Using In-Flight Chains to Build a Scalable Cache Coherence Protocol

SAMANTIKA SUBRAMANIAM and SIMON C. STEELY, Intel Corporation
WILL HASENPLAUGH, Intel Corporation and MIT
AAMER JALEEL, CARL BECKMANN, and TRYGGVE FOSSUM, Intel Corporation
JOEL EMER, Intel Corporation and MIT

As microprocessor designs integrate more cores, scalability of cache coherence protocols becomes a challenging problem. Most directory-based protocols avoid races by using blocking tag directories that can impact the performance of parallel applications. In this article, we first quantitatively demonstrate that state-of-the-art blocking protocols significantly constrain throughput at large core counts for several parallel applications. Nonblocking protocols address this throughput concern at the expense of scalability in the interconnection network or in the required resource overheads. To address this concern, we enhance nonblocking directory protocols by migrating the point of service of responses. Our approach uses in-flight chains of cores making parallel memory requests to incorporate scalability while maintaining high-throughput. The proposed cache coherence protocol called *chained cache coherence*, can outperform blocking protocols by up to 20% on scientific and 12% on commercial applications. It also has low resource overheads and simple address ordering requirements making it both a high-performance and scalable protocol. Furthermore, in-flight chains provide a scalable solution to building hierarchical and nonblocking tag directories as well as optimize communication latencies.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures; B.3.2 [**Hardware**]: Design Styles

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Cache coherence, tag directories, nonblocking, synchronization

ACM Reference Format:
Subramaniam, S., Steely, S. C., Hasenplaugh, W., Jaleel, A., Beckmann, C., Fossum, T., and Emer, J. 2013. Using in-flight chains to build a scalable cache coherence protocol. ACM Trans. Architec. Code Optim. 10, 4, Article 28 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2541228.2541235

1. INTRODUCTION

Many-core systems are attractive for the high-performance computing and scientific markets, as evidenced by the recently released Intel Xeon Phi [Jeffers 2012]. Managing application data coherence efficiently on a socket with many cores is an important requirement because even applications that are traditionally parallelized using message-passing are starting to use shared memory optimizations by placing multiple MPI tasks on a socket [Graham and Shipman 2008]. Maintaining coherence involves tracking shared address locations in ordering points such as tag-directories, as well as sending *probe* messages to forward data, and *invalidation messages* to eliminate stale copies of data. Thus, the performance and efficiency of coherence protocols is

Author's address: S. Subramaniam, MS: HD2-330, 77 Reed Road, Hudson, MA 01749.

DOI: http://dx.doi.org/10.1145/2541228.2541235

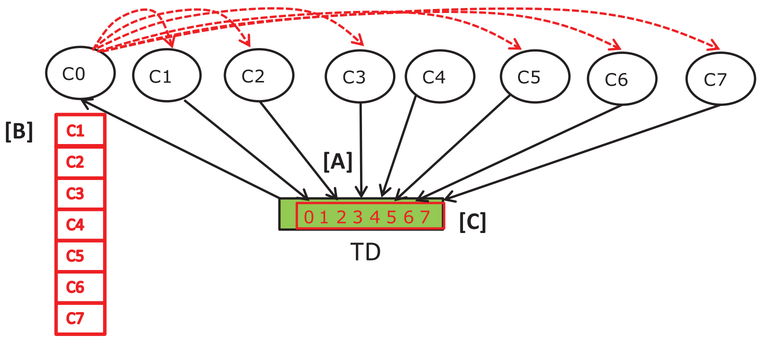

Fig. 1. Coherence scalability concerns in many-core systems.

intricately tied to the sharing in applications. High-performing coherence protocols can be implemented with low resource overheads in small systems. In many-core systems, however, the number of outstanding requests in the memory hierarchy increases, which increases contention in addresses. Applications designed for these systems also have large working sets resulting in long memory accesses. Furthermore, workloads that model the synchronization overheads in shared memory applications such as CLOMP demonstrate that the performance of even small but frequently accessed parallel regions can be a major factor in the overall system performance [Bronevetsky 2009]. Thus, naïively scaling the coherence mechanism can degrade performance and impact bandwidth [Gharachorloo et al. 2000].

Figure 1 shows the interactions of eight cores in a coherent system and highlights key bottlenecks. The figure shows seven cores (C1 through C7) requesting the same data residing in C0's cache. The tag directory (TD) processes these requests and sends probes to C0. The first scalability concern, marked as [A] in the figure, is to achieve high throughput when processing multiple requests to the same address at the TDs. Modern blocking protocols serialize request processing either by queuing requests at the TDs or by sending back-off messages to other conflicting requestors to guarantee correctness, which could hamper throughput and increase occupancy of the system [Chaiken et al. 1990; Marty and Hill 2008; Laudon and Lenoski 1997]. Conversely, protocols that remove this blocking feature either have network ordering requirements to handle protocol races that are difficult to implement or require a large number of resources and bandwidth at the requesting cores to keep up with the increased throughput leading to the second scalability challenge [Gharachorloo et al. 2000; Marty 2008]. The second scalability concern, marked as [B] in the figure, is to maintain low resource overheads at the cores, which implies that any single core should only receive a small and bounded number of probes for data. Coherence protocols that allow a core to receive probes from and provide data to all the other cores for an address might have to maintain a large number of buffers at the cores to hold these requests as well as issue multiple responses. Finally, the third scalability concern, marked as [C] in the figure, is in state storage at the TDs. As we add cores to the coherence domain, each TD entry needs to maintain more state to track these additional cores. If this state grows linearly with the number of cores, scalability of the system can be severely impacted. All three scalability challenges worsen as core counts increase as is the trend in most HPC systems.

Nonblocking protocols, which have been proposed in the past, preserve coherence while maintaining high throughput, by employing instant updates to the TD that take the system to the final state without making any intermediate states visible. While instant updates achieve concurrency of request processing and address the throughput scalability challenge at the TD, they do not reduce the resource and response overheads at the core, which could now receive multiple forwarded requests for data. In our work, we propose to improve the scalability of nonblocking directory protocols by enhancing them with the principle that the point of service of responses should migrate between the cores. Migration of response delivery is achieved by building an in-flight ordered chain of cores making parallel conflicting requests. This in-flight chain is temporary and does not require permanent pointers in the private caches, making it a scalable solution. Using properties of this temporary chain, we can guarantee that only one buffer is required at each core to hold a forwarded request as well as ensure that response delivery work is distributed among the cores. Instant updates augmented with in-flight chains addresses the first two scalability challenges described earlier and form the foundation of our novel cache coherence protocol called *Chained Cache Coherence* (CCC).

This article makes the following key contributions:

(1) We present the design and evaluation of Chained Cache Coherence or CCC, a novel, nonblocking, high-performance, and scalable directory-based protocol that only requires point-to-point address ordering of probes and can be implemented in modern networks.
(2) Using cycle-accurate simulations and data-sharing characterizations, this article presents, to the best of our knowledge, the first quantitative analysis to show that state-of-the-art blocking directory protocols, even those with migratory sharing optimizations significantly constrain performance for parallel applications.
(3) Finally, this article shows both quantitatively and by use of formal verification that the concept of in-flight chains can be directly applied to build nonblocking and race-free hierarchical TDs. This extension to CCC can further minimize the latency of communication and provide a scalable representation of state storage, thus addressing the third challenge. This extension also makes CCC, to the best of our knowledge, the first nonblocking and hierarchical directory protocol that addresses the key scalability challenges of throughput and resource overhead.

The rest of the article is as follows. Section 2 quantitatively presents the scalability challenges of modern blocking protocols and motivates the need for nonblocking TD. Section 3 describes the potential bottleneck of nonblocking TDs and describes the in-flight chains used in CCC that remove this bottleneck. This section also depicts how CCC can naturally handle all protocol races and its scalability advantages. Section 4 presents the design of hierarchical CCC, which uses the concept of in-flight chains to build a nonblocking, hierarchical, and race-free TD, thus addressing all three scalability challenges. We present the evaluation of CCC and hierarchical CCC in Section 5 and compare it to a highly optimized predictive protocol. We also present a sensitivity analysis for CCC that shows that it performs well when scaled to larger core counts as well as when used in a network with smaller queues. In Section 6, we compare CCC with nonblocking, nondirectory protocols, in particular the token coherence class of protocols and discuss the scalability benefits of CCC over these protocols. Finally, we contrast CCC with other protocols that have focused on improving scalability of large-scale systems.

2. MOTIVATION

In this section, we describe blocking directory protocols and demonstrate their potential scaling challenges. We then motivate the use of nonblocking protocols.

2.1. Challenges in Blocking Directory Protocols

Directory-based protocols can handle multiple outstanding requests to an address by ensuring that only a single transaction is processed at a time, while stalling other requests in the interim period [Hagersten and Koster 1999; Lenoski et al. 1992; Marty

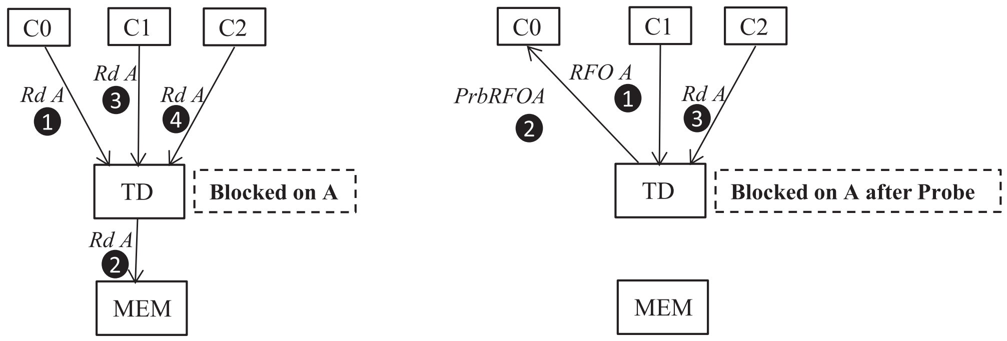

Fig. 2. (a) Blocking directory example showing requests from C1 and C2 being blocked. (b) Blocking directory example showing unnecessary blocking of request from C2.

