Fractal++: Closing the Performance Gap between Fractal and Conventional Coherence

Gwendolyn Voskuilen and T. N. Vijaykumar

School of Electrical and Computer Engineering, Purdue University

{geinfeld,vijay}@ecn.purdue.edu

Abstract

Cache coherence protocol bugs can cause multicores to fail. Existing coherence verification approaches incur state explosion at small scales or require considerable human effort. As protocols' complexity and multicores' core counts increase, verification continues to be a challenge. Recently, researchers proposed fractal coherence which achieves scalable verification by enforcing observational equivalence between sub-systems in the coherence protocol. A larger sub-system is verified implicitly if a smaller sub-system has been verified. Unfortunately, fractal protocols suffer from two fundamental limitations: (1) indirect-communication: sub-systems cannot directly communicate and (2) partially-serial-invalidations: cores must be invalidated in a specific, serial order. These limitations disallow common performance optimizations used by conventional directory protocols: reply-forwarding where caches communicate directly and parallel invalidations. Therefore, fractal protocols lack performance scalability while directory protocols lack verification scalability. To enable both performance and verification scalability, we propose Fractal++ which employs a new class of protocol optimizations for verification-constrained architectures: decoupled-replies, contention-hints, and fully-parallel-fractal-invalidations. The first two optimizations allow reply-forwarding-like performance while the third optimization enables parallel invalidations in fractal protocols. Unlike conventional protocols, Fractal++ preserves observational equivalence and hence is scalably verifiable. In 32-core simulations of single- and four-socket systems, Fractal++ performs nearly as well as a directory protocol while providing scalable verifiability whereas the best-performing previous fractal protocol performs 8% on average and up to 26% worse with a single-socket and 12% on average and up to 34% worse with a longer-latency multi-socket system.

1. Introduction

Cache coherence protocols facilitate shared-memory functionality in multicore and multiprocessor systems. As such, coherence protocol bugs can cause system failure. Unfortunately, verifying that a given protocol is correct and deadlock- and livelock-free is difficult. Straightforward state-exploration to check that each reachable system state is valid (e.g., only one writing core) quickly runs into scalability problems [24]. For instance, the state count for even 8 cores exhausts time and memory constraints. Other verification approaches [22,25,27] exploit inherent protocol symmetries to achieve more scalable verification. Though powerful, these approaches do not guarantee verification scalability. While the approaches have successfully verified a few protocols (e.g., S3.mp[23]), identifying whether and which symmetries exist in a given protocol is protocol-specific and generally requires considerable human effort. As such, many commercial protocols are verified only partially (e.g., SCI [9,11,12,26], Nehalem QPI despite using a 1000-node cluster running for a year [7], and HyperTransport [32]), and bugs do occur [5,13]. Thus, coherence verification remains an open problem, requiring increasing effort as systems add more shared-memory components (e.g., caches and GPUs).

Where prior verification approaches try to verify existing protocols, a new approach called *fractal coherence* [33] constrains protocols to facilitate verification. Specifically, fractal coherence enforces *observational equivalence* [20] between larger systems and their smaller sub-systems. This symmetry requires that the externally-visible actions of a larger system and a smaller sub-system are indistinguishable. Fractal coherence guarantees correctness irrespective of the system scale, provided a small sub-system is verified (using conventional techniques like state exploration).

Enforcing observational equivalence implies constructing larger self-similar systems from small sub-systems, naturally imposing a logical hierarchy on the protocol. The logical hierarchy restricts sub-systems' interactions with each other (e.g., sub-systems communicate only via common parents). The TreeFractal protocol [33] uses a tree architecture to achieve this logical hierarchy. Because cache misses traverse the hierarchy, TreeFractal's architecture greatly increases miss latency and hurts performance. Our previous work, FlatFractal [30], addresses this problem by decoupling the protocol's logical hierarchy from the architecture. FlatFractal folds the hierarchy into a flat, directory-like architecture that eliminates the miss traversal latency, greatly improving performance. However, the logical hierarchy's restrictions *fundamentally* disallow some key protocol performance optimizations employed by conventional directory coherence. FlatFractal does not address this issue, which degrades performance in all fractal protocols, regardless of the architecture. The degradation is especially apparent in multi-socket systems where communication (i.e., indirections and invalidations) involves long latencies. For example, in 32-core single-socket simulations FlatFractal performs on average 8% and up to 26% worse than a directory protocol, whereas in a longer-latency 32-core four-socket system, FlatFractal performs on average 12% and up to 34% worse. Thus, protocol designers must choose between scalable verification (i.e., fractal protocols) and scalable performance (i.e., directory protocols). In this paper, we obviate this choice by addressing fractal protocols' two key limitations to achieve comparable performance scalability to conventional protocols.

Fractal protocols' first limitation, *indirect-communication*, arises because observational equivalence between sub-systems fundamentally disallows direct communication between sub-systems across the hierarchy (i.e., cores). This limitation forces cores to send coherence responses (data and acknowledgments) to one another indirectly via the logical hierarchy. In contrast, conventional protocols employ *reply-forwarding*, where cores respond directly to requestors to reduce latency. Despite introducing both subtle races, which make protocols complex and error-prone, and the potential for network deadlocks (avoided by extra virtual networks), reply-forwarding is used in many real protocols due to its performance advantages (e.g., SGI Origin [15]). The second limitation, *partially-serial-invalidations*, occurs because fractal protocols invalidate caches in a specific, partially-serial order fundamentally required by the logical hierarchy. In contrast, conventional protocols' invalidations are fully parallel. Both limitations hurt read-write sharing, prevalent in commercial and scientific shared-memory programs.

To address these fundamental limitations, we propose *Fractal++*, a fractal protocol which includes a new class of performance optimizations for verifiability-constrained coherence protocols. Fractal++ includes three optimizations, two for the first limitation and one for the second limitation.

For the protocols' first limitation, indirect-communication, we observe that when cache blocks are read-write shared, directing cache-to-cache responses through the logical hierarchy has two negative performance effects relative to conventional protocols: (1) *internal*: the latency that requestors incur from request to access is greater, and (2) *external*: the latency that other requestors incur while waiting for a current request to be handled is greater. To this end, we propose two optimizations, *decoupled-replies*, which target the internal effect, and *contention-hints*, which target the external effect.

To reduce internal latency, we make the key observation that observational equivalence limits only externally-visible actions but not internal actions. Accordingly, we decouple cores' receipt of data and acknowledgment messages from externally-visible actions by splitting a data or acknowledgment message into two messages each: an early-reply which responding cores send directly to requestors and a parallel fractal confirmation sent through the fractal protocol's logical hierarchy. Upon receiving an early-reply, a requestor proceeds with execution *before* the fractal confirmation arrives. Thus, early-replies reduce a requestor's internal latency to that seen with reply-forwarding in conventional directories. However, allowing requestors to expose externally-visible effects of execution before the fractal confirmation arrives might violate the fractal protocol. For example, if a requestor immediately applies an incoming invalidation to the data received via an early-reply before the fractal confirmation, the requestor may re-validate the data on receiving the later fractal confirmation, potentially violating coherence. Hence, until the confirmation arrives, the requestor can proceed only with internal actions for the affected block (e.g., cache hits). Further, the extra bandwidth needed for fractal confirmations is under 4%.

