Scalably Verifiable Cache Coherence

by

Meng Zhang

Department of Electrical and Computer Engineering
Duke University

Date:_____

Approved:

_____
Daniel J. Sorin, Supervisor

_____
Chris Dwyer

_____
Alvin R. Lebeck

_____
Benjamin C. Lee

_____
Bruce M. Maggs

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University

2013

ABSTRACT

Scalably Verifiable Cache Coherence

by

Meng Zhang

Department of Electrical and Computer Engineering
Duke University

Date:_____

Approved:

_____
Daniel J. Sorin, Supervisor

_____
Chris Dwyer

_____
Alvin R. Lebeck

_____
Benjamin C. Lee

_____
Bruce M. Maggs

An abstract of a dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University

2013

Copyright by
Meng Zhang
2013

Abstract

The correctness of a cache coherence protocol is crucial to the system since a subtle bug in the protocol may lead to disastrous consequences. However, the verification of a cache coherence protocol is never an easy task due to the complexity of the protocol. Moreover, as more and more cores are compressed into a single chip, there is an urge for the cache coherence protocol to have higher performance, lower power consumption, and less storage overhead. People perform various optimizations to meet these goals, which unfortunately, further exacerbate the verification problem. The current situation is that there are no efficient and universal methods for verifying a realistic cache coherence protocol for a many-core system.

We, as architects, believe that we can alleviate the verification problem by changing the traditional design paradigm. We suggest taking verifiability as a first-class design constraint, just as we do with other traditional metrics, such as performance, power consumption, and area overhead. To do this, we need to incorporate verification effort in the early design stage of a cache coherence protocol and make wise design decisions regarding the verifiability. Such a protocol will be amenable to verification and easier to be verified in a later stage. Specifically, we propose two methods in this thesis for designing scalably verifiable cache coherence protocols.

The first method is Fractal Coherence, targeting verifiable hierarchical protocols. Fractal Coherence leverages the fractal idea to design a cache coherence protocol. The self-similarity of the fractal enables the inductive verification of the protocol. Such a verification process is independent of the number of nodes and thus is scalable. We also design example protocols to show that Fractal Coherence protocols can attain comparable performance compared to a traditional snooping or directory protocol.

As a system scales hierarchically, Fractal Coherence can perfectly solve the verification problem of the implemented cache coherence protocol. However, Fractal Coherence cannot help if the system scales horizontally. Therefore, we propose the second method, PVCoherence, targeting verifiable flat protocols. PVCoherence is based on parametric verification, a widely used method for verifying the coherence of a flat protocol with infinite number of nodes. PVCoherence captures the fundamental requirements and limitations of parametric verification and proposes a set of guidelines for designing cache coherence protocols that are compatible with parametric verification. As long as designers follow these guidelines, their protocols can be easily verified.

We further show that Fractal Coherence and PVCoherence can also facilitate the verification of memory consistency, another extremely challenging problem. One piece of previous work proves that the verification of memory consistency can be decomposed into three steps. The most complex and non-scalable step is the verification of the cache

coherence protocol. If we design the protocol following the design methodology of Fractal Coherence or PVCoherence, we can easily verify the cache coherence protocol and overcome the biggest obstacle in the verification of memory consistency.

As system expands and cache coherence protocols get more complex, the verification problem of the protocol becomes more prominent. We believe it is time to reconsider the traditional design flow in which verification is totally separated from the design stage. We show that by incorporating the verifiability in the early design stage and designing protocols to be scalably verifiable in the first place, we can greatly reduce the burden of verification. Meanwhile, we perform various experiments and show that we do not lose benefits in performance as well as in other metrics when we obtain the correctness guarantee.

Dedication

To my parents.

Contents

Abstract iv

List of Tables xii

List of Figures xiii

Acknowledgements xv

1. Introduction 1

1.1 Cache Coherence in Many-core Era 2

1.2 Verification Problem of Cache Coherence Protocols 4

1.3 Incorporating Verification into Cache Coherence Design 6

1.4 Thesis Statement and Contributions 8

1.5 Thesis Outline 10

2. Background and Related Work 12

2.1 Verification Definition and Approaches 12

2.1.1 Verification through Simulation 13

2.1.2 Verification through Formal Methods 13

2.1.2.1 Model Checking 14

2.1.2.2 Theorem Proving 16

2.1.3.3 Combining Model Checking and Theorem Proving 17

2.2 Design Space of Cache Coherence Protocols 18

2.3 Status and Challenges of Cache Coherence Verification 21

2.3.1 Verification of Flat Cache Coherence Protocols 22

2.3.2 Verification of Hierarchical Cache Coherence Protocols 25
2.4 Designing Verifiable Cache Coherence Protocols 27
2.4.1 Related Work in Design for Verifiability 28
2.4.2 Verifiability Analysis of Cache Coherence Protocols 29
3. Fractal Coherence: Verifiable Hierarchical Cache Coherence 31
3.1 Concept of Fractal Coherence 31
3.2 System Architecture 33
3.3 Verification Methodology 35
3.3.1 Verification of Minimum System 35
3.3.2 Verification of Fractal Behavior 37
3.3.3 Proof of Cache Coherence for Arbitrary N-node System 39
3.4 Case Study: TreeFractal 43
3.4.1 System Design 43
3.4.1.1 Two-node System Design 44
3.4.1.2 Scaled System Design 46
3.4.2 Verification Procedure and Results 51
3.4.2.1 Verification of Minimum System 52
3.4.2.2 Equivalence Checking of Fractal Behavior 53
3.4.3 Evaluation 54
3.4.3.1 Storage Overhead 54
3.4.3.2 Simulation Methodology 57
3.4.3.3 Performance Results and Analysis 58

3.5 Fractal Directory: Extension of TreeFractal 62
3.5.1 Insufficiency of TreeFractal 63
3.5.2 System Design of Fractal Directory 63
3.5.3 Performance Evaluation 65
3.5.4 Difficulty in Verification Process 66
3.5.4.1 State Explosion Problem in Verification 68
3.5.4.2 Reducing State Space by Optimizing Verification Process 68
3.5.4.3 Demanding for a Verifiable Flat Protocol 73
3.6 Design Space of Fractal Coherence 74
3.7 Summary 76
4. PVCoherence: Verifiable Flat Cache Coherence 77
4.1 Parametric Verification for Cache Coherence 78
4.1.1 Different Approaches to Parametric Verification 78
4.1.2 A Mostly Automated Method: Simple-PV 79
4.2 Limitation of Simple-PV 84
4.3 System Overview of PVCoherence 86
4.4 Design Guidelines for PVCoherence Protocols 88
4.5 Case Study: PV-MOESI 99
4.5.1 An Optimized Protocol: OP-MOESI 99
4.5.2 Converting OP-MOESI to PV-MOESI 100
4.5.3 Verification of PV-MOESI 104
4.5.4 Evaluation 106

4.5.4.1 Methodology and System Configuration 106
4.5.4.2 Performance Results 107
4.5.4.3 Scalability Analysis 109
4.5.4.4 Storage Overhead 109
4.6 Combining Fractal Coherence and PVCoherence 111
4.7 Summary 113
5. Leveraging Fractal Coherence and PVCoherence to Verify Memory Consistency 115
5.1 Difference between Cache Coherence and Memory Consistency 116
5.2 Decomposing the Verification of Memory Consistency 117
5.3 Architecting Memory System to Facilitate the Verification of Consistency 119
6. Conclusions and Future Work 124
References 128
Biography 137

List of Tables

Table 1: System configuration of TreeFractal and baselines 59

Table 2: System configuration of Fractal Directory and baseline 67

Table 3: High-level specifications of OP-MOESI and PV-MOESI. Ignores transient states. Differences between OP-MOESI and PV-MOESI are in bold font in PV-MOESI specification. 102

Table 4: Simulation Configurations of PV-MOESI and OP-MOESI 107

Table 5: TSO ordering 121

List of Figures

Figure 1: Scalability problem in verification of cache coherence 32

Figure 2: Possible binary tree structures 34

Figure 3: Minimum systems of different degree trees 34

Figure 4: Observational equivalence for maintaining fractal behavior 38

Figure 5: Lemmas for proof of cache coherence in any N-node system 43

Figure 6: System architecture of TreeFractal 47

Figure 7: An example of naïve design which violates the fractal behavior 48

Figure 8: Correct implementation to ensure fractal behavior 50

Figure 9: Optimized observational equivalence checking 55

Figure 10: Runtime normalized to Directory 62

Figure 11: Runtime of on-chip caching protocols normalized to Directory 62

Figure 12: System structure of Fractal Directory 65

Figure 13: Runtime of Fractal Directory 67

Figure 14: Speedup with "Recall" 68

Figure 15: Minimum system of Fractal Directory 69

Figure 16: Observtional equivalence checking of Fractal Directory 69

Figure 17: Reduce the state space for verification 74

Figure 18: Making a traditional Snooping protocol fractal 75

Figure 19: Simple-PV Verification Process 80

Figure 20: Parametric Model 81

Figure 21: System Architecture of PVcoherence 87
Figure 22: Components impacted by Guildeline #2 90
Figure 23: Scenario #2 forbidden by Guideline #3 92
Figure 24: Runtime comparison: OP-MOESI vs PV-MOESI 110
Figure 25: Network traffic overhead of PV-MOESI 110
Figure 26: Performance Scalability 110
Figure 27: Architecture for shared-memory system 121

Acknowledgements

I have received substantial help from my mentors, colleagues and families throughout my PhD study at Duke. I would like to express my great gratitude towards them. Without their supports, it is impossible for me to complete the studies and be able to write this thesis.

First of all, I would like to thank my advisor, Daniel Sorin, one of the greatest mentors I have ever seen. Dan not only introduces me into this field, supervises me on my research work, but also impacts me on my life attitude with his personality. Through numerous discussions with him, I learnt how to think deeply and creatively, how to improve communication and writing skills, how to manage time and balance life. As I overcome the difficulties during my studies, Dan has been always patient with me and influences me with his passion and determination. I will continue to emulate his example in my career. I am also grateful to Prof. Alvin Lebeck, who collaborates with us on two of our papers. His extensive experience and insightful thoughts are valuable to our work, helping bring them to the public. I would also like to thank all other committee members, Prof. Chris Dwyer, Prof. Ben Lee, and Prof. Bruce Maggs for taking their time to serve on my committee, and also for the feedback they have provided.

I am very fortunate to be in the Duke computer architecture group. It includes a lot of talented and interesting people, who contribute to the pleasure and freedom

atmosphere. Particularly, I would like to thank a few individuals among them. Bogdan Romanescu helped me speed and complete my very first project at Duke. He also gave me useful suggestions on PhD life and career plan. Anita Lungu not only worked with me on my first paper, but also being a good friend to talk and share feelings. It is a pleasure to sit next to Blake Hechtman, who is always energetic and enthusiastic with research. Blake gave me useful feedback on my work and helped me a lot in solving the infrastructure problems. There are still many other peers in this group that have helped me and made my graduate life easier: Fred Bower, Vince Mao, Adam Jacobvitz, and Ralph Nathan.

Finally, my greatest gratitude goes to my husband, Dongtao and my son, Grayson. The life of a PhD student can be tough and dull, but they have lightened it up. I would like to thank Dongtao for cheering me up every time I was in low spirit and discussing with me every time I raise a question. He is continuously supporting and caring for me. My little boy, who cannot speak yet, gives me big smiles every day, encouraging me to pursue the study even in the most difficult time. I also want to thank my mom, who left her hometown and stay with us helping take care of Grayson. If it were not her help, I would not be able to complete this thesis in time. I am forever indebted to them.

1. Introduction

Single-core systems have given way to multi-core systems, or more accurately, many-core systems, which may have hundreds or even thousands of cores on a single chip. To better utilize the underlying hardware resources, parallel programming is becoming ubiquitous. However, it is extremely challenging, even for experts, to develop correct and efficient parallel programs. A shared memory is an effective way to alleviate the burden of parallel programming, since it provides the programmer with the view of a single memory address space, which greatly reduces the need of explicit data partitioning and movement. In a shared-memory many-core system, all cores have access to the entire memory locations. The communications among different cores are via the read and write operations to the shared memory.

However, together with the benefit for parallel programming, the shared-memory paradigm also brings several problems, an important one among which is cache coherence. Cache coherence problem occurs when there are one or multiple levels of caches sitting between the cores and the memory. Each core has its private cache and may or may not share caches with other cores. Such cache hierarchies greatly reduce memory accesses and improve performance, but cause multiple copies of the same block to exist in the system. Data inconsistency may happen in these copies, either between the cache and the memory, or among the caches themselves. Both scenarios can lead to a

read being unable to retrieve the updated value of the latest write. A read of a stale value can ruin the correctness of the program. Therefore, we must ensure that all writes are correctly propagated in a timely fashion and data of the same block have a consistent view across all copies. To achieve this goal, we need a mechanism to coordinate the behaviors of all the caches as well as the cores and the memory. Such a mechanism is called a cache coherence protocol. In particular, we confine our discussion to hardware cache coherence, which is implemented in most of today's computer systems. Software cache coherence is beyond the scope of this thesis.

1.1 Cache Coherence in Many-core Era

In a cache coherence protocol, every participating component (cores, caches, and memory) is described as a state machine, and the protocol itself is a compound state machine with all these components interacting with each other. Cache coherence protocols may have quite different features according to various system requirements. No matter what specific design choices it has made, a cache coherence protocol must satisfy two invariants to ensure correctness. The first one is permission invariant. It says that the protocol must enforce the so-called single-writer, multiple-reader (SWMR) invariant [89]. SWMR means for each block of memory, at any given time, the block either has a single writer or zero or more readers. The second one is data invariant, which states that the protocol must ensure that a read of a block returns the value of the

most recent write to that block. Although cache coherence protocols may have quite different implementations, as long as the protocol can be verified to satisfy these two invariants, we are ensured that the protocol does not have a bug.

As a key factor in the overall cost and performance, cache coherence has been under extensive research for a long period and achieved great improvement. Now, with the advent of many-core era, the design of cache coherence protocols meets a new challenge: scalability. The ever expanding system requires the cache coherence protocol to scale in all different aspects. The first aspect is performance. Most previous optimizations for a cache coherence protocol have performance gain as the ultimate goal. It is a question, though, whether the performance enhancement can still be maintained as the system scales, as we know that some optimizations are naturally not scalable. Secondly, the storage overhead and power consumption of a cache coherence protocol also need to be taken into consideration for scalability. Nowadays, a single chip, which only has a limited area and power budget, may integrate a variety of functional units. So we would like to minimize the resources allocated to the cache coherence protocol. Finally, as the components in the system increase, the utilization of the interconnection network increases. It is important that the cache coherence protocol only consumes a modest bandwidth and does not lead to a burst or congestion in the network.

To satisfy the above requirements in scalability, people have designed highly aggressive cache coherence protocols with a variety of optimization techniques for current cache hierarchies, which may include one or multiple levels of caches. However, there is an important aspect that is usually overlooked, that is, whether we are able to verify the designed scalable cache coherence protocol is correct in the first place.

1.2 Verification Problem of Cache Coherence Protocols

A buggy cache coherence protocol might lead to a catastrophic failure of the shared-memory system that employs this protocol. It is quite important to carefully design and inspect the protocol. However, the carefulness is insufficient to ensure the correctness of a modern cache coherence protocol. To ensure the reliability of the protocol, verification must be done as part of implementing cache coherence protocols for real systems. This type of verification is actually "***design verification***", which occurs before the product is manufactured. Whenever we mention "verification" in the thesis, we are referring "design verification".

There are two major approaches to performing the verification of a cache coherence protocol: simulation and formal verification. Simulation is to run benchmarks, stress tests, and random code sequences and check the output to see whether the cache coherence protocol is correct. Formal verification, on the contrary, mathematically proves that the cache coherence protocol satisfies the correctness properties. The main

difference between the two methods is that simulation only explores certain path of the state space, while formal verification explores the whole reachable state space.

Simulation or formal verification may work for very simple or unrealistic cache coherence protocols, but both of them have difficulty in verifying modern cache coherence protocols implemented in today's computer systems. As the size and complexity of a cache coherence protocol increase, simulation may only cover a small percentage of paths no matter how long it runs. This inability to reach all system states limits the ability of simulation methods to find subtle bugs. Previous work [19], [26], [33], [86] shows cache coherence protocols can still have bugs even after extensive simulations. Therefore, recent research has been focusing on formal verification of cache coherence protocols, which is a complete method. Formal verification also has its unavoidable problems. The most prominent problem is formal verification fundamentally requires a huge amount of resources, either in the form of memory, runtime or human efforts, which exceeds the capability that current formal verification tools could provide. As the system scales, the requirement also goes up. The increasing complexity of cache coherence protocols further exacerbates the problem. To achieve high performance, current cache coherence protocols allow multiple outstanding requests and concurrent operations. This concurrency implies numerous transient states and a much bigger state space. Moreover, there is a trend towards hierarchical cache

coherence protocols. Several levels of protocols interacting with each other further complicates the verification [23].

To allow people to determine the correctness of the designed cache coherence protocol, the verification of the cache coherence protocol, just as all the other aspects, like performance, power consumption, etc., also needs to scale as the system expands and gets more complex. Unfortunately, few people have ever considered the scalability of verification and we would like to explore this area in this thesis.

1.3 Incorporating Verification into Cache Coherence Design

In a traditional design process, verification is performed at a late stage where most design choices have already been made. It is the verification team's responsibility to try their best to get the verification work done. However, it is acknowledged that verification consumes a large amount of resources during the whole process, increasing both the cost and the time to market. For example, when verifying the Pentium 4 processor, Bob Bentley and Rand Gray formed a large team and spent several years performing the verification [4]. It has been estimated that about 60-70 percent of the development cost of a system is spent on verification [1], [46]. Even with such a great effort consumed, verification still cannot fulfill all its responsibilities on many occasions. Considering this fact, the traditional design paradigm may not be optimal.

Taking the cache coherence protocol design as an example, there is a tension between performance and verification difficulty. Architects make great efforts to improve the performance of the protocol, which often leads to increasing amounts of parallelism and larger state space that needs to be explored. The increased parallelism and state space make the verification of protocols more difficult. In the traditional design flow, priority is given to the architects and they focus only on the performance. They are allowed to do any optimizations they want to achieve that goal. Then the cache coherence protocol is passed to the verification team, who are responsible for trying out various techniques to prove the correctness of the protocol. Unfortunately, the cache coherence protocol is usually too complex for a complete proof. The verification team thus has to either verify a scaled down system, which does not necessarily guarantee the correctness of a larger system, or employ some incomplete methods (e.g., simulation) to get the verification done. Neither of these methods is satisfying.

