Appears in the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010 (MICRO 2010)
Atlanta, Georgia, December, 2010

Fractal Coherence: Scalably Verifiable Cache Coherence

Meng Zhang
Dept. of ECE
Duke University
Durham, NC, USA
mz28@duke.edu

Alvin R. Lebeck
Dept. of Computer Science
Duke University
Durham, NC, USA
alvy@cs.duke.edu

Daniel J. Sorin
Dept. of ECE
Duke University
Durham, NC, USA
sorin@ee.duke.edu

***Abstract*—We propose an architectural design methodology for designing formally verifiable cache coherence protocols, called Fractal Coherence. Properly designed to be fractal in behavior, the proposed family of cache coherence protocols can be formally verified correct for systems with an arbitrary number of cores, using existing, automated formal tools. We show, by designing and implementing a specific Fractal Coherence protocol, called TreeFractal, that Fractal Coherence protocols can attain comparable performance to traditional snooping and directory protocols.**

Keywords: multicore, cache coherence, formal verification

I. INTRODUCTION

A buggy cache coherence protocol might lead to a catastrophic failure of the shared-memory system that employs this protocol. Carefully designing and inspecting the protocol is necessary. However, this method is insufficient to ensure the correctness of a modern cache coherence protocol. To achieve high performance, current cache coherence protocols allow multiple outstanding requests and concurrent operations. This concurrency implies numerous transient states and nondeterministic behavior due to race conditions. To ensure the reliability of the protocol, design verification (also known as "design validation" or "pre-silicon validation") must be done as part of implementing cache coherence protocols for real systems.

Traditionally, the verification of cache coherence protocols is mostly performed via simulations [5, 33]. The system is simulated while running benchmarks, stress tests, and random code sequences, and the output is checked to see whether the cache coherence protocol is correct. However, testing is extremely unlikely to reach all possible states no matter how long it runs, especially when testing a complex cache coherence protocol with many caches, each of which can have cache blocks in many different states. This inability to reach all system states limits the ability of simulation methods to find subtle bugs. Previous work [7, 9, 11, 28] shows cache coherence protocols can still have bugs even after extensive simulations.

Formal verification can be used to overcome the incompleteness of simulation. Formal verification falls into two categories [10]: 1) model checking, which employs automated tools to verify the design and 2) theorem proving, which relies heavily on the user's guidance to complete the proof. We constrain our focus in this paper to the first category because theorem proving is not widely used; the user effort to guide a theorem prover through a non-trivial proof is generally impractical. Automated formal verification tools have been used to help verify the correctness of cache coherence protocols. The tools traverse all possible states in the system and thus the method is complete. Despite this completeness advantage over simulation, formal verification has its own unavoidable problems. The most obvious is the "state explosion problem"; as an exhaustive method, the state space explodes exponentially as the number of cores increases. Despite efforts to optimize these tools, even configurations with only a few cores saturate state-of-the-art formal verification tools and only small systems can be formally verified. Unfortunately, there is currently no guarantee that the correctness of a small system implies the correctness of a much bigger system [3]. In this sense, even if we have a formally verified cache coherence protocol with 4 cores, we cannot trust that it is bug-free if we implement it for a 64-core system. Our work seeks to provide this guarantee by changing the way we design coherence protocols.

We propose, from the perspective of architects, a new methodology for designing cache coherence protocols, called *Fractal Coherence*. Our goal is to design cache coherence protocols for large, many-core systems such that the protocols are verifiable using existing, automated, easy-to-use formal tools. Fractal Coherence protocols originate from fractal theory and leverage the self-similarity characteristic of fractals. The smallest complete system, called the *minimum system*, is small enough to be easily verified coherent using existing formal verification tools without incurring the state explosion problem. Larger systems can be verified fractal in behavior by showing that any scale of the system behaves exactly the same with regard to coherence. The whole system can then be proved coherent using induction, and thus the verification of a Fractal Coherence protocol can be scaled to any arbitrary N-node system. Fractal Coherence eliminates the burdensome and error-prone user work to abstract the system and fit it into the verification tools.

We design a concrete example of a Fractal Coherence protocol, called TreeFractal, based on the proposed methodology. The primary difference between TreeFractal and traditional protocols is that TreeFractal introduces an interface component to maintain the fractal behavior at each scale. TreeFractal connects the interfaces, the cores, the caches and the memories to construct a shared-memory system. We present the straightforward steps to verify cache

coherence on this system. We then evaluate TreeFractal with full system simulation and show that it achieves comparable performance to typical snooping and directory protocols.

Previous research pursued optimized methods to verify the cache coherence protocol of a system *after* the protocol is designed. In contrast, we incorporate verifiability of the cache coherence protocol into the early design stage. This idea corresponds to the concept of "design for verifiability" presented by Milne [24], which is a counterpart to the "design for testability" in the formal verification area. Taking formal verification effort as a design constraint can ease verification effort, improve product quality, and reduce a product's time to market. In all, we make the following contributions:

- We propose Fractal Coherence: a design methodology for formally verifiable cache coherence protocols based on fractal theory. Fractal Coherence ensures each scale of the system has the same behavior with regard to coherence. Then the verification of cache coherence for the minimum system can be scaled to larger systems. We show that through straightforward verification with existing tools, any arbitrary N-node system can be proved cache coherent.
- We implement a specific Fractal Coherence protocol, TreeFractal, and present the verification process for any arbitrary N-node system with this protocol. This process includes formal verification of the minimum system and an equivalence check showing that the whole system has fractal behavior.
- We experimentally evaluate TreeFractal using full system simulation and show it has comparable performance to traditional snooping and directory protocols without adding significant implementation costs.

II. FRACTAL COHERENCE

Our goal is to design a coherence protocol that is within the capability of existing, automated verification tools, yet still scales to many-core systems. Considering the two systems in Fig. 1, we assume that System A is a shared-memory system that is small enough to be verified coherent by existing formal tools. System A is part of a much larger System B. We want to formally verify that System B is cache coherent, but System B is large and way beyond the capability of existing tools. Intuitively, if there is a certain kind of similarity between System A and System B, we may be able to extend the verification of System A to the scale of System B.

This intuition inspires us to use fractal theory. A fractal is a shape that can be split into parts in which each part is a reduced-size copy of the whole [18]. At any scale, the fractal appears exactly identical. We focus on the cache coherence behavior of each scale instead of only the structure. Thus, if System B has fractal behavior and System A is a reduced-size copy of System B, then we can prove the cache coherence of B based on the cache coherence of A.

We propose Fractal Coherence, a class of coherence protocols that leverages the self-similarity characteristic of fractal theory to enable the verification of large scale systems. A system with Fractal Coherence is architected in a manner

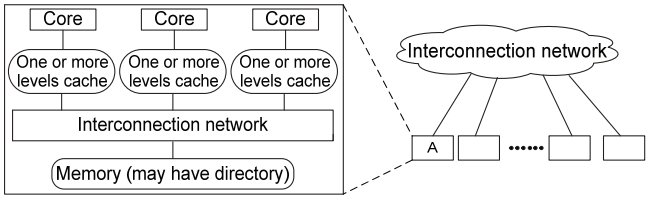


Figure 1. Scalability problem in verification of cache coherence

that is formally verified to be fractal in behavior with regard to coherence.