Our second optimization, *contention-hints,* reduces the external latency seen by contending requestors waiting for the current request to complete. The hints notify current accessors of waiting requestors, allowing the accessors to replace the contended block to the shared last-level cache (LLC) before a subsequent request arrives. The replacement allows waiting requestors to incur shared LLC hits rather than longer remote misses. While the hints can also reduce the external latency in conventional protocols, the hints are more effective on fractal protocols' longer latencies. In contrast to dynamic self-invalidations (DSI) [16], which must predict when a shared block needs to be invalidated, contention-hints target known (not predicted) contending accesses where immediate invalidation is necessary.

Finally, we address fractal protocols' second limitation of partially serial invalidations and acknowledgments. We propose employing *fully-parallel-fractal-invalidation* messages based on the key observation that while acknowledgment collection must be in the fractal order, invalidations can be sent in any order. We achieve this decoupling via additional control messages. Fortunately, the extra messages are small, and while acknowledgment collection is fast and can be serial, the far-slower sending of invalidations and acknowledgments each occurs in parallel.

Unlike decoupled-replies, contention-hints and fully-parallel-fractal-invalidations do not violate the fractal property and hence are part of the fractal protocol. Together, Fractal++'s three optimizations improve performance scalability for read-write sharing and preserve fractal's verifiability.

While Fractal++'s new class of optimizations apply to all fractal protocols, combining them with FlatFractal's architecture brings Fractal++ close to the performance level of conventional protocols. Thus, unlike existing protocols, Fractal++ is scalable in *both* performance and verifiability.

To summarize, our contributions are:

- a new class of performance optimizations to alleviate fundamental restrictions in verifiability-constrained fractal protocols;
- decoupled-replies decouple data and acknowledgment receipt from externally-visible actions to allow direct

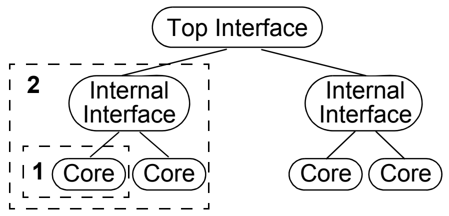

Figure 1. Observational equivalence

communication between requestor and responder while retaining the fractal property;

- contention-hints convert contending remote misses, whose latency is elongated by indirect communication, into shared LLC hits;
- fully-parallel-fractal-invalidations replace fractal protocols' partially-serial-invalidations with parallel invalidations and parallel acknowledgments where only the acknowledgment collection is serial;
- and, in 32-core simulations, compared to a directory protocol, Fractal++ using the FlatFractal architecture performs nearly as well while providing scalable verifiability, whereas FlatFractal performs 8% on average, and up to 26%, worse with a single-socket system and 12% on average, and up to 34%, worse with a longer-latency multi-socket system.

The rest of the paper is organized as follows. We provide background on fractal coherence in Section 2. We describe Fractal++'s optimizations and their verification in Section 3. Section 4 describes our experimental methodology. We present our results in Section 5, discuss related work in Section 6, and then conclude in Section 7.

2. Fractal Coherence: Background

Recall from Section 1 that fractal protocols [33] guarantee scalable verifiability by enforcing observational equivalence [20] among sub-systems in a larger system, which yields a hierarchical protocol. Figure 1 shows a fractal protocol organized as a binary tree, with two two-core sub-systems composed to form a larger system. Leaves are cores, and other nodes are coherence interfaces which act as directories for their child sub-systems. For observational equivalence, the interactions between each core (e.g., sub-system 1) and its parent must be identical to those between each sub-system (e.g., sub-system 2) and its parent. Because the interactions of sub-systems are identical to those of the larger system, only the smallest sub-system needs to be verified.

Fractal protocols impact performance in two ways. First, because the smallest sub-system must necessarily be quite small (i.e., to be verified using conventional techniques), the logical hierarchy is deep. A deep architectural hierarchy forces coherence messages to traverse many levels, increasing latency. Second, to enforce observational equivalence, the protocols restrict communication among sub-systems, disallowing conventional optimizations.

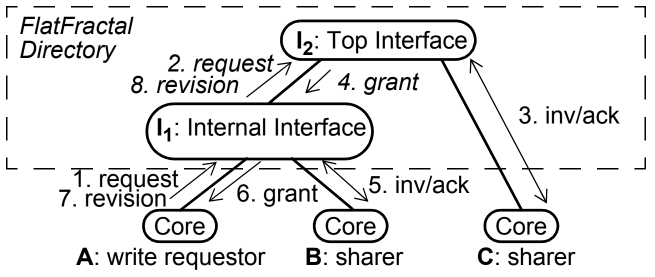

Figure 2. Fractal protocol

2.1 Fractal Coherence: Logical Protocol

To see how observational equivalence affects fractal protocols, consider the logical tree hierarchy in Figure 2. Suppose core *A* issues a write request for a block that is shared by cores *B* and *C* (step 1). Because *A*'s parent, I_1, does not have write permission for the block, I_1 passes the request to the top interface, I_2 (i.e., the request is serialized at the root of the sub-system with write permissions). I_2 then invalidates its non-requesting sharer *C* (step 3). This action is similar to that of a conventional directory protocol. However, the response action differs. In a conventional protocol, reply-forwarding would allow *C* to send responses (acknowledgments or data) directly to *A*. Unfortunately, such direct communication violates observational equivalence because (1) cores receive replies not received by the internal and top interfaces and (2) the number of interactions among cores increases with system scale. As such, *C* returns the acknowledgment indirectly by sending it to I_2. This *indirect-communication* is the first limitation described in Section 1.

On receiving the acknowledgment, I_2 grants write permission to the requesting sub-system (step 4). Because I_1 and I_2 are observationally equivalent, I_1 follows the same set of actions as I_2. I_1 invalidates non-requesting sharer *B* (step 5) and then I_1 grants write permission to *A* (step 6). Finally, as done in conventional protocols to signify request completion and to update the coherence state, *A* issues a revision message which is forwarded up the hierarchy (steps 7 and 8). This example demonstrates fractal protocols' second limitation, *partially-serial invalidations*. As each interface invalidates its non-requesting sub-system before granting permission to its requesting sub-system, sub-systems cannot invalidate their internal sharers (e.g., *B*) before the external sharers (e.g., *C*) are invalidated. These serial invalidations are much slower than the conventional parallel ones.

2.2 FlatFractal: Protocol Architecture

As stated in Section 1, the original fractal protocol, TreeFractal [33], adopts a hierarchical architecture which forces cache misses to incur many network hops and lookups as they traverse the hierarchy. To improve performance, our previous work, FlatFractal [30], decouples the logical and architectural hierarchies and employs a flattened directory-like architecture which eliminates the slow traversals without destroying observational equivalence or the logical hierarchy. Conceptually, a FlatFractal directory is observationally equivalent to the sub-system consisting of the logical hierarchy's top and internal interfaces (dashed box in Figure 2). As such, requests incur one hop to and from the FlatFractal directory rather than many hops between interfaces (i.e., FlatFractal eliminates the italicized hops in Figure 2). Likewise, requests look up the single directory rather than looking up each interface sequentially.

While FlatFractal addresses TreeFractal's architectural problems, FlatFractal inherits fractal's fundamental protocol limitations — indirect-communication and partially-serial-invalidations — which degrade performance relative to conventional protocols, especially for multi-socket systems where latencies are longer. For example, in Figure 2, the FlatFractal protocol still invalidates core *C* first (step 3) and only then invalidates core *B* (step 5). As a result, FlatFractal performs worse than a conventional directory protocol. This paper focuses on these limitations.

