage the Neo methodology, one must design a protocol to fit the Neo class of transition systems. While hierarchical directory protocols such as NeoMESI seem a natural fit, the theory provides little insight on how a variety of system configurations and architectures can be modeled as Neo Systems. For our final contribution in this work, we examine different types of coherence protocols, network topologies, and cache organizations that architects might be interested in and discuss how they can be modeled as Neo Systems.

6.1 Heterogeneous Protocols

In large shared-memory systems, architects might be interested in employing heterogeneous coherence protocols to enable scalability and design simplicity [4, 5, 19]. In multi-chip processors, for example, coherence protocols on each chip have been integrated into larger inter-chip coherence protocols [3, 18, 22].

Observe that heterogeneous protocols would have cache controllers from different subprotocols that potentially behave differently. However, the Neo theory requires leaves to be identical. Within the bounds of the Neo theory, we present a solution to this apparent disparity. One can simply model each leaf as containing the behavior of all possible leaves. Upon initializing the protocol, the directories (which are allowed to be different internal nodes) initialize the leaves to which they are composed so as to make the leaves behave as is appropriate for that subprotocol. Note that this solution does not affect the scalability of verification of safety or the *implementation relation*. This is because the model checker would, at runtime, not traverse the superfluous states and transitions of leaves after the leaves have been initialized by their intermediate or root nodes to the partition of state transitions that is needed for the appropriate subprotocol.

6.2 Snooping Protocols

Snooping protocols are a class of coherence protocols that rely on an ordered broadcast network, typically a bus, as the ordering point for coherence requests [9, 29]. In typical snooping protocols, cache controllers broadcast requests to all controllers via a shared bus. The bus produces an total ordering of all requests for each cache block and is snooped on by all controllers. This ordering, observed by all controllers, determines the coherence permissions that each controller has.

Architects might be interested in using the Neo theory to compose snooping subprotocols into larger, verifiable hierarchies [39]. In order to verify the Neo properties for each flat subprotocol, as is required by the theory, one must model broadcast and snooping behaviors in Neo Systems. At a first glance, Neo Systems seem to only support unicast communication occurring in single transitions. We provide an approach for modeling snooping within the bounds of the theory. The key is to model the behaviors of the broadcast network in internal and root nodes. The internal or root node would collect all requests from the controllers (via *input* transitions), order and buffer the requests, then send messages to each controller through a string of *output* transitions, thus completing a broadcast/snoop cycle.

6.3 Ring Protocols

Ring interconnection networks appear in several processors such as IBM Power 4 [33] and Intel's Larrabee Microarchitecture [28] because they are often cost-effective and simple to implement. Coherence protocols built on such interconnects can leverage the ordering properties that ring topologies naturally provide. For example, in unidirectional rings where two requests for a block cannot bypass each other, one node can simply be made the ordering point at which a total ordering of requests is made. That node would be responsible for marking requests as *active*.

Architects might be interested in composing flat rings into a hierarchy of rings or a ring of rings as in KSR1 [8]. Recall that in Neo Systems, leaf nodes are identical and communication is symmetric among all siblings. However, coherence protocols in rings typically require unidirectional communication, where each node sends messages only to the left or right node. This unidirectional communication can be modeled in Neo Systems by storing, as part of the leaf state, a variable that holds the index of the next node in the ring and a Boolean that indicates if a leaf is the ordering point. These variables would determine which node each leaf communicates with and which node is responsible for marking requests as active. Also, encoded in the initial state of the internal nodes would be the values of these variables for every leaf. The initial transition of the internal node would then be to send messages to each leaf to instantiate these variables appropriately.

6.4 Non-inclusive Cache Hierarchies

While NeoMESI featured an inclusive cache hierarchy, several processors have used non-inclusive cache hierarchies to maximize the total cache capacity [17, 20]. In non-inclusive cache hierarchies, it is not required that a block cached in an upper level cache (say, L1) be also cached in a lower level cache (say, L2). The Neo theory, on the other hand, requires that each subhierarchy (Open Neo System) *implement* an L1 (leaf), which might appear to necessitate an inclusive hierarchy. We argue that non-inclusive hierarchies are also permitted by the Neo theory. Observe that, to maintain the Neo Safety property and the Safe Composition Invariant, one only needs to keep track of permissions and indices of sharers and owners in subhierarchies, i.e., the directory state. The actual data of the cache blocks in the states of internal nodes and leaves do not factor into the maintenance of these properties. So, while metadata (the directory state) still needs to be inclusive to maintain the Neo properties, cache data management and evictions will be based on the underlying cache properties such as cache size, associativity, and replacement policies.

6.5 Banked Shared Caches

In a many-core processor with large shared caches, splitting the shared caches into banks can be useful for reducing the data access latency and alleviating the bottleneck that one centralized cache controller could otherwise cause. Typically, the address space is statically partitioned and each bank services requests from only one partition. We believe there is no theoretical limitation in modeling as a Neo System a coherence protocol that supports banking. One can simply think of the protocol as a collection of multiple independent Neo Hierarchies, one for each bank. These independent Neo hierarchies would not interact with each other. As long as each one is verified, the system can have an arbitrary number of them.