However, the above dilemma does not necessarily need to be the case. If the designers have more information about what features the verification people would prefer and what they would like to avoid, they may have made wiser design choices. This information is particularly helpful when the intuition of the architect to design the system is not the same as what happens later in the verification stage. For example, a specific feature that improves the performance only a tiny bit may actually lead to a

disaster for the verification team, or a feature the architect discards considering it not useful may significantly ease the verification. We would like to avoid both of these situations.

We propose to incorporate verifiability of the cache coherence protocol into the early design stage instead of considering it late. This idea follows the concept of "design for verifiability" presented by Milne [74], which is a counterpart to the "design for testability" in the formal verification area. We expect that taking formal verification effort as a first-class design constraint has the potential to ease verification effort, improve product quality, and reduce a product's time to market.

1.4 Thesis Statement and Contributions

In summary, we make the following contributions:

1. We propose to incorporate verification into the design stage of cache coherence protocols in order to ease verification. Specifically, we suggest to tradeoff between verifiability and other design aspect, such as performance, power consumption, etc., so that the system is scalably verifiable.
2. We propose Fractal Coherence: a design methodology for hierarchical 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 scale of 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. We experimentally evaluate TreeFractal using full system simulation and show it has comparable performance to traditional cache coherence protocols without adding significant implementation costs.

3. We propose PVCoherence which provides a set of design guidelines for flat cache coherence protocols design. Following these guidelines, architects can design flat cache coherence protocols that can be parametrically verified with any number of nodes. We describe the design process of a PVCoherence protocol, called PV-MOESI, that can be verified with model checking tools and limited human intervention for any arbitrary number of nodes. We experimentally compare the performance, network traffic, scalability, and storage overhead of PV-MOESI and a highly optimized protocol, OP-MOESI, that cannot be verified with Simple-PV. The results show that following the guidelines we present does not seriously impact the protocol's performance or costs.

4. We extend the idea of scalably verifiable cache coherence to the domain of memory consistency verification. We show that using a scalably verifiable cache coherence protocol in the system can ease the verification of memory consistency model implementation.

1.5 Thesis Outline

In the following parts of the thesis, Chapter 2 provides some background knowledge about verification and the approaches to performing verification. It also discusses the basic concept of cache coherence protocols, current status of cache coherence verification and the related work in the field of design for verification. Chapter 4 through Chapter 6 describe the major three contributions of the thesis. Chapter 3 discusses Fractal Coherence, a method to design a cache coherence protocol in a fractal way so that the self-similarity enables the protocol to be inductively verified independent of the number of nodes. It also provides the design process of specific Fractal Coherence protocols. Any protocol from the family of Fractal Coherence must have a hierarchical structure. Chapter 4 proposes PVCoherence to design a flat protocol which can be scalably verified and describes how to convert a common protocol to a PVCoherence protocol. PVCoherence is a complement of Fractal Coherence. Chapter 5 extends the discussion to the verification of memory consistency and talks about how

Fractal Coherence and PVCoherence can facilitate the verification of memory consistency. Chapter 6 concludes the paper and proposes some future work.

2. Background and Related Work

This chapter provides some background knowledge of verification and describes related work in the verification of cache coherence protocols. We first introduce the concept of verification and different approaches of verification (Section 2.1). After that, we provide an overview of cache coherence protocols design space (Section 2.2). Then we describe the related work in the verification of cache coherence protocols and point out the existing problem and difficulty in these verification techniques (Section 2.3). Finally, we illustrate the motivation for the proposal of designing a verifiable cache coherence protocol and talk about previous work in design for verification as well as in verifiability analysis of cache coherence protocols (Section 2.4).

2.1 Verification Definition and Approaches

Verification is the process of checking whether the developed system meets a set of specifications. Verification process, on the contrary to design process, starts from the implementation of the system and ends up in confirming the implementation complies with the specification. During this process, we may find design bugs which we must fix. There are two main approaches to performing verification: simulation-based verification and formal verification. The following sub-sections discuss the pros and cons of them in more detail. Particularly, since this thesis focuses on formal verification, the majority of this section will discuss formal verification.

2.1.1 Verification through Simulation

Simulation-based verification is to run the simulator as many cycles as possible to uncover design bugs. The inputs to the simulator are a variety of test cases, which can be either manually generated test vectors or pseudo-random inputs. From the methodology point of view, simulation may apply to any system no matter how big it is since we can designate how many resources can be allocated to the simulator and control for how long the simulation will run. However, the main disadvantage of simulation is it is usually incomplete. We can verify only those cases that are encountered during the simulation. As the design complexity increases, the ratio of the test cases over the overall problem state space declines. The lower the coverage, the less we can trust the simulation.

2.1.2 Verification through Formal Methods

Another important approach is formal verification, which mathematically proves that a design satisfies a set of properties. Both the design (an implementation) and the properties (a specification) are formally specified. The most important advantage of formal verification is that it is a complete method. Formal verification is able to find corner cases that might be missed in simulation. Completeness is quite important for current designs since they have become so complex and it is extremely difficult to pinpoint where the corner cases might be and how to manifest them through simulation.

Formal verification also has disadvantages compared to simulation, depending on which formal verification method is used. There are two main methods of formal verification: model checking and theorem proving. The following sub-sections discuss the two methods in more detail.

2.1.2.1 Model Checking

Model checking is a state-based method. It exhaustively examines the entire reachable state space of the system to check whether the desired properties hold. A typical model checking process includes three tasks: modeling, specification and verification [8]. Modeling is formally describing the system in a finite state machine description language; specification is formally specifying certain properties that the system must satisfy. We need to be careful that the specification exactly describes what we want from the system. Verification is then performed by a model checker that traverses all the reachable states and checks whether the model satisfies the specification. If the model checker finds a verification failure, then a counterexample will be generated.

The most prominent advantage of model checking over theorem proving is model checking is amenable to use. First, model checking is highly automated. A model checker can perform an automatic search procedure to determine whether the specification holds in all states without any human intervention. Second, model

checking facilitates the debugging process. As mentioned earlier, in face of a failure, a model checker will provide a counterexample, which is a sequence of states that leads to a violation of the specification. The counterexample gives useful information to the user to debug the system and find design errors.

However, model checking is limited by the state explosion problem. Since model checking employs an exhaustive state space traversal method, the resources used in the process, such as memory and runtime, will eventually become a bottleneck [3]. The traditional way to perform model checking is through explicit state enumeration, which explicitly stores the states in the table. To overcome the inefficiency and large memory requirement in the explicit state enumeration of model checking, there have been a number of proposals to compress the state space [47], [79], [93]. A notable one of them is symbolic algorithm [18], [68], which uses ordered binary decision diagrams (OBDDs) [17] to represent the transition relationship. An OBDD representation of the state graph is much more compact than the explicit state enumeration. However, none of these methods can fundamentally solve the state explosion problem; instead, they only defer the advent of the explosion. Generally speaking, current designs are usually too complex for model checking to thoroughly verify. What people often do is to abstract the model or reduce the scale of the system in order to formally verify it, which cannot guarantee

the correctness of the original system. Due to the state space explosion problem, model checking fails to provide a general solution for realistic systems.

2.1.2.2 Theorem Proving

Different from model checking, theorem proving is a proof-based method. It derives a proof from the desired properties to show that the system adheres to these properties. In theorem proving, a system is modeled in a more expressive way than in model checking, using higher-order logic or set theory. Then, a human-driven interactive correctness proof is performed. The verification process in theorem proving can be modularized and structured into layers. In this way, the proof is reduced to a series of sub-proofs to facilitate verification [43].

Theorem proving's superiority over model checking lies in its ability to deal with much more complex systems. It can employ a variety of techniques like structural induction to prove an infinite state system [27]. Moreover, some properties that are hard to describe in model checking can be easily described in theorem proving due to its high expressiveness.

The main problem with theorem proving is that it demands a significant human input to define and guide the proofs, meaning a great deal of expertise and time is needed [31]. This also means the verification process is prone to errors caused by human intervention. Another major disadvantage is that if the theorem prover fails to prove the

correctness of the system, we cannot obtain any information about why and how the failure happens. Not being able to provide counterexample makes it rather inconvenient to debug in theorem proving.

2.1.3.3 Combining Model Checking and Theorem Proving

We have described the two mainstream methods of formal verification, model checking and theorem proving. We can see they have strength and weakness in different aspects. Model checking is superior in its automaticity and amenability, while theorem proving is superior in its applicability and capability. With model checking, we can more easily and quickly verify a system, but current model checking tools saturate even when verifying a very small system. With theorem proving, we can verify more complex and larger systems, but the process of performing the verification is laborious and limited to very few experts.

Since both model checking and theorem proving have their own pros and cons, there are a few methodologies proposed to combine the two methods in a certain way. The goal is to enable the verification of complex systems in a more automatic fashion, and thus achieve the advantages of both worlds. There are two major approaches to combine model checking and theorem proving. The first approach is to add theorem proving techniques to model checkers [7], [66]. This approach enables model checkers to analyze large or even infinite state space by using techniques such as data abstraction,

assume-guarantee reasoning, etc. The second approach is to implement model checking algorithms in theorem provers [16], [29]. In the theorem proving process, if a subset of the problem can be translated into a finite state model checking problem, we can leverage the power of model checking algorithm to verify it. The details of these methods are beyond the scope of this thesis. We will only discuss a specific combinational method related to our research later in Section 4.

2.2 Design Space of Cache Coherence Protocols

The development of cache coherence protocol dates back to 1980s. We will not describe the history of cache coherence; instead, we provide an overview of the design space of current cache coherence protocols. The following aspects are among the most important design choices.

Protocol type. It is an important decision to choose the type of protocol we will use. Usually we have two options: snooping protocol and directory protocol. Snooping protocols usually rely on a bus to provide ordering for all coherence transactions. Requests that arrive at the bus will be broadcast to all the cores in the same order. Each core then snoops to see if it has a copy the data and responds accordingly. Snooping protocols are widely implemented in earlier small-scale machines because they are simple [34], [39], [40], [49], [81]. However, the broadcasting feature causes the major limitation for a snooping protocol to scale to large scale systems.

Directory protocols, on the other hand, avoid broadcasting and have better scalability, making it more popular in modern systems [3], [9], [56], [57], [77]. Directory protocols use a directory to record the state of each memory block. With this extra information, the protocol replaces the broadcasting with point to point messages. There are various methods to implement the directory. The most straightforward one is a full-map directory [57] which tracks the states of all memory blocks in every cache. However, due to the huge storage overhead, it is not commonly used in current computer systems. There have been a great number of optimization techniques proposed to reduce the size of the directory [20], [21], [78], [87].

States and transactions. A cache coherence protocol usually has several stable states and a few transient states. The stable states usually include M(odified), O(wned), E(xclusive), S(hared), I(nvalid). M means only this cache has a valid copy of the data, and the data in the memory is stale. S means the cache has a valid copy of the data and the data in the memory is also up-to-date. There can be other caches that also hold a valid copy of the data. I means the cache has no valid copy of data for this block. O and E are for optimization purpose. O enables a cache to respond to a read request without copying back the data to the memory. In this case, there can be multiple valid copies in caches, while the memory does not have an up-to-date copy. E allows a cache to upgrade to state M without asking for the permission. A cache in E state means the

cache and the memory both have valid copies of data, but no other cache has a valid copy of data.

In order to achieve high performance, people usually assume transactions to be non-atomic, so we need to introduce transient states. The number of transient states varies according to the aggressiveness of the protocol. The existence of transient states greatly increases the complexity of the cache coherence protocol.

Interconnection network. As the number of cores increases, how to connect those cores, caches, memories and directories becomes more important since the topology may impact the performance and power consumption. For a small number of cores, crossbar, bus, or ring may work well, as shown in [48], [51]. However, when the system scales, these topologies are not practical. Instead, we require more scalable topologies, such as torus, mesh, etc. Another important feature of the interconnection network is whether it provides point-to-point ordering. With ordering, the design of cache coherence protocol can be much easier since many race conditions can be avoided. However, unordered networks allow more optimizations, such as adaptive routing [24], [38], which provides more flexibility and avoids network congestion. Thus the ordering is another tradeoff architects need to consider before making the decision.

Write policy. There are two kinds of write policies: write-invalidate and write-update. Write-invalidate means a cache needs to invalidate all other caches before it

writes to the block, while write-update means the cache needs to update the data in all caches that have the block when it performs the write. The write-update policy propagates data faster than the write-invalidate policy, but it uses a lot more bandwidth and complicates the memory consistency [2], [89] . Due to this fact write-invalidate policy is more widely used in current cache coherence protocols.

Besides the above basic design options, there are still a variety of details that people can choose to improve their cache coherence protocols. These optimizations can help improve performance, reduce the storage overhead, or reduce the network traffic [45], [63], [91]. For example, in a directory protocol, non-blocking directory allows subsequent requests to proceed without being blocked at the directory. This technique can improve performance since it greatly reduces the waiting time on the critical path. We do not discuss various optmizations in this thesis as they are not related to our work, but it is worth mentioning that most of these optimizations can still apply to our proposed cache coherence protocols.

2.3 Status and Challenges of Cache Coherence Verification

To verify cache coherence protocols, formal verification is preferable to simulation-based verification. The reasons are tow-folded. First, as mentioned in Section 2.1, simulation has low coverage in face of complex systems, while formal verification is always complete. Current cache coherence protocols in many-core systems are so

aggressively optimized that a large number of concurrent events may occur. Not being able to gather all possible combinations of inputs and outputs, simulation is very likely to miss corner cases which are actually buggy. As an important functional unit in the memory hierarchy, a cache coherence protocol cannot tolerate bugs, which may lead to a system failure. We want a complete correctness guarantee of the protocol, which requires formal verification. A secondary reason for employing formal verification for cache coherence protocols is that the components involved in a protocol are usually described as several state machines that interact with each other. This state-based structure is amenable to the current formal verification tools since it is very convenient to model the cache coherence protocol in the tool.

Due to the above reasons, in the following sub-sections, we only include the research work related to formal verification of cache coherence protocols. Hereafter, whenever we mention verification, we only refer to formal verification.

2.3.1 Verification of Flat Cache Coherence Protocols

Most of the research done to verify cache coherence protocols is for flat (not hierarchical) protocols. Depending on which formal verification method the work uses, it can be categorized as a model checking based method, theorem proving based method or a joint method combining model checking and theorem proving.

The model checking method is widely used in small scale examples. Stern and Dill [92] used Murphi [33] to verify the cache coherence protocol of IEEE standard for scalable coherent interface. Clarke et. al [26] used SMV [70] to formally model and verify a cache coherence protocol described in IEEE Futurebus+ standard. Fong Dong et. al [84], [85] proposed to verify cache coherence protocols based on a symbolic state expansion procedure and employed it to verify the coherence in S3.mp (Sun's Scalable Shared memory MultiProcessor). All of the work performs the verification using a fully automated model checking tool and the procedure is very straightforward. However, they all have to deal with the state space explosion problem. There have been various techniques proposed to reduce the number of states, but even with these techniques applied, those automated tools still have a limitation on the number of cores. The inability to scale is the biggest disadvantage of model checking a cache coherence protocol, since for current cache coherence protocols, there is no guarantee that the correctness of a small system implies the correctness of a much bigger system [8]. Even if we can formally verify a cache coherence protocol with 4 cores as sanity check, we cannot trust that the protocol is bug-free if we implement it for a 64-core system.

Theorem proving is more powerful in verifying complex protocols in larger systems due to its scalability. Akhiani et al. [5] used TLA+ language [73], which is based on first-order logic and set theory, to verify the cache coherence protocol for the Alpha

memory model. Moore [76] proved the correctness of a write invalidate cache coherence with the ACL2 theorem proving system [50] with arbitrary number of nodes. Park and Dill [83] employed the general purpose theorem prover PVS [80] to the verify the safety properties of the cache coherence protocol in Stanford FLASH multiprocessor [53]. The problem with theorem proving is a huge amount of time and human efforts are needed to guide the verification. For example, Park and Dill [83] shows that it is a laborious process even to formulate the invariants against which the model would be verified.

We can see that model checking has automaticity, but is limited by the number of nodes in a cache coherence protocol, while theorem proving has capability, but is limited by the amount of human efforts and ingenuity required. As mentioned earlier, there is some work done to combine the capabilities of model checking and theorem proving. The verification of cache coherence protocols can also leverage the combinational method. For different approaches, the ratio of model checking and theorem proving varies, and the roles of model checking and theorem proving also vary. For example, in [67] , theorem proving is used to split the problem to sub problems that can fit into the model checking tools, so the verification results of all sub problems together confirm the cache coherence protocol is correct. While in [25] , since model checking is not able to handle the large number of caches, an abstraction is done to simplify the cache coherence model, which introduce spurious bugs due to over approximation. Then

theorem proving acts as the human reasoning process to remove the spurious bugs found during the model checking process.

Ideally, the combination of model checking and theorem proving can find an optimal point between automaticity and capability. However, it is a big question that how general the method can be. Put another way, what kind of cache coherence protocols and properties can be verified using this method. Most previous research only shows examples of simplified or unrealistic cache coherence protocols and we doubt that modern cache coherence protocols are naturally compatible with this method.

In all, for verifying flat cache coherence protocols, various methods have been proposed, but many of them suffer from the automaticity or capability problem. There do exist some methods that aim to achieve the advantages of both worlds, but no one has studied what kind of cache coherence protocols these methods can apply to.

2.3.2 Verification of Hierarchical Cache Coherence Protocols

As multiple levels of cache come to exist, there is a demand for cache coherence protocols to change from "flat" to "hierarchical" [37], [41], [57], [64]. For each level of cache, there is a coherence protocol independent of protocols for other levels. The most important advantage of hierarchical cache coherence protocols is the scalability and flexibility. In a many-core system with hundreds or thousands of cores, it is not practical to have all cores in the same coherence domain. It can easily cause a bottleneck and

hinder performance. With a hierarchical structure, we can partition the cores to different clusters and probably run workloads on different domains. Moreover, we can use different cache coherence protocols at different levels if they have distinct requirements.

The verification of hierarchical cache coherence protocols has not been as actively studied as that of flat protocols. Compared to flat protocols, hierarchical cache coherence protocols are even more difficult to verify. Hierarchical protocols usually involve more components and more types of state machines, which imply a much larger state space. It is usually impossible to verify a hierarchical cache coherence protocol with only automated tools even for a small scale system.