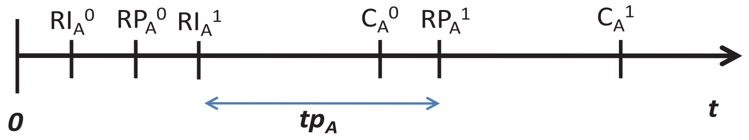

Fig. 3. Transaction timeline for a blocking protocol shows processing delay for request A from Core 1 (tp_A).

and Hill 2008]. Blocking protocols accomplish this by marking a TD entry for a particular address as being in use or blocked after being accessed by a request. Subsequent requests to the same address from different cores are held back from being processed or forced to back off and retry until the TD entry is unblocked. Unblocking a directory entry is done by sending a completion message to the TD. Blocking preserves the global order between conflicting memory requests since every request is completed and acknowledged by the TD before it processes another one.

Although blocking TDs ensure coherence, they can cause unnecessary delays in the processing of transactions and forgo some concurrency as shown in the following two examples. Figure 2(a) shows a request from core C0 that misses the cache and is sent out to memory. When this request accesses the TD on the path to memory, it blocks the TD entry until the response returns. This means that any other request to the same address, such as the read requests from cores C1 and C2, will get stopped at this ordering point in the interim period, ensuring that coherence is maintained, as shown in the figure. In Figure 2(b), a read-for-ownership request for address A (RFO A) from core C1 at the TD generates a probe to receive the data from core C0, which has a copy of A. At the same time, another core, C2, issues a read request for A (Rd A) that reaches the TD. At this point, the TD has the necessary information to generate a probe message to the C1 and service this request in the shadow of the previous request. However, it is blocked from doing so until it receives a completion message from C0 stating that the directory (and thus the system) is in a stable and coherent state.

2.1.1. Impact on Throughput and Occupancy. Consider Figure 3, which shows a timeline of transactions from two cores in a system-enforcing coherence through blocking. $RI_A{}^0$ is the time at which a request is issued for address A from Core 0, $RP_A{}^0$ is when the request for A from Core 0 is processed, and $C_A{}^0$ is when the response for A is sent to Core 0, thus completing the transaction. $RI_A{}^1$, $RP_A{}^1$, and $C_A{}^1$ are corresponding times for a request for address A issued from Core 1. The timeline shows that in a blocking protocol, $RP_A{}^1$ is blocked until $C_A{}^0$ is received leading to a transaction processing delay, tp_A, in the system. Although it may seem that individual transaction latencies might not be impacted by blocking, concurrent work for each transaction such as

Table I. Sharing Patterns in Typical Parallel Applications

Workload	Percentage of shared blocks	Percentage of accesses to shared blocks	Percentage of accesses to shared blocks (>2 threads)
BT	10	66	60
SP	8	38	31
LU	4	77	65
CG	30	22	20
MG	9	30	24
FT	3	17	15
IS	5	15	10

forwarding probes and collecting acknowledgements also get delayed and could impact the throughput of the application.

Stalled requests also increase occupancy (the time that a message occupies a queue slot) at the TDs because they have longer lifetimes in the network queues. High occupancy can impact the performance of the system because it creates contention in the queues as observed in prior studies [Chaudhuri et al. 2003]. Blocking protocols that send back-off messages to conflicting requestors instead of queuing, also generate more messages in the network queues, and in Section 5.5, we quantitatively show that nonblocking protocols are able to get high performance even with smaller queues.

2.1.2. Enhanced Blocking Protocols. While blocking has been the industry and academic standard to designing most directory-based protocols, there have been several proposals that use predictive sharing techniques to reduce the impact of the directory indirection and thus mitigate the impact of blocking [Kaxiras and Georgios 2010; Cheng et al. 2007; Martin et al. 2003a; Mukherjee and Hill 1998; Kaxiras and Goodman 1999; Cox et al. 1993]. These optimizations typically predict producer–consumer sharing patterns and convert 3-hop (requestor-directory-responder) coherence protocols into 2-hop (requestor-responder) protocols. They can cut the latency of directory accesses but require high accuracy to avoid costly mispredictions. Additionally, in several HPC and scientific applications, the datasets do not completely fit in the on-chip caches and require many memory accesses, which not only increases the blocking duration but also reduces the accuracy of these predictions. Consequently, these proposals have not been adopted in industry designs although they provide performance benefits over blocking directories. In this article, we show that CCC removes the throughput bottleneck at the TD itself in a nonspeculative manner, which can reduce the need for additional predictive performance-optimizing mechanisms. Furthermore, in Section 5.2, we demonstrate that CCC even outperforms one recently proposed predictive mechanism.

2.2. Data Sharing in Applications

The timeline in Figure 3 shows that blocking is a serious concern when there is address contention among threads. Intuitively, for a parallel application to be scalable, memory requests should be balanced across the memory blocks in the data structure. However, several parallel applications experience significant amount of read sharing even at small thread counts [Jaleel et al. 2006; Woo et al. 1995]. In this section, we characterize the sharing patterns of parallel applications using two studies. The first study focuses on the spatial component of data sharing and was done using published methodology [Jaleel et al. 2006]. Table I illustrates the data-sharing patterns observed when executing eight application threads of some common HPC applications from the NAS suite [Bailey 1994]. Table I depicts shared cache blocks that are currently resident

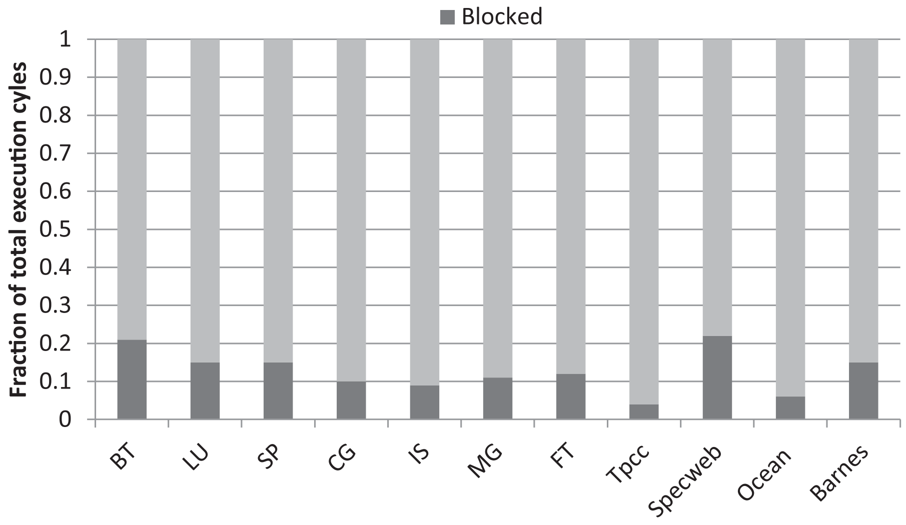

Fig. 4. Cycles/accesses stack showing fraction of execution cycles lost due to blocked requests.

in the cache and not blocks that have been shared in the past. Only read sharing is captured because this highlights the real benefit of nonblocking directories. The first column is the workload under consideration, whereas columns 2 through 4 present the sharing metrics. As expected, the second column shows that most applications have a small percentage of shared cache blocks. On the other hand, the third column shows that there is significant amount of accesses to these shared blocks. Finally, the fourth column shows that when there is sharing, it generally involves more than two threads. These results are not surprising because many of the NAS applications (BT, LU, and SP) exhibit nearest-neighbor communication patterns, whereas others have some producer–consumer sharing in phases (CG and MG).

The temporal component of data sharing is another factor contributing to the performance of a coherence protocol. Blocking can significantly impact throughput if the threads simultaneously access shared data. The second study, illustrated in Figure 4, depicts the temporal component of data sharing in HPC, scientific, and commercial applications by showing the fraction of execution cycles that are lost due to stalled requests at TDs. Similar to conventional CPI stacks, this data is collected from the time a request is issued into the memory hierarchy until it receives its response. We use a cycle-accurate simulator for this study the details of which are described in Section 5.1. As can be seen, in most applications, between 10% and 20% of the execution cycles are due to blocked TD stalls. These blocked cycles also correspond to the performance improvements that will be shown during the evaluation of the CCC protocol. These results show the prevalence of simultaneous data sharing in parallel applications. This phenomenon has also been observed in other works that explored global synchronization in parallel applications [Singh et al. 1992].

An interesting point to note is that large shared caches have often been used in small-scale systems to improve the performance of shared data. In sockets with large core counts ($\geq$64), however, these large banked shared caches are not considered attractive due to the increased effective access latency and thread contention [Morgan 2012]. Recently released many-core HPC machines also appear to be moving away from shared caches [Morgan 2012]. Although this article focuses on data sharing, there is significant code sharing as well, which will exacerbate the scaling challenges in protocols.