Nonetheless, one could imagine a system configuration that would allow addresses to dynamically change partitions and be cached in different banks. It is not clear if the Neo theory supports such systems. Fortunately, we are not aware of any systems that allow this behavior.

7 RELATED WORK

In Fractal Coherence [41], Zhang et al. propose designing coherence protocols with self-similarity imposed in the logical organization of the nodes. Then, one can model check the protocol for only a small number of nodes but, by induction, guarantee correctness for an arbitrary number of nodes. Voskuilen et al. propose optimizations to the protocol implemented in Fractal Coherence [36, 37]. They provide flat, directory implementations of Fractal Coherence that allow optimizations such as request forwarding and parallel invalidations. In PVCoherence, Zhang et al. show how one can design realistic flat coherence protocols to fit a mostly-automated parametric verification method [40]. Finally, Matthews and Sorin et al. adapt the Fractal approach to designing verifiable dynamic power management protocols [25, 31]. However, none of these works satisfy our goal of designing realistic hierarchical coherence protocols to be formally verifiable for all tree configurations.

The Manager-Client Pairing (MCP) framework sought to enable the composition of pre-verified heterogeneous coherence protocols into a unified hierarchy [4, 5]. *Manager* agents that manage coherence permissions (e.g., directories) are paired with *client* agents (e.g., private caches) of a higher tier in the hierarchy. Coherence is maintained across the hierarchy through a permission-checking interface with which managers and clients must communicate to satisfy requests. However, MCP is theoretically flawed [24]. MCP assumes that one need only verify each subprotocol to guarantee the correctness of the hierarchy. However, each subprotocol is pre-verified *without* any interactions with the permission-checking interface. Hence, there is no formal guarantee about their behavior when they are composed together and forced to interact with the permission-checking interface. Moreover, the MCP framework does not enable integrating protocols that could each have an arbitrary number of nodes, which would be required to support arbitrary tree configurations.

Ros et al. explore ways of reducing the complexity in hierarchical coherence protocols [27]. Their approach is to eliminate complex operations such as recursive invalidations in favor of self-invalidations and write-throughs within a cluster. Unlike them, our focus is on designing hierarchical protocols specifically to be verifiable. While design simplicity generally improves verifiability, hierarchical protocols such as theirs are still well beyond the reach of current automated verification tools.

Ladan-Mozes et al. present Hierarchical Cache Consistency (HCC) [21], which embeds a coherence protocol on a fat-tree interconnection network. HCC guarantees forward progress by enforcing properties that coordinate coherence permissions between parents and children and ensures that a tree hierarchy is adhered to in all communication between cores and any memory bank. HCC was verified *manually* and its proof applies to only one protocol. However, our focus is on showing how one can architect a wide range of coherence protocols to enable their push-button verification.

Vijayaraghavan et al. [34, 35] use a theorem prover to verify a specific coherence protocol for arbitrary tree configurations. They concede that theorem proving requires tremendous manual effort and time by verification experts. But they argue that this disadvantage is offset by the fact that, at the time of their work, model checkers could not support parametric verification of hierarchical protocols. Their protocol was much simpler than our NeoMESI, with only MSI permissions and no data forwarding allowed among sibling nodes, among other simplifications. Nonetheless, their protocol required 12,000 lines of theorem prover code to be verified. Moreover, future theorem proving based approaches are still likely to require grueling manual effort. On the other hand, our work seeks to enable the design of future hierarchical coherence protocols to be amenable to fully-automated (push-button) parametric verification for arbitrary configurations of the hierarchy.

8 CONCLUSION

In this work, we showed how the recent Neo theory can be used to design realistic hierarchical coherence protocols specifically to be verifiable with fully automated formal verification tools. As such, we presented NeoMESI—the first realistic hierarchical coherence protocol to have these highly desirable properties. In designing and verifying NeoMESI, we found that certain desirable optimizations render protocols unverifiable with the Neo theory and current verification tools. We evaluated NeoMESI to determine the effects of these optimizations, comparing it to protocols that have these optimizations. We found that our verifiable NeoMESI performs statistically on-par with these protocols. Finally, we showed how the Neo theory can be used to model other kinds of protocols that may not appear as natural a fit as NeoMESI. We hope that our work in this paper can enable future work in verification-aware coherence protocol design using the recent theoretical developments of the Neo theory.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under grant CCF-142-1167. Additionally, this work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We thank Jesse Bingham for his input and advice on this work.