One of the earliest work that verified hierarchical cache coherence protocol is by McMillan and Schwalbe [71]. They used SMV [70] to verify the protocol of Gigamax multiprocessor, which has two levels of snooping protocol. However, in their verification, many details about the protocol are abstracted away due to the consideration of complexity. The over simplification makes the work itself not that interesting since the correctness of the higher level does not guarantee the correctness of the actual protocol. Some other work in this field usually considers one level of the protocol at a time and abstracted away other levels. However, this kind of separation usually lacks proof and may involve bugs [23] .

One piece of the most important work of verifying a hierarchical cache coherence protocol recently is by Chen et al. [23]. They developed a compositional approach based on assume-guarantee reasoning to decompose hierarchical protocols into a set of abstract protocols. The decomposition greatly reduces the verification complexity. Then each level is verified at one time. The theoretical way of separating the hierarchical cache coherence protocol ensures the correctness of all different levels implies the correctness the overall verification. However, there is no indication that their method can scale to cache coherence protocols with more levels since it still involves too much reasoning.

As seen from above, there is no standard method of verifying a hierarchical cache coherence protocol. This area is far from mature and a lot of issues need to be studied. In Chapter 3, we will show how we can standardize the verification of a hierarchical protocol by designing the protocol in a special way.

2.4 Designing Verifiable Cache Coherence Protocols

As discussed in Section 2.3, currently there is no clear answer what is the most efficient way to verify a flat cache coherence protocol and what kind of protocols can be verified. And the techniques for verifying a hierarchical cache coherence protocol are far from mature. Although experts in verification have struggled to come up with innovative methods to fulfill their responsibility, the current situation is many realistic cache coherence protocols are still impossible to verify. We, as architects, would like to

solve this problem from a different perspective. We follow the "design for verifiability" idea and propose methodologies for designing verifiable cache coherence protocols so that the verification is easier.

2.4.1 Related Work in Design for Verifiability

Milne [74] presents the concept of "Design for verifiability", which proposed to limit the designer's freedom so that the hardware they designed would have fewer states and thus need less verification effort. Milne gave an example of inserting clocked latches between blocks of combinational logic to reduce the number of states. Although very simple, this example provides us with the hint that we may need to sacrifice some area and speed when modifying the design to gain the convenience in formal verification. However, as long as the cost is acceptable, we have a good reason to make such modification.

In Curzon and Leslie's paper [30], they investigate the notion "Design for verifiability" presented by Milne. They tried to find the particular part which had a high impact on verification effort. By changing the implementation of this part, they removed the problem without sacrificing too much performance. Arvind et al. [8] more directly suggested, from hardware point of view, that formal verification should be got into the design flow so that formal methods can be more widely used. Lungu and Sorin [58] explore how the microprocessor should be designed so that the verification is easier. We

follow these concepts and agree that architects should add verifiability as a first class design constraint together with metrics such as performance, power, and reliability.

2.4.2 Verifiability Analysis of Cache Coherence Protocols

In the context of cache coherence protocols, we are unaware of any work that proposes to design a cache coherence protocol with the verification effort as a first class constraint. There exist several pieces of work that explored the verifiability of designed cache coherence protocols. Martin [62] argues that directory protocols are superior to snooping protocols with regard to formal verification effort. His conclusion is based on qualitative analysis considering desirable properties for verification. Marty [65] compared the formal verification efforts of different cache coherence protocol designs and showed their protocol is more amenable to formal verification. However, they primal purpose was for performance instead of verification and it is not clear how general their protocol could be.

Two other papers discuss verification or design complexity after designing cache coherence protocols. HCC [55] is organized hierarchically and they argue that this tree organization facilitates the verification of liveness and consistency. HCC is verified manually, unlike the largely automated verification in our work. Vantrease et al. [95] propose an atomic coherence protocol that avoids races and is thus simpler; we expect it

would be easier to verify than a typical non-atomic protocol, but verification is not discussed in the paper.

A more recent work is by Beu et al. [12], which leverage a coherence design framework called MCP [13] for composing heterogeneous protocols in a hierarchical fashion. Beu et al. show that, if each of the building block protocols is verified correct, then the hierarchical protocol is also correct by inductive reasoning. The main weakness of their paper is the lack of formal proof.

Our work is different from the above research in that we encourage architects to consider verification effort as a first class design constraint and incorporate it in early design stages. Moreover, all of designed cache coherence protocols can be formally verified with mostly automated tools instead of human analysis.

3. Fractal Coherence: Verifiable Hierarchical Cache Coherence

In this chapter, we propose a methodology for designing a hierarchical cache coherence protocols, called Fractal Coherence. Fractal Coherence enables the protocols to be verifiable using existing, automated, easy-to-use formal tools even for large, many-core systems. The idea and design methodology of Fractal Coherence are described from Section 3.1 through Section 3.3. In Section 3.4, a concrete example of a Fractal Coherence protocol, TreeFractal, is illustrated in detail and evaluated with full system simulation to compare with typical snooping and directory protocols. Section 3.5 shows another example, Fractal Directory, which has a larger degree tree as its structure and is more realistic than TreeFractal. Section 3.6 discusses the design space of Fractal Coherence and Section 3.7 is the summary of this chapter.

3.1 Concept of Fractal Coherence

Fractal Coherence protocols originate from fractal theory and leverage the self-similarity characteristic of fractals. Considering the two systems in Figure 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

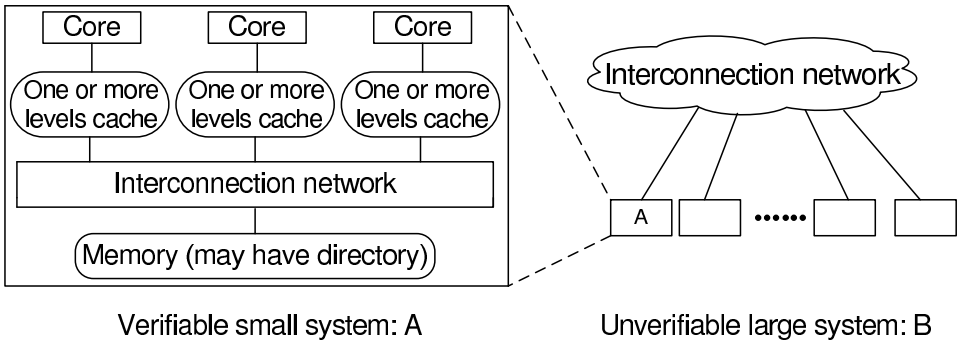

Figure 1: Scalability problem in verification of cache coherence

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

3.2 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 thesis, 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. Figure 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. Interfaces may also associate with caches. 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

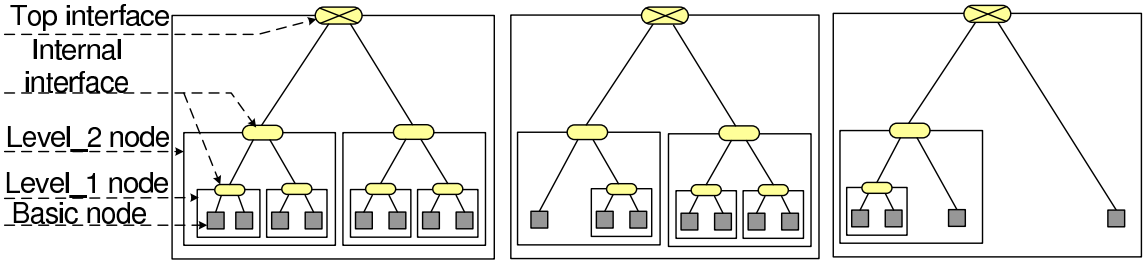

Figure 2: Possible binary tree structures

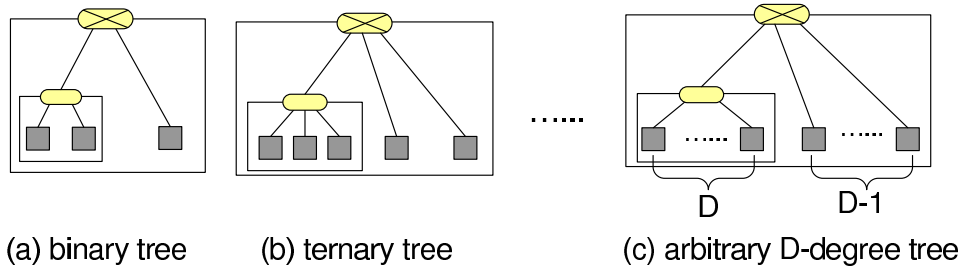

Figure 3: Minimum systems of different degree trees

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. Figure 3 shows the minimum system for a binary tree, a ternary tree, and any D-degree tree.

3.3 Verification Methodology

To formally verify that a fractal system is cache coherent with any arbitrary number of nodes, two verification steps are needed. The first step is to verify that the smallest scale of the system is coherent. The second step is to show that the system is fractal with respect to coherence. We then present an inductive proof that these two verification steps are the only formal verification steps needed. 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.

3.3.1 Verification of Minimum System

Ideally, we could perform the verification of minimum system with an automated model checking tool, as long as the minimum system is small enough. If the state space of the minimum system is beyond the capability of existing tools, we probably need to combine the strength of both model checking and theorem proving. We will show the example that explodes model checking tools later in Section 3.5. In this section, we will only focus on the common case in which a model checking tool is able to verify the minimum system.

As discussed in Section 2.1.3, the model checking process includes modeling, specification, and verification. We illustrate several key points in these processes when

we verify a cache coherence protocol. First, modeling has to accurately capture the behavior of the cache coherence protocol no matter what tool and language it uses. For example, non-atomic protocol transactions should not be assumed atomic. This abstraction may cause misses of important transactions. However, there are still several reduction techniques that can safely apply to the modeling of a cache coherence protocol. 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 [90]. These optimizations can all be employed in modeling the minimum system of Fractal Coherence and they help reduce the state space. Second, specification has to precisely state the properties that the protocol must satisfy. 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) the read always retrieve the value of the latest write, 3) no state machines will ever enter deadlock, and 4) the system is making forward progress at all times (i.e., there is no livelock). Finally, 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.

3.3.2 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 Figure 3(a) as an example and show in Figure 4 the two relationships needed to be verified equivalent. We construct a 4-node binary tree, shown in Figure 4(b), by expanding node A in Figure 4(a) to node A' in Figure 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 Figure 4(c) and A' in Figure 4(d) have the same environment and thus they behave the same. This

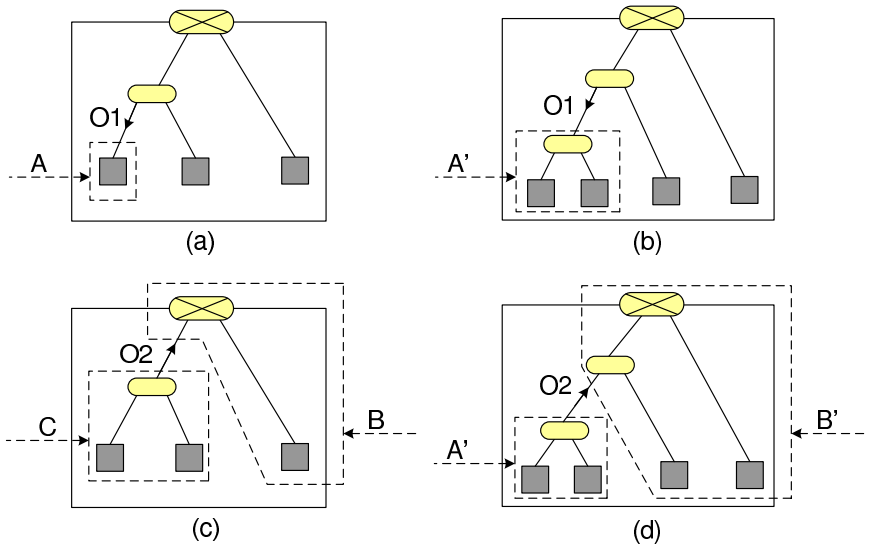

Figure 4: Observational equivalence for maintaining fractal behavior

verification ensures that the two basic nodes in A′ behave the same as they do in a coherent system (Figure 4(c)). The two verifications together ensure that the new system (Figure 4(b)) has the same behavior as the previous one (Figure 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" [75], 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 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 3.4.2.2.

3.3.3 Proof of Cache Coherence for Arbitrary N-node System

We claimed that the formal verification steps described in Section 3.3.1 and Section 3.3.2 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, …, DL}.

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 Figure 5.

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

Lemma 2 (Figure 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 (Figure 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 (Figure 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 (Figure 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.■

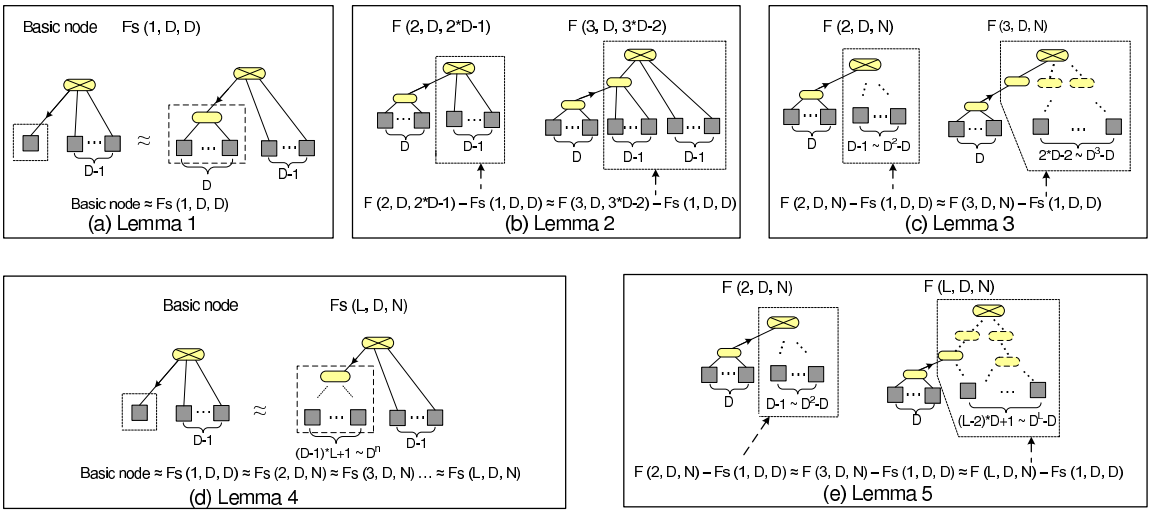

Figure 5: Lemmas for proof of cache coherence in any N-node system

3.4 Case Study: TreeFractal

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.

3.4.1 System Design

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 Figure 2) contains duplicate cache tags for all cache blocks beneath it in the tree. We call these interfaces Tags. We build up the system from a simple two-node system to a scaled system.

3.4.1.1 Two-node System Design

We start our design from a two-node system, illustrated in Figure 6(a). It consists of two basic nodes and a Top Tag. The basic node, shown in Figure 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. If the Top Tag receives a Get-Shared (GetS) 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.

3.4.1.2 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 Figure 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,

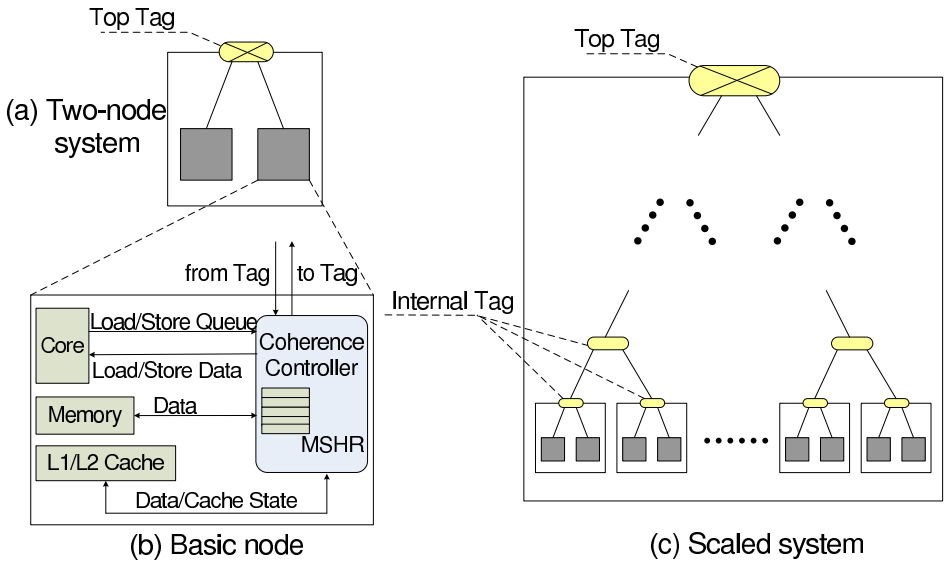

Figure 6: System architecture of TreeFractal

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 Internal Tag must send the request

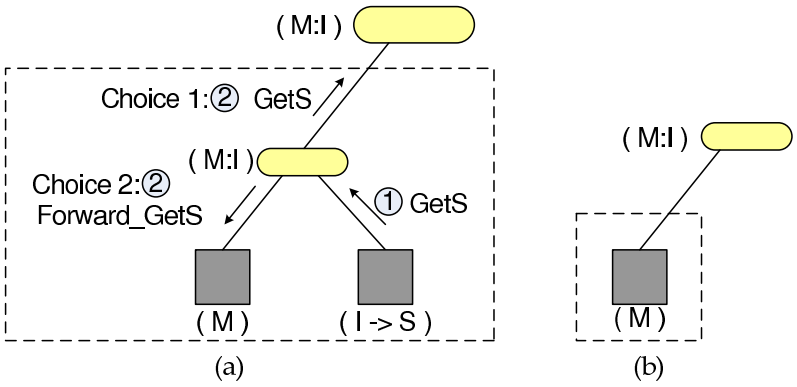

Figure 7: An example of naïve design which violates the fractal behavior

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 Figure 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 Figure 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 3.3.2, 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 Figure 8. In Figure 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:ISD, 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 transfers the data to the right node

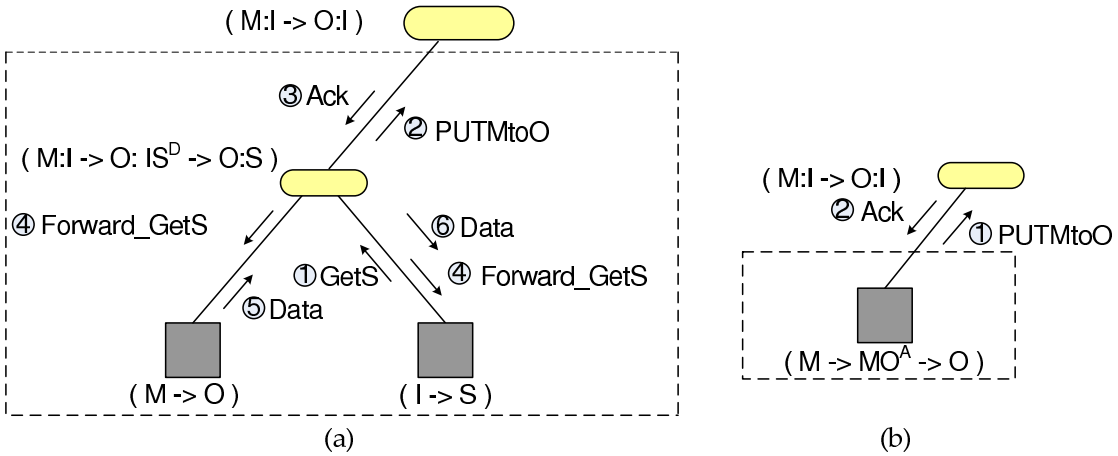

Figure 8: Correct implementation to ensure fractal behavior

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 Figure 8(b). The basic node is allowed to generate a PutMtoO request and change to MOA, 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.