2.3 Verifying Fractal Protocols

Fractal protocols are verified in two steps. First, a minimal complete system is verified using conventional state exploration (e.g., *Murphi* [8]). Minimal systems include at least one of each type of coherence interface (e.g., the system shown in Figure 2). The second step uses a formal verification tool (e.g., *CADP* [10]) to check for observational equivalence of the interactions between the nodes in the minimal system (e.g., the *core-to-internal*, *internal-to-top*, and *core-to-top* interactions in Figure 2 are *all* equivalent). Both verification steps use the protocol's RTL implementation. In contrast, model-checking verification methods abstract an implementation into a model and verify the model, raising the possibility that the implementation and model differ (discussed later in Section 6). Finally, FlatFractal is proved to be observationally equivalent to fractal in [30].

3. Fractal++

To address the above fundamental protocol limitations, we propose Fractal++, a fractal protocol which includes a new class of fractal protocol optimizations. Specifically, we propose *decoupled-replies* and *contention-hints* to address fractal protocols' indirect-communication limitation, and *fully-parallel-fractal-invalidations* to address fractal protocols' partially-serial-invalidations limitation. These optimizations are scalably verifiable using fractal's verification framework and together close the performance gap between fractal and conventional directory protocols.

3.1 Decoupled-replies

Recall from Section 1 that for read-write sharing, indirect-communication causes two negative effects relative to conventional protocol performance: (1) internal: the latency a requestor sees from request to access is greater and (2) external: contending requestors see a greater latency while waiting for a current request to complete. We consider the internal effect here and the external effect in the next section.

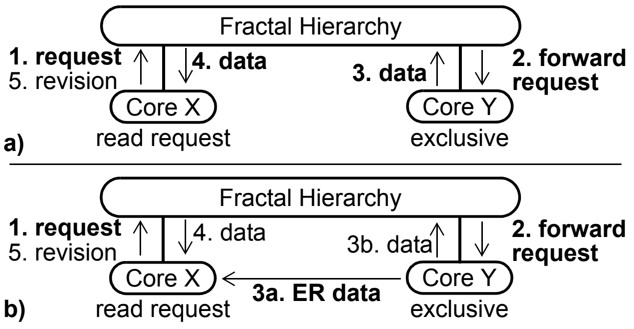

Figure 3. Read-write sharing in fractal (a) without and (b) with decoupled-replies; actions in bold are in the critical path

Figure 3a shows the latency incurred in a fractal protocol by a core *X*'s read request for a block that is exclusive at core *Y*. The fractal hierarchy (which may be physical as in TreeFractal or logical only as in FlatFractal) receives the request (hop 1) and forwards it to *Y* (hop 2). *Y* sends its data reply to the hierarchy (hop 3) which forwards it to *X* (hop 4). Recall that hop 5, the revision message (Section 2.1), is off the critical access path and does not add to the internal latency. Conventional directory protocols use reply-forwarding to replace hops 3 and 4 with a single direct hop from *Y* to *X*, reducing the internal latency from four hops to three; this reduction is especially beneficial in longer-latency multi-socket systems. Unfortunately, reply-forwarding violates observational equivalence as explained in Section 2.1. To obtain reply-forwarding's low internal latency without sacrificing verifiability, we make the key observation that observational equivalence limits only externally-visible actions but not internal actions. Accordingly, we propose *decoupled-replies* (DRs) to decouple data and acknowledgment receipt from the fractal protocol's externally-visible actions. To this end, DRs employ two reply messages: a non-fractal direct early-reply (ER) which allows cores to proceed with execution and a fractal-path confirmation. To retain verifiability, the ER enables only internal actions for the affected block (e.g., cache hits); externally-visible actions are delayed until the slower, fractal-path confirmation reply is received.

Figure 3b shows core *X*'s internal latency using DRs. Hops 1 and 2 remain the same as before. During hop 3 however, core *Y* sends an ER directly to *X* (hop 3a) in parallel with a fractal confirmation (hops 3b and 4). After just three hops, *X* receives the ER and proceeds with its access. The fractal protocol completes the access as normal after forwarding the fractal confirmation to *X*. DRs for invalidation acknowledgments are handled similarly, but the requestor waits to receive all ER acknowledgments before proceeding.

By decoupling data and acknowledgment receipt from the visible fractal protocol behavior, DRs effectively reduce internal latency. If not carefully handled, however, DRs can violate the protocol's fractal property, affecting both correctness and verifiability. For example, if a cache sent an ER with data and then wrote the data before sending the fractal confirmation, the requestor would consume incorrect data from the ER. To avoid such issues, ERs should not cause any divergence from the fractal protocol's behavior. To this end, we stipulate that ERs (1) be non-speculative: the fractal confirmation should confirm the ER without triggering any corrective action; and (2) cause no externally-visible effects: locally consuming an ER should not cause new externally-visible protocol interactions.

For the first requirement, we ensure that even though ERs are not fractal, they do not influence the fractal protocol's behavior. For example, in Figure 3b, because the hierarchy correctly serializes and only then forwards *X*'s request to *Y*, *Y*'s fractal reply is non-speculative. Similarly the ER is also non-speculative and *X* can safely consume the ER before the fractal confirmation arrives.

For the second requirement, that consuming ERs cause no new interactions, we ensure that caches conceal the receipt of an ER. This key issue arises because DRs create a time window between the receipt of an ER (when the access completes) and its fractal confirmation (when the fractal protocol completes the request). Revealing the ER receipt during this window may cause new interactions that violate observational equivalence and correctness. For example, in the fractal protocol with coupled replies, caches do not issue upgrades until the reply arrives and the request is complete. Therefore, if a cache issues an upgrade request after receiving the ER but before the fractal confirmation arrives, the upgrade would not only violate observational equivalence but also may cause errors as the protocol would not have transitions to handle this event. To prevent such violations, we disallow externally-visible events that would not have been possible without ERs until the window closes. For locally-generated events such as upgrades or replacements, we stall until the fractal confirmation arrives. Because the confirmation is guaranteed to arrive (i.e., the protocol is deadlock-free), stalling does not cause deadlocks. Further, our results show that such stalling is infrequent and does not impact performance. For our benchmarks, on average just 16% of read misses that use DRs (i.e., are misses to read-write shared blocks) stall due to an attempted upgrade within the DR window. Likewise, if a cache receives externally-generated events such as invalidations or forwarded requests, the cache must handle the events as if the ER had not been consumed, as done in the original fractal protocol which is deadlock-free (e.g., nack the message to be retried or queue the message and handle when the fractal reply arrives). By adhering to the fractal protocol's behavior in all cases, DRs avoid diverging from the fractal protocol. We track the DR window via an extra state in the cache's local protocol. The extra state is not visible to the fractal protocol and hence does not affect observational equivalence. Finally, while the ER frequently arrives before the fractal confirmation, the ER may occasionally arrive later. In this case, the access can complete when the fractal confirmation is received but, to ensure that the ER is expected, the DR window stays open until the ER arrives. Again, because the protocol is deadlock-free, ERs are guaranteed to arrive and the window eventually closes.

We now address two details of DRs. First, requestors must know (1) to expect ERs because a reply without an ER may come from the directory (no reply-forwarding case) or an ER and a confirmation may come from another cache (reply-forwarding case); and (2) in the case of invalidation acknowledgments, how many to expect. Second, DR's two replies increase bandwidth usage. To ensure that requestors expect ERs, responding caches set a flag in both the ER and confirmation replies. Including the flag in both replies ensures that the requesting cache knows whether to expect ERs regardless of which response arrives first. Invalidation acknowledgments additionally require a count indicating how many ER acknowledgments to expect (as in conventional protocols). A minor issue arises because the logical hierarchy does not centrally track the number of sharing caches; each interface tracks only its child sub-systems. Therefore, in a hierarchical architecture like TreeFractal, obtaining the count is difficult (i.e., no central location to look up sharers). Fortunately, FlatFractal's directory encapsulates all the interfaces and so records all the sharers in a central location. Thus while the logical protocol allows DRs for acknowledgments, the architecture affects how easily the sharer count can be acquired.