In the rest of this section, we first introduce the fractal system architecture (Section II.A). We then describe the two steps needed to verify that a given system with a Fractal Coherence protocol is correct (i.e., maintains coherence) for any arbitrary number of cores. The first step is to verify that the smallest scale of the system is coherent (Section II.B). The second step is to show that the system is fractal with respect to coherence (Section II.C). We then present an inductive proof that these two verification steps are the only formal verification steps needed (Section II.D). Unlike the two verification steps, which are part of the design flow for each Fractal Coherence protocol developed, the inductive proof need only be performed once to show that the two verification steps are sufficient.

A. System Architecture

To ensure fractal behavior, Fractal Coherence requires a hierarchical *logical* structure. However, Fractal Coherence does not place any requirements on the *physical* topology of the system. The hierarchical logical structure can be implemented on any kind of physical topology, such as a 2D mesh, torus, ring, etc. Hereafter, when we refer to a system's structure, we are referring to its logical structure. In this paper, we confine our discussion to the tree structure with a consistent degree at each level, but we believe our methodology can also apply to other hierarchical logical structures.

The tree structure in Fractal Coherence can be either a balanced tree or an unbalanced tree, because Fractal Coherence does not rely on the fractal *structure*; instead, it relies on the fractal *behavior*. Fig. 2 shows several possible binary tree structures for Fractal Coherence. The shadowed square components are *basic nodes* (corresponding to the leaf nodes in the tree structure), which may have a number of caches, cores and memories. The elliptical shape components are the *interfaces* (corresponding to the internal nodes in the tree structure) that support the fractal behavior. Depending on its position in the system, an interface can be categorized as a *top interface* (corresponding to the root node in the tree structure) or an *internal interface* (corresponding to internal nodes except the root node in the tree structure). Two or more basic nodes and a top interface or an internal interface compose a level_1 node. Iteratively, two or more level_n-1 nodes and a top interface or an internal interface compose a level_n node, where "n" is the height of the node's tree. For a tree structure with a given degree, we can determine the *minimum system*. It is the smallest complete system that

includes all the different types of components used in larger systems. The minimum system consists of a top interface, an internal interface with all its children, and other basic node(s) directly beneath the top interface. Fig. 3 shows the minimum system for a binary tree, a ternary tree, and any D-degree tree. To formally verify that a fractal system is cache coherent with any arbitrary number of nodes, two verification steps are needed. First, the minimum system must be verified cache coherent (Section II.B). Second, the whole system must be verified fractal in behavior so that we can leverage the self-similarity (Section II.C). We discuss these two verification steps next, before proving that these two steps are sufficient.

B. *Formal Verification of Minimum System*

To formally verify the minimum system, we need a description of the system (modeling) and a specification of correctness properties (specification). Regardless of the kind of language the verification tool accepts for describing the models and properties, the modeling has to accurately capture the behavior of the cache coherence protocol, and the specification has to precisely state the properties that the protocol must satisfy in order to maintain coherence among all caches and memories. Then, the tool performs the verification by walking through each possible state of the entire system (i.e., including the states of all coherence controllers) to ensure that all states adhere to the specified properties. We discuss several key points in the modeling and specification processes.

Modeling. To model the cache coherence protocol, several reduction techniques have been widely used. For example, modeling only one block in the cache and memory instead of all the blocks is sufficient to verify the cache coherence protocol; the data values themselves can also be abstracted away since they have no impact on coherence [29]. These optimizations can all be employed in modeling the minimum system of Fractal Coherence. It is worth pointing out that these techniques cannot eliminate the state explosion problem, because the number of states still explodes exponentially with the number of cores. Moreover, the more complicated the cache coherence protocol, the more states it has. Therefore, although we claim the minimum system can have arbitrary configurations, the architects must ensure that the system is within the capability of existing verification tools in order to avoid the state explosion problem.

Specification. The correctness properties of a cache coherence protocol are usually specified in invariants or temporal logic. More specifically, the tool needs to verify the following properties: 1) each block can have either one writer or multiple readers at any given time, 2) no state machines will ever enter deadlock, and 3) the system is making forward progress at all times (i.e., there is no livelock).

This verification process is straightforward. It can be automatically completed by a wide range of existing tools without requiring the user to abstract away system details or implement complicated optimizations. We show the detailed verification process of the minimum system of a specific fractal system in Section IV.A.

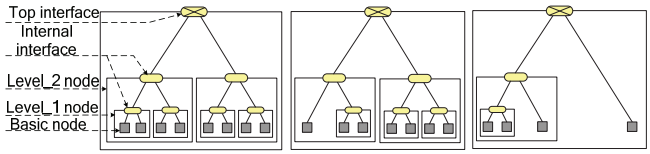
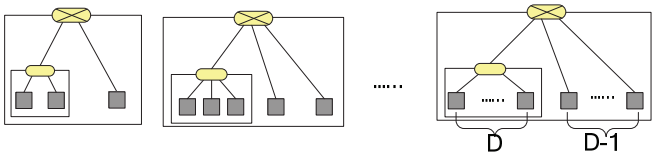


Figure 2. Possible binary tree structures



(a) binary tree (b) ternary tree (c) arbitrary D-degree tree

Figure 3. Minimum systems of different degree trees

C. *Verification of Fractal Behavior*

After verifying that the minimum system is coherent, we need to show that the whole system has fractal behavior in order to leverage the self-similarity to prove that larger scale systems are coherent. By fractal behavior, we mean that a system scales in a manner such that the behavior of the larger system is always the same as the smaller system. Fractal behavior ensures that coherence is maintained while scaling the system.

We need "equivalence checking" to verify that each scale of the system behaves the same. Because the system is constructed by iteration, it is sufficient to verify only the equivalence between the level_1 node and the level_2 node. We take the binary tree in Fig. 3(a) as an example and show in Fig. 4 the two relationships needed to be verified equivalent. We construct a 4-node binary tree, shown in Fig. 4(b), by expanding node A in Fig. 4(a) to node A' in Fig. 4(b). To satisfy fractal behavior requirements, two verification steps must be performed. First, we must verify that A and A' have the same behavior as observed at point O1 in Figure 4(a) and Figure 4(b). This verification enables the system to scale based on substituting A with A'. The rest of the system cannot tell the difference after the substitution and has the same behavior as before. Second, we must verify that B and B' have the same behavior as observed at point O2 in Figure 4(c) and Figure 4(d). This equivalence means that C in Fig. 4(c) and A' in Fig. 4(d) have the same environment and thus they behave the same. This verification ensures that the two basic nodes in A' behave the same as they do in a coherent system (Fig. 4(c)). The two verifications together ensure that the new system (Fig. 4(b)) has the same behavior as the previous one (Fig. 4(a)).