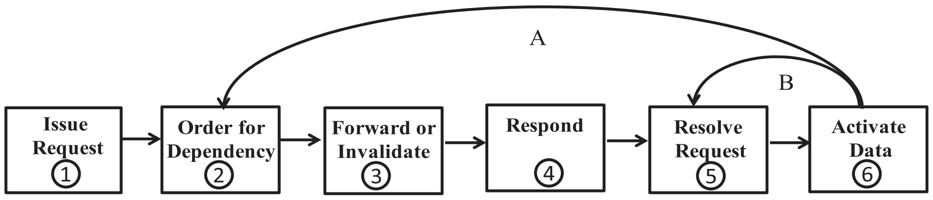

Fig. 5. Cache coherence pipeline.

2.3. Principle of Nonblocking Directory Protocols

Another approach to designing directory-based coherence protocols is to ensure that messages reach their destinations in the order that they were sent out from the TD. This approach does not block TD entries on requests, thus achieving high throughput at the TDs.

Consider Figure 5, which presents a general cache coherence mechanism as a series of pipelined operations that occur both at the core and in the network. In the figure, the TD determines an order between the requests that have been issued to it and sends out a probe to receive the data or invalidations in case the request is for ownership (RFO). The probed and invalidated cores, in turn, respond either with data or acknowledgements to ensure that no stale copies of data are present in the hierarchy. When the requesting core receives its responses, it resolves the request, which typically involves writing the cache and then activating the data for use.

When there are multiple simultaneous in-flight requests to the same address, a blocking TD chooses one request to order, which has to resolve and notify the TD that the system is in a globally coherent state before it can begin processing other requests. This leads to a stall in the system, as shown by arrow A. However, there is considerable parallelism in stages 3 and 4 that can occur in the shadow of processing the next request. If the coherence stall is moved from ordering to resolution and the TD is *instantly updated* such that all subsequent requests know which cores to probe or invalidate, a globally coherent view can be maintained even before a globally coherent state is reached. This design principle is used in the AlphaServer GS320, which uses nonblocking TDs. Consequently, a request only has to stall in stage 5, where the outstanding request has to have received the data and/or necessary acknowledgements reducing the system stall to arrow B [Gharachorloo et al. 2000].

2.4. Disadvantages of Prior Nonblocking Directory Protocols

Nonblocking TDs can maintain correct coherence by making instant updates to the TD entry with the requesting core information and sending the probe or invalidation messages to the appropriate cores as an atomic operation. An atomic update means that once the TD is updated and the probe or invalidation message is sent to a core, they have to be processed at the core without any intervening operations that might make the global state of the system inconsistent with the TD view. The AlphaServer paper describes in detail the potential deadlock that can occur in nonblocking TDs if this restriction is not maintained and the actual implementation of the GS320 used stringent total ordering guarantees across all addresses in the network channel carrying probes to avoid the deadlock. Implementing total ordering in a network requires a nonscalable, switch-style interconnect that is impractical in modern mesh or torus networks. Additionally, the requirement to maintain total order between different addresses can create further stalls in the message queues and lead to high message buffering at the core boundaries, both of which are scalability challenges. In Section 5.7, we present the number of additional stalls that are generated due to total ordering of probes in the network.

A proposed optimization hypothesized that the AlphaServer GS320 could have been implemented with just point-to-point ordering of probes and did not require total ordering across all the probe channels [Yongquin et al. 2009]. However, this hypothesis is not true in modern interconnection networks like a mesh or torus, with banked TDs that are distributed across a tiled network. Here, each TD bank independently orders requests that it receives, and deadlock can occur in a nonblocking protocol unless total ordering of probes across all addresses is employed. In Section 3.3, we demonstrate how in-flight chains as implemented in our work can remove the total ordering requirement.

An alternate approach to total ordering to eliminate the deadlock that can arise in nonblocking TDs is to store away all probe messages that cannot be immediately handled for later processing, as described by Kong et al. [1999]. This approach allows the coherence protocol to process any ready probe message. But in the absence of an optimized algorithm that limits the number of probes received at a core, this would mean that every core in the system would have to maintain a buffer large enough to hold probes from every other core in the system. The additional resource overhead due to high message buffering at the core boundaries is a key scalability challenge, as described earlier in the article.

3. A CHAINED CACHE COHERENCE APPROACH TO A NONBLOCKING AND SCALABLE PROTOCOL

In the previous section, we showed that simultaneous sharing was prevalent in high-performance computing applications and introduced the concepts and overheads of nonblocking directory protocols. In this section, we first expand on the bottleneck in nonblocking protocols and then propose a method to augment them with in-flight chains so as to retain their high throughput while incorporating scalability.

3.1. Potential Bottleneck in a Nonblocking Protocol

If the coherence stall is moved from the ordering to the resolution stage without considering the ramifications on scalability, then stage 5 in Figure 5 can become a bottleneck. Because the TD can now order and start processing multiple requests to an address before even one of them is completed, a core can receive probes from potentially every other core in the coherence domain. As an example, consider a read-for-ownership request for address A, denoted as RFO A, which is issued from a core C0. This request misses in the TD and is sent to memory. It will be many cycles before C0's cache receives the data. In the interim period, any other request for address A is forwarded to C0 as a probe. Because C0 has to wait to receive the data from memory before it can service these probes, it has to maintain buffers to store the messages. In addition, once C0 receives the data, it has to send responses to all the cores that have sent it forwarded requests. Note also that network arbitration rules might mean that C0 could have to wait several cycles between sending these responses. Figure 1 illustrated this scenario as scalability concern "B." However, a nonblocking protocol can avoid forwarding all the probes to a single core if it can ensure that the point of service of response delivery migrates between the cores making simultaneous conflicting requests. Thus, migration of response delivery can remove the scalability bottleneck in a nonblocking directory protocol.

3.2. In-Flight Chains of Last Accessor Cores to Migrate Point of Service of Response Delivery

In MOESI-based cache coherence protocols, the owner core of an address is responsible for servicing on-socket requests for the address as well as for keeping memory up to date. However, the owner does not have to be the core that services all requests for data. Any core that has the up-to-date value in its cache and has the storage capability to process messages and forward data can service a request.

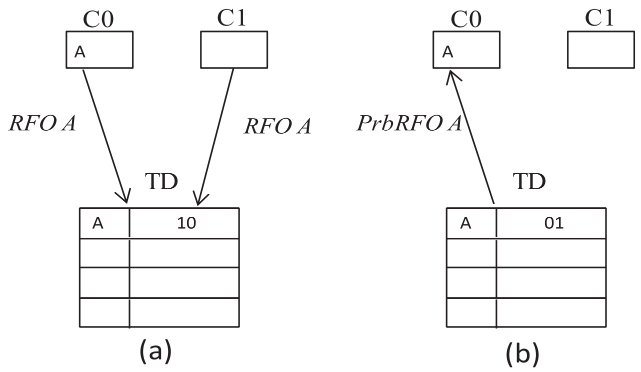

Fig. 6. Instant updates at the TD when ordering two requests in (a) take the system to the final state in (b) even though the caches have yet to be updated.

Prior works have made this observation in nondirectory-based protocols such as the Token Coherence protocol [Raghavan et al. 2008] and in bus-based protocols [Rajamony et al. 2005]. The SCI protocol also used this principle to maintain permanent pointers between cores in a linked list [Gustavson and Li 1996; Chaiken et al. 1990]. CCC, however, is the first protocol to apply this principle to improve the scalability and reduce the resource overheads of nonblocking TDs without increasing the storage required in the core caches.

The approach used in CCC to migrate the point of service of responses is to designate every core that accesses a line as the *last accessor* of a line and transfer responsibility to service requests from a last accessor to the next last accessor on read and RFO requests. The protocol supports last accessor migration by building an in-flight chain of every accessor to the line and only storing the location of the current last accessor in the TD to send the next probe. Last accessor migration by way of chaining ensures that every core in the chain is sent at most one probe (following which responsibility to service a request is transferred) and thus has to maintain only a single buffer entry and forward data to at most one other core, solving both the problems of buffering scalability and response delivery scalability. Note that the chains used in CCC are temporarily built during a period of conflicting accesses to the TD, and unlike the previously mentioned SCI protocol, no permanent pointers are required in the core caches and no special maintenance regarding insertion or removal of members in the chain is necessary. In Section 7, we further contrast the in-flight chains used in CCC with other protocols that used chaining or linking of requesting cores.

3.3. Implementing the CCC Protocol

In this section, we describe the implementation of the CCC protocol using instant updates and in-flight chains of last accessors.

3.3.1. Implementing Instant Updates in CCC. Consider Figure 6(a), which shows the interactions between two cores, C0 and C1, that are making read-for-ownership requests to address A (RFO A). C0 already has the data due to a prior read operation but now wants to update the data, whereas C1 needs to read the data and write to it. The bit vector field in the TD, which tracks the sharers of a line and is known as the core-valid or CV vector, indicates that C0's cache has the only copy of A in the socket. Let us assume that the TD ordered the RFO A from C1 before the RFO A from C0. In Figure 6(b), we see the states of the system after the request are ordered at the TD. The CV vector is instantly updated to show that C1 has a copy of the data but C0's copy is invalidated; as a result, no other access to A will be forwarded to C0 even though the data in C0's cache is yet to be invalidated. The PrbRFO (probe to get data for the RFO request) sent to C0 will eventually invalidate its copy and send the data to C1.