The second detail is DR's increased bandwidth usage. Because the extra confirmations are small control messages, our experiments show that DRs incur a small 4% bandwidth overhead compared to a directory protocol.

Finally, while we formally verify that DRs do not diverge from the fractal protocol's behavior later in Section 5.1,we clarify a correctness issue here. The DR window is unlike that used to enforce correct ordering for out-of-order loads and allowing a requestor to proceed upon ER receipt does not violate the memory consistency model; out-of-order loads make requests out of order but DRs affect only replies and ERs are sent only after the request is properly serialized.

3.2 Contention-hints

DRs allow requestors to avoid waiting for the fractal protocol's slower indirect-communication. However, contending requests from other threads must still wait for the fractal protocol to complete. We refer to this latency seen by other requestors as the *external latency effect* of indirect communication. As many contending requests may wait for each other, fractal's greater external latency adds up much faster than the latency in conventional protocols, and long latencies in multi-socket systems worsen the effect. In a single-socket system, Raytrace for example, worsens by 9% due to fractal's higher external latency, even though external latency affects only contending requests. To reduce this overhead, we propose our second optimization, *contention-hints*.

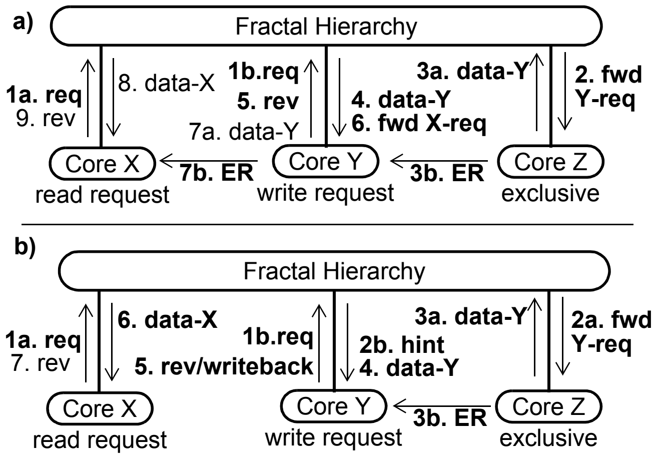

Figure 4. Read-write sharing in fractal (a) without and (b) with contention-hints; actions in bold are in core X's critical path.

Contention-hints are best-effort coherence requests that notify current accessors of contending requestors so that the accessors write back the contended block when their access completes (i.e., upon a reply from the fractal hierarchy). This writeback allows a contender to incur a faster shared LLC hit instead of a remote miss, reducing total latency (external plus internal). Figure 4a shows fractal without contention-hints when core *X* issues a read and core *Y* issues a contending write for a block that core *Z* owns (steps 1a, b). As *Y*'s write reaches the hierarchy first, *X*'s read waits for *Y*'s request to complete. The waiting may be via nacking, queuing at the directory, or forwarding to the owner. As described in previous sections, the hierarchy handles *Y*'s request (steps 2-5) while *X* waits. Finally, on receiving *Y*'s revision at step 5, the hierarchy forwards *X*'s read to *Y* (step 6). *Y* replies to the hierarchy (step 7a) and sends an early-reply to *X* (step 7b). Thus from request to access, *X* incurs a total of 7 hops. In contrast, conventional directory protocols eliminate steps 3a and 4 (3b remains), and incur only six hops.

Figure 4b shows the same scenario with contention-hints. Steps 1-4 remain the identical to Figure 4a except that on receiving *X*'s request, the hierarchy sends a contention-hint to the current accessor, core *Y* (step 2b). Because of the hint, when *Y*'s request completes after step 4, *Y* not only sends a revision to the hierarchy but also writes back the block (step 5). Importantly, the fractal protocol allows *X* to write back a block at any time after its access completes and so the hints do not cause any behavior disallowed by the protocol. Further, contention-hints do not use direct communication between sub-systems and therefore, unlike DRs, do not violate observational equivalence and can be a part of the fractal protocol. Thus, *X*'s read incurs an LLC hit and the hierarchy immediately returns data to *X* (step 6). Thus for contended misses, the hints enable fractal protocols to achieve miss latencies similar to conventional protocols'. We note that the hints can also help conventional protocols though the hints are more effective on fractal protocols' longer latencies.

Three details of contention-hints are: when to send the hints, their effect on accessor misses, and their bandwidth overhead. Figure 4b demonstrates contention-hints when a read waits for a current write. However, in total, three contention cases exist: a read waiting for a write, a write waiting for a write, and a write waiting for a read (the fourth case, two reads, is not a conflict). In the first two cases the current and only accessor is a writer. As such, once the writer writes back the block, an ensuing request is guaranteed a shared LLC hit. In the third case however, where a write waits for a current read access, other readers may also exist. Therefore, even if the current accessor self-invalidates, a writer may still need to invalidate the other sharers. As such, we send contention-hints to current write accessors but not read accessors. Differentiating one reader versus multiple readers would add complexity and did not significantly improve performance in our experiments because while writes delay both readers and writers, reads cause only writers to wait.

The second detail, that contention-hints increase misses, arises because the hints encourage immediate replacement of contended blocks. Such blocks are often accessed multiple times by the accessor, so replacing too quickly can increase misses. To avoid this problem, we (1) use contention-hints only on accesses that also use DRs so the accessor can exploit the DR window to hold on to the block and (2) delay writeback for a period of *hint-wait* cycles (including the DR window) to prevent the accessor from replacing too quickly.

To reduce the bandwidth overhead, the third detail, we send a single contention-hint to a current accessor, even if many contending requests are seen. A flag in the fractal hierarchy tracks whether a hint has been sent. Thus, we replace a single contending requestor's forwarded request with a single contention-hint to avoid bandwidth overhead.

Both contention-hints and dynamic self-invalidations (DSI) [16] invalidate blocks to reduce request latency. However DSI targets all shared blocks and attempts to predict when a block should be replaced to (1) avoid incoming invalidations and (2) avoid replacing before the core finishes accessing the block. Such prediction is hard. In contrast, contention-hints apply only to accesses where contention actually exists and therefore do not predict when to invalidate a block. Unless the block is released quickly, the contending request would soon invalidate the block anyway.

Finally, while we formally verify Fractal++, including contention-hints, later in Section 5.1, we discuss two correctness issues here. First, in Figure 4, the contention-hint arrives before *Y*'s request completes, but the reverse may also occur (e.g., a contending request may arrive late or the hint may be delayed in the network). In this case, the accessor simply drops the hint (i.e., the hints are best-effort). At worst, the contending request will incur a longer latency remote miss. Further, in the rare case that an accessor receives an unnecessary contention-hint (e.g., the contending requestor was

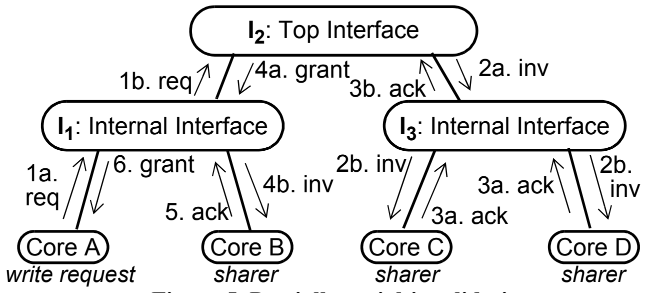

Figure 5. Partially-serial-invalidations

nacked and has not retried yet), the accessor will, at worst, incur a needless miss upon another access. Thus, the hints do not violate the fractal protocol's correctness. The second correctness issue is that, like reply-forwarding, the hints create a request-request dependence (i.e., contending requests trigger hint requests) and therefore can incur network deadlock (Section 1). Fortunately, unlike reply-forwarding where the directory must forward requests, the hints are best-effort and can be dropped if the outgoing network queue is full.