3.4.2 Verification Procedure and Results

In Section 3.3, 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.

3.4.2.1 Verification of Minimum System

We chose the well-known Murphi [33] 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 Figure 3(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.

3.4.2.2 Equivalence Checking of Fractal Behavior

To verify fractal behavior, we employ CADP's [36] equivalence checker, Bisimulator [11]. 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 3.3.2 to ensure the fractal behavior. Specifically, the models in

Figure 4 can be reduced to Figure 9, which reduces the number of components included in the system, while provides the same equivalence guarantee.

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.

3.4.3 Evaluation

For TreeFractal to be viable, its verifiability advantage must not come with significant storage overhead or performance degradation. We quantitatively analyze the storage overhead and experimentally evaluate TreeFractal using full system simulation.

3.4.3.1 Storage Overhead

TreeFractal is a viable option for architects only if its implementation cost is not

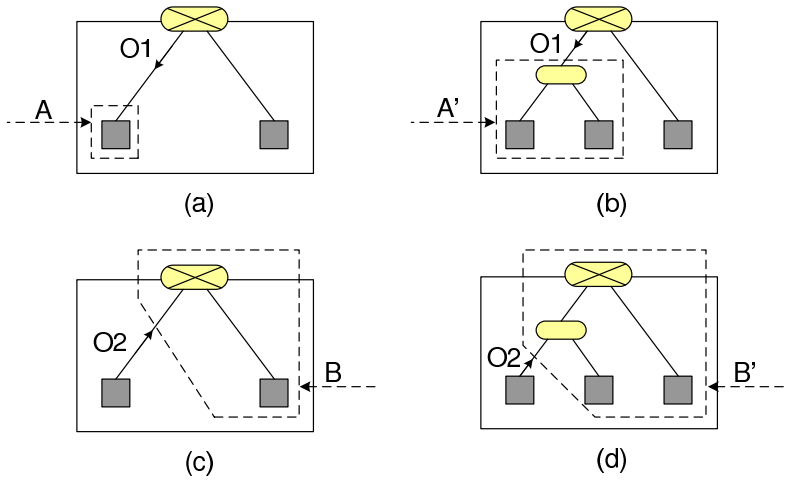

Figure 9: Optimized observational equivalence checking

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.

3.4.3.2 Simulation Methodology

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.

We evaluate TreeFractal using a full-system simulator, Virtutech Simics [59], extended with the Wisconsin GEMS toolset [61]. 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 [54]. 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.

3.4.3.3 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 [6], we ran each benchmark multiple times with small pseudo-random perturbations of the memory latency and averaged the results of all runs. shows the runtime (lower is better) for the three protocols normalized to the runtime of Directory. The error bars represent +/- one standard deviation. From Figure 10, 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

Table 1: System configuration of TreeFractal and baselines

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

as 65.3% (in Apache). This performance improvement is due to two reasons. First, for smaller 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 3.4.1, 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 Figure 11. 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.

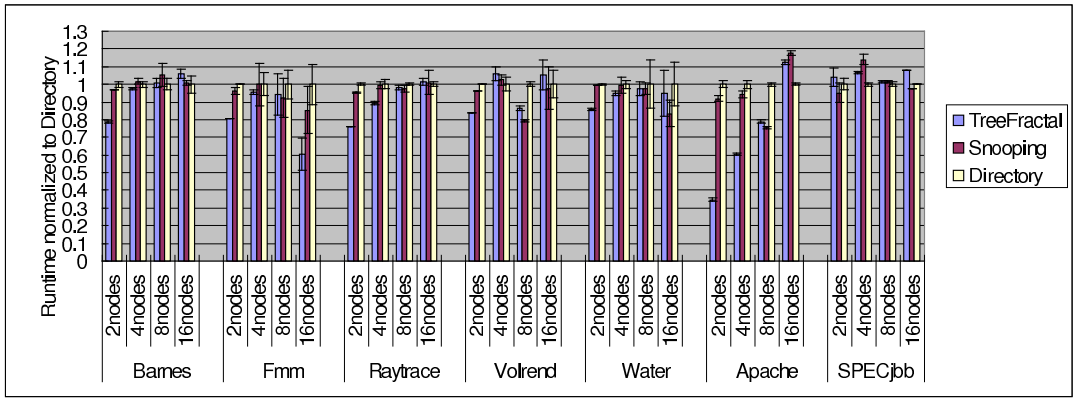

Figure 10: Runtime normalized to Directory

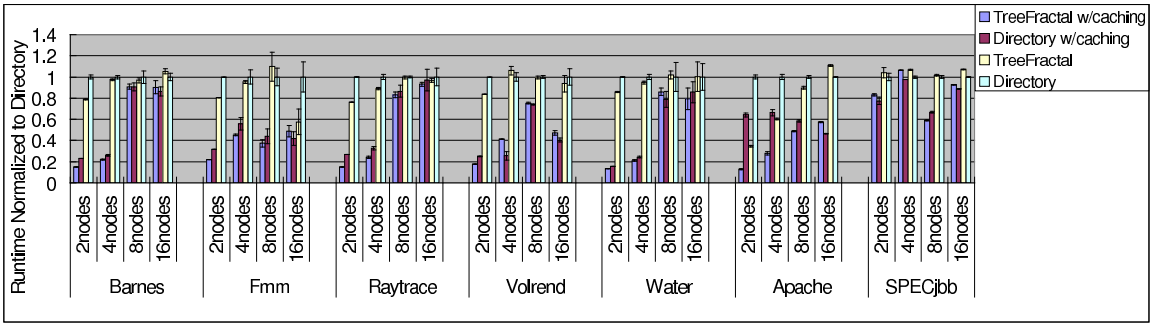

Figure 11: Runtime of on-chip caching protocols normalized to Directory

3.5 Fractal Directory: Extension of TreeFractal

The TreeFractal design, as a preliminary example, can provide some insight into the performance/storage overhead of Fractal Coherence protocols. However, it is still a little far from prevalent on-chip cache coherence protocols. In this section, we propose a more advanced Fractal Coherence design which is realistic enough to be implemented for current memory systems.

3.5.1 Insufficiency of TreeFractal

Although TreeFractal can achieve comparable performance with traditional snooping and directory protocols, there are still several aspects with TreeFractal that makes it not quite suitable for many-core chips. First, TreeFractal requires a binary tree. With such a small degree, the performance/storage overhead may not scale as well as a larger degree tree. Second, TreeFractal has only one level of cache which is private to each core. As discussed in Section 2.2, hierarchical caches are the trend for future many-core chips since it helps improve scalability and flexibility. Third, in TreeFractal, the memory controller is attached with each core. We design such a structure originally with the purpose to ease equivalence checking. However, it is more common for current systems to connect the memory controllers to the last level cache. We propose a more advanced design which still adheres to the design guidelines of Fractal Coherence but avoids the above problems. We call it Fractal Directory since it is more like a traditional directory protocol.

3.5.2 System Design of Fractal Directory

Fractal Directory is modeled loosely after Intel Nehalem protocol, while still maintains the character of fractal behavior as a member of Fractal Coherence protocols family. Therefore, Fractal Directory can be scalably verified and is realistic enough to be implemented for current memory systems.

The main difference from Fractal Directory versus TreeFractal is that in Fractal Directory 1) there are multiple levels of caches composing a hierarchical structure; 2) the internal interface and top interface are co-located with caches; 3) the memory is off-chip and connects to the last level cache via memory controllers. The system structure is shown in Figure 12.

To be more specific, each core has a private, write-back L1 cache. Four cores share a L2 cache that is interleaved to 4 banks based on addresses. The structure inside the box in Figure 12 can be considered as a cluster. Each bank of L2 cache has a coherence controller that handles the coherence transactions. The L2 cache also has a directory which is embedded in the cache and sharing the same tag structure with the cache. The L2 cache maintains inclusion with the L1 cache. The directory in the L2 cache has a bit-vector for each L1 cache beneath it and also has a pointer indicating the owner. All L2 banks connect to a shared L3 cache which is also divided into many banks. The L3 cache also maintains inclusion with all L2 caches. It stores a bit-vector for each cluster (4 L1 caches and 1 L2 cache) and also a pointer for the owner (a cluster).

Because of the inclusion property, the L2 cache and L3 cache may have quite high associativity. For example, if the L1 cache has an associativity of 4, then the L2 cache needs to have an associativity of 16 in order to keep all L1 cache tags in the worst case, and the L3 cache needs to have an associativity of 64 in order to keep all L2 cache

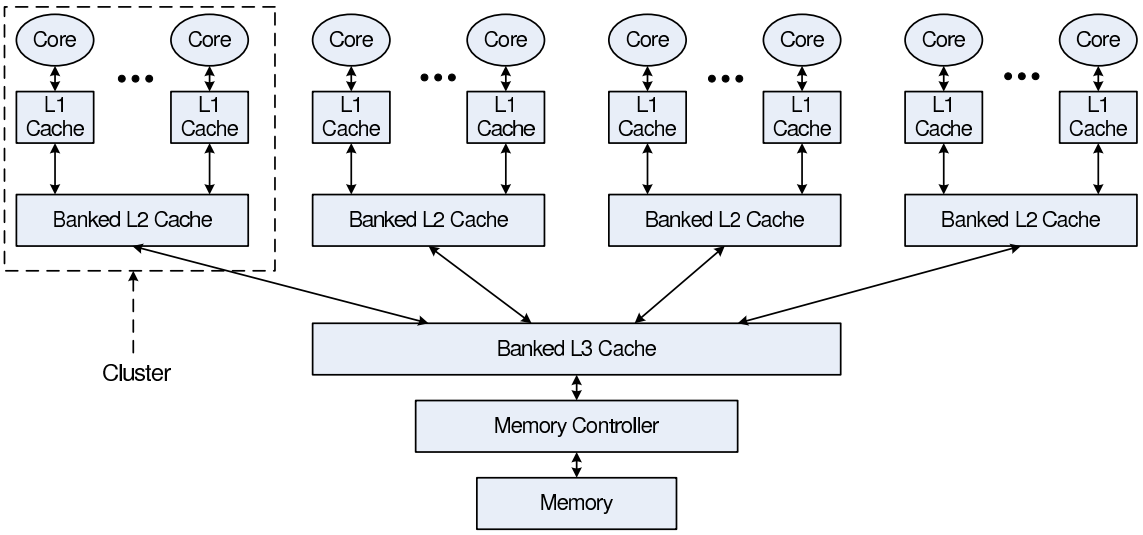

Figure 12: System structure of Fractal Directory

tags in the worst case. Such a high associativity will result in a high latency and huge power consumption when the cache is accessed. We can limit the associativity of the cache by using "Recall" mechanism [28]. "Recall" is the process of the cache sending invalidations to the sharers or sending forwarded request to the owner for retrieving the data it has in order to evict a block. After the "Recall" process finishes, the block being recalled can be evicted and there is room for the new request block which is in a conflict with previous block.

3.5.3 Performance Evaluation

We evaluate Fractal Directory the same way as we did with TreeFractal. Since Fractal Directory is quite similar to a directory protocol, it is reasonable to compare it with a directory protocol that does not have fractal consideration, but has all the other

parameters the same as Fractal Directory. In this way, we can know exactly how the fractal requirement impact performance. The system parameters are shown in Table 2 and the result of runtime normalized to that of the baseline directory is shown in Figure 13. As expected, the performance of Fractal Directory is comparable to the baseline directory protocol since there are only a few coherence transitions that need to be specially designed to maintain fractal behavior.

Besides the comparison between Fractal Directory and a normal directory protocol, we do another evaluation to show the impact of the "Recall" mechanism on the performance. According to [28], when the size of the L2 cache is more than twice the aggregate size of all L1 caches, the "Recall" mechanism is not likely to degrade performance. In our results, surprisingly, "Recall" actually improves performance at an average of 13% across all benchmarks, as shown in Figure 14. After studying the detailed statistic, we find that the ratio of "Recall" is very low, less than 1%, while the cycles saved due to accesses to the lower associativity directory help improve the performance.

3.5.4 Difficulty in Verification Process

We originally thought that the verification process of Fractal Directory would be as straightforward as TreeFractal. However, we were too optimistic about the capability

Table 2: System configuration of Fractal Directory and baseline

Common Parameters for Three Protocols	
Processor parameters	
Number of cores	16
Clock frequency	2 GHz
Memory parameters	
Memory	2 GB, 160 cycle
Network parameters	
Link bandwidth	32 GB/s
Link latency	1 cycle
Topology	2D Torus
Cache parameters for Fractal & Baseline	
Cache line size	64 byte
Split L1 I&D cache	32 KB, 2 way, 2 cycle
L2 cache	1 MB, 4 banks, 16 way, 12 cycle
L3 cache	8 MB, 4 banks, 64 way, 40 cycle
Cache parameters for Recall	
Cache line size	64 byte
Split L1 I&D cache	32 KB, 2 way, 2 cycle
L2 cache	1 MB, 4 banks, 8 way, 7 cycle
L3 cache	8 MB, 4 banks, 16 way, 14 cycle

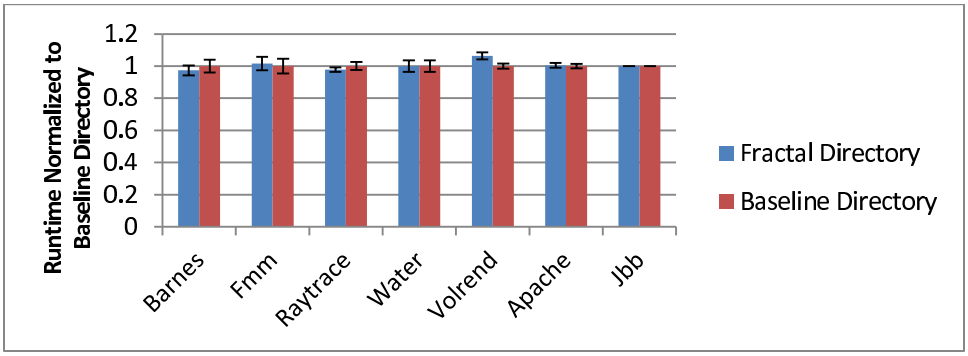

Figure 13: Runtime of Fractal Directory

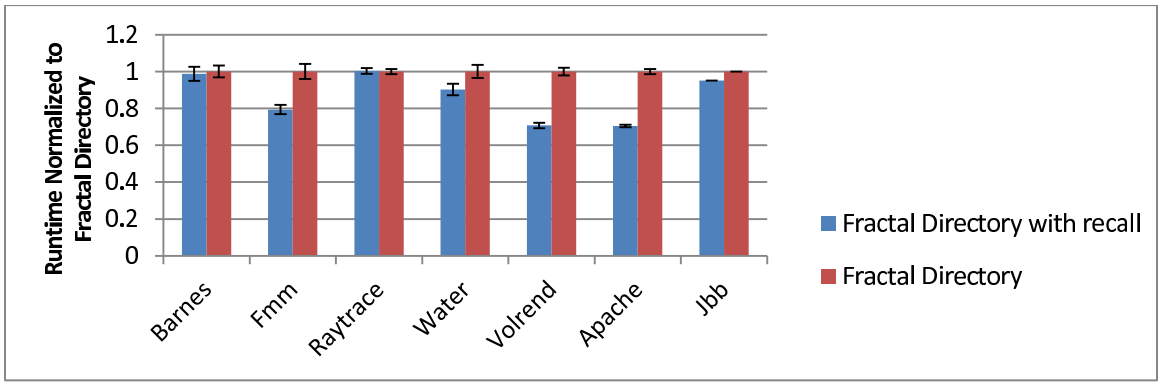

Figure 14: Speedup with "Recall"

of current automated verification tools. We soon encountered the state explosion problem in the verification of the minimum system.

3.5.4.1 State Explosion Problem in Verification

As a tree of degree 4, the minimum system of Fractal Directory has 7 L1 caches, 1 L2 cache, and 1 L3 cache, shown in Figure 15. With so many nodes in a hierarchical system, no current automated tool is able to verify the cache coherence protocol. Moreover, the equivalence checking processes include even more cores than the minimum system does, as shown in Figure 16. The state explosion problem encourages us to further reduce the state space and simplify the verification process so that the model is able to fit in the tool.