Since instant updates allow the TD to process parallel requests without blocking, a core could receive a read probe followed by an invalidation message (generated due to an RFO issued by a third core). These two messages should be serviced in the order they were sent by the TD so that the correct version of data is preserved. To ensure correct processing of these messages, we designate three virtual channels in our protocol. VcReq carries all requests, VcResp carries all responses and invalidation-acknowledgements (sent in response to RFO requests to maintain sequential consistency), and VcPrb carries all probes and invalidation messages. The VcReq and VcResp channels in CCC have no ordering constraints similar to the AlphaServer GS320 protocol. The VcPrb channel in that protocol, however, had to maintain total order across all addresses as described earlier. Because in-flight chains allow the TD to only send one probe to any given core after which the service of response is migrated to the next last accessor, the VcPrb channel in CCC only requires address ordering implying that only messages for a particular address in that channel need to travel in order from one source to one destination. Address ordering can be implemented as multiple interleaved in-order channels and, unlike total ordering used in the AlphaServer GS320, has no negative impact on buffering or scalability.

Ordering decisions at the TD are conveyed to requesting cores using an Order Marker (OM) message (similar in function to the marker message in the AlphaServer GS320). OMs travel on the VcPrb and are subject to the same address-ordering constraints as probes and invalidations. An OM signals to the core that its request was ordered at the TD and that any probe or invalidation received before the OM has to be processed instantaneously bevcause it refers to a prior version of data whereas a probe or invalidation received after the OM can only be processed after the core has completed its request.

3.3.2. Implementing In-Flight Chains in CCC. Because all requesting cores that miss in their core caches access the TD to obtain the latest coherent version of data, we could consider building the in-flight chain at the TD itself. However, this would require significant storage at the TD (potentially as large as the in-flight chain) as well as require the TD, which is already a contended resource, to be involved in the generation of responses to each member of the chain. Another resource that could be used to hold the chain is the cache line. However, this approach requires providing storage in each cache line to hold the core id of the next member in the chain. A key benefit of in-flight chains in CCC is that they are only maintained for requests that are still outstanding, so the TD and the cache do not have to be aware of the existence of the in-flight chain. All the TD has to maintain is the current last accessor to an address.

Let us consider Figure 7, which shows an in-flight chain being built by three cores, all waiting on read data for address A. The last accessor in the figure for A is C2 because this was the last core to access A. As shown in the figure, C0 and C1 are part of the chain and have been last accessors at some point. Because the Miss Address File (MAF), sometimes called the *outstanding buffer,* already maintains storage in the core for every in-flight request, it can easily be extended to also maintain the chain. Every MAF entry is augmented with a field that holds the target identification for the next core in the chain. Because every core only receives one probe before it transfers ownership of a line, the number of bits in this field only has to be large enough to store the value of the last core in the node. The MAF entry also requires a field that denotes if a probe has been received for this address before the data is available and its type (Read/RFO). Because the size of the MAF (32 entries in recent processor designs) is

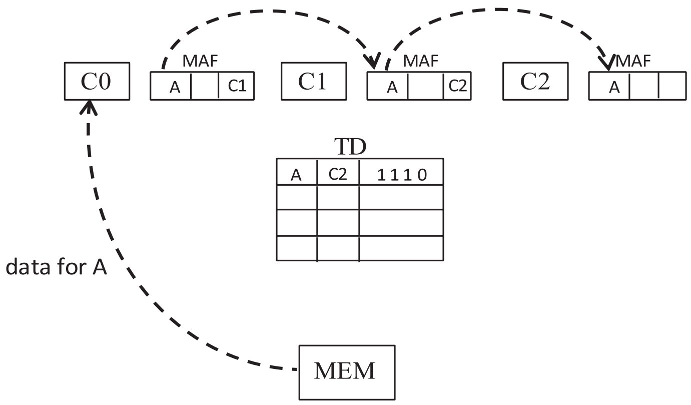

Fig. 7. In-flight chaining of parallel requests for address A between C0, C1, and C2. Instant updates at TD show C2 as last accessor even though C0 is yet to receive data.

much smaller than that of the cache or the TD, it is a scalable solution to hold the next member in the chain without significant are overhead.

In the figure, C0 and C1 each received one probe which was stored away in their MAF entry so that they know to send a fill response to C1 and C2, respectively. When C0 receives the data from memory, it processes the MAF entry, uses the data, and then forwards it to C1, which in turn forwards it to C2. As the figure shows, even if cores are added to the chain, only a single buffer entry needs to be kept at every MAF entry and only a single response needs to be delivered providing scalability. Similarly, the MAF entry has to also hold storage for an invalidation message that it may receive.

As a result of in-flight chaining, every probe is directed to the last accessor for an address. A concern might be that the last accessor is not necessarily the closest core to the requesting core in the network, which could increase resolution latency. However, communication locality implies that the probability of sharing tends to be higher in neighboring cores as opposed to randomly distributed cores mitigating any adverse impact of chaining [Bailey 1994]. In Section 5, we show that the performance of CCC assuming "perfect chaining" (where the responses in the chain are all serviced directly from the head of the chain) is not significantly higher than in-flight chaining.

3.4. Avoiding Protocol Races in the CCC Protocol

Correctness of a coherence protocol is often measured by how it avoids races resulting in deadlock. The first possible coherence race is the late race. A late race occurs if a probe for a request reaches the last accessor after it has started eviction of a cache line. The problem occurs because at the time the probe is generated at the TD, the last accessor has the line in a valid state. However, the last accessor could have started the victim response in the VcResp channel as the probe is making its way up in the VcPrb channel. CCC solves the late race by maintaining a valid copy of the data that is being written back at the core until an acknowledgement is received from the TD that it has recorded the eviction and that no further probes for data will be sent to the core. There are different implementations that can be used to hold this data. Either it can reside in the cache until the acknowledgement arrives or a separate structure that is sometimes present in modern cache hierarchies called the *Victim Buffer* (VB) can be used to hold this data temporarily. Because this storage is only required to hold evicted cache lines temporarily, it does not have to grow due to scaling of core counts making it a scalable solution.

The second commonly occurring race in coherence protocols is the early race. An early race occurs if a probe for a request arrives at the last accessor core before the data arrives and the last accessor has to decide which version of data the probe should be serviced with. We have described this scenario during the discussion of instant updates earlier in the article. We use the OM message to resolve this ambiguity because any probe that arrives at the MAF entry before the OM has been ordered before the request. Because an MAF can only hold one outstanding request per address, it is trivial to match up OMs and probes. Furthermore, due to in-flight chaining, which transfers service ownership on accesses, we are guaranteed that at most one probe will be sent from the TD to the last accessor core. Therefore, even if the probe has to be buffered, it does not impact scalability.

3.5. Other Scalability Advantages of Chained Cache Coherence

3.5.1. Avoiding Nacks, Retries, and Timeouts. Negative acknowledgements or nacks are typically used in protocols to resolve races and to avoid deadlocks that may occur due to resource dependencies. Nacks and retries are undesirable for several reasons; they add uncertainty in the processing of requests in the protocol, they can impact fairness (e.g., if a particular core request is always having to retry), and they have a negative effect on scalability because these messages add to the buffering requirements and queue sizes. CCC does not run into deadlock issues caused by resource dependencies at the cores due to the properties of in-flight chains as well as in the TDs due to instant updates. Thus, the CCC protocol does not require or support any form of negative-acknowledgments, which improves its scalability and efficiency.

3.5.2. Scalability of Message Buffers. Protocols that block or stall could put undue pressure on the message buffers or channels, which can cause back-pressure, contention and overall inefficiency in the protocol. Blocking protocols have the obvious challenge, that multiple requests to a line get blocked at the TD, which means that the queues may have to be large enough to support these blocked requests. Protocols based on total ordering requirements may also need large queues because ready messages will have to wait until they reach the head of the queue to be processed. Consider, for example, a back-invalidation issued to make space in the TD is sent to a core. If that core is already processing a request for the same address, it will not process the back-invalidation until the request is complete. This could potentially hold up an important probe for another address behind it that requires data causing slowdowns in the protocol. Because CCC does not block during request completions and does not have total ordering requirements, it is able to process or buffer away probes and responses as soon as they are received at their destinations.

4. USING IN-FLIGHT CHAINS TO BUILD A NONBLOCKING AND HIERARCHICAL PROTOCOL

An added benefit of CCC is that its principles can be applied toward building race-free and nonblocking hierarchical TDs. Blocking protocols with hierarchy avoid coherence races because each level of the hierarchy is blocked until the request completes and the TD receives an acknowledgement for the same. In a high-throughput nonblocking protocol like CCC, however, each level of the TD could be processing a different request because the stall is moved to resolution at the core, which makes building a hierarchical and correct protocol very challenging [Zhang 2010]. This section explores the benefit of hierarchical TDs and shows how CCC can incorporate hierarchical TDs using race-free and scalable techniques.

Instant updates augmented with in-flight chains addressed the first two scalability concerns described in Figure 1. The third concern was in the storage required to track the coherence state. As the number of cores (and thus private caches) in a coherence

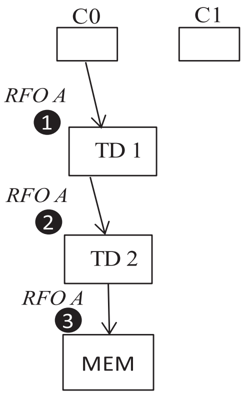

Fig. 8. First access in a hierarchy has to make an extra hop to go to memory.