3.3 Fully-parallel-fractal-invalidations

Recall from Section 1 that in addition to the indirect-communication limitation addressed in the previous sections, fractal protocols fundamentally must invalidate sharers in a specific, partially serial order. These partially-serial-invalidations arise because fractal protocols enforce observational equivalence at each interface in the hierarchy so that all interfaces invalidate their non-requesting sub-system's sharers (external sharers) before granting write permission to their requesting sub-system (Section 2.1). Only on receiving this grant can the requesting sub-system invalidate its own internal sharers. Figure 5 shows the logical protocol's actions when core *A* issues a write for a block shared by cores *B, C,* and *D*. In the figure, parallel actions share a number and letter (e.g., both 2b); actions that occur during a single traversal up or down the tree (and hence, in a flattened architecture require a single hop) share a number but not letter (e.g., 2a and 2b). On receiving *A*'s request (steps 1a, 1b), I_2 invalidates the right sub-system, including external sharers *C* and *D* (steps 2a, 2b). Importantly, to preserve observational equivalence with caches, which send a single acknowledgment to their parent, I_3 waits for both *C*'s and *D*'s acknowledgments before sending a single acknowledgment to I_2 (steps 3a, 3b). I_2 then grants write permission to the requesting (left) sub-system (step 4a) which invalidates its internal sharer *B* (steps 4b and 5). Finally, I_1 grants *A*'s request (step 6). In contrast, conventional protocols use just four steps to receive *A*'s request, invalidate *B*, *C*, and *D* in parallel, and grant *A*'s request. For simplicity, in this example we do not consider reply-forwarding or decoupled-replies which reduce latency in conventional and fractal protocols, respectively, by one hop (Section 3.1). Thus, fractal's fundamental external-before-internal serial invalidation order significantly reduces performance (i.e., 7 steps up and down fractal's hierarchy versus 4 to and from the directory in conventional) and long latencies in multi-socket systems exacerbate the loss.

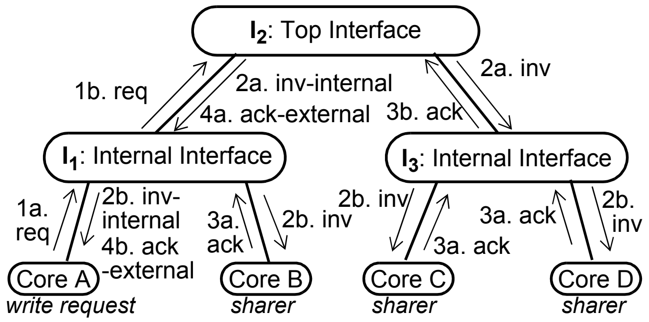

Figure 6. Fully-parallel-fractal-invalidations

To alleviate the performance problem, we propose *fully-parallel-fractal-invalidation* messages to enable parallel invalidation without violating the fractal property. We make the key observation that while acknowledgment collection must be in the fractal order, invalidations can be sent in any order. To achieve this decoupling, we observe that in existing fractal protocols, the message that grants write permission to a requesting sub-system (i.e. step 4a in Figure 5) serves two distinct functions: (1) to notify a sub-system that its request is being handled and it should invalidate its internal sharers and (2) to notify the sub-system that all external sharers have returned their acknowledgments. If instead, a sub-system were notified that its request was being handled when the upper interfaces began invalidating the external sharers, the sub-system could invalidate its internal sharers in parallel. To this end, we change the fractal protocol to separate the two functions into two independent coherence messages, *invalidate-internal* and *acknowledge-external*, without losing observational equivalence.

Figure 6 shows the above example with fully-parallel-fractal-invalidations. As before, when I_2 receives *A*'s request (step 1a, 1b), I_2 invalidates I_3 which includes external sharers *C* and *D*. However, rather than wait for I_3's acknowledgment, I_2 issues an *invalidate-internal* in parallel to I_1 (step 2a). This message allows I_1 to invalidate any internal sharers. To preserve observational equivalence, I_1 takes the same actions as I_2 and sends an invalidation to *B* and an invalidate-internal message to *A* in parallel (step 2b). Because *A* has no children, *A* simply ignores the message. Thus, invalidate-internal enables parallel invalidations to *B, C,* and *D*.

Because invalidations are now distinct from acknowledgments, I_1 requires an additional notification when I_2 receives the external acknowledgments. As in the original fractal protocol, to preserve observational equivalence with caches, sub-systems may send only a single acknowledgment message to their parents. Therefore, Figure 6 shows I_3 receiving an acknowledgment from both *C* and *D* before acknowledging I_2 (step 3a and 3b). As such, while invalidations can be sent in parallel without violating observational equivalence, acknowledgments must be collected serially. I_2 notifies its

Table 1. Hardware parameters

Cores	32, in-order
L1+L2 Caches	Split I&D, Private, 32KB 4-way set associative + 256KB 8-way set associative, write-back, 64B block, LRU replacement, 2 cycles
L3 Cache	Unified, Shared, 64 MB 32-way set associative, write-back, 32 banks, LRU replacement, 10 cycles
Network	Mesh, 5 cycles for router+link
Coherence	MOSI full bit-vector directory at L3, 10 cycles
Memory	16 GB, 200 cycles
Multi-socket	32-core; 8 cores/chip; 75-cycle inter-chip latency
Fractal++	*hint-wait*=70 cycles (Section 3.2)

requesting sub-system of the acknowledgments' receipt using *acknowledge-external*. When I_1 receives all the acknowledgments for sharers external to its own requesting sub-system, namely I_2's acknowledge-external and *B*'s acknowledgment, I_1 sends an acknowledge-external to *A* and the write completes. Importantly, while acknowledgments are collected serially, the parallel invalidations enable *B*, *C*, and *D* to send their acknowledgments in parallel (step 3a). Thus I_1 likely will already have *B*'s acknowledgment when I_2's acknowledge-external arrives and can immediately notify *A* (step 4). Thus, acknowledgment collection is fast and can be serial, and the far-slower sending of invalidations and acknowledgments each occurs in parallel. Consequently, the number of traversals up and down the hierarchy (four) is equivalent to a conventional protocol's hops. Like contention-hints, fully-parallel-fractal-invalidations uphold observational equivalence and are part of the fractal protocol.

4. Methodology

We simulate Fractal++ using the FlatFractal architecture and compare to a high-performance directory protocol. We use full-system simulation with Wisconsin GEMS-2.1 [18] and Simics [17] to simulate a SPARC-based multicore running Solaris 10. Using the hardware parameters in Table 1, we simulate both a 32-core single socket system and a 4-socket system with 8 cores per socket. Table 2 describes our benchmarks: three commercial workloads and six scientific applications from Splash-2 [31]. To account for statistical variations, we use randomly-perturbed runs to achieve 95% confidence [4] for at most a 2% variation on the mean.