These two verification steps are both "equivalence checking." Intuitively, A' has more state machines than A, and B' has more state machines than B. They cannot have exactly the same transitions. However, for verifying fractal behavior, we need to show only that they behave in a manner that is "observationally equivalent" [25], which means the external world cannot tell the difference between the two systems. The observational equivalence allows several transitions in the more complex system to match one transition in the simple system. For example, considering a

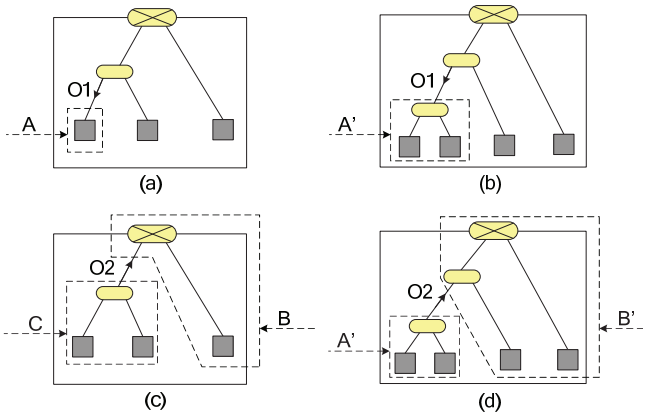


Figure 4. Observational equivalence for maintaining fractal behavior

simple MSI protocol without transient states, if A is in state S, the observationally equivalent states in A' are S:S, S:I and I:S, where the state before the colon is the state of the left child in A', and the state after the colon is the state of the right child in A'. The transitions between S:S, S:I and I:S are considered "internal" because they have no impact on the external world. The three states S:S, S:I and I:S are collapsed to one. We can say that A', taken as a "node as a whole," is in state S, meaning the external world considers A' to be a single node in state S. By this collapsing, A' can be simplified to have the same states and transitions as A, and B' can be simplified to have the same states and transitions as B. The external world cannot tell apart A and A' or B and B'.

This equivalence checking is also a formal method because it explores all possible states in the system. Therefore, the tool used for this verification should be an exhaustive tool. Many formal tools are able to do equivalence checking and they accept different kinds of description languages. We will show a detailed verification process of the fractal behavior of TreeFractal in Section IV.B.

D. Proof of Cache Coherence for Arbitrary N-node System

We claimed that the formal verification steps described in Section II.B and Section II.C are the only steps the architect of a Fractal Coherence protocol must perform to verify the correctness of an arbitrary N-node system with Fractal Coherence. In this section, we prove by induction why these two steps are sufficient.

Definition 1. We use F (L, D, N) to denote a system that has L levels, D degrees for each level, and N basic nodes in all. The subscript "s" in Fs (L, D, N) denotes that the system is a sub-system inside a larger system and not a complete system itself. In F (L, D, N) and Fs (L, D, N), L={1,2, …, m}, D={2,3, …, n}, and N={(D-1)*L+1, (D-1)*(L+1)+1, …, D^L}.

From Definition 1, we know that the minimum system can be written as F (2, D, 2*D-1). Note that only when we use a binary tree (D=2), the number of basic nodes can be contiguous; otherwise we can have only discrete increments of (D-1) for the number of basic nodes, because we assume each level of the tree structure has the same degree. We could relax this constraint, since the missing children can be considered as always in state I and have no impact on coherence.

Definition 2. Given two systems A and B, where A is larger than B, we use the symbol "≈" to represent observational equivalence, and we use the symbol "–" to represent the subtraction of a subsystem from a larger system. So A ≈ B means A is observationally equivalent to B, and A – B represents the rest of the system after removing a subsystem B from System A.

We now present five lemmas that we use in our proof. Each lemma is illustrated in Fig. 5.

Lemma 1 (Fig. 5a). *Basic node ≈ Fs (1, D, D)* by the verification result of Section II.C.

Lemma 2 (Fig. 5b). *F (2, D, 2*D-1) – Fs (1, D, D) ≈ F (3, D, 3*D-2) – Fs (1, D, D)* by the verification result of Section II.C.

Lemma 3 (Fig. 5c). *F (2, D, N) – Fs (1, D, D) ≈ F (3, D, N) – Fs (1, D, D)* by a generalization of Lemma 2 based on using Lemma 1 to do substitution.

Lemma 4 (Fig. 5d). *Basic node ≈ Fs (1, D, D) ≈ Fs (2, D, N) ≈ Fs (3, D, N) … ≈ Fs (L, D, N)* by iteration on Lemma 1.

Lemma 5 (Fig. 5e). *F (2, D, N) – Fs (1, D, D) ≈ F (3, D, N) – Fs (1, D, D) ≈ F (L, D, N) – Fs (1, D, D)* by iteration on Lemma 3 and by using Lemma 4 to do substitution.

Theorem. Any N-node system is coherent.

Proof.

1) Base case: when N = 2*D -1, it is the minimum system. The cache coherence of the minimum system is formally proved (Section II.B). Note that when N<2*D-1, the system can be formally proved coherent by just using Lemma 1 to do substitutions.

2) Inductive step: We assume that, when N=k*(D-1), the system is coherent. We must prove that, when N= (k+1)*(D-1), the system is still coherent. To expand the k*(D-1) node system into the (k+1)*(D-1) node system, we substitute a basic node in the k*(D-1) node system with a Fs (1, D, D) that we call A'.

Proposition 1. *For the other N-1 nodes and the A' subsystem, coherence is still maintained.* Based on Lemma 1, after substituting a basic node with A', the rest of the system cannot see the difference and maintains the same behavior. At the same time, A' as a whole maintains the same coherence states as the previous basic node does.

Proposition 2. *A' maintaining coherence indicates that all of its children maintain coherence.* Based on Lemma 5, the rest of the system after subtracting A' from the N=(k+1)*(D-1) node system is observationally equivalent to the rest of the system after subtracting A' from the N=k*(D-1) node system. Thus A' behaves the same in the two systems. We know that, in the N=k*(D-1) node system, A' as a whole as well as each basic node of A' maintain coherence, because the N=k*(D-1) node system is cache coherent (the inductive assumption). Therefore, in the N=(k+1)*(D-1) node system, A' as a whole maintaining coherence is sufficient to ensure that each basic node in A' maintains coherence.

Based on Proposition 1 and Proposition 2, we can conclude that any N-node system is coherent. ■