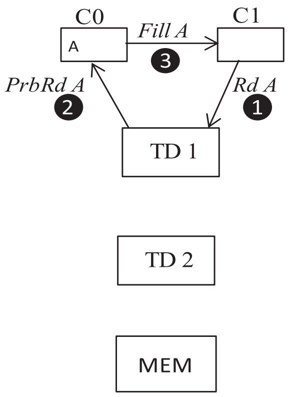

Fig. 9. Short-circuiting for a read within a domain optimizes a three-hop protocol.

domain increase, the CV vector grows with the respective number of cores. Every core adds one bit in each CV vector and the number of entries in the TDs would have to grow to accommodate the extra cache to avoid contention; thus, the storage requirement grows by O (n*n), where n is the number of cores. Proposals to reduce the CV size by representing multiple cores with a single bit have been studied. Imperfect CV bits, as they are popularly known, result in spurious probes and invalidate messages and cause TDs to fill up with entries that cannot be cleaned out unambiguously, thus requiring more TD entries. Prior hierarchical TDs [Wallach 1992; Yang et al. 1992] arranged the core caches in domains with a TD that tracks the state and ordering for all the cores in a domain. Following this trend, the TDs can be organized in domains themselves, resulting in a second-level TD (TD2) that tracks the state and ordering for all the first-level TDs (TD1). The storage requirement for a hierarchical TD organization grows by O (n *log n) because the addition of a core only adds a bit in the CV vector in the respective domain. In-flight chains can be used to extend nonblocking TDs to introduce hierarchy in a race-free manner. To incorporate hierarchy in CCC, each TD entry is augmented with a domain state field that tracks the state of the line in the local domain as well as in other domains. This field has three possible values indicating exclusive ownership in a domain, shared access permission, and lack of access permission in a domain, respectively.

Hierarchical TDs can also reduce message latencies. In communication paradigms like nearest neighbor communication, stencil algorithms, and halo-exchanges, locality exists primarily within a small domain of cores and thus sharing tends to be higher in neighboring cores than in randomly distributed cores [Bailey 1994]. Consider Figure 8, which depicts the path of communication for a request that misses in the on-die caches and goes to memory. A hierarchy of two levels as shown in the figure requires the request to make two hops to go memory because the first hop indicates the data in not present in C0's domain and the second hop indicates it is not present in the on-die caches. This two-hop process may seem like a disadvantage as opposed to a single hop in a flat directory. But note that each of the TDs (TD1 and TD2) will be smaller than one flat directory and thus has lower access latency. Figure 9 shows the true performance benefit from hierarchy. As the data for request A from C1 is present in its domain itself, this request can avoid going down to TD2 and is able to get a response from its local neighbor, C0, which reduces the response latency. In general, if a request's data can be serviced within its domain, then this request can avoid going down in the hierarchy and is able to get a response from its local neighbor, which

Table II. System Configuration

Parameter	Configuration
Core	In-order, 2-threads, 4-way, 32 outstanding misses
L1 Cache	32KB, Private, 4-way, 1-cycle,
L2 Cache	512KB, Private, 8-way, 10-cycle
MAF, VB	32-entry, FA
TD1, TD2	128KB, 8-way, 10-cycle
Interconnect Network	2-D mesh, 16-entry queues, 1-cycle link latency
Memory	150-cycle latency

reduces the response latency. This phenomenon is called *short-circuiting*. CCC uses the domain state field to determine whether reads or RFOs or both can short-circuit the hierarchy. Because CCC implements nonblocking by building in-flight chains of requests, the TDs can maintain independent local domain chains as well as global chains without any impact on throughput. The second key feature in the CCC protocol that enables building race-free hierarchies is that probes sent between levels are always processed without stalling because they travel on the VcPrb channel, thus avoiding deadlock.

5. EVALUATION AND VERIFICATION

In the preceding sections, we qualitatively enumerated the benefits of CCC over other protocol scaling techniques. In this section, we quantify these benefits in terms of performance speed-ups and other metrics that measure stalls. We also present the performance of hierarchical CCC and demonstrate its sensitivity to core counts and queue sizes.

5.1. Methodology

5.1.1. Simulation Environment. We employ a cycle-accurate performance simulator to implement and study CCC that is based on the Asim performance modeling infrastructure [Emer et al. 2002]. The baseline system models a socket with 64 tiles arranged in an 8x8 mesh interconnection network. The configuration of each tile (core, MAF, VB, TD1 slice, TD2 slice) as well as the memory and the interconnection network is summarized in Table II. As described earlier, we do not model an L3 due to the increased access latency and thread contention at such large core counts. The TD1s are banked and are arranged column-wise such that a domain comprises eight tiles. This implies that every memory access issued by a core only has to check the TD1 on its column to determine if it can be serviced in its domain itself. The TD2s are arranged row-wise such that once a home for a particular address is determined on the interconnection network, it has a specific TD1 tile and a specific TD2 tile that can service it. Thus, all messages make at most one turn when they go from the TD1 to the TD2. The memories are placed on three sides of the tiled network to reduce hop counts on memory accesses.

For the experiments reported in the article that do not use hierarchy, we force all requests to go to the TD2 to access shared data even if it is in the same domain. The simulation infrastructure does not use an operating system but is augmented with an application scheduler. For the HPC workloads studied, the impact of system-level events are not significant, whereas for the server workloads, full-system traces are used so the impacts of system-level events are modeled. The baseline model uses a blocking directory-based MOESI protocol, which is similar to the CMP_directory protocol modeled in the gem5 simulation infrastructure, and unblocks TDs by sending acknowledgements once the requesting core completes its transaction [Binkert et al. 2011]. In addition to the CCC protocol, we also study an enhanced blocking protocol by using one recently proposed predictive sharing optimization that avoids directory indirection by using "writer prediction" [Kaxiras and Georgios 2010]. We model the PC-based predictor; however, rather than deal with the complexity of handling mispredictions in their protocol, we only forward data directly from the producer to the consumer when a "peek" into the directory indicates that the prediction is correct. Thus, the baseline shows the upper-bound performance potential of writer prediction without the penalty of recovery.

5.1.2. Workloads. Because of their computing resources, large-scale systems are generally attractive to the following market segments: HPC, scientific computing, parallel servers, and supercomputing. We chose seven Class C applications from the NAS Parallel benchmark suite as being representative of the HPC domain and ran the default problem size for the class [Bailey 1994]. We use Barnes (16,384 nodes, 123 seed) and Ocean (258∗258 array) from the Splash-2 benchmark suite to represent the scientific computing similar to other published studies [Woo et al. 1995]. The online transaction processing and Web server needs of server applications are modeled with in-house Tpcc and Specweb full-system traces. We forward 100B instructions and then simulate 500 million instructions for each workload. Simulations continue to execute until at least one thread in each domain (column in the topology) executes 500 million instructions. The supercomputing domain is modeled using three specialized workloads. First, the RandomAccess workload from the HPCC benchmark suite, where the dominant pattern is random read-modify-write updates to a hash table. Because this benchmark was designed to measure the giga updates per second of the system, which is a critical marker of supercomputing systems, it is denoted as GUPS in the results. We also studied a linear barrier algorithm, denoted as Linear Barrier, in which all cores reach a barrier at a random address continuously to study the benefit that a nonblocking protocol with hierarchy would have on such a heavily shared pattern. Finally, a more realistic tree-based barrier is explored. The total area sizes for the synthetic workloads were realistically chosen to be larger than the individual core caches. Studying a wide spectrum of workloads helps to analyze the benefits and overheads of a coherence protocol.

5.2. Improving Performance over Blocking

In-flight chains using instant updates allow simultaneous conflicting requests to be processed without blocking, which could result in higher throughput at the TD. Figure 10 presents the performance advantages of three protocol designs over the baseline blocking protocol. We group the applications based on their market segment.

The first data point, BL_WrPred, explores the upper-bound performance of a blocking protocol that is enhanced with writer prediction. Because this datapoint does not include the latency of mispredictions, it provides an upper bound on the performance of a blocking protocol. The results indicate that writer prediction alleviates some of the penalties imposed by blocking TDs and improves the performance across all the applications. The key aspect is whether the application can service a significant portion of the accesses out of the cache itself. In Tpcc, in particular, predictive sharing achieves all the benefit that a nonblocking protocol with hierarchy provides because it has a small footprint that fits mostly in the cache. In the barrier workloads, however, any thread could reach the barrier in any order which impacts the prediction accuracy. The fact that CCC has higher performance overall than predictive sharing in a completely nonspeculative manner indicates that it is more likely to be implemented and scaled as core counts increase, which could cause thrashing in the predictor. Nevertheless, CCC can also be augmented with a sharing predictor to improve its performance.

The second data point in this plot, CCC_Original, demonstrates the performance of the original CCC protocol without optimized short-circuiting in the hierarchy. As

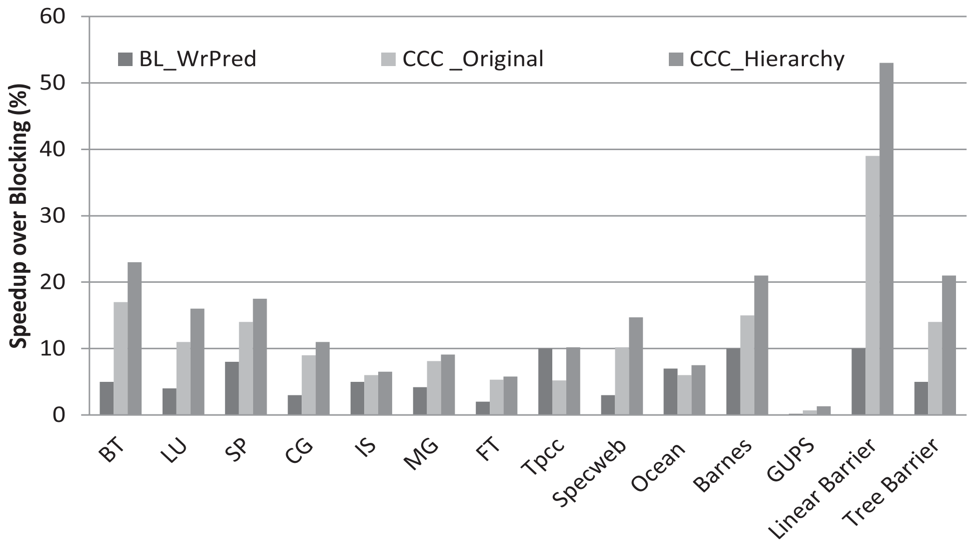

Fig. 10. Performance speed-up using in-flight chaining and instant updates relative to a blocking protocol.