5. Experimental Results

We verify Fractal++ and then compare performance of Fractal++, FlatFractal, and a conventional directory protocol. Next, we analyze Fractal++'s performance scalability. Finally, we evaluate each of Fractal++'s three optimizations.

5.1 Verification

As described in Section 2.3, we verify Fractal++, which includes decoupled-replies (Section 3.1), contention-hints (Section 3.2) and fully-parallel-fractal-invalidations (Section 3.3) via two automated steps: (1) using Murphi to verify correctness for a minimal complete system and (2) using CADP to verify that observational equivalence holds among sub-systems. For both steps, Fractal++ uses the same 3-core minimal system as TreeFractal (Section 2.3).

Table 2. Benchmarks

Commercial (com)
Apache 2.2.11 [28] and SURGE v1.3 [6] with http 1.1 capability; 20,000 file repository (~500MB); 3200 clients; 25ms think time; warm up for 1,500,000 transactions; measure 2000 transactions.
Online Transaction Processing (OLTP) PostgreSQL v9.2.0 [2] and Open Source Development Labs Test Suite DBT-2 v0.40[1] modeling TPC-C-like specifications; 3GB database; 20 warehouses; 512 users with 0 think time; warm up for 50,000 transactions; measure 200 transactions.
SPECjbb2005 v1.07[3] with Sun J2SE v1.5.0 JVM; 1.5 warehouses per processor with 0 think time; warm up for 350,000 transactions; measure for 10,000 transactions.
Scientific (sci)
SPLASH-2: We run *barnes, raytrace*, *ocean*, *water-spatial, volrend,* and *water-n^2* from SPLASH-2[31] using default parameters. For *raytrace*, we render the "car" object provided by SPLASH-2.

While decoupled-replies (DRs) are non-fractal and require extra care, contention-hints and fully-parallel-fractal-invalidations are part of a new fractal protocol and are verified via the above steps. For DRs, we verify in the first and second steps, respectively, the stipulations in Section 3.1 that DRs are non-speculative and have no externally-visible effects. We add invariants to the first step to verify that when a reply is issued by a cache, the associated request has been serialized. Some of the vital invariants are (1) that when a cache issues a reply, the requestor is guaranteed to receive that reply, (2) that when a cache issues an acknowledgment, no new sharers can join until after the write completes, and (3) that the confirmation allows the same internal behavior (e.g., read or write hit) as the ER allows. In the second step, we additionally verify that the new fractal protocol without DRs is observationally equivalent to the protocol with DRs.

5.2 Performance

We compare the performance of Fractal++ and the FlatFractal protocol, both using the best previous architecture, the FlatFractal architecture, normalized to that of a conventional directory protocol (Directory). Recall that FlatFractal avoids TreeFractal's architectural overhead, but still incurs the fundamental protocol limitations of indirect-communication and partially-serial-invalidations that affect *all* fractal protocols (Section 2.2). Comparing FlatFractal to Directory isolates the impact of these limitations on performance. We also show *FlatFractal-no-limits*, a protocol where we artificially remove from FlatFractal the limitations of indirect-communication and partially-serial-invalidations. While *FlatFractal-no-limits* is not fractal (and not verifiable), this protocol shows whether performance factors other than the limita-

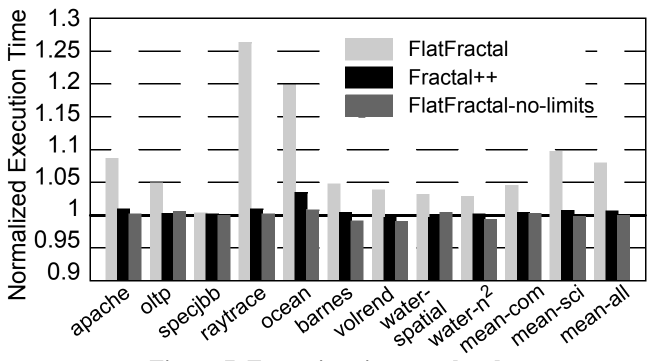

Figure 7. Execution time overhead

tions exist as compared to Directory. Figure 7 shows normalized execution times along the Y-axis (lower is better) of a 32-core single-socket system for each of our benchmarks (X-axis). The commercial and scientific benchmarks behave differently, so on the right we show means for each class separately (*mean-com* and *mean-sci*, respectively) as well as the overall mean (*mean-all*).

The fractal limitations degrade FlatFractal's performance by 8% compared to Directory. Because the limitations hurt read-write sharing (Section 1), *apache*, *raytrace*, and *ocean*, which have significant read-write sharing, incur much greater degradations of 9%, 26%, and 20%, respectively. Further, all benchmarks except one (*specjbb*) lose some performance, indicating the prevalence of read-write sharing.

In comparison, Fractal++ addresses the protocol limitations and therefore performs nearly as well as Directory. Fractal++ lags in performance by 3% for only one benchmark, *ocean*, because contending readers often arrive late, delaying contention-hints. Thus, unlike either FlatFractal (which degrades performance) or directory protocols (which are difficult to verify), Fractal++ achieves both performance and verifiability. Finally, *FlatFractal-no-limits* performs as well as Directory, confirming that the protocol limitations alone cause FlatFractal's performance degradation.

To explain the above results, Table 3 shows the average total, read, and write miss latencies for FlatFractal and Fractal++ normalized to Directory's total, read, and write latencies, respectively. We omit FlatFractal-no-limits because its latencies match Directory's. As expected from its performance degradation, FlatFractal's miss latencies are higher than Directory's (columns *A*, *C*, and *E*). Because partially-serial-invalidations, and to a lesser degree, indirect-communication, disproportionately delay write requests, FlatFractal's write misses (column *E*) generally incur greater degradation than its read misses (column *C*). Still, the average read miss latency (column *C*) worsens because indirect-communication affects both reads and writes. Fractal++ in comparison, uses decoupled-replies and contention-hints to achieve a similar average read latency to Directory (column *D*). As mentioned in Section 3.1, stalling upgrades within the DR window does not degrade Fractal++'s read miss latency

Table 3. Miss latencies normalized to directory

Benchmarks	Total Miss Latency		Read Miss Latency		Write Miss Latency	
	(A) Flat-Fractal	(B) Fractal++	(C) Flat-Fractal	(D) Fractal++	(E) Flat-Fractal	(F) Fractal++
apache	1.09	1.01	1.12	1.00	1.07	1.02
oltp	1.06	1.01	1.04	1.00	1.09	1.03
specjbb	1.00	1.00	1.00	1.00	1.01	1.00
raytrace	1.35	0.99	1.33	1.00	1.33	0.98
ocean	1.28	1.01	1.35	1.01	1.18	1.02
barnes	1.07	1.01	1.05	1.00	1.58	1.17
volrend	1.28	1.04	1.25	1.02	1.38	1.12
water-spat	1.03	1.00	1.02	1.00	1.31	1.07
water-n^2	1.21	1.03	1.19	1.00	1.24	1.09

because on average (over all benchmarks), just 9% of misses are read misses that use DRs, and 16% of such misses incur upgrades. Along with fully-parallel-fractal-invalidations, these optimizations also improve Fractal++'s store latency (column *F*) relative to FlatFractal (column *E*). However, in some cases (e.g., *barnes* and *volrend)*, Fractal++'s store latency lags compared to Directory's because decoupled-replies and contention-hints approach but do not equal the latency of reply-forwarding (e.g., contending requestors may arrive late causing late contention-hints). Overall, however, Fractal++ achieves an average total miss latency nearly equal to Directory's (column *B*), matching the performance shown in Figure 7. *Barnes* and *volrend* have lower write miss rates, so the benchmarks' higher write miss latencies do not degrade total miss latencies.