3.5.4.2 Reducing State Space by Optimizing Verification Process

We made two optimizations in the verification steps which greatly help reduce the state space, which is shown in Figure 17. We first change the cache coherence

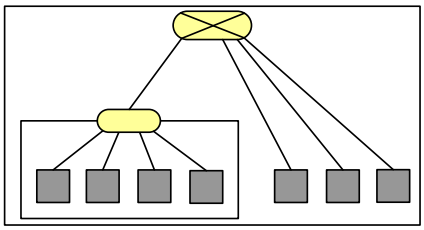

Figure 15: Minimum system of Fractal Directory

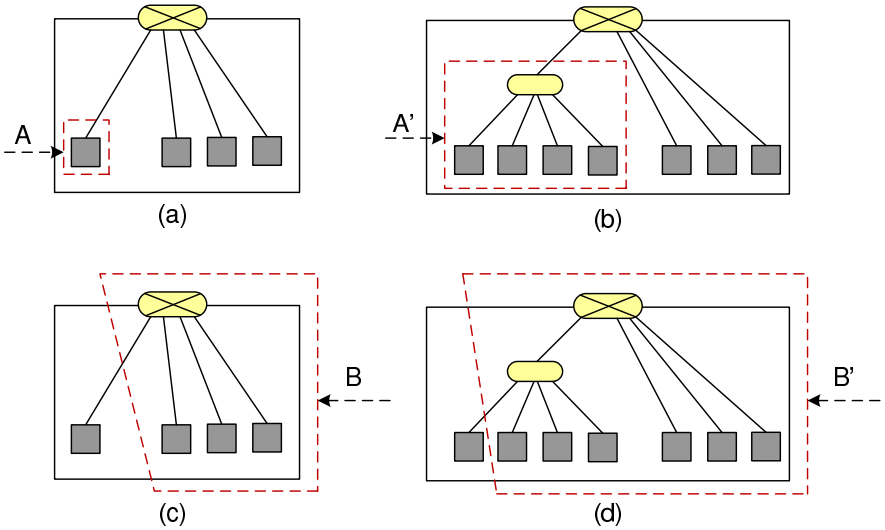

Figure 16: Observtional equivalence checking of Fractal Directory

protocol involved in the minimum system from a hierarchical one to a flat one. A flat protocol is much easier than a hierarchical one regarding verification efforts. Then we reduce the number of nodes involved in the equivalence checking by using an unconstrained environment. We now show why these modifications are correct and how they can be done.

In the previous proof, we need to verify a hierarchical cache coherence protocol in the minimum system because we would like to include all different components of the system, both internal interfaces and top interfaces, in the verification of cache coherence. If we only have a flat protocol in the minimum system, the internal interface will be excluded and we cannot make sure whether it does something bad that ruin the correctness of the system. However, we can incorporate the verification of the internal interface to the equivalence checking. The basic idea is to add state mapping checking to the previous observational equivalence checking. The state mapping checking ensures that the internal interface is actually doing the right thing without including them in the minimum system.

We can leverage Park et al.'s aggregation checking idea [82] to perform the equivalence checking. The idea was originally introduced to check that an implementation of a protocol is consistent with its specification. The implementation is a fine-grained description of the execution, and the specification is an abstraction of the protocol with coarse-grained atomicity. The key point is to use an aggregation function to map an implementation state to a specification state by completing any committed but incomplete transactions. Then an invariant is checked about this mapping to ensure that the two are actually consistent. This aggregation method is ideal for our purpose since it can verify both observational equivalence and state mapping. We can leverage

this method with moderate modifications. First, during the aggregation, we need to use an explicit state mapping between the subsystems we are proving equivalent, which we specify by ourselves beforehand. The state mapping we use is different from Park et al. since they only require an implicit state mapping. Second, the systems verified to be equivalent by Park et al. are closed systems, whereas our subsystems are open ones that interact with the environment. Therefore, we not only need to check the state mapping, but also make sure the messages sent to and received from the environment by the two subsystems being observed are consistent.

We can take a look at how Fractal Directory fits into this aggregation method. The small system can be considered as a specification and the large system can be considered as an implementation. For example, as shown in Figure 4(a) , A is the specification and sub-system A′ is the implementation. A always has atomic transitions, and sub-system A′ has many internal transitions. The commit point in sub-system A′ is when any request or reply message arrives at the internal interface. After a message passes through this interface, the message is in its post-commit stage and needs to be processed until the end. The aggregation function is designed in a way that it drains out all the committed messages in all buffers inside sub-system A′. So by executing all committed but incomplete transactions, it hides the internal transitions and leaves us only the transitions we are interested in, which is exactly what the observational

equivalence requires. And during the aggregation process, explicit state mapping is checked whenever the states change.

Another important advantage of using this aggregation method is that we can perform the equivalence verification using the Murphi model of the larger subsystem. So the verification of the minimum system and the equivalence can be done with the same tool, which removes the effort to translate the model between different tools. To do this, we only need to add an aggregation function and a state mapping function to the Murphi model, and insert assertions in the model as equivalence invariants that check the state mapping and message consistency. Murphi can then perform the equivalence verification automatically. It is worth mentioning that Murphi enables the checking of the equivalence to be performed "on-the-fly," which means the verification does not incur any increase in the state space compared to the state space of the larger subsystem. That is to say, if we can verify the correctness of the larger subsystem without a state explosion problem, it is guaranteed that the equivalence checking will not incur state explosion either.

With Park's aggregation method, we can change the minimum system to a flat protocol. However, the equivalence checking still involves too many nodes that will explode the tool. An effective way is to verify the observational equivalence in an unconstrained environment instead of a specific environment, as shown in Figure 17.

For model checking purposes, it means the environment would non-deterministically choose messages to send into the system. It is crucial to make sure that the protocol would behave properly in a completely unconstrained environment. To do this, we need to define some simple constraints in the protocol, such as to ignore bogus messages. Another benefit of using an unconstrained environment in the verification can ensure that the complex system and the single basic node will behave exactly the same no matter what environment they are put in. In this way, we can remove the second step in the equivalence checking since whether the environment keeps the same is not important for the behavior of the system we are interested in.

We have done an experimental implementation in a scaled-down system to confirm our optimizations in the verification steps, but not yet for the 4 degree Fractal Directory, which will be future work.

3.5.4.3 Demanding for a Verifiable Flat Protocol

After performing the several reduction steps in the previous section, we were able to have a minimum system with 4 L1 caches and 1 L2 cache only. We feed this model into the Murphi model checker. Surprisingly, it had not finished after a day's running on our machine. We stopped the program and have the colleagues from Intel to run the model on their 60 machine clusters. They are able to finish the model in a few hours. However, they suggested that increasing one or two nodes would explode

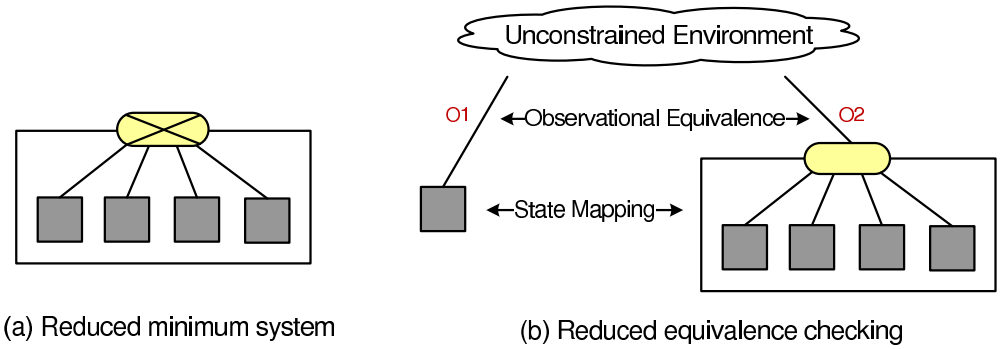

Figure 17: Reduce the state space for verification

their tool, which has an upper limit of 100 billion states in total. This situation makes us to ponder on the question how we can verify a minimum system without such a small limitation on the number of nodes. Ideally, we would like to verify the minimum system with any arbitrary number of nodes. Chapter 4 will discuss the research in this area in more detail.

3.6 Design Space of Fractal Coherence

As mentioned in Section 3.2, 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

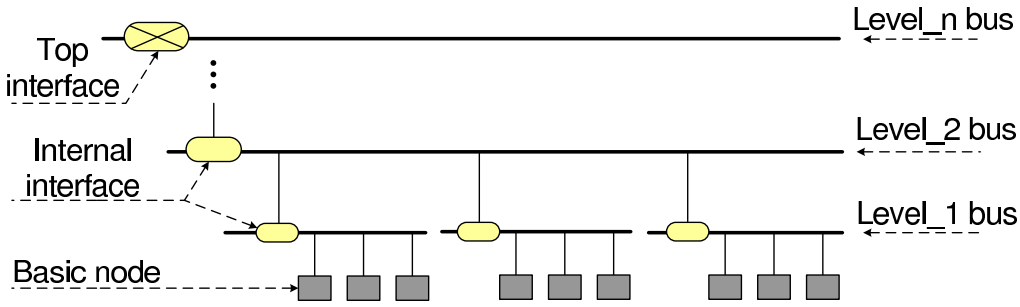

Figure 18: Making a traditional Snooping protocol fractal

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 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 Figure 18, 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.

3.7 Summary

Formal verification of a hierarchical cache coherence protocol is an extremely difficult problem without a standard and complete method to solve. This difficulty is due to the complexity of the hierarchical system itself as well as the inability of current automated tools. This section proposes a design methodology called Fractal Coherence for designing verifiable cache 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. To improve the insufficiency of TreeFractal, we further develop a more realistic directory protocol, called Fractal Directory. Fractal Directory is quite similar to prevalent directory protocols with limited changes to adhere to Fractal Coherence and also performs comparably. However, the verification of Fractal Coherence encountered some difficulty due to state space explosion, which encourages us to further reduce the state space of the verification steps. During the verification process, we find the necessity to have a verifiable flat cache coherence protocol and this is the main topic of Section 4.

4. PVCoherence: Verifiable Flat Cache Coherence

Fractal Coherence proposes to design a hierarchical protocol that can be verified for an arbitrary number of cores. A later piece of work by Beu [12] also proposes a method, called MCP, to design hierarchical cache coherence protocols in a way that enables inductive verification. The key insight in both of these works is that hierarchical designs can be inductively verified for arbitrarily sized systems—but only assuming that the base case or the building blocks can themselves be verified. To satisfy this assumption, TreeFractal is limited to a small degree tree organization that incurs significantly more latency and storage overhead for directory controllers than if one could verify a base case with a higher degree tree. Similarly, MCP assumes that the building blocks it composes together can be verified, which is true only for small building blocks.

As seen from Section 3.5.4.3, an optimized cache coherence protocol with 5 L1 caches and 1 L2 cache may explode the widely-used model checking tool (Murphi). This limitation restricts the design space of hierarchical cache coherence protocols. In this work, our goal is to architect arbitrarily large flat (non-hierarchical) protocols such that they can be verified using a mostly-automated methodology. These flat protocols can be used either on their own or as building blocks in inductively verified hierarchical protocols [12], [96]. To achieve this goal, we use a previously developed technique called

parametric verification. The key idea of parametric verification is to treat the number of nodes—in this work on coherence protocols, a node is a core plus its private cache(s)—as a parameter instead of as a concrete number. Parametric verification can prove that certain properties are true for the system regardless of the value of the parameter.

There have been several proposals for how to perform parametric verification (PV), and in this work we focus on a method that is highly automated. We believe that automation is critical for usability by non-experts. We use a method developed by Chou et al. [25] and McMillan [69] that makes heavy use of automated tools plus a relatively small amount of manual intervention.

4.1 Parametric Verification for Cache Coherence

In this section, we explain parametric verification (PV) at a level that is relevant to architects who would want to design protocols that can be verified with this method instead of delving deeply into the theoretical foundations of PV.

4.1.1 Different Approaches to Parametric Verification

There are a number of methodologies for PV, with greatly varying levels of automation. At one extreme, any modern theorem proving system is capable of doing PV, given enough human insight and effort. At the other extreme, there are fully automated approaches [10], [35]. Unfortunately automation can be costly; such methods typically suffer from extreme limitations on the protocols that can be verified and/or

high computational complexity (i.e., the "automation" is of no practical use). In the middle ground we find approaches [32] that are more automated than pure theorem proving but do require some manual intervention. The method we use for PVCoherence falls into this space and has seen perhaps the most success in academia and industry.

4.1.2 A Mostly Automated Method: Simple-PV

Chou et al. [25] propose a simple method, which we call Simple-PV [1], to parametrically verify cache coherence protocols. This method is based on McMillan's compositional reasoning theory [69]. The main advantage of Simple-PV, compared to other PV methods, is that it leverages automated tools where possible to minimize the required manual effort, and it is of practical use for realistic designs.

Consider a system with an arbitrary number of nodes, N. We illustrate the Simple-PV process for verifying this system's coherence protocol in Figure 19 and now discuss each step. As shown in Figure 19, we start with a non-parametric model, like in a typical non-parametric verification.

Step #1: Automatically Create Parametric Model

The first step in Simple-PV is to create a parametric model from the non-parametric model. Consider the system with N nodes in Figure 20. Starting with two

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

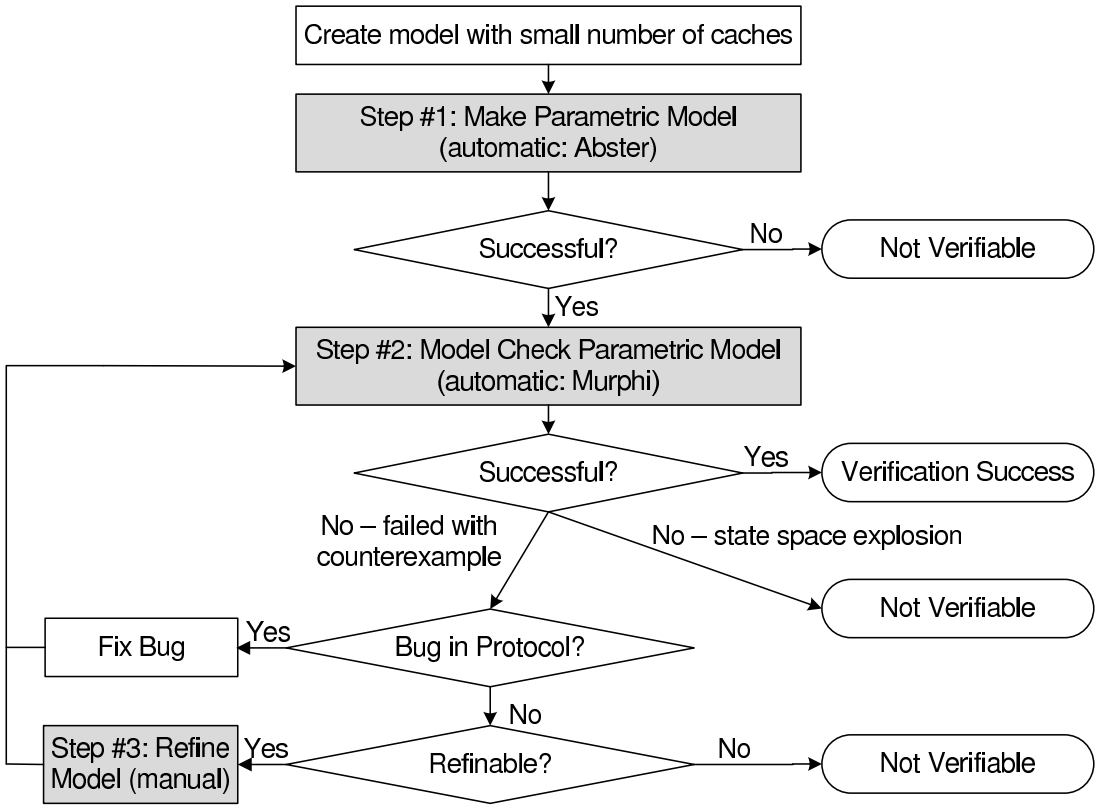

Figure 19: Simple-PV Verification Process

concrete nodes[2] in the non-parametric model, we then abstract the other N-2 nodes into a single "Other" node that we refer to as OtherNode. OtherNode represents the behaviors of all N-2 nodes and we must ensure that the parametric model permits all possible behaviors that the concrete nodes can do as well as the actions those abstracted nodes can do to them.

[2] The number of concrete nodes to instantiate depends on the protocol, for reasons explained at the end of this section, but it tends to be two or three.

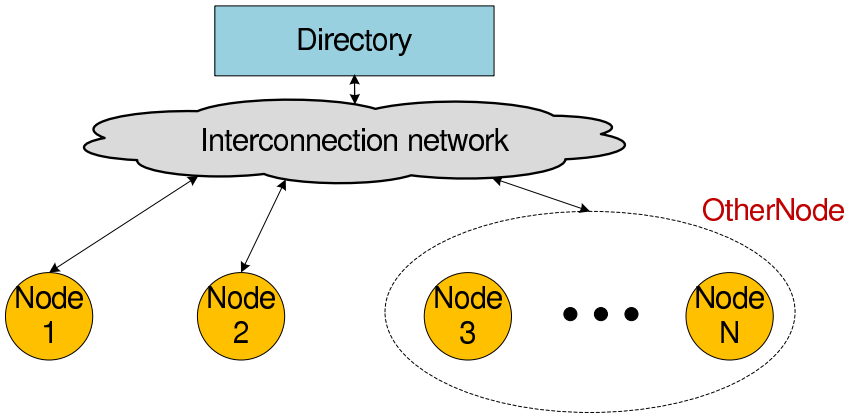

Figure 20: Parametric Model

We perform this process of abstraction with a fully automated tool called Abster [94] that was developed by Intel. Abster automatically generates the behavior for OtherNode, and it helps greatly in avoiding tedious and error-prone manual abstraction. The key point of abstraction is that it must preserve all the behaviors of OtherNode that could occur. Thus, Abster conservatively over-approximates the behavior of the N-2 nodes that it abstracts; that is, the automatically generated OtherNode is likely to exhibit behaviors that would not be possible had we instead instantiated N-2 concrete nodes. This over-approximation leads to a challenge that we address in Step #3.

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

Step #2: Automatically Model Check the Model

If Abster successfully creates a parametric model, we use Murphi to automatically model check this model. If Murphi succeeds, the protocol is coherent and we are done. If Murphi fails, there are four possible scenarios:

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

Step #3: Manually Refine the Model

Because Abster over-approximates when it abstracts the N-2 nodes into OtherNode, it is possible that, in Step #2, Murphi discovers spurious violations of invariants. When this happens, the verifier must manually intervene and refine the

parametric model by modifying OtherNode. Based on the counter-example provided by Murphi, the verifier modifies OtherNode to restrict its behavior.

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

The key to refinement is to both constrain the behavior of OtherNode and also check that these constraints are valid. Thus for each constraint we add to OtherNode's behavior, we add an invariant that Murphi checks, and this invariant is that the constraint is justified (i.e., true for a non-abstracted model). Furthermore, this added invariant is checked on the concrete nodes[3]. In the PV literature, such an invariant is called a lemma, and we adopt this terminology here.

Steps #2 and #3 represent an iterative process of identifying spurious violations in Murphi and refining the model accordingly. The process ends when either (a) Murphi successfully verifies the parametric model, in which case we know the protocol is correct for any arbitrary number of nodes, or (b) the iterative refinement process does not eventually result in a model that Murphi can verify, in which case we consider the

---

[3] It appears that checking the lemma on the concrete nodes when OtherNode has been constrained is circular. However, verification literature has shown that the circularity is broken using an induction over time along with symmetry, and hence the method is sound [25], [69].

protocol to be incompatible with Simple-PV. We discuss why protocols may be incompatible with Simple-PV in Section 2.4.