Table III. Percentage of Stalls Due to Blocked TDs in the Baseline

Wkd	BT	LU	SP	CG	Tpcc	Specweb	Barrier	Tree Barr
% Blocked	6.9	9.4	6.7	3.6	8.9	15.3	32.3	15.3

described in Section 2.2, the applications BT, SP, and LU all show nearest neighbor communication patterns with significant sharing among many threads. As a result, they have over 10% speed-up due to the high-throughput nature of CCC. Specweb has a large memory footprint, and thus many of the stalls are to memory, which increases the time the TD is blocked. In Barnes, the main data structure results in a majority of single producer, multiple consumer patterns which can benefit from the high-throughput of a non-blocking directory. These results correspond to the lost cycles due to blocking that was demonstrated in Figure 4 and show that CCC eliminates almost all the negative impact of blocking with negligible overhead. GUPS is a randomized pattern of read-modify-write operations to a hash table with negligible sharing and so has little performance benefit from eliminating blocking. The tree barrier workload is a radix 8 algorithm that matches the simulated configuration. Finally, the extreme linear barrier example shows the upper bound on performance that a scalable, nonblocking protocol like CCC can achieve. In this workload, the TD was blocked for more than 40% of the execution time. The performance for the barrier workloads also shows that when there are many small and frequently accessed parallel regions, as is the case in many synchronization primitives, CCC can provide much higher performance than even a highly optimized blocking protocol.

We use several statistics to analyze and verify the performance data. First, the data shown in Figure 4 shows that blocked TDs add significant cycles to the execution time. Analyzing the data showed that if one of the stalls was to memory, they had the most impact. Second, Table III represents the percentage of stalls in the baseline caused due to conflicting requestors at the TDs for the most interesting workloads. The data shows that the barriers and producer–consumer patterns suffer the most. One way to improve the performance of blocking protocols is to stall requests only if they will modify data and allow all read requests that conflict to be processed and instantly update the TD. The problem with this solution is that there needs to be storage to hold all these read

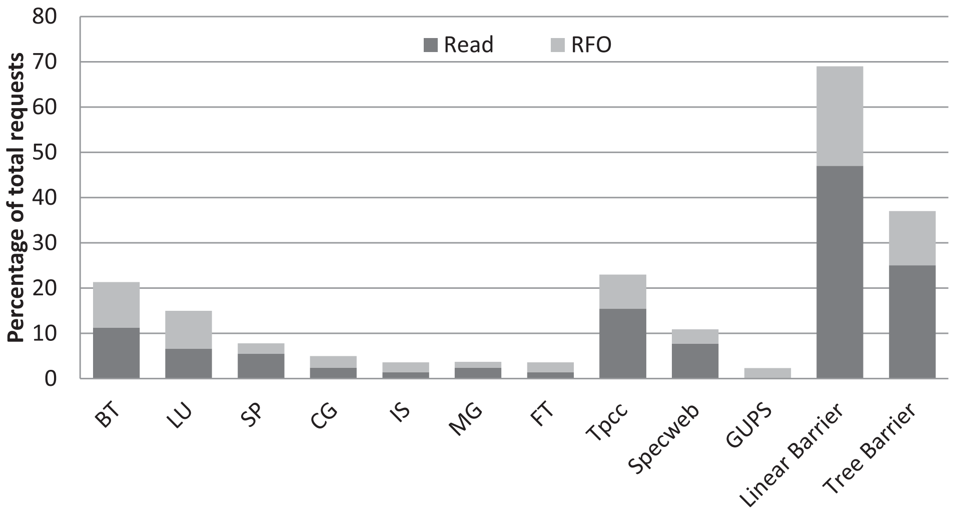

Fig. 11. Requests that can be short-circuited.

probes, which leads to a scalability problem. An in-flight chain of last accessors solves the bottleneck in nonblocking directories with a scalable storage solution. As a last analysis experiment, we built an analytical model to verify the HPCC GUPS workload.

5.3. Optimizing Local Communication with Hierarchy

Section 4 illustrated the performance benefits of short-circuiting the hierarchy using an example access pattern. In this section, we quantify this benefit depicted in the third data point in Figure 10 labeled as CCC_Hierarchy. The results are normalized to the baseline blocking protocol without hierarchy, so these results indicate the cumulative impact of designing a nonblocking and scalable protocol and being able to short-circuit the hierarchy. As shown in the figure, the linear barrier workload has significant locality of communication because it represents a single miss being brought in from memory, which can service the entire domain. On the other hand, the benefit in performance for many of the HPC applications is primarily achieved by the nonblocking nature of the TD. We also measure the number of reads and RFO requests that could be short-circuited, which is shown in Figure 11. As expected, more reads than RFOs are short-circuited, since RFOs have to invalidate all valid copies of data and can only be serviced completely in their domain if they have exclusive access.

5.4. Applying “Perfect Chaining”

In this section, we focus on the possible adverse performance impact of in-flight chaining if the last accessor core jumps around the network. To study this effect, we simulate the CCC protocol with perfect chaining, that is, we implement CCC as it is but assign performance latencies for response delivery as though the responses were sent out by the head of the chain. Figure 12 demonstrates this configuration. The first interesting observation from these results is that across the spectrum of workloads perfect chaining only impacts a few workloads (CG, Specweb, and Barnes). The second interesting observation is that adding hierarchy in the TD organization mitigates the impact of chaining as every local domain has its own last accessor that can be used to short-circuit the hierarchy whereas the owner core may belong in another domain. This phenomenon can be seen in CG and Barnes. The original protocol with perfect chaining has slight improvement, but when hierarchy is introduced, the performance

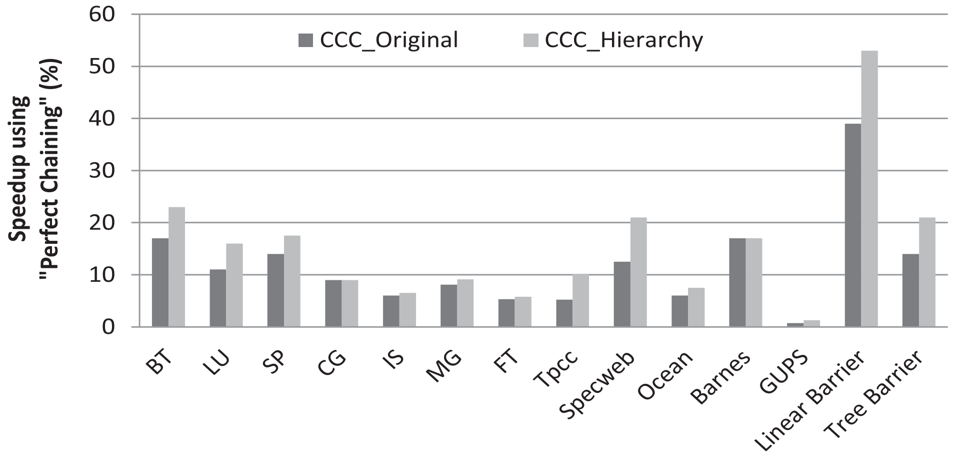

Fig. 12. Impact of perfect chaining.

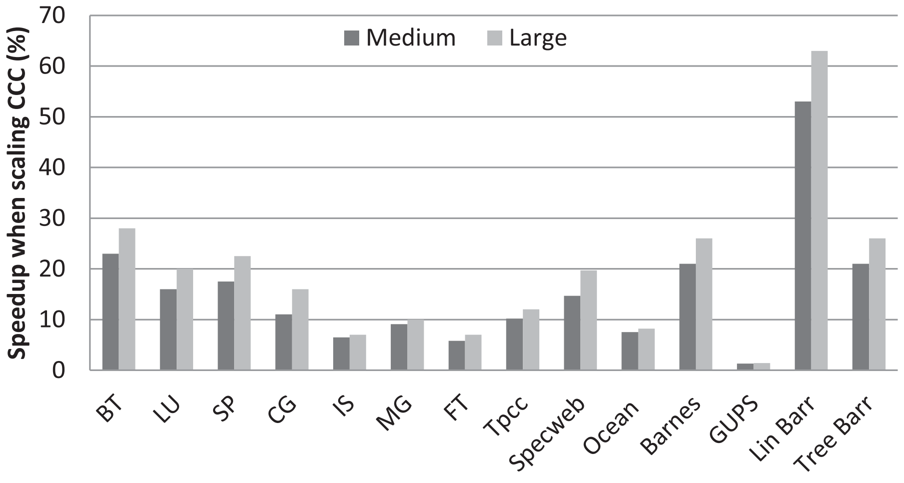

Fig. 13. Performance of CCC on larger sockets.

is slightly worse than CCC_Hierarchy because the benefit of hierarchy is crossed out due to the head of the chain being in a different domain. Specweb, on the other hand, has a performance speed-up, since the dominant pattern is an RFO followed by reads and the reads in a domain do not reach the TD in a linear manner.