5.2.1 Multi-socket Performance

As mentioned in Section 1, compared to single-socket, multi-socket systems incur long latencies for indirection and invalidations. As multi-socket systems are increasingly common, we evaluate Fractal++'s benefits for these systems. Figure 8 shows execution time along the Y-axis for a 32-core system with 8 cores per socket (Table 1) for Fractal++ and the FlatFractal protocol normalized to Directory's execution time (lower is better). Benchmarks are shown along the Y-axis. Fractal's fundamental limitations, indirect-communication and partially-serial-invalidations, hurt FlatFractal's performance by 12% on average. In the benchmarks with more read-write sharing, *apache*, *raytrace*, and *ocean*, FlatFractal's performance lags by 17%, 34%, and 23%, respectively. Even *OLTP* and *volrend*, which have moderate sharing, show significant losses of 10% and 8%, respectively. Because multi-socket systems' latencies are longer than single-socket systems' (5-cycle on-chip network hop versus 75-cycle inter-chip delay in Table 1), these losses exceed those for the single-socket system in Section 5.2. In contrast, Frac-

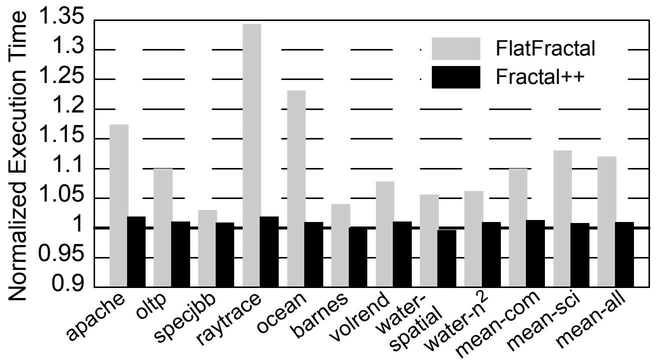

Figure 8. Multi-socket execution time overhead

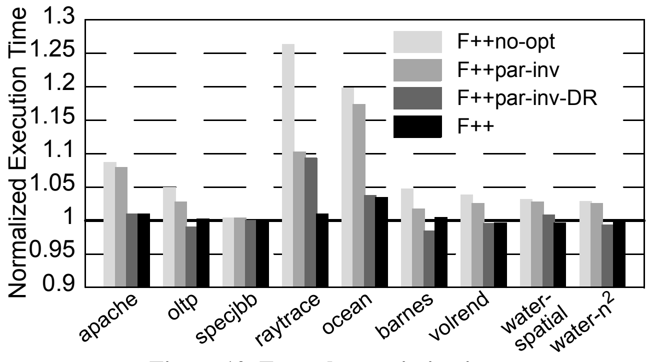

Figure 10. Fractal++ optimizations

tal++'s decoupled-replies and contention-hints reduce indirection overhead and fully-parallel-fractal-invalidations reduce the invalidation overhead so that Fractal++ performs nearly as well as Directory. Thus Fractal++ attains good performance even in higher-latency systems.

5.3 Performance Scalability with More Cores

In this section we compare the performance scalability of Fractal++, FlatFractal, and Directory. Good performance scalability is essential as multicores' core counts continue to grow. Figure 9 shows single-socket execution times for 16-core and 32-core FlatFractal and Fractal++ normalized to 16-core and 32-core Directory, respectively, along the Y-axis (lower is better). Benchmarks are shown along the X-axis. FlatFractal's performance degradation increases with core count, demonstrating its worse performance scalability. For one benchmark, *raytrace*, the trend reverses: 16-core FlatFractal performs worse than 32-core FlatFractal compared to their respective base cases. *Raytrace* has a heavily-contended work queue and the low cache-to-cache network latencies at 16 cores increase the number of caches misses on contended blocks. In contrast, Fractal++ performs nearly as well as Directory as the system scales, indicating that Fractal++ is as scalable in performance as directory protocols.

5.4 Impact of Optimizations

We now assess the individual performance impacts of Fractal++'s optimizations: decoupled-replies, contention-hints, and fully-parallel-fractal-invalidations. Because the optimizations each affect an independent aspect of the fractal protocol, their impacts are additive. Figure 10 shows Fractal++'s execution time normalized to Directory's for a 32-core single-socket system along the Y-axis as we add each optimization. From left to right, we show Fractal++ without optimizations (*F++no-opt*) which is the same as FlatFractal, Fractal++ with fully-parallel-fractal-invalidations (*F++par-inv*), Fractal++ with both fully-parallel-fractal-invalidations and decoupled-replies (*F++par-inv-DR*), and full Fractal++ including contention-hints (*F++*). The X-axis shows the benchmarks. Different optimizations benefit different benchmarks, indicating that both fundamental fractal limitations, partially-serial-invalidations and indirect-communication, degrade performance. *OLTP*, *raytrace*, *ocean*, and *barnes* incur partially-serial-invalidations and thus improve with fully-parallel-fractal-invalidations. DRs benefit *apache*, *oltp*, *ocean*, *volrend*, *water-spatial*, and *water-n^2* which incur indirect-communication for read-write shared misses. Contention-hints are especially useful in *raytrace* where many contending requestors wait for others' accesses to complete.

Finally, as noted in Section 3.2, directory protocols may also use contention-hints. Figure 11 shows execution time for Directory with contention-hints normalized to time without contention-hints (X-axis) for each benchmark (Y-axis). Because Directory's latencies without hints are already low, contention-hints do not improve performance. In *raytrace*, contention-hints cause the blocks to be written back too quickly and increase misses. Therefore, using directory without contention-hints as the baseline is valid.

5.5 Fractal++ Overheads

Recall that Fractal++ incurs bandwidth overhead from decoupling responses for decoupled-replies (Section 3.1) and from dividing fractal protocols' single request-grant

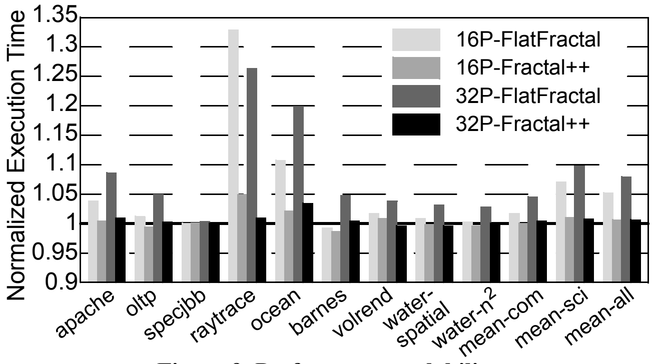

Figure 9. Performance scalability

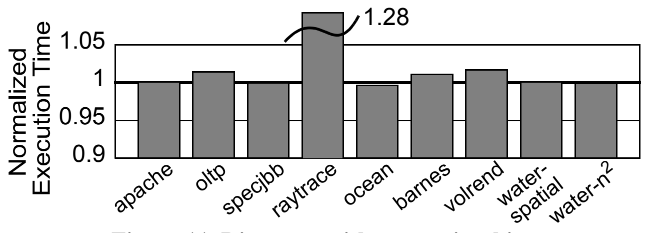

Figure 11. Directory with contention-hints

Table 4. Bandwidth normalized to Directory

Benchmarks	(A) FlatFractal (bytes*links/miss)	(B) Fractal++ (bytes*links/miss)
apache	1.11	1.03
oltp	1.08	1.02
specjbb	1.01	1.00
raytrace	1.34	1.10
ocean	1.21	1.06
barnes	1.06	1.02
volrend	1.17	1.05
water-spatial	1.08	1.02
water-n^2	1.27	1.08

message into two separate messages for fully-parallel-fractal-invalidations (Section 3.3). Contention-hints replace one request message with another and do not incur overhead (Section 3.2). Table 4 shows bandwidth, measured as the product of bytes and network hops per miss, for Fractal++, FlatFractal, and Directory, normalized to Directory's bandwidth (all 32-core single-socket systems). As expected, the lack of reply-forwarding increases FlatFractal's bandwidth overhead relative to Directory (column *A*). Fractal++ reduces overhead relative to FlatFractal because (1) DRs allow the larger data reply to travel directly between caches while only a control message reply incurs the longer indirect path and (2) fully-parallel-fractal-invalidations add just one control message per shared write request. However, decoupled-replies' extra messages do cause Fractal++ to incur a small overhead relative to Directory (column *B*).

