Fractal Consistency: Architecting the Memory System to Facilitate Verification

Meng Zhang[1], Alvin R. Lebeck[2], Daniel J. Sorin[1]

[1]*Dept. of Electrical and Computer Engineering,* [2]*Dept. of Computer Science, Duke University*

Abstract—One of the most challenging problems in developing a multicore processor is verfiying that the design is correct, and one of the most difficult aspects of pre-silicon verification is verifying that the memory system obeys the architecture's specified memory consistency model. To simplify the process of pre-silicon design verification, we propose a system model called the Fractally Consistent Model (FCM). We prove that systems that adhere to the FCM can be verified to obey the memory consistency model in three simple, scalable steps. The procedure for verifying FCM systems contrasts sharply with the difficult, non-scalable procedure required to verify non-FCM systems. We show that FCM systems do not necessarily sacrifice performance, compared to non-FCM systems, despite being simpler to verify.

Index Terms—Memory Consistency, Multicore, Verification, Validation

1 Introduction

For a shared memory multiprocessor, the memory consistency model specifies the correct behavior of the memory system [1]. In particular, the consistency model specifies the legal, software-visible orderings of reads and writes performed by different threads. The memory consistency model is a software-visible interface and enables the programmer to write multithreaded code without having to reason about the hardware implementation.

Architects must design a multiprocessor's 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 [5].

To minimize the likelihood of memory consistency bugs escaping into shipped processors, industrial development teams spend a vast amount of time and effort in the process of pre-silicon (static) design verification (also known as design validation). The current state of the art is to design the system and then verify it using a combination of testing and formal verification. The combination is due to the fact that both testing and formal verification have their own advantages and unavoidable problems. Traditional testing 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 [2, 6] 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.

Considering the difficulty in designing a memory system and verifying that it satisfies its consistency model, we propose a *system model*, called the *Fractally Consistent Model (FCM)*, that enables easier pre-silicon verification. The key is that, for systems that adhere to the FCM, 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, FCM enables verification to be done in three parts, each of which is self-contained and scalable. FCM simplifies verification by explicitly considering verification early in the development process.

Some previous work has considered verifiability inside the memory system. Fractal Coherence [10] presents a design methodology for verifiable cache coherence, but coherence is not sufficient. Consistency defines architectural correctness, whereas coherence is a microarchitectural mechanism that helps to support consistency. Landin et al. [8] proposed the use of race-free interconnection networks to facilitate reasoning about how performance optimizations affect SC and Processor Consistency (PC). With race-free networks, they could re-factor SC and PC into clear invariants. While these invariants could have been used during the verification process, the authors did not consider verification, which is the focus of our work.

Manuscript submitted: 07-Oct-2010. Manuscript accepted: 28-Oct-2010. Final manuscript received: 01-Nov-2010.

2 Background

The keys to why FCM systems are easier to verify are in two pieces of prior work. In Dynamic Verification of Memory Consistency (DVMC) [9], the authors decompose the process of verifying memory consistency into three simpler verification steps.[1] Of those three verification steps, two are simple and scalable, but the third step is not. To make that step simpler and scalable, we use the result of the other relevant prior work: Fractal Coherence [10]. We now discuss the relevant aspects of both DVMC (Section 2.1) and Fractal Coherence (Section 2.2).

2.1 Dynamic Verification of Memory Consistency

The DVMC paper [9] proved that if a memory system satisfies three invariants, then the implementation conforms to the memory consistency model. 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 order. 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.

2.2 Fractal Coherence

To facilitate verification of cache coherence, the non-scalable verification step from Section 2.1, we leverage Fractal Coherence [10]. Fractal Coherence is a design methodology for cache coherence protocols that can be formally verified using existing, automated, easy-to-use formal verification tools. Fractal Coherence protocols are designed in a way that enables fractal behavior, i.e., each scale of the system has exactly the same behavior. Using existing, automated formal verification tools, we can both verify the smallest scale of the system, called the minimum system, and we can also verify that the behavior of the system is fractal. We can then use induction to prove that the system, regardless of size, is coherent.

We are unaware of any other protocols that are fractal. Traditional snooping and directory protocols are not fractal, nor are existing hierarchically designed protocols. Logical hierarchy is necessary to achieve fractal behavior, but it is not sufficient. Our experience has shown that achieving fractal behavior requires detailed, explicit effort and does not occur "naturally", even in hierarchical protocols.

3 FCM Design and Verification

Combining the results from the prior work discussed in Section 2 allows us to decompose the verification of FCM systems into three simpler, scalable verification steps. These three verification steps correspond to three distinct self-contained portions of the system model, as illustrated in Fig. 1: core (Section 3.1), reordering mechanism (Section 3.2), and cache-coherent memory system (Section 3.3).

3.1 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.

3.2 Allowable Reordering

Depending on the specific consistency model, the FCM 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] such as FIFO write buffers (processor consistency, TSO, x86) and coalescing, un-ordered 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 1. 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 sup-

[1] In the DVMC work, the goal is dynamic (runtime) verification, rather than static design verification, but that difference does not matter here.