REFERENCES

[1] Alaa R Alameldeen and David A Wood. 2003. Variability in architectural simulations of multi-threaded workloads. In *International Symposium On High Performance Computer Architecture (HPCA).*

[2] Arvind, Nirav Dave, and Michael Katelman. 2008. Getting formal verification into design flow. In *International Symposium on Formal Methods.*

[3] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *ACM SIGARCH Computer Architecture News.*

[4] Jesse G. Beu, Jason A. Poovey, Eric R. Hein, and Thomas M. Conte. 2013. High-speed formal verification of heterogeneous coherence hierarchies. In *International Symposium On High Performance Computer Architecture (HPCA).*

[5] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte. 2011. Manager-client pairing: A framework for implementing coherence hierarchies. In *International Symposium on Microarchitecture (MICRO)*.
[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark Hill, and David Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2.
[8] Henry Burkhardt, Steven Frank, Bruce Knobe, and James Rothnie. 1992. Overview of the KSR1 computer system. *Technical Report KSR-TR-9202001, Kendall Square Research, Boston.*
[9] Alan Charlesworth. 2001. The Sun Fireplane system interconnect. In *ACM/IEEE Supercomputing Conference.*
[10] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD).*
[11] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaidi. 2013. Invariants for finite instances and beyond. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD).*
[12] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha ZaÄśdi. 2012. Cubicle: A parallel SMT-based model checker for parameterized systems. In *International Conference on Computer-Aided Verification (CAV).*
[13] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha ZaÄśdi. 2012. German's Protocol. http://cubicle.lri.fr/examples/german.ctc.cub.html.
[14] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro.*
[15] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. 1992. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design (ICCD).*
[16] Silvio Ghilardi and Silvio Ranise. 2010. MCMT: A model checker modulo theories. In *International Joint Conference on Automated Reasoning (IJCAR).*
[17] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C Valentine. 2003. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal* 7, 2.
[18] Gary Gostin, Jean-Francois Collard, and Kirby Collins. 2005. The architecture of the HP Superdome shared-memory multiprocessor. In *International Conference on Supercomputing (ICS).*
[19] Erik Hagersten and Michael Koster. 1999. WildFire: A scalable path for SMPs. In *International Symposium On High Performance Computer Architecture (HPCA).*
[20] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. 2001. The microarchitecture of the Pentium® 4 processor. In *Intel Technology Journal.*
[21] Edya Ladan-Mozes and Charles E Leiserson. 2008. A consistency architecture for hierarchical shared caches. In *Symposium on Parallelism in Algorithms and Architectures (SPAA).*
[22] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, W-D Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. 1992. The Stanford DASH multiprocessor. *Computer* 25, 3.
[23] Milo MK Martin, Mark D Hill, and Daniel J Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7.
[24] Opeoluwa Matthews, Jesse Bingham, and Daniel Sorin. 2016. Verifiable Hierarchical Protocols with Network Invariants on Parametric Systems. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD).*
[25] Opeoluwa Matthews, Meng Zhang, and Daniel J Sorin. 2014. Scalably verifiable dynamic power management. In *International Symposium on High Performance Computer Architecture (HPCA).*
[26] Seungjoon Park, Satyaki Das, and David L. Dill. 2000. Automatic checking of aggregation abstractions through state enumeration. In *Computer-Aided Design of Integrated Circuits and Systems.*
[27] Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. 2015. Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies. In *International Symposium On High Performance Computer Architecture (HPCA).*
[28] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG).*
[29] Balaram Sinharoy, Ronald N Kalla, Joel M Tendler, Richard J Eickemeyer, and Jody B Joyner. 2005. POWER5 system microarchitecture. *IBM Journal of Research and Development* 49, 4.5.
[30] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* (2011).
[31] Daniel J Sorin, Opeoluwa Matthews, and Meng Zhang. 2014. Architecting dynamic power management to be formally verifiable. In *Design Automation Conference (DAC).*
[32] Murali Talupur and Mark R. Tuttle. 2008. Going with the Flow: Parameterized Verification Using Message Flows. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD).*
[33] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. 2002. POWER4 system microarchitecture. *IBM Journal of Research and Development* 46, 1.
[34] Muralidaran Vijayaraghavan. 2016. *Modular verification of hardware systems.* Ph.D. Dissertation. Massachusetts Institute of Technology.
[35] Muralidaran Vijayaraghavan, Adam Chlipala, and Nirav Dave. 2015. Modular deductive verification of multiprocessor hardware designs. In *International Conference on Computer Aided Verification (CAV).*
[36] Gwendolyn Voskuilen and TN Vijaykumar. 2014. Fractal++: Closing the performance gap between fractal and conventional coherence. In *International Symposium on Computer Architecture (ISCA).*
[37] Gwendolyn Voskuilen and T. N. Vijaykumar. 2014. High-performance Fractal Coherence. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*
[38] Deborah A Wallach. 1992. *PHD: A hierarchical cache coherent protocol.* Ph.D. Dissertation. Massachusetts Institute of Technology.
[39] Andrew W Wilson Jr. 1987. Hierarchical cache/bus architecture for shared memory multiprocessors. In *International Symposium on Computer Architecture (ISCA).*
[40] Meng Zhang, Jesse D. Bingham, John Erickson, and Daniel J. Sorin. 2014. PVCoherence: Designing flat coherence protocols for scalable verification. In *International Symposium On High Performance Computer Architecture (HPCA).*
[41] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. 2010. Fractal coherence: Scalably verifiable cache coherence. In *International Symposium on Microarchitecture (MICRO).*