6. Related Work

Coherence verification methods fall into three major classes: state space exploration, model-checking, and semi-automated theorem proving. State exploration checks each of a protocol's reachable states for correctness (e.g., no co-existing reader and writer caches; no livelock or deadlock) [24]. However, state space grows exponentially with system size (e.g., caches); hence this approach becomes impractical from time and memory standpoints at fairly small scales (e.g., 8 cores). While some schemes exploit protocol symmetries to reduce the state space [14], such schemes only delay the inevitable state explosion.

The second class, model-checking [22,25,27], uses abstractions to verify protocol properties and is more powerful than state space exploration. Parameterized model-checking [19,27] uses scale-independent abstractions to achieve scalable verification. While model-checking has been successfully applied to some well-known protocols (e.g., Futurebus [29]), model-checking incurs three drawbacks that prevent its widespread use. First, many models rely on protocol interactions and thus can incur state explosion. Second, finding appropriate abstractions for a given protocol requires considerable human effort. Third, the use of abstractions raises the possibility that the verified model and protocol differ. Thus model-checking does not guarantee scalable verifiability in general. Finally, the third class, semi-automated theorem proving [21], requires even more human effort than model-checking and is therefore less commonly used.

The above approaches try to verify existing protocols. In contrast, recent work designs protocols for scalable verifiability. PVCoherence [34] provides guidelines for designing protocols that can be verified using current parameterized model-checking approaches. While such guidelines increase the likelihood that a particular protocol is verifiable, PVCoherence does not guarantee that such a protocol is scalably verifiable. In contrast, fractal coherence [33] guarantees scalable verifiability. As detailed in Section 2, fractal protocols incur both architectural and fundamental protocol limitations which hurt performance. Our previous work, FlatFractal [30], addresses the architectural constraints (Section 2.2). However, all fractal protocols suffer from the fundamental protocol limitations addressed by Fractal++.

Finally, as noted in Section 3.2, while dynamic self-invalidations (DSI) [16] predict when to replace a block, contention-hints avoid this difficult prediction.

7. Conclusion

Coherence verification continues to be a challenging problem. A promising approach, fractal coherence, provides scalable verifiability by enforcing observational equivalence among sub-systems within the protocol (the fractal property). Unfortunately the fractal property imposes two fundamental limitations on protocols: *indirect-communication* and *partially-serial-invalidations*. For read-write sharing which is prevalent in commercial and scientific applications, these limitations hurt performance especially in multi-socket systems where indirection and invalidations involve long latencies. In contrast, conventional directory protocols improve performance via reply-forwarding and parallel invalidations, respectively, which both violate the fractal property.

To address these fundamental limitations, we proposed Fractal++ which uses novel performance optimizations for verifiability-constrained coherence protocols. These optimizations preserve fractal protocols' verification scalability while providing similar performance scalability to directory protocols. Specifically, we proposed *decoupled-replies* and *contention-hints* to address *indirect-communication* and *fully-parallel-fractal-invalidations* to address *partially-serial-invalidations*. Importantly, we showed that Fractal++ is scalably verifiable using the existing fractal coherence verification framework. In 32-core simulations of single- and four-socket systems, we showed that Fractal++ performs nearly as well as a directory protocol whereas the best-performing previous fractal protocol performs 8% on average and up to 26% worse with the single-socket and 12% on average and up to 34% worse with the longer-latency multi-socket system.

As coherence protocols grow in complexity and multicores include more components (e.g., caches, GPUs), performance and verifiability will continue to be increasingly important and increasingly difficult to obtain together, especially for multi-socket systems with long latencies. Fractal++ enables systems to achieve both performance (unlike previous fractal protocols) and verification (unlike conventional directory protocols) even as systems (and latencies) scale.

References

[1] Open source development labs database test suite 2 v0.40 http://osdldbt.sourceforge.net/.

[2] Postgresql. v9.2.0. http://www.postgresql.org/.

[3] The standard performance evaluation corporation. specJBB2005 suite. http://www.spec.org/jbb2005/.

[4] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[5] AMD. Revision Guide for AMD Family 14H Models 00h-0Fh Processors, revision 3.18. February 2013.

[6] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. Technical report, 1998.

[7] B. Bentley. Simulation Driven Verification. Presentation at *Design Automation Summer School*. 2009.

[8] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors,* pages 522 –525, 1992.

[9] A. Felty and F. Stomp. A correctness proof of a cache coherence protocol. In *Proceedings of the 11th Conference on Computer Assurance,* pages 128-141, 1996.

[10] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, 2011.

[11] S. Gjessing, S. Krogdahl, and E. Munthe-Kaas. A linked list cache coherence protocol: verifying the bottom layer. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 324-329, 1991.

[12] S. Gjessing, S. Krogdahl, and E. Munthe-Kaas. A top down approach to the formal specification of SCI cache coherence. In *Computer Aided Verification, 3rd International Workshop,* pages 83-91,1991. Springer.

[13] Intel. Intel Xeon Processor E7-800/4800/2800 Product Families Specification Update. August 2013.

[14] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, Aug. 1996.

[15] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, 1997.

[16] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture,* pages 48-59, 1995.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.

[18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, Nov. 2005.

[19] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 179-195. Springer, 2001.

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

[21] 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*, pages 288–296, 1996.

[22] F. Pong and M. Dubois. The verification of cache coherence protocols. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[23] 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,* volume 966 of *Lecture Notes in Computer Science*, pages 287-300, Springer, 1995.

[24] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[25] F. Pong and M. Dubois. Formal verification of complex coherence protocols using symbolic state models. *Journal of the ACM*, 45(4):557–587, July 1998.

[26] U. Stern and D. Dill. Automatic verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings,* pages 21-34, 1995.

[27] C. Tsun Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer Aided Design*, pages 382–398. Springer, 2004.

[28] Apache HTTP Server, v2.2.11. The apache software foundation. http://httpd.apache.org/.

[29] K. Williams and R. Esser. Verification of the Futurebus+ cache coherence protocol: a case study in model checking. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26*, pages 65–71, 2004. Australian Computer Society, Inc.

[30] G. Voskuilen and T. N. Vijaykumar. High-performance fractal coherence. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 701-714, 2014.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The Splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[32] P. Yeung and K. Larsen. Practical assertion-based formal verification for SoC designs. In *Proceedings of the 2005 International Symposium on System-on-Chip,* pages 58-61, 2005.

[33] M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd IEEE/ACM International Symposium on Microarchitecture*, pages 471–482, 2010.

[34] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin. PVCoherence: Designing flat coherence protocols for scalable verification. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, 2014.