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

One issue in Simple-PV is choosing the number of concrete nodes in the parametric model. This number is a function of both the protocol and the invariants. A rigorous explanation of the theory behind choosing the number of concrete nodes [25] is beyond the scope of this thesis, but we provide the intuition here. Essentially, we must have enough concrete nodes such that we can describe every possible situation and invariant. For example, if we have only one concrete node, then we cannot describe the invariant that two different (concrete) nodes cannot both be in M(odified) state at the same time. In this situation, at least two concrete nodes are needed. Our protocol in this paper requires two concrete nodes, but some other protocols may require three.

4.2 Limitation of Simple-PV

One limitation of Simple-PV is that, as mentioned previously and illustrated in Figure 19, some possible protocols are incompatible with Simple-PV. This limitation of Simple-PV is understandable, because there is a trade-off between the expressiveness of a logical formalism and the difficulty of its decision problem [43]. To be more specific,

Simple-PV seeks to maximize the usage of automated tools and reduce the human effort. The automated tools ease the verification process, but heavy reliance on automation may somewhat limit the kinds of protocols we can verify.

Krstic [52] provided the most formal treatment and justification of Simple-PV in the literature. His restrictions are more or less in line with those we take in this paper. Krstic did not delve into the ramifications of these theoretical constraints on the design of actual cache coherence protocols.

Another limitation of Simple-PV is that it has not been demonstrated on modern system models. The two example coherence protocols in Chou et al.'s paper [25] are both verifiable, but they are from a long time ago and not suitable for today's multicore processors. (For example, these protocols assume that each core is its own chip with its own dedicated link to its own portion of the distributed memory.) It is unclear whether we can design modern multicore protocols that are compatible with Simple-PV. And, if we can design protocols that are compatible with Simple-PV, it is unclear whether the constraints imposed to ensure compatibility are too costly in terms of performance or storage.

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

4.3 System Overview of PVCoherence

In this section, we present the system architecture based on which we design cache coherence protocols. Currently multicore processors usually employ a multi-level cache hierarchy, in which each core has one or more private caches and all cores share a last level cache. This system model includes Intel's Nehalem [88] and AMD's Barcelona [28]. We illustrate this system model in Figure 21.

Coherence Protocol: We assume a directory-like coherence protocol, because snooping protocols do not scale to many-core systems. We describe the operation of the cache coherence protocols in this paper using terminology common to both Sorin et al. [89] and the protocols that are distributed with the gem5 simulator [15]. The coherence requests are: "GetS" to obtain Shared (read-only) access, "GetM" to obtain Modified (read-write) access, "PutM" to writeback a Modified block, "PutO" to writeback an Owned block, and "PutE" to writeback an Exclusive block.

Cache Hierarchy: The last-level L2 cache is inclusive with respect to the L1 caches. To maintain inclusion, evicting a block from an L2 requires invalidating that block from any L1 caches that hold it. With an inclusive L2, the L2 tags can be to create a co-located directory cache. That is, each L2 block holds the directory state for that block. Because the L2 is inclusive, the directory cache has the state of all blocks present

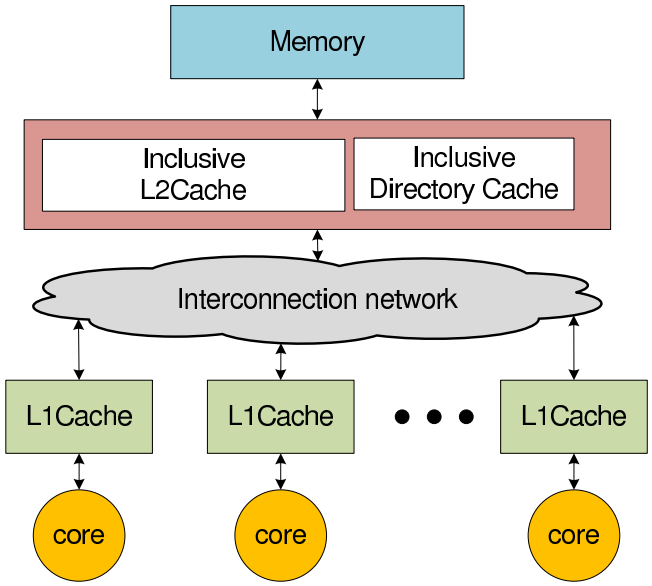

Figure 21: System Architecture of PVcoherence

in one or more L1 caches, and a miss in the L2 implies that the block's state is I(nvalid) and leads to a memory access.

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

4.4 Design Guidelines for PVCoherence Protocols

We seek to design cache coherence protocols that can be parametrically verified with Simple-PV. However, not all features of a protocol are compatible with the use of Simple-PV. Our goal is to explore the design space of cache coherence protocols and discover which features make Simple-PV impossible. We do not claim that the list of design decisions we study is complete—because there are so many possible ways to design a protocol—but we believe we have included many of the most important design decisions. We strive to consider design decisions and features that are common to current protocols.

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

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

Guideline #1: All nodes must be identical.

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

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

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

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

This guideline most directly impacts coherence protocol design by prohibiting the use of counters (that count the number of nodes). Typical directory protocols often use counters to aid in collecting acknowledgments, such that a core waits to write to a block until it has received acknowledgments from some number of other cores that had been sharing the block.

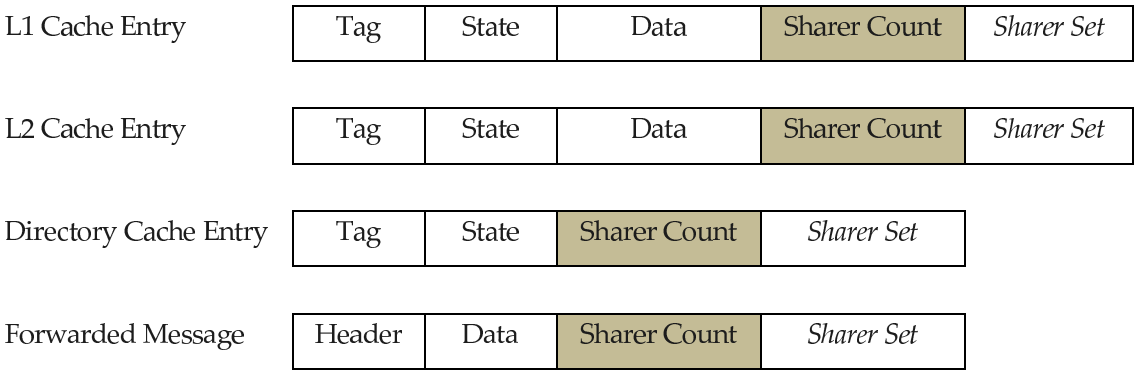

Figure 22: Components impacted by Guildeline #2

Figure 22 shows the protocol components that may be impacted by Guideline #2. The gray entries indicate storage and message fields that we need to avoid: sharer counters. Instead of a sharer counter, we need to use a bit vector to denote the sharer set. This constraint leads to potential overhead in two ways. One overhead is storage, because a bit vector consumes more storage than a counter. The other overhead is network traffic, because a message containing a sharer set is larger than an equivalent message containing a sharer counter.

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

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

This guideline has a significant impact on coherence protocol design. Adhering to this guideline prohibits us from designing protocols in which we enforce point-to-

point ordering for a virtual channel that has a queue depth that depends on the number of nodes. Some queues have a depth that does not depend on the number of nodes, such as a queue of requests from one L1 cache to the L2; the depth of this queue depends on the number of outstanding requests the L1 can have, which is both small and not parameterized.

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

1. Consider a system in which all of the L1 caches share a queue of requests to the L2. This queue has a depth that is proportional to the number of nodes. Such a queue is compatible with Simple-PV only if it is unordered.
2. Consider a protocol that relies upon point-to-point ordering of forwarded coherence requests from the directory to each L1. Many protocols rely on this ordering to avoid races that would otherwise complicate the protocol and require additional messages to acknowledge message reception. However, ordering of this queue is not compatible with Simple-PV if the number of forwarded messages that can be in this queue is a function of the number of nodes. Unfortunately, in many protocols, this situation is possible. For example, if Core C1 has requested Modified permissions for

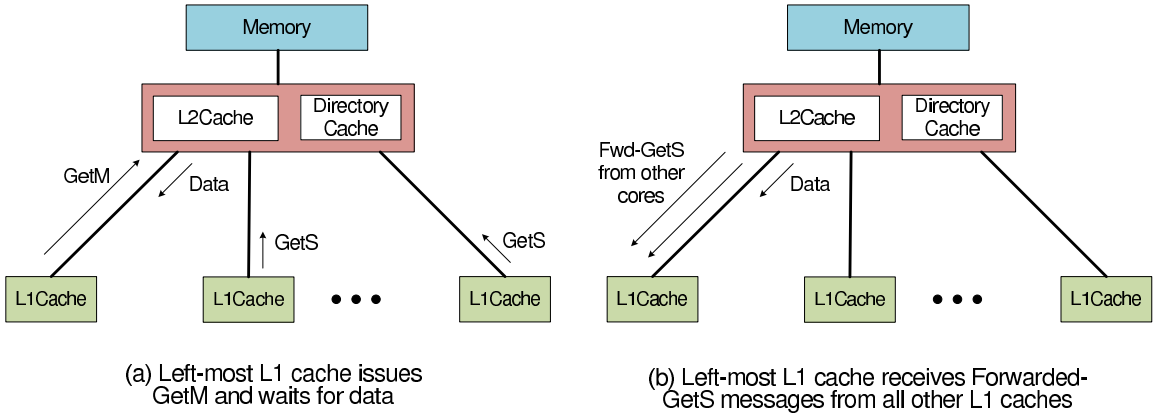

Figure 23: Scenario #2 forbidden by Guideline #3

block B, the directory could forward subsequent GetS requests for B to C1 from every other core before C1 receives the data for B and can start responding to the GetS requests that have filled its queue. We illustrate this scenario in Figure 23. Most protocols do not rely on ordering of the forwarded GetS requests in this example[4]; nevertheless, the possibility of having a number of messages in the queue that depends on the number of nodes precludes ordering any messages in this queue.

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

In our protocol models (and in all model checking work we have seen), arrays are used to represent channels and messages are entries in these arrays. For example,

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

we can specify the buffer of requests from Core C1's L1 cache to the L2 cache as buffer_L1_C1_to_L2[SIZE], where SIZE is the number of entries in the buffer. It is common to designate SIZE as a concrete value (e.g., 4) or as a variable that is equal to the number of cores. The latter situation can occur, for example, in a queue of forwarded requests from the directory to a given L1 cache; as explained in the discussion of Guideline #3, such a queue could hold forwarded GetS requests from all other cores. Although the buffer depth in this example is a function of the parameterized number of nodes, the protocol is still compatible with Simple-PV, because the array is parameterized in only one dimension.

The problem for Simple-PV appears only when we want to specify an array that is parameterized in more than one dimension. The consequence of this constraint is that it affects a common protocol design option. Namely, it precludes us from letting a core that issues a GetM collect all of the acknowledgment messages from cores that were invalidated by the GetM. In this scenario, Core C1 issues a GetM to the L2 and the L2 sends an invalidation to all cores with Shared copies of the block. In most protocols, the invalidated cores send acknowledgments to C1. However, that implies that we have buffers from each core to each other core. Because the number of nodes is parameterized, we thus have a two-dimensional parameterization with a structure like AcknowledgmentBuffers[N][N].

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

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

Observations about Specific Optimizations

The above four guidelines are basic rules for protocol design. However, even if a protocol follows all of these guidelines, the subtle details of the protocol can affect the

protocol's compatibility with Simple-PV. We explore the design space to determine which design choices are compatible with Simple-PV and which are not.

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

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

1. We add the stable state E(xclusive).
2. We add the stable state O(wned).

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

The impact of these optimizations on Simple-PV varies. Adding the "E" state (Optimization 1) has zero impact. Because the protocol is still conservative in that it allows only one transaction at a time, we find we can still verify it without adding more

lemmas. Adding the "O" state (Optimization 2) requires two lemmas; for example, one lemma constrains OtherNode's behavior based on whether the L2's coherence state indicates that a concrete L1 is in state S or not.

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

Optimization 6 is not compatible with Simple-PV. Assume that one of the N-2 abstracted L1 caches is the Modified owner of a block. L1 cache C0 has the block in state I and issues a GetM to the L2 and transitions to transient state IM (in I, waiting to go to M). The L2 forwards C0's GetM to OtherNode and immediately changes the directory state to indicate that C0 is the owner[5]. Before C0 receives data from the owner (in OtherNode), another abstracted L1 cache issues a GetM to the L2. The L2 forwards this request to C0 and changes the directory state to indicate that the owner is OtherNode. C0 still does not have data, and it changes its block state to the transient state IMI (in I, waiting to go to M, will do one store when it gets the data, and then will go back to I). At

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

this point in time, cache C1 goes through the same process that C0 has just gone through and also ends up in state IMI.

Now the problem for Simple-PV emerges. C0 and C1 are in the same state and both are eligible to receive data. In real programs, it does not matter whether C0 or C1 receives data first as long as all nodes view the same ordering. However, C0 and C1 are not allowed to maintain the writable copy of the data at the same time because it violates the SMWR invariant. In the unabstracted protocol, this scenario cannot happen, but it may happen in the abstract one, since the OtherNode can cause impossible behaviors. Therefore, we must have a constraint to prevent this behavior. Unfortunately, this constraint requires adding another concrete node in the abstracted model, which results in more concurrency and bigger state space. Our extensive experiment shows that the refinement process never converges after adding the third node, meaning that the abstracted protocol cannot be verified with Murphi. Thus, removing completion messages make the protocol with this optimization incompatible with Simple-PV.

Conclusions: Designing a coherence protocol that can be verified with Simple-PV requires adhering to several guidelines. The list in this section is not exhaustive but it illustrates many of the issues that arise in the design of typical coherence protocols.

4.5 Case Study: PV-MOESI

Following the above guidelines, we can design PVCoherence protocols. The common feature of all PVCoherence protocols is that they can be formally verified using Simple-PV. Although all PVCoherence protocols obey the design guidelines presented in Section 4, there can still be considerable variation between different PVCoherence protocols. In this section, we show the design process of one PVCoherence protocol, called PV-MOESI. The protocol is based on the system architecture in Figure 21. To highlight the ramifications of designing a protocol to be compatible with Simple-PV, we compare and contrast the design of PV-MOESI with a protocol we call OP-MOESI. OP-MOESI is similar to typical multicore protocols, and it provides high performance but it cannot be verified using Simple-PV. In the design of PV-MOESI, we try to keep it as similar to OP-MOESI as possible, only modifying it when necessary to satisfy the constraints of Simple-PV.

4.5.1 An Optimized Protocol: OP-MOESI

The OP-MOESI coherence protocol is a fairly standard directory protocol that is optimized for performance. OP-MOESI is similar to other prevalent protocols [28], [88]. OP-MOESI has five stable L1 cache coherence states (MOESI) and more than 30 transient states to improve performance. An L2 block can be in a similar set of states, except that an L2 block cannot be in E (there is no use for it) and it can be in one of two "stale"

states: M(s) and O(s). These stale states denote when there is an L1 that has the block in M or O, respectively, and that L1 potentially has a more up-to-date value of the data than the L2.

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

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

We specify OP-MOESI at a high level in Table 3. This specification omits all of the complexity of transient states, but it provides the big picture of how the protocol works.

4.5.2 Converting OP-MOESI to PV-MOESI

Although highly optimized, OP-MOESI cannot be verified with Simple-PV. Abster fails to generate an abstracted model for OP-MOESI and thus we cannot run it through Murphi. In this section, we create PV-MOESI by modifying OP-MOESI to

satisfy the guidelines in Section 4. The specification of PV-MOESI is alongside the specification of OP-MOESI in Table 3, with PV-MOESI's differences highlighted in bold.

1. For GetM transactions, we remove the counter in the response message from the L2 to the requesting L1. We replace it with a sharer set that is, unfortunately, larger than the counter (C bits compared to log2C bits).
2. For GetM transactions, we have the L2, instead of the L1 requestor, collect the acknowledgments from L1 caches that are invalidated. After collecting all acknowledgements, the L2 sends a single acknowledgement to the L1 requestor. This modification adds one more network hop for the transaction.
3. We remove the point-to-point ordering in all virtual channels. This is the most significant change in the protocol because it leads to more races. The races happen when an L1 receives a forwarded request or an invalidation while in a transient state. PV-MOESI handles these races in the usual fashion (with extra messages and extra transient states) but without ever blocking. These races are not unique to PV-MOESI but rather a well-known issue for protocols that cannot rely on point-to-point ordering. Handling the races introduces some complexity but is manageable.

Table 3: High-level specifications of OP-MOESI and PV-MOESI. Ignores transient states. Differences between OP-MOESI and PV-MOESI are in bold font in PV-MOESI specification.