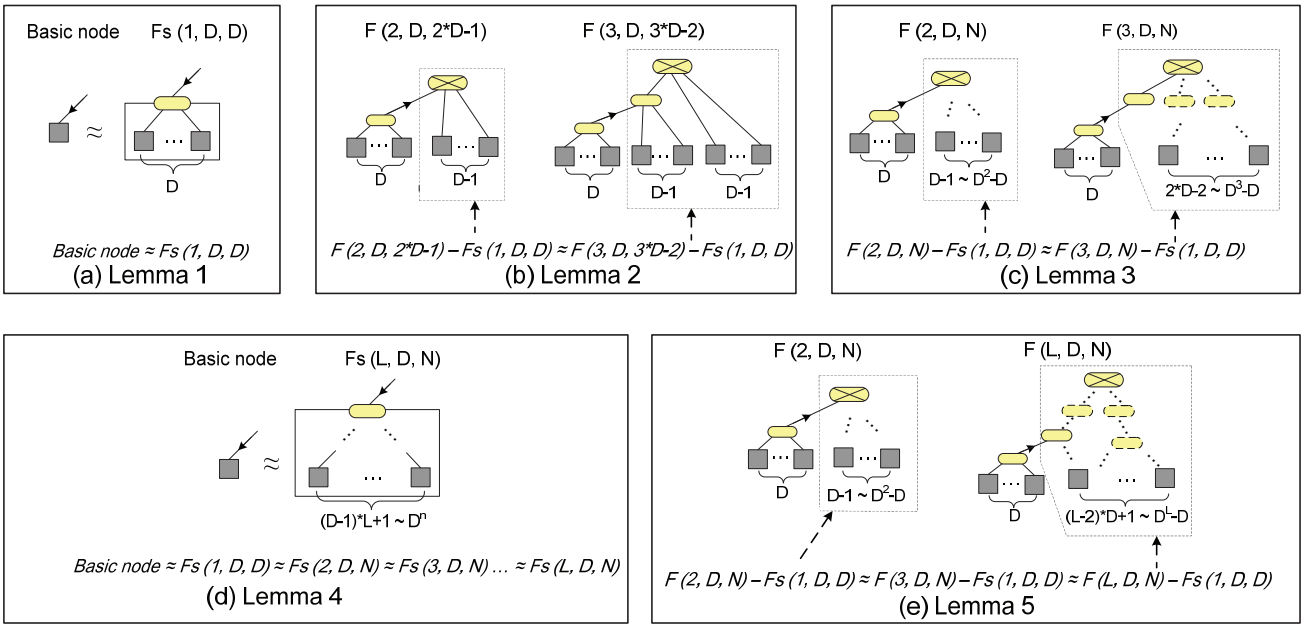


Figure 5. Lemmas for proof of cache coherence in any N-node system

III. Implementation of a Specific Fractal Coherence Protocol

There are many different possible Fractal Coherence protocols. We implemented a specific protocol, which we call TreeFractal, to show that the fractal design methodology is viable. TreeFractal uses a binary tree as both the logical structure and network topology, although this is not required. In TreeFractal, each interface that maintains fractal behavior (see Fig. 2) contains duplicate cache tags for all cache blocks beneath it in the tree. We call these interfaces Tags. We now discuss the two-node system design (Section III.A), the scaled design (Section III.B) and the implementation costs of TreeFractal (Section III.C).

A. Two-Node System Design

We start our design from a two-node system, illustrated in Fig. 6(a). It consists of two basic nodes and a Top Tag. The basic node, shown in Fig. 6(b), consists of a core, a private L1 cache, a private L2 cache, a portion of the shared memory and a coherence controller. The coherence controller is responsible for communicating with the core, the cache, the memories and its parent Tag. The coherence controller also has MSHRs to allow for multiple outstanding requests. The Top Tag holds copies of the cache tags and coherence states of its two children, and it serves as the serialization point for coherence transactions in the two-node system. In the two-node system, the Tag is called the "Top" Tag to distinguish it from "Internal" Tags in larger systems.

The TreeFractal coherence protocol is a MOSI protocol with numerous transient states that is neither snooping nor directory, although it has some features in common with both of those well-known classes of protocols. The coherence controller responds to load and store requests from the core. If the coherence controller cannot satisfy a load or store, it issues a coherence request up to the Top Tag. When the Top Tag receives a coherence request from one of its children cores, it looks up the state of the block in both of its children. We denote this state using *X:Y* notation, where X is the state of the block in the left child and Y is the state of the block in the right child. For example, the Top Tag state S:O denotes that the left child has the block in state S and the right child has the block in state O. Based on the states in the children, the Top Tag forwards the request down to either one or both of them (similar to directory protocols). Because the Top Tag is the serialization point for all transactions, it always forwards a request back to the requestor, so that the requestor knows when its request is ordered with respect to other coherence requests (similar to snooping protocols). We now present three examples to illustrate how this protocol works:[1]

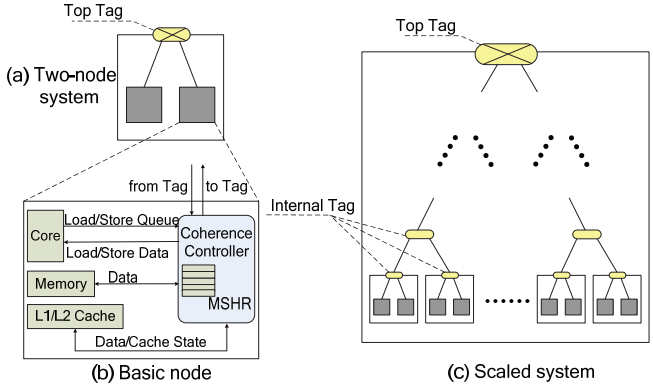


Figure 6. System architecture of TreeFractal

1) If the Top Tag receives a Get-Shared (GetS)

[1] The whole specification of TreeFractal can be found online: http://arch.cs.duke.edu/micro2010/TreeFractal/

coherence request for a block that is in state I:I (invalid in both children), it forwards the GetS down to the requestor and to the home node for the block (i.e., the node that has the portion of the memory space including this block) based on the block's address. The home sends its reply up to the Top Tag, and the Top Tag forwards the reply down to the requestor. If the requestor is the home, the reply does not need to go up to the Top Tag and then back down to itself.

2) If a GetS request from the left child reaches the Top Tag in the state I:M, the Top Tag forwards the GetS to both the left and right children and changes its state to S:O. The right child's coherence controller replies to the Top Tag with the data and changes its state to O, and the Top Tag forwards the reply to the left child to complete the transaction.

3) This third example highlights an important feature of TreeFractal. If a GetS from the left child reaches the Top Tag in state I:S, the Top Tag forwards the request to both children, and the right child replies to the Top Tag, which forwards the reply to the left child. In this example, a node in state S responds to a coherence request, which is not typical in snooping or directory protocols.

To avoid deadlock due to circular dependences among coherence messages of different types, TreeFractal requires four virtual networks. Requests from the basic nodes to the Top Tag go into the request network. The Top Tag forwards the requests to one or both basic nodes through the forwarded request network. Replies from the basic nodes to the Top Tag go into the reply network. The Top Tag forwards the replies to the requestor through the forwarded reply network.

B. Scaled System Design

The two-node system can scale to any arbitrary N-node system by adding Internal Tags between the Top Tag and the basic nodes and making the system structure a binary tree, as shown in Fig. 6(c). In the scaled system, just as in the two-node system, requests and replies go up the tree and forwarded requests and forwarded replies go down the tree. However, the requests and replies in the scaled system do not need to go all the way up to the Top Tag each time as the two-node system requires, because the requests and replies need only go up to the highest common ancestor Tag of all destinations. For example, consider a 4-node system in which the cores are numbered starting from the left as 1, 2, 3, 4, and the block is in states M, I, I, I in these four cores. If Core 2 issues a Get-Modified (GetM) request to its Internal Tag, that Internal Tag is in state M:I and forwards the GetM to both children (Core 1 and Core 2). The Internal Tag does *not* need to send the request up to the Top Tag in this situation. Core 1 replies to the Internal Tag and the Internal Tag forwards the reply to Core 2. This entire transaction is invisible to the Top Tag (and Core 3 and Core 4), which views the node as a whole consisting of Core 1, Core 2, and their Internal Tag as being in state M the entire time. As an example of a request that the Internal Tag must send up, consider a GetM that reaches an Internal Tag in state I:I. The

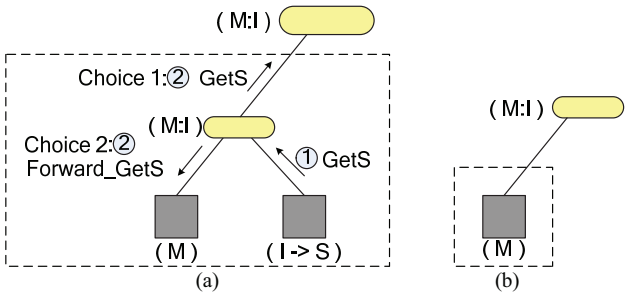


Figure 7. An example of naïve design which violates the fractal behavior

Internal Tag must send the request up to its parent Tag in case there is a node elsewhere in the system that is in a valid coherence state and needs to observe the GetM. One might think we can scale the system by simply expanding the two-node system. However, if not carefully designed, the Internal Tag can break the fractal behavior. We show a specific case to illustrate how a naïve design would violate the fractal behavior.