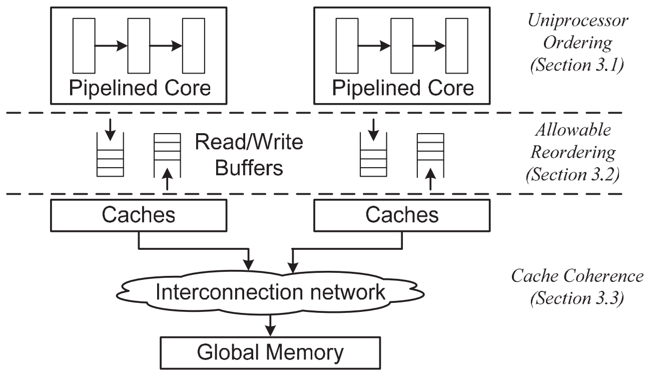
Pipelined Core
Pipelined Core
Uniprocessor Ordering (Section 3.1)
Read/Write Buffers
Allowable Reordering (Section 3.2)
Caches
Caches
Interconnection network
Cache Coherence (Section 3.3)
Global Memory

Figure 1. Architecture for shared-memory system

TABLE 1 TSO ORDERING

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

posed 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.

3.3 Cache Coherence

The third part of an FCM 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. Traditional snooping and directory protocols are not scalably verifiable, which motivates us to choose for FCM a coherence protocol design, called Fractal Coherence, that is scalably verifiable.

Fractal Coherence Structure. Fractal Coherence protocols need a hierarchical logical structure, such as a tree. The physical interconnection has no restriction. The components in a Fractal Coherence protocol are shown in Fig. 2. The shadowed square components are *basic nodes*, which may have a number of caches, cores and memories. The elliptical shape components are *Interfaces* that support the fractal behavior. There exist two kinds of Interfaces, named according to their positions in the system, i.e., *Top Interface* and *Internal Interface*. We use a specific protocol, TreeFractal, to illustrate the design steps of Fractal Coherence protocols, as shown in Fig. 2(a) and (b).

Smallest System Design. The design process involves two steps: the design of the smallest system, which includes only the basic nodes and the Top Interface, and the design of the scaled system, which includes all different kinds of components in the system. The smallest system for a D-degree tree in TreeFractal is shown in Fig. 2(a). The Top Interface holds copies of the cache tags and coherence states of its D children, and it serves as the serialization point for coherence transactions. For each child, there is a coherence controller responsible for the communication with the core and the Top Interface.

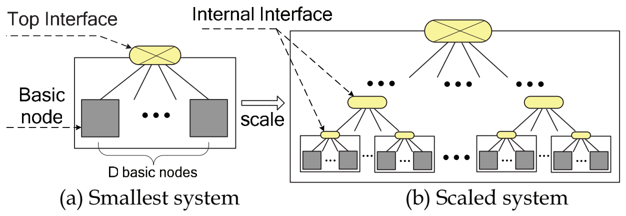

Figure 2. Design steps of a Fractal Coherence protocol

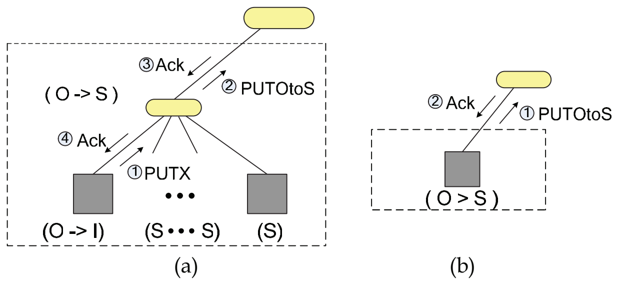

Figure 3. Correct implementation to ensure fractal behavior

TreeFractal is similar to both snooping and directory protocols, but it is not the same as either of them. The coherence controller in the basic nodes responds to load and store requests from the core. Depending on whether the requests can be satisfied within the node itself, the coherence controller decides whether to issue a coherence request up to the Top Interface. The Top Interface is responsible for responding to the coherence requests from its children cores. After receiving a request, the Top Interface looks up the state of the block in all its children. Based on these states, the Top Interface might need to forward the request down to its child/children, similar to directory protocols. As the serialization point for all transactions, the Top Interface always needs to forward 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.

Scaled System Design. The second step is the scaled system design. The smallest system can scale to any arbitrary N-node system by adding Internal Interfaces between the Top Interface and the basic nodes as shown in Fig. 2(b). In the scaled system, requests and replies also go up the tree and forwarded requests and forwarded replies also go down the tree, however, they do not go all the way to the Top Interface each time as the smallest system does. They only need to go up to the highest common ancestor Interface of all destinations.