5.5. Performance Scalability of CCC

Although the results presented in the article so far were for a 64-tile system, a practical design point for running long and detailed simulations, we also studied the final hierarchical CCC design using a larger socket with 128 tiles using an 8×16 mesh interconnect and appropriate scaled queue sizes. All the other simulation parameters are kept the same. Figure 13 shows that the trends are similar to the medium-sized 64-tile socket. However, the difference in the barrier workloads is higher because these workloads are directly impacted by scaling core counts.

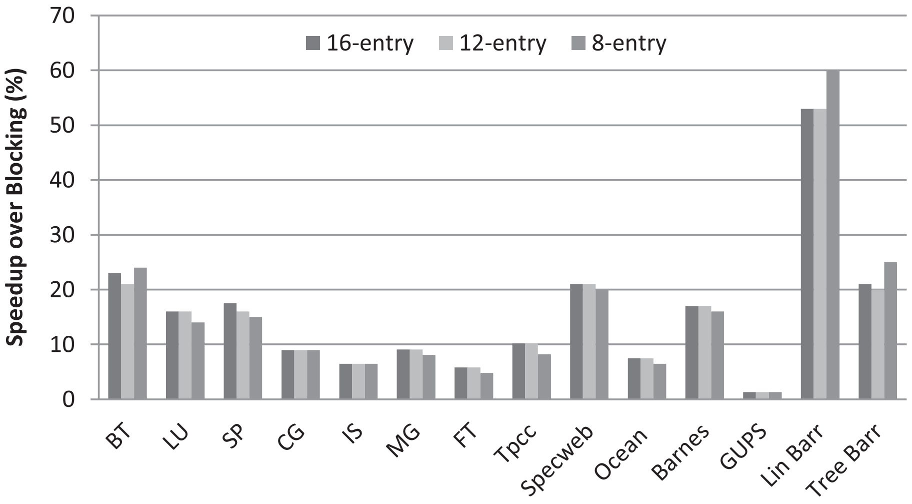

Fig. 14. Sensitivity of CCC to network queue sizes.

5.6. Sensitivity of CCC to Queue Sizes

Sizing network queue structures appropriately can significantly affect the performance of a coherence protocol. We perform a sensitivity study that measures the performance benefits of CCC relative to a blocking protocol when using smaller network queues (8 and 12 entries), as presented in Figure 14. The analysis shows that, in general, the CCC protocol performs just as well with 12-entry and 8-entry queues. However, for the barrier workloads, we find that the speedup is even higher at the 8-entry configuration as the smaller queues degrade the performance of the blocking protocol more than that of CCC. Smaller queues in the interconnection network can reduce energy consumption and area making CCC a scalable and efficient solution.

5.7. Scalability Advantages of CCC over Total Ordering

In-flight chains allow the building of a deadlock and livelock-free protocol without requiring total ordering across different addresses in the probe channel. In addition to the implementation infeasibility of building a totally ordered network as described earlier in the article, total ordering could also pose a performance problem, especially when there are outstanding requests for addresses for which cores receive probes. In these scenarios, the probes have to wait for the data to reach the core before they can be processed, causing them to experience long stalls in the queues. To study the impact that total order has on the rate of probe processing at the cores, we analyzed the number of probe messages that are blocked due to a stall at the head of the probe queue from the TDs to the cores. Figure 15 shows the percentage increase in probes that are stalled in a protocol implementing total ordering over the in-flight chains used in CCC. The data indicates that the same type of workloads that perform well with CCC (those that simultaneously access shared data) also experience many stalls in a totally ordered network because these workloads often have probes at the head of the queue that are waiting for fills.

5.8. Verification

We verified the original and hierarchical CCC protocol using the Murphi checker [Dill 1996]. We used Murphi to study three major aspects of our protocol: deadlock free, which means that every state reached in Murphi simulation is one from where progress can

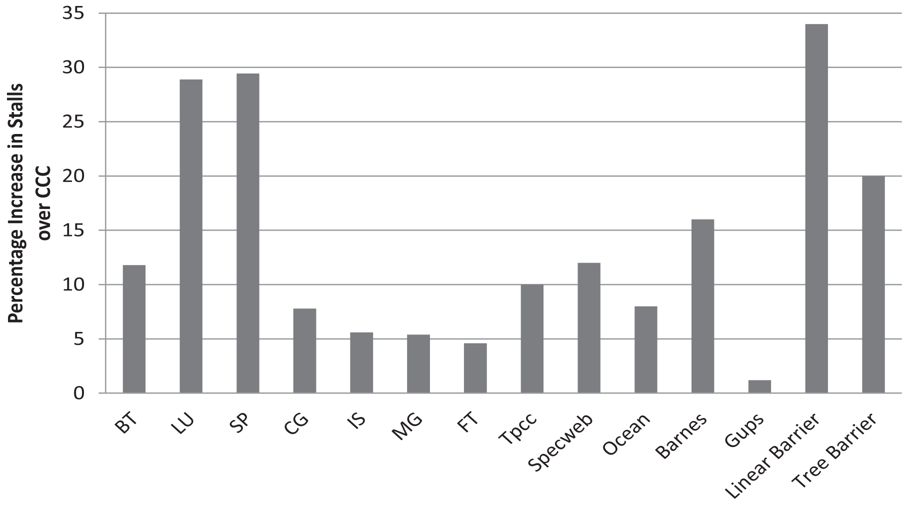

Fig. 15. Percentage increase in stalls in a totally ordered network.

be made; MOESI property, which states if there is a line in the modified state in any cache the hierarchy, then no other valid copy can exist in; and the value property, which states that the last value to be written for any line is the value that any subsequent read for the line receives. To verify systems, Murphi exhaustively expands the state graph of a system and walks all the paths in the state graph. State enumeration in Murphi is vulnerable to explosion if scale of the system is very large and so we chose to verify a smaller system but with multiple domains so that all the protocol transactions were encountered. We use multiple addresses in our verification because victims and back-invalidations occur only in the presence of multiple addresses.

6. COMPARISON OF CCC WITH DISTRIBUTED COHERENCE PROTOCOLS

An alternate coherence framework is the Token Coherence protocol, which attempts to avoid the latency of directory accesses [Martin et al. 2002, 2003b]. In this framework, coherence is maintained by tracking a pre-established number of tokens for each block of shared memory. The key idea is to separate the high-performance features of the protocol from the correctness and resorting to retries in the event of synchronization races. While eliminating directory accesses can enable a two-hop low-latency protocol, races can cause persistent retries that are much slower and consumer bandwidth. The base implementation of token coherence is described as relying on broadcasts in the network to get the low latency of cache-to-cache transfers in snooping protocols. However, such a system would be overwhelmed by bandwidth limitations as we scale the cores on a socket. In broadcast-based token coherence for a system with 64-cores, a read request would generate several unnecessary messages to search for data as opposed to the request, probe, and response messages generated in a directory-based protocol and the original paper on token coherence shows that a basic version of the protocol can use twice the interconnect bandwidth of a basic directory protocol. CCC provides the low bandwidth requirements of directory-based protocols while improving their throughput and scalability using in-flight chains.

Variations on token coherence protocols use adaptive mechanisms with both snooping and directory accesses to mitigate the bandwidth impact; however, these hybrid protocols are more complex to implement [Bilir et al. 1999; Martin et al. 2002]. Prior work has explored the feasibility and performance of a hierarchical Token Coherence protocol to improve its scalability [Marty et al. 2005]. In such a system, the coherence protocol would first send out broadcasts in its local domain and if none of the local caches respond with the data, the request will be forwarded to the global domain. This additional latency in sending out a global request could have an adverse impact on the performance. In contrast, the hierarchical CCC protocol is able to use in-flight chains to optimize local requests as well as efficiently identify and issue requests that need to go to the global TD level.

7. RELATED WORK

7.1. Blocking Directory Protocols

The Scalable Coherent Interface (SCI) implements a blocking TD using a linked-list directory structure, which leads to design complexity and latency issues, especially when accessing nodes in the middle of the linked list [Gustavson and Li 1996]. SCI is also a strict request-reply blocking protocol and cannot use CCC's probes and OMs without adding a third virtual channel. Although SCI uses chains to link sharing cores, it requires forward and backward permanent pointers stored in each cache line, whereas CCC only builds temporary forward pointing chains in the outstanding MAF buffer (32-entries) leading to far less area requirement. CCC's in-flight chains are also likely to be smaller than SCI's linked lists leading to faster data propagation and higher performance. SCI also has to traverse the doubly linked list for invalidations and patch up head and tail pointers during evictions, whereas CCC only uses a small victim buffer to maintain correct state. Consequently, we believe that it is nontrivial to make SCI a race-free nonblocking protocol. The novelty in CCC is that the last requestor/accessor can augment a nonblocking protocol to make it scalable.

Directory CMP is a hierarchical blocking coherence protocol that uses safe and transient state to resolve races [Marty and Hill 2008]. As the number of cores being integrated on a chip increase, however, the probability that the protocol hits unsafe or blocked states increases.