A Non-Fractal Design. As shown in Fig. 7(a), for a single block, the Internal Tag has a left child in state M, and a right child in state I, so the state in the Internal Tag is M:I. Observed from the external world, the node as a whole (i.e., the Internal tag and its two children) should appear in M for the block to maintain fractal behavior. Therefore, the Top Tag is in state M:I, too. Then the right child issues a Get-Shared (GetS) coherence request. The GetS request arrives at the Internal Tag, and then the Internal Tag needs to decide where and how to send the request. Intuitively, the Internal Tag has two options. First, as in snooping protocols, it could issue a GetS up to the Top Tag. Second, as in directory protocols, it could issue a Forward_GetS to the owner (the left child) and then change state to O:S, which is equivalent to state O as observed by the external world. However, both options result in the violation of fractal behavior. As shown in Fig. 7(b), if a basic node has a block in state M, for that block, it will neither issue a GetS to the Top Tag nor silently change to state O. In Section II.C, we have shown that to ensure fractal behavior, a necessary observational equivalence relationship is that the basic node and the level_1 node behave the same as seen by the external world. But in this example we do not make them have the same behavior, which violates the foundation of our methodology.

A Fractal Design. Our method to deal with the above problem is to add some new states and message types. For this case, the correct implementation is shown in Fig. 8. In Fig. 8 (a), The Internal Tag issues a Put-From-M-to-O (PutMtoO) request up to the external world, meaning the node as a whole (the subsystem outlined by the dashed box) would like to change from M to O. After receiving the acknowledgment of the PutMtoO from the external world, the Internal Tag forwards the GetS to both the left child and the right child. The forwarded GetS is sent to *both* children because the Internal Tag is the ordering point. Then the state of the Internal Tag changes to $O:IS^D$, meaning the right child is in I, trying to go to S, and waiting for the data. After the data comes back from the left node, the Internal Tag

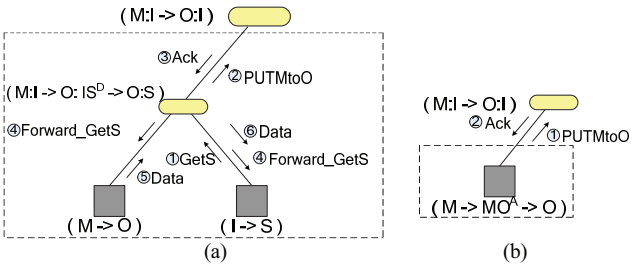


Figure 8. Correct implementation to ensure fractal behavior

transfers the data to the right node and changes to O:S. The node as a whole appears to be in state O. To ensure the observational equivalence, there must also be a PutMtoO action in the basic node, as shown in Fig. 8(b). The basic node is allowed to generate a PutMtoO request and change to MO^A, meaning it is in M, trying to go to O, and waiting for the acknowledgment. After receiving the acknowledgment from the Top Tag, the basic node changes to state O. This scenario is impossible for a single node in a real system since a core will not choose to change from M to O. However, to ensure the fractal behavior, we need to incorporate such transitions in the basic node state machine in the minimum system design and formally verify it.

Besides the given examples, the Internal Tag has many other specifically designed transitions to maintain the fractal behavior of TreeFractal. For example, we have a node in S, instead of memory, respond to coherence requests with data. Therefore, if a node in S would like to evict the shared block, it must explicitly notify the Top Tag in order to update the state. Another example is when the Internal Tag is in the state S:S, meaning both the left and the right child are in S. If either of them evicts the block and changes to I, the Internal Tag does not issue any request to the external world since the node as a whole is still in S. The Internal Tag state changes to S:I or I:S. When the second eviction arrives, the Internal Tag must issue an explicit Put-Shared (PutS) coherence request to the external world and change state to I:I. In the scaled system, an important decision is whether a certain action should be visible to the external world and how it should be displayed to the external world. With a properly designed Internal Tag, we can scale the system to any number of nodes while still maintaining the fractal behavior.

C. *Implementation Cost*

TreeFractal is a viable option for architects only if its implementation cost is not far greater than the costs of existing, non-fractal protocols. Consider a system with N cores, a total number of B blocks that are cached on these N cores, and a total number of M blocks that are distributed evenly across the memories at the N cores. We now discuss the implementation cost of TreeFractal and a full-map directory, respectively.

TreeFractal. The implementation cost of TreeFractal stems mainly from the storage overhead of the Tags at each level. Since the Tag stores only the addresses and states of the cache blocks beneath it, the storage overhead is much less compared to a full-map directory structure that tracks the states of all the blocks in the memory. The address of a block is $\log_2 M$ bits long. For TreeFractal, which has fewer than 64 coherence states, 6 bits is enough for a Top Tag or Internal Tag entry that stores the state of a block in one of its children. For the Top Tag, which has B entries (i.e., the total number of cached blocks in its children is B), the storage overhead is $(\log_2 M+6)*B$. The storage overhead of one of the two Internal Tags just beneath the Top Tag is $(\log_2 M+6)*(B/2)$ bits. Since there are two such Internal Tags at this level, the storage overhead at this level is still $(\log_2 M+6)*B$ bits. For a system with N cores, the total number of levels is $\log_2 N$. Thus, the total storage overhead for all Tags is $(\log_2 M+6)*B*\log_2 N$ bits.

Full-map Directory. An entry in the full-map directory has an N-bit sharer list, a $\log_2 N$-bit owner field, and a 2-bit tag. Therefore, the total directory storage is $(N+ \log_2 N+2)*M$.

For some common values of N (16), B (32MB cache/ 64B block size), and M (64GB memory/64B block size), we found TreeFractal's storage overhead is less than 1/300 that of a directory protocol's storage overhead. TreeFractal uses less storage because it can leverage multicasting as a message comes down the tree. A Tag has greater associativity than a direct-mapped directory, which means its access time is longer and power consumption is larger compared to an equal-sized directory. However, considering the large difference in their sizes, we believe the Tag's size advantage outweighs its associativity disadvantage.