The most important task in the scaled system design is to ensure the fractal behavior. Simply expanding the smallest system without careful consideration can easily break the fractal behavior. We give a specific example to show how to maintain the fractal behavior. As shown in Fig. 3(a), for a single block, the Internal Interface has a child in state O (wned), and all other D-1 children in state S (hared). Observed from the external world, the node as a whole (i.e., the Internal Interface and its children) appears in O since it has the ownership. Then the child in O issues a PutO (meaning changing from state O to I(nvalid)) coherence request to evict that block. When the PutO arrives at the Internal Interface, we must consider the implication of the PutO to determine how the Internal Interface should respond to it. For the node as a whole, the PutO means it gives up the ownership and becomes a sharer of the block. So the final state should be S instead of I. If the Internal Interface simply sends a PutO to the Interface above it, the above Interface cannot distinguish it from a common PutO and will send an acknowledgement back for it to change to state I. But the transition we need here is from O to S. Considering that this O to S transition is impossible for a single node, we need to add a new message type, called Put-From-O-to-S (PutOtoS), to both the scaled system and the smallest system to maintain fractal, shown in Fig. 3 (a) and (b). The PutOtoS notifies the above Interface the transition is from O to S. The above Interface will send acknowledgement back for this PutOtoS request.

The given example is one of the many specifically designed transitions to maintain the fractal behavior. It shows the Internal Interface has to decide in what form a request should be displayed to the outside world depending on its actual implication. With a properly designed Internal Interface, we can scale the system to any number of nodes while maintaining the fractal behavior.

Verification Process: As proven in the original Fractal Coherence paper [10], Fractal Coherence protocols can be formally verified to maintain cache coherence for any number of cores, using two scalable verification steps. First, we must verify that the minimum system is coherent Second, we must verify the fractal behavior, i.e., the larger-scale systems behave like the minimum system. Both verification steps can be performed with existing automated formal tools. The inductive proof that these two steps are sufficient for the scalable verification is based on the correctness of the minimum system (as the base case) and the fractal behavior of Fractal Coherence protocols (as the inductive step).

4 Performance analysis

FCM is a system model that facilitates the verification of consistency. For FCM to be viable, it is important that the performance of systems that conform to FCM is comparable to the performance of traditional, non-FCM systems. FCM could potentially sacrifice performance due to two constraints in the system model.

One constraint is that we draw hard lines between the cores and the reordering mechanisms and between the reordering mechanisms and the cache-coherent memory system. This rigid system partitioning may sacrifice performance compared to a system that integrates these parts more tightly. However, given that many commercial machines already enforce these partitions, we believe that FCM is probably not leaving much performance on the table due to partitioning, but it is an interesting open question whether tighter integration offers significant performance benefits.

The other constraint is that FCM systems use Fractal Coherence protocols instead of traditional snooping or directory protocols. However, we have experimentally shown [10] that Fractal Coherence protocols have comparable performance to traditional snooping and directory protocols. Therefore, this constraint will not hinder performance. The above two aspects give us confidence in the performance of Fractal Consistency.

5 Conclusion

How to design a memory system and completely verify that it complies with the corresponding memory consistency model is difficult. Traditional methods usually aim to find bugs after the design has been completed, which is laborious and expensive. We propose a system model, FCM, based on three invariants. If the designer fits their system into this model, the verification can be decomposed into three steps that are self-contained and scalable. This system model eases the verification of the memory system by considering verification early in the design stage.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CCF-0702434 and CCF-0811290.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer,* Vol. 29, No. 12, 1996.

[2] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. Shared Memory Consistency Protocol Verification against Weak Memory Models: Refinement via Model-Checking. *In Proc. of 14th Int'l Conference on Computer-aided verification*, 2002.

[3] K. Gharachorloo. Memory Consistency Models for Shared-memory Multiprocessors. *PhD Thesis,* Stanford University, 1995.

[4] P.B. Gibbons and E. Korach. Testing Shared Memories. *SIAM J. Computing*, vol. 26, no. 4, 1997.

[5] S. Hangal, D. Vahia, C. Manovit, J. J. Lu, and S. Narayanan. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *Proc. of 31st Annual Int'l Symposium on Computer Architecture*, 2004.

[6] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying Sequential Consistency on Shared-memory Multiprocessor Systems. *In Proc. of 11th Int'l Conference on Computer-aided verification*, 1999.

[7] M. D. Hill, A. E. Condon , M. Plakal , and D. J. Sorin. A system-level Specification Framework for I/O Architectures. In *Proc. of 11th Annual ACM Symposium on Parallel Algorithms and Architectures,* 1999.

[8] A. Landin, E. Hagersten, and S. Haridi. Race-free Interconnection Networks and Multiprocessor Consistency. In *Proc. of 18th Annual Int'l Symposium on Computer Architecture*, 1991.

[9] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. *IEEE Transactions on Dependable and Secure Computing*, Vol. 6, No. 1, 2009.

[10] M. Zhang, A. R. Lebeck, and D. J. Sorin. "Fractal Coherence: Scalably Verifiable Cache Coherence. To appear in *43rd Int'l Symposium on Microarchitecture,* 2010.