	OP-MOESI	PV-MOESI
Structure		
L1 cache entry	64B data, tag, state={M,O,E,S,I}	
L2 cache entry	64B data, tag/bit vector to track L1 caches, directory state={I:I, S:S, O:S, M:I, O(s):O, M(s):M}	
Core C1 has load miss on block B in its L1, sends GetS to L2		
L2 = I:I	L2 gets block from memory and sends it to C1; L2 adds C1 to bit vector; L2 →M(s):M; C1's L1 → E	*same as OP-MOESI*
L2 = S:S or O:S	L2 sends data to C1; L2 adds C1 to bit vector; C1's L1→S	*same as OP-MOESI*
L2 = M:I	L2 sends data to C1; L2→M(s):M; C1's L1→E	L2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→E
L2 = O(s):O C2 is the owner	L2 forwards GetS to C2; L2 adds C1 to bit vector; C2 sends data to C1; C1's L1→S	L2 forwards GetS to C2; L2 adds C1 to bit vector; C2 sends data to C1; **C1 sends Completion to L2;** C1's L1→S
L2 = M(s):M C2 is the owner	L2 forwards GetS to C2; L2 adds C1 to bit vector; L2→O (s):O; C1's L1→S	L2 forwards GetS to C2; L2 adds C1 to bit vector; **C1 sends Completion to L2;** L2→O (s):O; C1's L1→S
Core C1 has store miss on block B in its L1, sends GetM to L2		
L2 = I:I	L2 gets block from memory and sends it to C1; L2 →M(s):M; C1's L1→M	L2 gets block from memory and sends it to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→M
L2 = S:S or O:S	L2 sends data to C1 with number of sharers and sends invalidations to sharers; sharers send acks to C1; L2 →M(s):M; C1's L1→M	**L2 sends invalidations to sharers; sharers send acks to L2; L2 sends data to C1 (without number of sharers);** **C1 sends Completion to L2;** L2

		→M(s):M; C1's L1 → M
L2 = M:I	L2 sends data to C1; L2→M(s):M; C1's L1→M	L2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1→M
L2 = O(s):O C2 is the owner	L2 forwards GetM to C2 with number of sharers and sends invalidation to sharers; C2 sends data to C1; sharers send acks to C1; L2 →M(s):M; C1's L1→M; C2's L1→I	L2 forwards GetM to C2 **(without number of sharers)** and sends invalidations to sharers; C2 sends data to C1; **sharers send acks to L2; L2 sends ack to C1**; **C1 sends Completion to L2;** L2 →M(s):M; C1's L1→ M; C2's L1→I
L2 = M(s):M C2 is the owner	L2 forwards GetM to C2; C2 sends data to C1; L2 →M(s):M; C1's L1 → M; C2's L1→I	L2 forwards GetM to C2; C2 sends data to C1; **C1 sends Completion to L2;** L2→M(s):M; C1's L1 → M; C2's L1→I
Core C1 has store miss on block B in its L1, but it has the data, sends Upgrade to L2		
L2 = S:S or O:S or O(s):O	L2 sends invalidations to sharers except C1; L2 sends ack to C1 with number of sharers; sharers send acks to C1; L2 →M(s):M; C1's L1 → M	L2 sends invalidations to sharers except C1; **sharers send acks to L2; L2 sends ack to C1; C1 sends Completion to L2;** L2 →M(s):M; C1's L1 → M
Core C1 wants to evict block B from its L1		
C1's L1=S	C1 immediately evicts block; C1's L1àI	
C1's L1=E	C1 sends PutE to L2 without data, waits for ack	
C1's L1=O or M	C1 sends PutO or PutM with data to L2, waits for ack	
L2 wants to evict block B		
L2 = I:I	L2 immediately evicts block	
L2 = S:S or O:S	L2 sends invalidations to L1 sharers, waits for acks, then evicts	
L2 = M:I	L2 writes data back to memory, waits for ack from memory, then evicts	
L2 = O(s):O	L2 sends GetM to L1 owner, sends invalidations to L1 sharers, waits for data and acks, then evicts	
L2 = M(s):M	L2 sends GetM to L1 owner, waits for data, then evicts	

After the above modifications, we find that the model can be abstracted by Abster. However, the abstracted model still cannot be verified by Murphi, regardless of how we try to refine it. This problem—which arises due to multiple in-flight GetM requests—was discussed at the end of Section 4, and we handle it by modifying how the protocol handles GetM requests. When the L2 receives a GetM it forwards the GetM and/or invalidations (as in OP-MOESI) but then blocks subsequent requests until it receives a Completion message from the L1 that requested the GetM. The L1 sends the Completion once it has received data and/or the acknowledgment from the L2. This protocol modification has some potential impact on performance due to blocking at the L2.

4.5.3 Verification of PV-MOESI

The verification of PV-MOESI follows the steps discussed in Section 4.2. We first use Abster to abstract the PV-MOESI protocol, and then verify the abstracted model with Murphi. As expected, the abstracted model failed in Mruphi due to the over-approximation in the abstraction step. To remove the spurious counterexamples, we need to manually add totally 7 lemmas during the refinement process. We will not list all lemmas here, but give an example to show how a counterexample is generated and how it guides us to generate the corresponding lemma.

Assume there is a transaction called T1 in the cache coherence protocol which handles the forwarded GetM request while the cache state is M. The designed action is the owner sends data to the requestor and changes its state to I. This works well with the model that contains only two concrete nodes. Then Abster does abstraction for the model. A similar transaction T1′ that describes the same scenario is generated for the OtherNode. Remember that the state information must be abstracted away in the OtherNode, meaning the constraint that the cache state is M is not existing any more. The loose of the constraint is to make sure that the abstracted model is conservative enough to permit all possible behaviors. However, it is usually too conservative so that unexpected actions may also happen. In this example, when running Murphi to model check the abstracted model, we encounter a counterexample due to the action that the OtherNode sends the data when it is not supposed to. The actual state of the OtherNode is "O" instead of "M" and it should perform another transaction T2′ instead of T1′. T2′ sends the requestor data as well as the number of acknowledges the requestor needs to collect. However, as the state has already been abstracted away, T1′ may take place when T2′ should happen. So the OtherNode sends the data to the requestor without number of acks and the requestor just happily changes to M. Since there are might be other sharers in the system, the SWMR invariant is violated.

Therefore, we need to add a constraint to T1′ to restrict its behavior. The constraint strengthens the precondition of the transaction by implicitly indicating the actual state of the OtherNode. It says that only when the directory in the L2 cache denotes the OtherNode is the owner, T1′ can take place. In this way, T1′ will not take place when the OtherNode is not in "M" state. After we add the constraint in the transaction, we also need to add it as a lemma to justify the constraint is indeed correct. This lemma appears to be circular reasoning, but is actually not, as discussed in Section 4.2.

4.5.4 Evaluation

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

4.5.4.1 Methodology and System Configuration

We evaluate OP-MOESI and PV-MOESI using the gem5 full-system simulator [15]. For both protocols, we keep the common architectural parameters the same: processor configuration, L1/L2 cache size, memory size, link latency, link bandwidth, etc. We calculated the access latency of storage structures using Cacti [54]. The system parameters are shown in Table 4.

Table 4: Simulation Configurations of PV-MOESI and OP-MOESI

Processor Core Parameters	
Cores	32 in-order x86 cores
Clock frequency	2 GHz
Cache and Memory Parameters	
Cache line size	64 bytes
Split L1 I&D caches	32 KB, 2-way, 2 cycle hit
L2 cache	inclusive with respect to L1s; 8MB split into 16 banks – each bank 512 KB, 8-way, 12-cycle hit
Memory	2GB, 160-cycle hit
Interconnection Network Parameters	
Topology	2D mesh
Link bandwidth	32 GB/s
Link latency	1 cycle

For benchmarks, we use the PARSEC benchmark suite [14], except for two benchmarks, streamcluster and fluidanimate, that are not compatible with gem5. We run each experiment multiple times to accommodate the natural variability in simulation runtimes [6]; error bars in graphs indicate plus/minus one standard deviation from the mean.

4.5.4.2 Performance Results

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

requiring the L2 to collect invalidation acknowledgments. The question is whether, in practice, these potential performance degradations occur.

In Figure 24, we plot the runtimes for both OP-MOESI and PV-MOESI, normalized to the runtime of OP-MOESI, for 32-core systems. While there are some differences in the runtimes, they are "within the noise." On some benchmarks, PV-MOESI even has a marginally shorter runtime than OP-MOESI, but these differences are also within the noise and are not meaningful speedups.

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

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

Network Overhead: PV-MOESI uses more interconnection network bandwidth than OP-MOESI. This extra bandwidth is mainly due to the extra messages caused by Completions. Intuitively, this bandwidth overhead should be small, but we

experimentally evaluated it to confirm this expectation. In Figure 25, we plot the total traffic consumed by PV-MOESI, normalized to the traffic consumed by OP-MOESI. For most benchmarks, the overhead is less than 5%, but it is as high as13.8% for canneal. Nevertheless, even for canneal, this extra network traffic has negligible impact on performance.

4.5.4.3 Scalability Analysis

Because our goal is to create protocols that are verifiable even as they scale to larger numbers of cores, we are interested in knowing PV-MOESI's performance scalability. We focus on one representative benchmark, blackscholes, and we show how its performance scales from 4-32 cores. In Figure 26, we compare the runtimes for OP-MOESI and PV-MOESI, normalized to OP-MOESI's 4-core runtime, as a function of the number of cores. We observe that PV-MOESI tracks OP-MOESI's performance for all core counts and is thus just as scalable—both up and down—as OP-MOESI. We also note that speedups are less than linear with core count, which is a function of the benchmark more than that of the protocol.

4.5.4.4 Storage Overhead

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

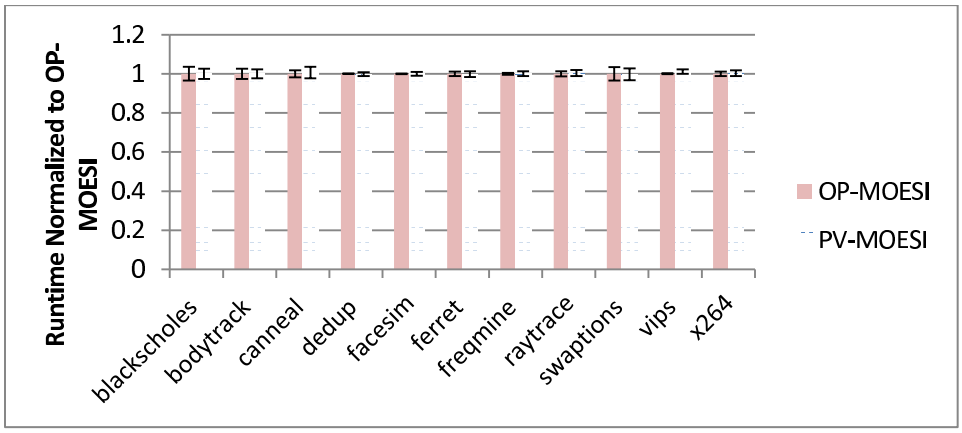

Figure 24: Runtime comparison: OP-MOESI vs PV-MOESI

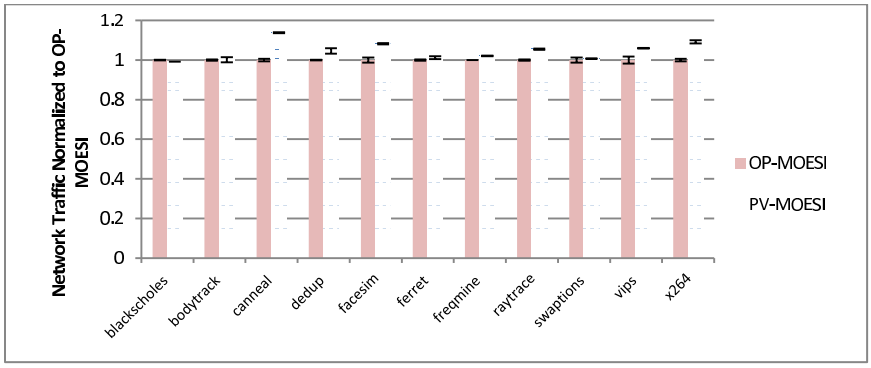

Figure 25: Network traffic overhead of PV-MOESI

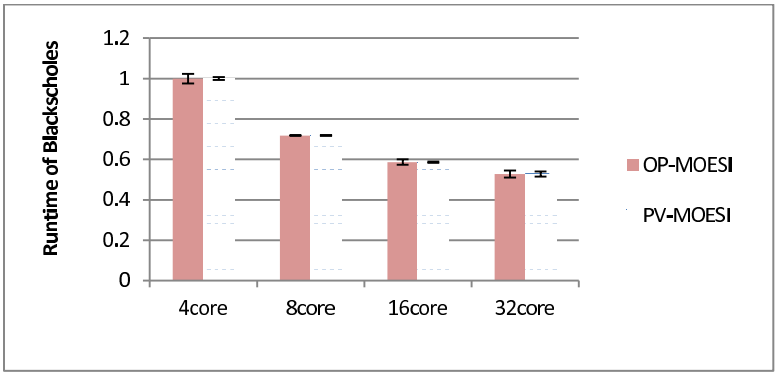

Figure 26: Performance Scalability

In the L2 cache, PV-MOESI requires a sharer set in the directory to record all L1 sharers. This is also true for OP-MOESI. Therefore, PV-MOESI adds no storage overhead compared to a protocol with a full-map directory. Those optimization techniques for reducing the storage cost of the directory, such as coarse directory, limited pointer directory [4], etc., can also be employed in PV-MOESI as long as they do not involve sharer counters.

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

4.6 Combining Fractal Coherence and PVCoherence

Fractal Coherence enables the scalable verification of a hierarchical protocol, while PVCoherence enables the scalable verification of a flat protocol. We now have solutions for both vertical and horizontal dimensions. A natural thought would be to combine the two methods and have a more general solution for a larger domain of systems. In this section, we discuss the possibility and difficulty in combining the two

methods. The actual implementation is not within the scope of this thesis, but an interesting piece of future work.

We use the system structure shown in Figure 17. Remember that we barely verified the cache coherence of the minimum system which has 4 L1 caches and 1 L2 cache with the directory, and we failed when we would like to increase by one more node. If we had designed the minimum system according to the guidelines of PVCoherence, it will be scalably verifiable with Simple-PV no matter how many L1 caches are included. The verification of the minimum system has become straightforward, but it is not clear how the equivalence checking can be done. Since the system shown in Figure 17(b) will also explode Murphi, we cannot directly apply Murphi and verify the observational equivalence and state mapping between the big system and the small system. A possible way might be to employ parametric verification also in the equivalence checking. The small system will be the same as before, while the large system will be parameterized with the node as the parametric type. Although it requires parametric verification in the equivalence checking step, it is still more amenable to verification compared to an arbitrarily designed hierarchical protocol.

We have mentioned in Section 3.5 that the equivalence checking will employ aggregation functions. So the difficulty is how to implement parametric verification with

those aggregation functions. In theory, we could still perform the abstraction and refinement steps. But in practice, these operations involve much more details and not as straightforward as before. For example, in each aggregation function, the involved nodes need to be abstracted, which may lead to spurious counterexamples and require refinement. This parameterization will couple with the normal parameterization of the system and we need to carefully handle them. No one has ever done parametric verification on aggregation functions, and we also anticipate it to be very challenging.

4.7 Summary

Due to state space explosion problem, automated tools are generally incapable of verifying a flat cache coherence protocol for a system with an arbitrary number of nodes. In this chapter, we explore the potential to design a flat cache coherence protocols such that they are amenable to formal verification and scalably verifiable. We use a mostly-automated form of parametric verification, called Simple-PV. We have shown that, with awareness of certain issues that affect parameterization, we can design protocols that are compatible with parametric verification. We call this kind of protocols PVCoherence protocols. Furthermore, our experimental results show that we can develop a PVCoherence protocol that is both compatible with parametric verification and achieves performance comparable to today's typical multicore coherence protocols. Together with Fractal Coherence, PVCoherence may enable us to verify a larger domain

of cache coherence protocols. However, the combination is not easy work considering the fact that the equivalence checking also requires parameterized verification.

5. Leveraging Fractal Coherence and PVCoherence to Verify Memory Consistency

Memory consistency is another fundamental problem with a shared memory many-core system. It is a guarantee of what hardware can provide for software with regard to the ordering of memory operations. Typically, this guarantee is specified as a memory consistency model, which is a set of rules that determines the allowed behavior of reads and writes from multiple threads [2]. Architects must design a shared memory system such that it satisfies the specified memory consistency model. However, the process of designing the memory system and then verifying that the design satisfies the specified consistency model is complicated and error-prone, even for seemingly simple consistency models like Sequential Consistency (SC). The complexity is due to both complicated hardware designs that seek to optimize performance and the need for architects and verification teams to reason about concurrency. As a confirmation of this complexity, subtle bugs have been uncovered in the designs of commercial processors [42]. In this chapter, we will discuss how to leverage the previously proposed design methodology of cache coherence protocols to facilitate the verification of memory consistency.

5.1 Difference between Cache Coherence and Memory Consistency

Before discussing the verification of memory consistency, we would like to distinguish memory consistency from cache coherence. They are both critical issues in a shared memory system with caches, but different in essence. There are at least the following several aspects we can tell them apart.

Software visibility. Cache coherence is a microarchitectural feature. The purpose of it is to make the caches invisible. The programmer cannot sense the existence of the coherence, or even caches. In contrast, memory consistency is an architectural feature. It provides a contract between the hardware and software by specifying what kind of orderings of reads and writes will be performed by different threads. Therefore, programmers must be aware of the memory consistency model before programming for the particular hardware. Then they can write multithreaded code without having to reason about the hardware implementation.

Memory location. In a cache coherence protocol, only one memory location is considered since accesses to different locations do not cause a data inconsistency. But memory consistency restricts the ordering of memory operations across all different locations.

Necessity. Cache coherence, or more accurately, hardware cache coherence, is not a must for a system. To the extreme, if the system does not have caches, we do not

need any coherence at all. However, any shared memory system needs to implement a certain kind of memory consistency model, no matter it is a strict one or a weak one.

Cache coherence and memory consistency seem totally unrelated. However, the majority of modern systems do implement their memory consistency models with coherent caches. This is because it is much easier to implement the consistency on top of cache coherence. Cache coherence, as a hardware optimization, enables the hardware to function correctly, and thus provide a correct interface for the memory consistency model.

5.2 Decomposing the Verification of Memory Consistency

To minimize the likelihood of memory consistency bugs escaping into shipped processors, industrial development teams spend a vast amount of time and effort in its verification. The current state of the art is to design the system and then verify it using a combination of simulation and formal verification. The combination is due to the fact that both simulation and formal verification have their own advantages and unavoidable problems. Traditional simulation methods are intuitive, but unlikely to cover every possible scenario in non-trivial, scalable systems. Formal verification of memory consistency, which is complete and good at uncovering subtle bugs, is extremely difficult. Previous work [22], [44] in this area is either too abstract to be realistic, non-scalable (i.e., using model checking), or requires huge amounts of manual

effort (i.e., using theorem proving). Thus, while formal methods are useful tools, their limitations have restricted the extent to which they can be used to verify memory consistency. Thus, whatever verification methodology is used, the current situation is that memory consistency cannot be completely verified for complex, modern processors.

There is one piece of work, called Dynamic Verification of Memory Consistency (DVMC) [72], which proved that if a memory system satisfies three invariants, then the implementation conforms to the memory consistency model. In this way, the complex process of verifying memory consistency is decomposed into three simpler verification steps. The three invariants are as follows.

1. **Uniprocessor ordering.** In a single-threaded system, a core's read from a given memory location should get the value from the last write to that location in program or-der. We must verify that this rule is not violated in a multithreaded system unless other cores access this memory location.
2. **Allowable reordering.** To improve performance, many consistency models reorder memory operation between when a core issues them and when they are performed in the cache. We must verify that the system does not reorder memory operations in ways that violate the consistency model. This is the only one of the three invariants that depends on the specific consistency model.