Caching Possibilities. Multicore chips encourage the use of on-chip caching, which is applicable to both directory protocols and TreeFractal. For directory protocols, on-chip caching of the directories, a well-known optimization, reduces the average latency of each directory access. Caching does not reduce the total cost of storage, though, since the full directory must still exist (off chip). For TreeFractal, which already has its complete Tag storage structures on chip, caching of Tags offers a similar cost/benefit tradeoff. Caching of Tags reduces the average latency of Tag accesses, although to a lesser degree than the latency reduction for directory caching, while it increases the total storage overhead.

IV. Verification Procedure and Results

In Section II, we discussed the two verification steps required to verify any Fractal Coherence protocol. Now we explain how we use two widely-used automated verification tools to perform these two verification steps for TreeFractal. We note that, although we use two specific tools to verify our implementation, there are numerous other verification tools that can do this work. They accept different languages and use different methods to specify the correctness of a system.

A. *Formal Verification of Minimum System.*

We chose the well-known Murphi [11] checker to verify the cache coherence of the minimum system. Murphi is straightforward since it employs the explicit state enumeration method to formally verify the system. Compared to (symbolic) model checking and symbolic state model methods, state enumeration expresses the system

more intuitively and is less likely to diverge from the real system. However, it is the most susceptible to the state explosion problem since it uses fewer techniques to overcome this problem. We seize the opportunity to use explicit state enumeration because we have already broken down the problem to small pieces and thus remove the state explosion problem as a constraint. This is a significant advantage over previous formal verification of cache coherence. Most previous approaches seek a method to avoid state explosion.

In Murphi, we model the minimum system shown in Fig. 4(a) which consists of one Internal Tag state machine, one Top Tag state machine and three coherence controller state machines. These state machines are simultaneously running and interacting with each other. The parallelism and interaction lead to the nondeterministic race conditions. The model includes several components: the structure of caches and Tags, the types of possible messages, the description of the events and the rules for transitions. We also specify the initial states of all the state machines to make sure Murphi knows where to start its traversal.

The properties we need to verify make use of four forms: in-line error statements, invariants, deadlock checking, and liveness checking. The in-line error statements are useful for finding common description errors and unused branches in case statements. The invariants are used to specify certain correctness properties. For example, we allow only one writer in the system at any time for a given block. The deadlock checking is inherent in Murphi when it traverses all possible states. The liveness checking is expressed in linear temporal logic to ensure the protocol is making progress.

Our results show that even such a small system took Murphi three hours to verify and 12,031,400 states were explored during this period. Increasing the number of cores will soon lead to the state explosion problem since the number of states increases exponentially.

B. *Equivalence Checking for Fractal Behavior*

To verify fractal behavior, we employ CADP's [12] equivalence checker, Bisimulator [6]. Bisimulator performs an on-the-fly comparison of the two input state machines modulo a given equivalence/preorder relation. In our case, the relation is observational equivalence. We verified the two kinds of observational equivalence discussed in Section II.C to ensure the fractal behavior. In CADP, a single state machine ─ like a basic node, a Top Tag, or an Internal Tag ─ is modeled as a process. A system with several state machines is modeled as a process that consists of several sub-processes running together and interacting with each other through queues. In our verification, we associate all these processes with a set of actions and parameters. We use actions to represent the process's interactions with other processes and the parameters to represent the states of this process. Since we do not care about the interactions between the sub-processes as long as the interactions cannot be noticed by the external world, we hide all these actions by considering them as invisible actions in the equivalence checking. The tool gives the results of the two equivalence checkings as "true", meaning the systems we are verifying are observationally equivalent when observed by the external world.

V. EXPERIMENTAL EVALUATION

For TreeFractal to be viable, its verifiability advantage must not come with significant performance degradation. We performed a series of experiments to compare TreeFractal with a typical MOSI snooping protocol (called Snooping) and MOSI directory protocol (called Directory). In Snooping, the memory controller implements an owner bit to determine whether memory should respond with data or broadcast the request to all the caches. Snooping has a separate address network (ordered) and data network (unordered). Directory is a typical directory-based protocol with a typical full-map directory. An entry in the directory includes the list of all sharers and the owner for one block. We designed Snooping and Directory for high performance; both protocols use many transient states in order to avoid stalling when messages arrive at coherence controllers.

A. *Target System and Configuration*

We evaluate TreeFractal using a full-system simulator, Virtutech Simics [17], extended with the Wisconsin GEMS toolset [20]. GEMS enables us to model the timing of the memory system. We compared TreeFractal to Snooping and Directory. For all three 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 Tags and directories using Cacti [31]. We simulate a CMP system with 2, 4, 8 and 16 cores. Each core is attached to a private L1 cache and private L2 cache and part of the memory. The system parameters are shown in Table 1.

B. *Performance Results and Analysis*

In this section, we quantitatively compare the performance of TreeFractal to Snooping and Directory. We use several benchmarks from the SPLASH-2 benchmark suite [32] and two commercial benchmarks, Apache and SPECjbb. All benchmarks have already been warmed up and checkpointed to avoid cold cache misses. Because of the inherent variability in parallel workload runtime [2], we ran each benchmark multiple times with small pseudo-random perturbations of the memory latency and averaged the results of all runs. Fig. 9 shows the runtime (lower is better) for the three protocols normalized to the runtime of Directory. The error bars represent +/- one standard deviation. From Fig. 9, we can see that TreeFractal performs comparably to Snooping and Directory. For all the benchmarks, the performance degradation is up to 11% compared to Directory, and up to 13% compared to Snooping.

We observe that for almost all benchmarks with 2 or 4 cores ─ except SPECjbb with 2 and 4 cores and volrend with 4 cores ─ TreeFractal outperforms Snooping and Directory. The performance improvement of TreeFractal over Directory can be as large as 65.3% (in Apache). This performance improvement is due to two reasons. First, for smaller

TABLE I. SYSTEM CONFIGURATION

Common Parameters for Three Protocols	
Processor parameters	
Number of cores	2, 4, 8, 16
Clock frequency	2 GHz
Cache parameters	
Cache line size	64 byte
Split L1 I&D cache	32 KB, 2 way, 2 cycle
Private L2 cache	512 KB, 2 way, 6 cycle
L1 and L2 exclusive	yes
Memory parameters	
Memory	2 GB, 160 cycle
Network parameters	
Link bandwidth	32 GB/s
Link latency	1 cycle
Specific Parameters for TreeFractal	
Level_1 Tag	144 KB, 4 way, 6 cycle
Level_2 Tag	288 KB, 8 way, 8 cycle
Level_3 Tag	576 KB, 16 way, 14 cycle
Level_4 Tag	1152 KB, 32 way, 24 cycle
Topology	Tree
Specific Parameters for Snooping	
Topology	Tree
Specific Parameters for Directory	
Directory for 2 nodes	20 MB, direct-mapped, 45 cycle
Directory for 4 nodes	32 MB, direct-mapped, 55 cycle
Directory for 8 nodes	52 MB, direct-mapped, 65 cycle
Directory for 16 nodes	88 MB, direct-mapped, 85 cycle
Topology	2D Torus

configurations, it takes much less time for TreeFractal to access the Tag than for Directory to access the directory because the Tag is on chip and much smaller than the directory. Second, as mentioned in Section III.B, in TreeFractal, we have a node in S respond to coherence requests with data instead of having the memory respond to the requests as is done in both Snooping and Directory. This method improves performance because the cache is much smaller than the memory and it is on chip and takes less time to access. To confirm this hypothesis, we compared the ratio of the number of coherence requests arriving at state S to the total number of coherence requests. The ratio for Apache with 2 cores is 0.3, but the ratio for SPECjbb with 2 cores is only 0.1. This statistic means that, for Apache with 2 cores, TreeFractal has more chances to reduce the latency by having a node in S respond to the requestor.