7.2. Nonblocking Directory Protocols

The nonblocking protocol in the AlphaServer GS320 uses a global switch to implement total ordering, which, as described earlier, is infeasible from a modern implementation perspective and can lead to high occupancy in the queues. The SGI Origin protocol is another nonblocking directory-based protocol that uses back-off messages to avoid deadlocks indicating to a requester core that it should directly send its request [Laudon and Lenoski 1997]. Since CCC is able to provide a high-throughput and nonblocking protocol without total ordering or the use of back-off massages, we believe it is more scalable.

7.3. Improving Performance of Shared Data

This body of work looked specifically at extending protocols to improve the performance of widely shared data [Kaxiras 1996]. The general principle in these works is that they provide extensions and solutions to map logically sharing tree algorithms to physical networks. Some of these solutions include building and invalidating a tree in logarithmic time [Nilsson 1992; Johnson 1993]. Other works build limited tree-based directories for shared data [Maa 1991]. CCC aims to provide a general framework to improve the performance of critical shared data and synchronization variables in a scalable manner. Since CCC is implemented using a general TD present in modern hierarchies and is based upon the nonblocking directory that was built in the AlphaServer product, we believe that it is more likely to be adopted.

7.4. Optimizing the Tag Directory Structure

Hierarchical or tree-based cache coherence organizations have been studied previously as a way to provide scalability in data storage and communication [Ladan-Mozes and Leiserson 2008]. The primary motivation in this work is to avoid timeouts and nacks by creating a unique path through the tree from each core to a given memory bank. Alternative approaches to reducing the data storage overhead include building tagless directories using bloom filters or by building efficient directories using a variable number of tags based on sharing patterns and optimized hash functions [Ferdman et al. 2011; Sanchez and Kozyrakis 2012; Zebchuk et al. 2009]. All these designs can be used in conjunction with the basic principle of migrating the point of service of responses in CCC to further improve its scalability.

7.5. Hardware Coherence and Software Coherence

Other work in cache coherence protocols has focused on highlighting the importance of hardware-based on-chip cache coherence [Martin et al. 2012]. This work shows that by following the goals of scalability, on-chip cache coherence can be made efficient and low cost, which is the same high-level goal of the CCC protocol.

8. CONCLUSIONS AND FUTURE WORK

Scalability in cache coherence protocols is a challenging problem facing designers and blocking protocols have serious concerns in being able to maintain high throughput. Nonblocking directory protocols as designed so far require significant resource overheads and impractical network ordering constraints to provide high performance. In this article, we exploit a key insight that service ownership of responses for data can be migrated by building in-flight chains of parallel memory requests. The chained coherence protocol built on instant updates and in-flight chains has higher performance than traditional and predictive blocking protocols with minimal channel ordering requirements as compared to prior nonblocking protocols. In-flight chains can also be used to incorporate hierarchy in the TD organization in a race-free and scalable manner. Future work may explore multisocket cache coherence, which is another communication bottleneck.

REFERENCES

Bailey, D. H. 1994. The NAS Parallel Benchmarks. www.davidhbailey.com/dhbpapers/npb-encycpc.pdf.

Bilir, E. E., Dickson, R. M., Hu, Y., Plakal, M., and Sorin, D. J. 1999. Multicast snooping: A new coherence method using a multicast address network. In *Proceedings of the International Symposim on Computer Architecture (ISCA'99)*.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S., Saidi, A., Arkaprava, B., et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Archit. News 39*, 2, 1–7.

Bronevetsky, G. G. 2009. CLOMP: Accurately characterizing OpenMP application overheads. *Int. J. Parallel Program. 37*, 250–265.

Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A. 1990. Directory-based cache coherence in large-scale multiprocessors. *Computer 23*, 6, 49–58.

Chaudhuri, M., Hienrich, M., Holt, C., Singh, J. P., and Hennessy, J. 2003. Latency, occupancy, and bandwidth in DSM multiprocessors: A performance evaluation. *IEEE Trans. Comput. 52*, 7, 862–880.

Cheng, L., Carter, J. B., and Dai, D. 2007. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'07)*.

Cox, A. and Fowler, R. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the International Symposium on Computer Architecture*.

Dill, D. L. 1996. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*. 390–393.

EMER, J., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C., MANNE, S., ET AL. 2002. Asim: A performance model framework. *IEEE Comput. 35*, 2, 68–76.

FERDMAN, M., LOTFI-KAMRAN, P., BALET, K., AND FALSAFI, B. 2011. Cuckoo directory: A scalable directory for many-core systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'11).*

GHARACHORLOO, K., SHARMA, M., STEELY, S., AND VAN DOREN, S. 2000. Architecture and design of AlphaServer GS320. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00).*

GRAHAM, R. AND SHIPMAN, G. 2008. MPI support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

GUSTAVSON, D. AND LI, Q. 1996. The scalable coherent interface (SCI). *Commun. 34*, 8, 52–63.

HAGERSTEN, E. AND KOSTER, M. 1999. WildFire: A scalable path for SMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'99).*

JALEEL, A., MATTINA, M., AND JACOB, B. 2006. Last level cache (LLC) performance of data mining workloads on a CMP—a case study of parallel bioinformatics workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'06).*

JEFFERS, J. 2012. Intel® many integrated core architecture: An overview and programming models. http://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL_Elec_Struct_WS_02062012.pdf.

JOHNSON, R. E. 1993. Extending the scalable coherent interface for large-scale shared-memory multiprocessors. PhD thesis, University of Wisconsin at Madison.

KAXIRAS, S. G. 1996. The glow cache coherence protocol extensions for widely shared data. In *Proceedings of the International Conference on Supercomputing.*

KAXIRAS, S. AND GEORGIOS, K. 2010. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro, 30*, 54–65.

KAXIRAS, S. AND GOODMAN, J. 1999. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of the International Symposium on High-Performance Architecture (HPCA'99).*

KONG, J., YEW, P.-C. Y., AND GYUNGHO, L. 1999. *A Non-blocking Directory Protocol for Large-Scale Multiprocessors.* Tech. rep. TR 99-012, University of Minnesota.

LADAN-MOZES, E. AND LEISERSON, C. 2008. A Consistency Architecture for Hierarchical Shared Caches. In *Proceedings of the Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08).*

LAUDON, J. AND LENOSKI, D. 1997. The SGI origin: A ccNUMA highly-scalable server. In *Proceedings of the International Symposium on Computer Architecture (ISCA'97).*

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W., GUPTA, A., HENNESSY, J., ET AL. 1992. The Stanford dash multiprocessor. *Computer 25*, 3, 63–79.

MAA, Y.-C. P. 1991. Two economical directory schemes for large-scale cache-coherent multiprocessors. *Comput. Archit. News* 19, 5, 10–18.

MARTIN, M. M., HARPER, P. J., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2003a. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03).*

MARTIN, M. M., HILL, M. D., AND SORIN, D. J. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM 55*, 78–89.

MARTIN, M. M., HILL, M. D., AND WOOD, D. A. 2003b. Token coherence: Decoupling performance and correctness. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03).*

MARTIN, M. M., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2002. Bandwidth adaptive snooping. In *Proceedings of the International Symposium on High-Performance Computer Architecture (ISCA'02).*

MARTY, M. R. 2008. Cache coherence techniques for multi-core processors. PhD thesis, University of Wisconsin.

MARTY, M. AND HILL, M. 2008. Virtual hierarchies. In *Proceedings of the International Symposium on Computer Architecture (ISCA'08).*

MARTY, M., BINGHAM, J., HILL, M., HU, A., MARTIN, M., AND WOOD, D. 2005. Improving multiple-CMP systems using token coherence. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'05).*

MORGAN, T. P. 2012. Intel teaches Xeon Phi x86 coprossor snappy new tricks: The interconnect rings a bell. http://www.theregister.co.uk/2012/09/05/intel_xeon_phi_coprocessor.

Mukherjee, S. S. and Hill, M. D. 1998. Using prediction to accelerate coherence protocols. In *Proceedings of the International Symposium on Computer Architecture (ISCA'98).*

Nilsson, H. 1992. The scalable tree protocol—a cache coherence approach for large-scale multiprocessors. In *Proceedings of the International Symposium on Parallel and Distributed Computing.*

Raghavan, A., Blundell, C., and Martin, M. M. 2008. Token tenure: Patching token counting using directory coherence. In *Proceedings of the International Symposium on Microarchitecture (MICRO'08).*

Rajamony, R., Shafi, H., Williams, D., and Wright, K. 2005. Chained cache coherency states for sequential non-homogeneous access to a cache line. US patent US20070083716 Al.

Sanchez, D. and Kozyrakis, C. 2012. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'12).*

Singh, J., Wolf-Dietrich, W., and Gupta, A. 1992. *SPLASH: Stanford parallel applications for shared memory.* Tech. rep., Stanford University, Stanford, CA.

Wallach, D. 1992. PHD: A hierarchical cache coherent protocol. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA.

Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA'95).*

Yang, Q., Thangadurai, G., and Bhuyan, L. 1992. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 3*, 3, 281–293.

Yongqin, H., Aidong, Y., Jun, L., and Xiangdong, H. 2009. A novel directory-based non-busy, non-blocking cache coherence. In *Proceedings of the International Forum on Computer Science-Technology and Applications (IFCSTA'09).* 374–379.

Zebchuk, J., Srinivasan, V., Qureshi, M., and Moshovos, A. 2009. A tagless coherence directory. In *Proceedings of the International Symposium on Microarchitecture (MICRO).* International Symposium on Microarchitecture.

Zhang, M. L. 2010. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the International Symposium on Microarchitecture.*

Received June 2013; revised September 2013; accepted August 2013