3. **Cache coherence.** Cache coherence requires that, for each block of memory, (a) its lifetime can be divided into epochs in which there is either a single writer or multiple readers, and (b) the value of the block at the beginning of an epoch is the same as the value at the end of the most recent read-write epoch. We must verify that the memory system is coherent, and this is the non-scalable verification step, because coherence involves all cores.

Note that these three invariants are sufficient but not necessary for a system to be memory consistent. For example, memory consistency can be satisfied by a system without cache coherence.

5.3 Architecting Memory System to Facilitate the Verification of Consistency

Based on the conclusion of DVMC, we propose a system model, called the Verifiable Consistent Model (VCM), which enables easier verification of memory consistency. It is worth mentioning that in the DVMC proposal, the goal is dynamic (runtime) verification, rather than static design verification as we do, but that difference does not matter here. The key of the VCM is that, for systems that adhere to it, verification of memory consistency is factored into three verification problems that are small and scalable. Instead of trying to verify memory consistency in one fell swoop, VCM enables verification to be done in three parts, each of which is self-contained and scalable. VCM simplifies verification by explicitly considering verification early in the

development process. These three verification steps correspond to three distinct self-contained portions of the system model, as illustrated in Figure 27: core, reordering mechanism, and cache-coherent memory system. We now discuss how to verify each part in this model and whether the verification step is scalable.

Uniprocessor Ordering

The Uniprocessor Ordering invariant requires us to design the core such that it is logically in-order. Fortunately, all cores are architecturally in-order, regardless of whether the microarchitecture is pipelined, superscalar, speculative, or out-of-order. All existing cores appear to execute instructions sequentially in program order.

Verification Process: Because cores are already logically in-order, validating that a core satisfies Uniprocessor Ordering is a process that already occurs during processor development. This validation process is well-understood and constrained to just the core, and thus there are no scalability challenges in validating Uniprocessor Ordering.

Allowable Reordering

Depending on the specific consistency model, the VCM permits the insertion of a reordering mechanism between each core and its cache hierarchy. The goal of the reordering mechanism is to improve performance. Sequential consistency does not permit a reordering mechanism, but other models permit reordering mechanisms [3]

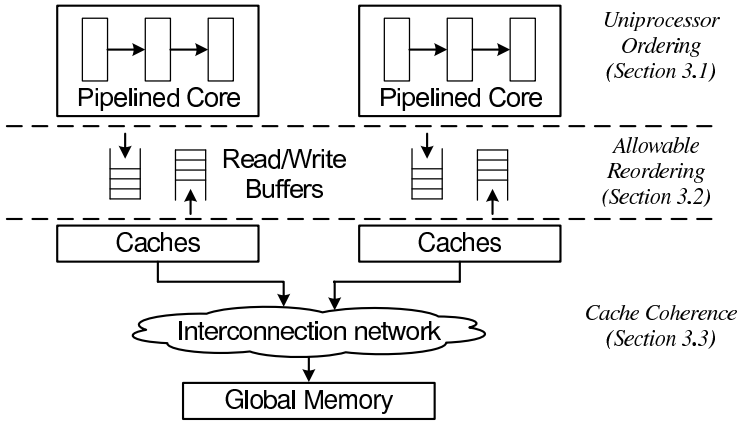

Figure 27: Architecture for shared-memory system

Table 5: TSO ordering

1st \ 2nd	Load	Store	Membar
Load	True	True	True
Store	False	True	True
Membar	True	True	True

such as FIFO write buffers (processor consistency, TSO, x86) and coalescing, unordered write buffers (weak ordering, Alpha, Power).

Verification Process: To verify the reordering mechanism is correct, we must verify that it does not permit reorderings that are prohibited by the consistency model.

To aid in this verification process, we can construct an allowable reordering table for a memory consistency model as presented in Hill et al. [7]. An ordering table for TSO is shown in Table 5. The first column corresponds to the first memory operation, and the first row corresponds to the second memory operation. "False" means there is no ordering requirement for the two operations and "True" means there is. For TSO, the

program order is relaxed when a write followed by a read to a different address, so the corresponding entry is "False".

Verifying the Allowable Reordering invariant is a simple process that is confined to just the reordering mechanism itself. For example, consider a system that is supposed to provide TSO and that has a FIFO write buffer between each core and its cache hierarchy. The verification process consists of verifying that the only reordering that can occur is for a load to pass an older store. Similar to the verification of Uniprocessor Ordering, the verification of Allowable Reordering has no scalability concerns, because it does not depend on the number of cores.

Cache Coherence

The third part of a VCM system is the cache-coherent memory system. We must design the memory system—including caches, interconnection network, memories, and coherence controllers—such that the coherence invariants are maintained. Then we need to verify that the design does maintain coherence. The previous two verification steps are simple and scalable, but the third step is not. Traditional snooping and directory protocols are difficulty to verify and not scalably verifiable. To make this step simpler and scalable, we leverage the ideas of prior work: Fractal Coherence and PVCoherence. Depending on the structure of the system, we choose the appropriate cache coherence protocols. If the system is flat and has only one level cache coherence

protocol, we design it in the way that adheres to PVCoherence. Otherwise, we design it as a Fractal Coherence protocol.

6. Conclusions and Future Work

In this thesis, we, from the high level, propose to design cache coherence protocols with the verifiability as a first-class constraint. We made this suggestion based on the fact that the correctness of a cache coherence protocol is critical to the system as well as the reality that current verification methods are ineffective at verifying protocols designed with only performance consideration. Specifically, we propose two design methodologies, following which, the cache coherence protocol will be easier to verify. Fractal Coherence is to design a hierarchical cache coherence protocol that can be inductively verified. The verification steps are reduced to only two steps, including the base case verification and self-similarity verification. Although effective to reduce the verification difficulty, Fractal Coherence can only apply to cache coherence protocols whose base case is manageable with current methodologies, including both automated tools and human reasoning. To explore how large the design space could be, we study the features that a cache coherence protocol in the base case should have so that it can be verified no matter how many nodes it has. We then propose PVCoherence, a design methodology for just flat cache coherence protocols. PVCoherence designates a set of guidelines and protocols that adhere to these guidelines can be parametrically verified independent of the number of nodes.

As architects, we definitely do not want to greatly degrade performance with our proposed design methodologies. It is almost for sure that we will have a little bit negative impact on the performance since we have design limitations in both Fractal Coherence and PVCoherence. There are a few optimizations that are incompatible with the two methods. However, evaluations with real benchmarks and optimized baseline protocols show that we can have confidence in both Fractal Coherence and PVCoherence since the performance impact is almost negligible. The overhead in other aspects, such as storage, and network traffic, are also acceptable.

A side benefit coming with Fractal Coherence and PVCoherence is they enable the easier verification of memory consistency. Verifying an implementation of a certain memory consistency model satisfies the model description is never easy work. Previous work, DVMC, simplify this problem into sub problems which are easier to handle. While a problem DVMC has not conquered yet is that the verification of cache coherence protocols is not scalable. Leveraging Fractal Coherence and PVCoherence to design the cache coherence protocol in the first place can thus solve the verification difficulty of memory consistency.

We believe we have done a significant job in incorporating verification into early design stage and showing how that helps improve the efficiency of verification. There is little work done in designing for verification before. But from our results "design for

verification" can gain us the benefits as important as that gained by "design for testing", which received much more attention. Traditional design flow has flaws in that architects do not consider what impact their design will exert on the verification team, partially because they are unaware what features are amenable to verification, what are not. Without delving into the analysis and comparison of verification effort for each individual feature, it is not easy to provide architects with a clear answer. We have taken the first step in this direction and given a few design guidelines and hints to architects in the cache coherence protocol design process. We encourage them to follow these methods, which greatly help their verification fellows while ensure them almost the same performance. We think many areas, such as the process verification, power verification, etc., should have the similar research proposals, since verification, for many of them, is also a problem not being solved.

Look into future, there are several directions we can pursue. The first one is to develop a thorough proof of combining Fractal Coherence and PVCoherence and have a real implementation of it showing how it facilitates verification. Both Fractal Coherence and PVCoherence partially solve the verification of cache coherence protocols from different perspectives. Combining them in the design of a cache coherence protocol will provide larger freedom to the system configuration. So the system is not only able to scale infinitely in the hierarchical dimension, but also in the horizontal dimension. The

second one is to automate the refinement step in the verification of a PVCoherence protocol. We think there is a relationship between the protocol feature and the constraints that need to be added. If we explore this relationship, it might be possible to automate or at least guide the refinement.

References

[1] J. Abella, R. Canal, and A. Gonzalez, "Power- and Complexity-Aware Issue Queue Designs," *IEEE Micro*, vol. 23, no. 5, pp. 50–58, 2003.

[2] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[3] A. Agarwal et al., "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 2–13.

[4] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 280–289.

[5] H. Akhiani et al., "Cache Coherence Verification with TLA+," in *Formal Methods, Volume II., Lecture Notes in Computer Science*, 1999, vol. 1709, p. 1871.

[6] A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," in *Proceedings of the 9th IEEE Symposium on High Performance Computer Architecture*, 2003, pp. 7–18.

[7] R. Alur and others, "MOCHA: Modularity in Model Checking," in *Proceedings of the 10th International Conference on Computer Aided Verification*, 1998, pp. 521–525.

[8] Arvind, N. Dave, and M. Katelman, "Getting Formal Verification into Design Flow," *Formal Methods, Lecture Notes in Computer Science*, pp. 12–32, 2008.

[9] L. A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 282–293.

[10] K. Baukus, Y. Lakhnech, and K. Stahl, "Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness," in *the 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2002, pp. 317–330.

[11] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu, "BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440, N. Halbwachs and L. Zuck, Eds. Springer Berlin Heidelberg, 2005, pp. 581–585.

[12] J. G. Beu, J. A. Poovey, E. R. Hein, and T. M. Conte, "High-Speed Formal Verification of Heterogeneous Coherence Hierarchies," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture*, 2013.

[13] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-Client Pairing: a Framework for Implementing Coherence Hierarchies," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 226–236.

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

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

[16] N. S. Bjørner et al., "Verifying Temporal Properties of Reactive Systems: A STeP Tutorial," in *Formal Methods in System Design*, 2000, pp. 1–45.

[17] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C–35, no. 8, pp. 677–691, 1986.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: 1020 States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[19] S. Burckhardt, R. Alur, and M. M. K. Martin, "Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement," in *the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[20] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, vol. C–27, no. 12, pp. 1112–1118, 1978.

[21] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[22] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared Memory Consistency Protocol Verification against Weak Memory Models: Refinement via Model-Checking," in *Computer Aided Verification, Lecture Notes in Computer Science*, 2002, pp. 123–136.

[23] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "Efficient Methods for Formally Verifying Safety Properties of Hierarchical Cache Coherence Protocols," *Formal Methods in System Design*, vol. 36, no. 1, pp. 37–64, 2010.

[24] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-Aware Coherence Protocols for Chip Multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[25] C.-T. Chou, P. Mannava, and S. Park, "A Simple Method for Parameterized Verification of Cache Coherence Protocols," in *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, 2004, vol. 3312, pp. 382–398.

[26] E. Clarke et al., "Verification of the Futurebus+ Cache Coherence Protocol," *Formal Methods in System Design*, vol. 6, no. 2, pp. 217–232, 1995.

[27] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996.

[28] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Apr. 2010.

[29] J. Crow, S. Owre, J. Rushby, N. Shankar, and and M. Srivas, "A Tutorial Introduction to PVS," *Workshop on Industrial-Strength Formal Specification Techniques*, Apr. 1995.

[30] P. Curzon and I. Leslie, "Improving Hardware Designs Whilst Simplifying Their Proof," in *Proceedings of the 3rd Workshop on Designing Correct Circuits*, 1996.

[31] D. Cyrluk, S. Rajan, N. Shankar, and and M. K. Srivas, "Effective Theorem Proving for Hardware Verification," in *Theorem Provers in Circuit Design (TPCD '94)*, 1994, vol. 901, pp. 203–222.

[32] S. Das, D. L. Dill, and S. Park, "Experience with Predicate Abstraction," in *Computer Aided Verification*, 1999, pp. 160–171.

[33] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 522–525.

[34] S. J. Eggers and R. H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 122–130.

[35] E. A. Emerson and V. Kahlon, "Exact and Efficient Verification of Parameterized Cache Coherence Protocols," in *Correct Hardware Design and Verification Methods*, 2003, pp. 247–262.

[36] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, "CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes," in *Computer Aided Verification*, vol. 4590, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, pp. 158–163.

[37] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren, "Architecture and Design of AlphaServer GS320," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 13–24.

[38] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 278–287.

[39] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983, pp. 124–131.

[40] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: a New Large-Scale Cache-Coherent Multiprocessor," in *Proceedings of the 15th Annual International Symposium on Computer architecture*, 1988, pp. 422–431.

[41] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," in *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, 1999, pp. 172–181.

[42] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 114–123.

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

[44] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems," in *In Proceedings of the 11th International Conference on Computer-Aided Verification*, 1999, pp. 301–315.

[45] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence Protocol Optimization for Both Private and Shared Data," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 45–55.

[46] R. Hum, "How to Boost Verification Productivity," *EE Times*, Jan. 2005.

[47] C. N. Ip and D. L. Dill, "State Reduction Using Reversible Rules," in *Proceedings of the 33rd annual Design Automation Conference*, 1996, pp. 564–567.

[48] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 589–604, Jul. 2005.

[49] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 1985, pp. 276–283.

[50] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[51] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way Multithreaded SPARC Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Apr. 2005.

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

[53] J. Kuskin et al., "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 302–313.

[54] H. P. Labs, "Cacti 5.3," *http://www.hpl.hp.com/research/cacti/*. .

[55] E. Ladan-Mozes and C. E. Leiserson, "A Consistency Architecture for Hierarchical Shared Caches," in *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 11–22.

[56] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 241–251.

[57] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 148–159.

[58] A. Lungu and D. J. Sorin, "Verification-Aware Microprocessor Design," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2007, pp. 83–93.

[59] P. S. Magnusson et al., "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

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

[61] M. M. K. Martin et al., "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[62] M. M. K. Martin, "Formal Verification and its Impact on the Snooping versus Directory Protocol Debate," in *Proceedings of the International Conference on Computer Design*, 2005.

[63] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 206–217.

[64] M. R. Marty and M. D. Hill, "Virtual Hierarchies to Support Server Consolidation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[65] M. R. Marty and others, "Improving Multiple-CMP Systems Using Token Coherence," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 328–339.

[66] K. L. Mcmillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking," 1998, pp. 110–121.

[67] K. L. Mcmillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," in *In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*, 2001, pp. 179–195.

[68] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[69] K. L. McMillan, "Verification of Infinite State Systems by Compositional Model Checking," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1999, pp. 219–234.

[70] K. L. McMillan, "Getting Started with SMV," *http://www.kenmcmil.com/psdoc.html*. .

[71] K. L. McMillan and J. Schwalbe, "Formal Verification of the Gigamax Cache-Consistency Protocol," in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1991, pp. 242–251.

[72] A. Meixner and D. J. Sorin, "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2006, pp. 73–82.

[73] S. Merz, "The Specification Language TLA+," in *Logics of Specification Languages*, D. Bjørner and M. Henson, Eds. Springer Berlin Heidelberg, 2008, pp. 401–451.

[74] G. J. Milne, "Design for Verifiability," in *Proceedings of the Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, 1989, pp. 1–13.

[75] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[76] J. S. Moore, "An ACL2 proof of Write Invalidate Cache Coherence," in *Computer Aided Verification*, vol. 1427, A. Hu and M. Vardi, Eds. Springer Berlin Heidelberg, 1998, pp. 29–38.

[77] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The Alpha 21364 Network Architecture," in *Proceedings of the 9th Hot Interconnects Symposium*, 2001.

[78] S. S. Mukherjee and M. D. Hill, "An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors," in *Proceedings of the International Conference on Supercomputing*, 1994, pp. 64–74.

[79] C. Norris Ip and D. Dill, "Better Verification through Symmetry," *Formal Methods in System Design*, vol. 9, no. 1–2, pp. 41–75, 1996.

[80] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Proceedings of the 11th International Conference on Automated Deduction*, 1992.

[81] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984, pp. 348–354.

[82] S. Park, S. Das, and D. L. Dill, "Automatic Checking of Aggregation Abstractions through State Enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1202–1210, Nov. 2006.

[83] S. Park and D. L. Dill, "Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions," in *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, 1996, pp. 288–296.

[84] F. Pong, M. Browne, A. Nowatzyk, and M. Dubois, "Design Verification of the S3.mp Cache-Coherent Shared-Memory System," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 135–140, Jan. 1998.

[85] F. Pong and M. Dubois, "A New Approach for the Verification of Cache Coherence Protocols," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 8, pp. 773–787, 1995.

[86] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois, "Verifying Distributed Directory-Based Cache Coherence Protocols: S3.mp, a Case Study," in *EURO-PAR '95 Parallel Processing*, vol. 966, S. Haridi, K. Ali, and P. Magnusson, Eds. Springer Berlin Heidelberg, 1995, pp. 287–300.

[87] R. T. Simoni,Jr., "Cache Coherence Directories for Scalable Multiprocessors," PhD Thesis, Stanford University, 1992.

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

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

[90] 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 *Proceedings of the 12th International Conference on VLSI Design - "VLSI for the Information Appliance,"* 1999, pp. 288–293.

[91] P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 109–118.

[92] U. Stern and D. L. Dill, "Automatic Verification of the SCI Cache Coherence Protocol," in *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995, pp. 21–34.

[93] U. Stern and D. L. Dill, "Combining State Space Caching and Hash Compaction," in *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, 1996, pp. 81–90.

[94] M. Talupur and M. R. Tuttle, "Going with the Flow: Parameterized Verification Using Message Flows," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, 2008, pp. 1–8.

[95] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, 2011, pp. 132–143.

[96] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal Coherence: Scalably Verifiable Cache Coherence," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

Biography

Meng Zhang was born on September 16th, 1984 in China. She received her M.E and B.E degrees from Beihang University in Beijing, China in 2008 and 2005 respectively, both with the honor of outstanding graduate thesis. From fall 2008, she joined the department of Electrical and Computer Engineering at Duke University and became a research assistant working with Professor Daniel J. Sorin. Her research focuses on designing verifiable cache coherence protocols for modern computers. She has published three papers and submitted another two in this area. She also did two internships with the architecture group of Nvidia Research in 2011 and 2012, where she designed cache coherence protocols for CPU and GPU heterogeneous systems. She was awarded ECE graduate fellowship from 2008 to 2009 and Nvidia Graduate Fellowship from 2012 to 2013.