As the number of cores increases, the advantages of having shorter Tag access latency and having a sharer respond to the requestor are reduced by the greater number of hops and larger Tag sizes in TreeFractal. However, even at 16 cores, TreeFractal still maintains performance that is comparable to Snooping and Directory. The results vary across the benchmarks, and we discuss two situations in which TreeFractal is outperformed. First, in Water, TreeFractal is outperformed by Snooping. On this benchmark, the root switch utilization of Snooping is only 1%, which is very low. Snooping is, unsurprisingly, performing well in a system with ample bandwidth for the given traffic. However, for other benchmarks that place more demand on the interconnection network, Snooping's performance does not scale well. The second benchmark we discuss is Apache. On Apache, TreeFractal performs 11% worse than Directory, but still 5% better than Snooping. From the statistics, we found the root switch utilization for Snooping is over 65%, which implies a possible bottleneck, while the root switch utilization for TreeFractal is only 15%. This data means that TreeFractal is less sensitive to the link bandwidth compared to Snooping. Therefore, TreeFractal's performance may not scale as well as Directory but it is more scalable than Snooping.

We further studied the impact of on-chip caching on Directory and TreeFractal. We found that both of them would benefit from on-chip caching of the storage structures they use, the directories and Tags, respectively. Because there are so many different possible caching schemes—different sizes, associativities, and latencies—we explored the potential of caching rather than any particular caching implementations. For both Directory and TreeFractal, we performed experiments in which we assumed perfect caching of directories and Tags; every cache access is a 1-cycle hit. The result is shown in Fig. 10. We see that the improvement in performance varies across different benchmarks and different numbers of cores. However, for the same benchmark and number of cores, the ranges of improvement for Directory and TreeFractal are similar. The results confirm that caching can benefit both Directory and TreeFractal and that their performances remain comparable with caching.

VI. FRACTAL COHERENCE DESIGN SPACE

As mentioned in Section III, there are different methods to design a Fractal Coherence protocol. One method is leveraging existing protocols and making them fractal by modifications. Note that existing snooping or directory protocols are not inherently fractal. One might think of connecting the nodes in an interconnection network that appears fractal in structure and implementing an existing protocol for it. However, these protocols do not have the support for the self-similarity in fractal behavior since this property is not a constraint in their designs. Consider a directory protocol where a single node issues a Get-Shared (GetS) coherence request. Making the protocol fractal by just attaching more cores to the interconnection network will not

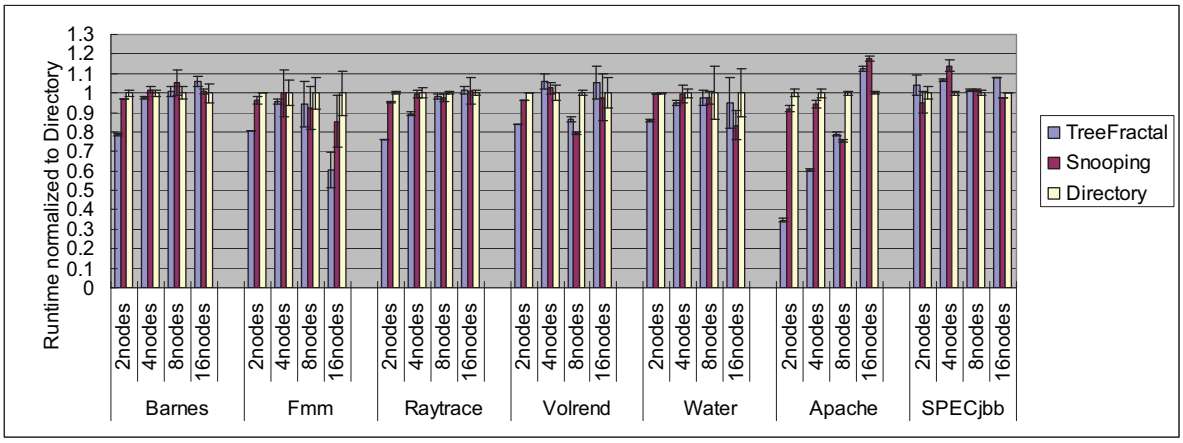


Figure 9. Runtime normalized to Directory

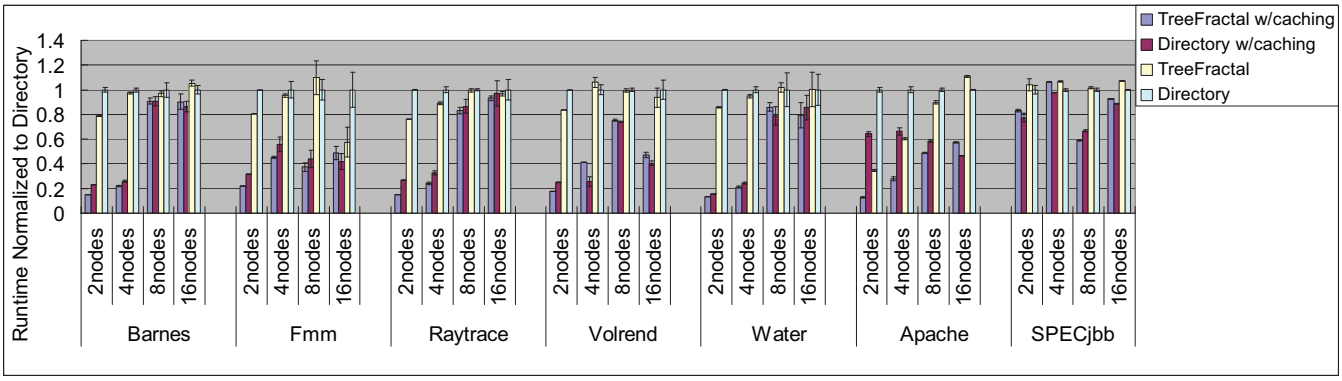


Figure 10. Runtime of on-chip caching protocols normalized to Directory

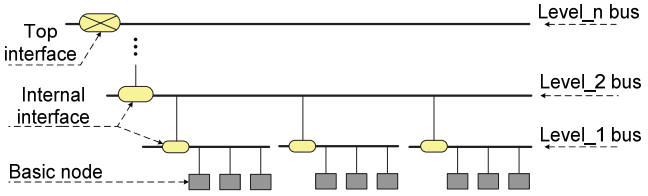


Figure 11. Making a traditional Snooping protocol fractal

lead to the node as a whole issuing a GetS in the same situation.

We now give an example of how to make a traditional snooping protocol fractal. As shown in Fig. 11, three basic nodes snoop on a level_1 bus. We attach an internal interface to this bus. The internal interface monitors all the transactions on this bus and determines which requests need to be forwarded to the higher level bus above it and which requests can be handled locally. Note that the internal interface must function in a way that guarantees that the three basic nodes beneath it behave the same as a single node when seen from the level_2 bus. By adding a number of internal interfaces and a top interface, we make the system have fractal behavior. If we can formally verify the coherence of the minimum system (by definition, composed of a level_1 bus, a level_2 bus, and 5 basic nodes) and the fractal behavior, we can prove the coherence for any arbitrary N-node system.

VII. Related Work

Fractal Coherence is based on a hierarchical structure and aims to enable the formal verification of cache coherence protocols via architectural innovation. Therefore, our work is related to all these aspects.

A. *Hierarchical coherence protocols*

Wilson [1] designed a large scale multiprocessor architecture based on hierarchies of shared snooping buses and caches in order to improve performance. DASH [15] employs both snooping protocols and directory protocols for different levels. Marty et al. [22] used a two-level virtual coherence hierarchy to support server consolidation. Fractal Coherence differs from previous hierarchical protocols in two important ways. First, as mentioned in Section VI, hierarchical coherence protocols are not inherently fractal, like Fractal Coherence, which makes them difficult to verify. Hierarchical coherence protocols, in fact, exacerbate the state explosion problem because they usually couple two protocols and thus add more corner cases [21]. Second, Fractal Coherence does not have different protocols for different levels while normal hierarchical protocols usually do. Applying one protocol for all levels simplifies the design.

B. Formal verification of cache coherence protocols

Clarke et al. [9] used SMV to formally model and verify a cache coherence protocol described in the IEEE Futurebus+ standard and found non-trivial bugs, but their largest configuration is only 8 processors. To make the formal verification of cache coherence protocols more scalable, researchers have pursued two paths.

One approach is optimizing automated tools to mitigate state space explosion [8, 14, 23, 27, 30]. However, many of them just postpone the state explosion problem instead of solving it. Others need the designer's experience in correctly modeling the protocol and have the risk verifying a different model from the real implementation. Moreover, some methods can be used only for certain protocols and cannot check liveness.

The other approach is using semi-automated theorem proving methods [16, 26]. Theorem proving uses extensive user guidance to perform a mathematical proof and is theoretically more scalable than model checking, but it is error-prone as a result of human intervention and not widely used due to the laborious verification work.

Different from these approaches, we divert our attention from the verification process, and instead focus on the architectural design. We want to ease the verification effort by designing architectures so that they can be verified with existing, fully automated formal tools. In this way, verification of cache coherence will not rely on the development of formal verification tools, which has long lagged behind architectural improvement.

C. Comparing the verification effort for different cache coherence protocols

There is another kind of work that analyzes the verification effort of existing or proposed protocols. Martin [19] argues that directory protocols are superior to snooping protocols with regard to formal verification effort. His conclusion is based on qualitative analysis. Marty [21] compared the formal verification efforts of different cache coherence protocol designs and showed their protocol is more amenable to formal verification. However, they did not design the architecture to ease verification. Our work is different from the above research in that we encourage architects to consider verification effort as a design constraint and incorporate it in early design stages.

VIII. Conclusion

Formal methods have gained importance in the verification of cache coherence protocols because the use of simulation to test protocols is unable to catch subtle bugs. Existing automated verification tools cannot handle large systems due to the state explosion problem. From the architects' perspective, we propose Fractal Coherence, a class of scalably verifiable coherence protocols. Fractal Coherence leverages the self-similarity of the fractal to enable the verification of any arbitrary N-node system. The verification of Fractal Coherence protocols is simplified to two straightforward, automated steps and does not incur the state explosion problem. We designed a Fractal Coherence protocol, TreeFractal, and verified it. By comparison to traditional snooping and directory protocols, we show that TreeFractal has comparable performance while maintaining the correctness guaranteed by formal methods.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CCF-0702434 and CCF-0811290. We thank Steven German, Alan Hu, and Paul Loewenstein for helpful consultations regarding formal verification. We thank Milo Martin for helpful feedback on an earlier draft of this paper.

References

[1] J. A. W. Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proc. of the 14th Annual Int'l Symposium on Computer Architecture*, 1987.

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

[3] Arvind, N. Dave and M. Katelman. Getting Formal Verification into Design Flow. In *Proc. of Formal Methods*, 2008.

[4] L. A. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of the 27th Int'l Symposium on Computer Architecture*, 2000.

[5] B. Bentley. Validating the Intel® Pentium® 4 Microprocessor. In *Proc. of Int'l Conference on Dependable Systems and Networks*, 2001.

[6] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: A Modular Tool for on-the-Fly Equivalence Checking. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[7] S. Burckhardt , R. Alur, and M. M. K. Martin. Verifying Safety *of* a Token Coherence Implementation by Parametric Compositional Refinement. In *Proc. of Verification, Model Checking, and Abstract Interpretation*, 2005.

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

[9] E. M. Clarke et al. Verification of the Futurebus+ Cache Coherence Protocol. *Computer Hardware Description Languages and Their Applications,* vol. 32, 1993.

[10] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys,* vol. 28, 1996.

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

[12] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of Computer Aided Verification*, 2007.

[13] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of Alphaserver GS320. In *Proc. of*

the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[14] C. N. Ip and D. L. Dill. Better Verification through Symmetry. *Computer Hardware Description Languages and Their Applications,* vol. 32, 1993.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor. In *Proc. of the 17th annual Int'l symposium on Computer Architecture*, 1990.

[16] P. Loewenstein. Verification of a Multiprocessor Cache Protocol Using Simulation Relations and Higher-Order Logic. *Lecture Notes in Computer Science,* vol. 531, 1991.

[17] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *Computer,* vol. 35, 2002.

[18] B. B. Mandelbrot. *The Fractal Geometry of Nature: W.H. Freeman and Company*, 1982.

[19] 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.

[20] M. M. K. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (Gems) Toolset. *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.

[21] M. R. Marty et al. Improving Multiple-Cmp Systems Using Token Coherence. In *Proc. of the 11th Int'l Symposium on High-Performance Computer Architecture*, 2005.

[22] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proc. of the 34th Annual Int'l Symposium on Computer Architecture*, 2007.

[23] K. L. Mcmillan and C. B. Labs. 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*, 2001.

[24] G. J. Milne. Design for Verifiability. *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, vol. 408, 1990.

[25] R. Milner. A Calculus of Communicating Systems. *Journal of Computer and System Sciences,* vol. 28, 1984.

[26] S. Park and D. L. Dill. Verification of FLASH Cache Coherence protocol by Aggregation of Distributed Transactions. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[27] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems,* vol. 6, 1995.

[28] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-Based Cache Coherence Protocols: S3.mp, a Case Study. In *Proc. of the 1st Int'l EURO-PAR Conference*, 1995.

[29] S. Srinivasan, P. S. Chhabra, P. K. Jaini, A. Aziz, and L. John. Formal Verification of a Snoop-Based Cache Coherence Protocol Using Symbolic Model Checking. In *Proc. of the 12th Int'l Conference on VLSI Design*, 1999.

[30] U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Proc. of Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.

[31] S. Thoziyoor et al. Cacti 5.1. *HP Technical Report,* 2008.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture*, 1995.

[33] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design & Test,* vol. 7, 1990.