Why On-Chip Cache Coherence is Here to Stay

A final revision of this manuscript will appear in Communications of the ACM, TBD 2012

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Mark D. Hill
University of Wisconsin
markhill@cs.wisc.edu

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

Abstract

Today's multicore chips commonly implement shared memory with cache coherence as low-level support for operating systems and application software. Technology trends continue to enable the scaling of the number of (processor) cores per chip. Because conventional wisdom says that the coherence does not scale well to many cores, some prognosticators predict the end of coherence.

This paper refutes this conventional wisdom by showing one way to scale on-chip cache coherence with bounded costs by combining known techniques such as: shared caches augmented to track cached copies, explicit cache eviction notifications, and hierarchical design. Based upon our scalability analysis of this proof-of-concept design, we predict that on-chip coherence and the programming convenience and compatibility it provides are here to stay.

1. Introduction

Shared memory is the dominant low-level communication paradigm in today's mainstream multicore processors. In a shared memory system, the (processor) cores communicate via loads and stores to a shared address space. The cores use caches to both reduce the average memory latency and reduce memory traffic. Caches are thus beneficial, but private caches lead to the possibility of cache incoherence. The current mainstream solution is to provide shared memory and to prevent incoherence using a hardware cache coherence protocol, making caches functionally invisible to software. The sidebar (Figure 1) reviews the incoherence problem and the basic hardware coherence solution.

Cache-coherent shared memory is provided by mainstream servers, desktops, laptops, and even mobile devices, and it is available from all major vendors, including Intel, AMD, ARM, IBM, and Oracle (Sun). Cache coherence has come to dominate the market for both technical and legacy reasons. Technically, hardware cache coherence provides performance that is generally superior to that achievable with software-implemented coherence. Cache coherence's legacy advantage is that it provides backward compatibility for a long history of software, including operating systems, that is written for cache-coherent shared memory systems.

Although coherence provides value in today's multicore systems, the conventional wisdom is that on-chip cache coherence will not scale to the large number of cores expected to be found on future processor chips [5, 10, 13]. Coherence's alleged lack of scalability is believed to be due to the poor scaling of the storage and traffic on the interconnection network that coherence requires, as well as concerns about latency and energy. Such arguments against coherence lead those authors to conclude that cores in future multicore chips will not employ coherence, but instead will communicate with software-managed coherence, loads/stores of scratchpad memories, or message passing (without shared memory).

This paper seeks to refute this conventional wisdom by presenting one way to scale on-chip cache coherence in which coherence overheads (*i.e.*, traffic, storage, latency, and energy) (a) grow slowly with core count and (b) are similar to the overheads deemed acceptable in today's systems. To do this, we synergistically combine known techniques such as: shared caches augmented to track cached copies, explicit cache eviction notifications, and hierarchical design. Using amortized analysis, we show that interconnection network traffic per miss need not grow with core count and that coherence uses at most 20% more traffic per miss than a system with caches but not coherence. Using hierarchical design, we show that storage overhead can be made to grow as the root of core count and stay small, e.g., 2% of total cache size for even 512 cores. We find negligible energy overhead and zero latency penalty for cache misses to data that are not actively shared, and our analysis suggests the relative miss penalty and energy overheads of accessing shared data do not increase appreciably with increasing core count.

Consequently, we predict that on-chip coherence is here to stay. Computer systems tend not to abandon compatibility to eliminate small costs, such as the costs we find for scaling coherence. In particular, systems will not likely wish to replace conventional operating systems now that they have been shown to scale using cache coherence [2, 3]. Boyd-Wickizer *et al.* [2] concur: "there is no scalability reason to give up on traditional operating system organizations just yet."

Some might object to retaining coherence, because many applications do not yet scale well with coherence. True, but is the root cause the algorithm, program implementation, or coherence? If, for example, frequent updates must be communicated to many readers, coherence and most alternatives will do poorly. At least with hardware coherence, programmers can concentrate on choreographing independent work rather than handling low-level mechanics.

Our claim for the continued viability of on-chip cache coherence does not imply that other communication paradigms will disappear. There will continue to be applications for which message passing is appropriate (e.g., scale out high performance computing) or for which incoherent scratchpad memories are appropriate (e.g., real-time systems), and we believe those communication paradigms will persist. However, we predict that they are likely to continue to coexist with on-chip cache-coherent shared memory.

2. Cache Coherence Today

To understand the issues involved in coherence's future, we must first understand today's cache coherence protocols. Rather than provide a survey of the coherence protocol design space, we instead focus on describing one concrete coherence protocol loosely based upon the on-chip cache coherence protocol used by Intel's Core i7 [17]. This system represents the state-of-the-art and can already

Sidebar: The Problem of Incoherence

Incoherence. To illustrate the incoherence problem, consider the multiple cores and corresponding *private caches* in the upper-right of the diagram below. If core 1 writes the block labeled *B* by updating its private cache only, subsequent reads by core 2 would see the old value indefinitely. This incoherence can lead to incorrect behavior. For example, if the block holds a synchronization variable for implementing mutual exclusion using a lock, such incoherent behavior could allow multiple cores into a critical section or prevent cores waiting for the release of the lock from making forward progress.

The coherence invariant. The mainstream solution to preventing incoherence is a hardware cache coherence protocol. Although there are many possible coherence protocols, they all maintain coherence by ensuring the *single-writer, multiple-reader (SWMR) invariant*. That is, for a given block, at any given moment in time, there is either:

- only a single core with write (and read) permission to the block (in state *M* for *modified*) or
- zero or more cores with read permission to the block (in state *S* for *shared*).

Enforcing coherence. The diagram below illustrates core 0 caching block *A* with read/write permission (state *M*) and cores 1 and 2 caching block *B* with read-only permission (state *S*). A write to block *B* by core 1 (which, in our example above, led to incoherence) is not allowed to update its read-only copy of the block. Instead, core 1 must first obtain write permission to the block. Obtaining write permission without violating the single-writer, multiple-reader invariant requires *invalidating* any copies of the block in other caches (core 2 in this case, as encoded by the tracking bits in the shared cache). Such actions are handled in hardware by cache coherence logic that is integrated into the cache controllers. Section 2 presents a current protocol (and describes the rest of the diagram). For more background, see Sorin *et al.* [18].

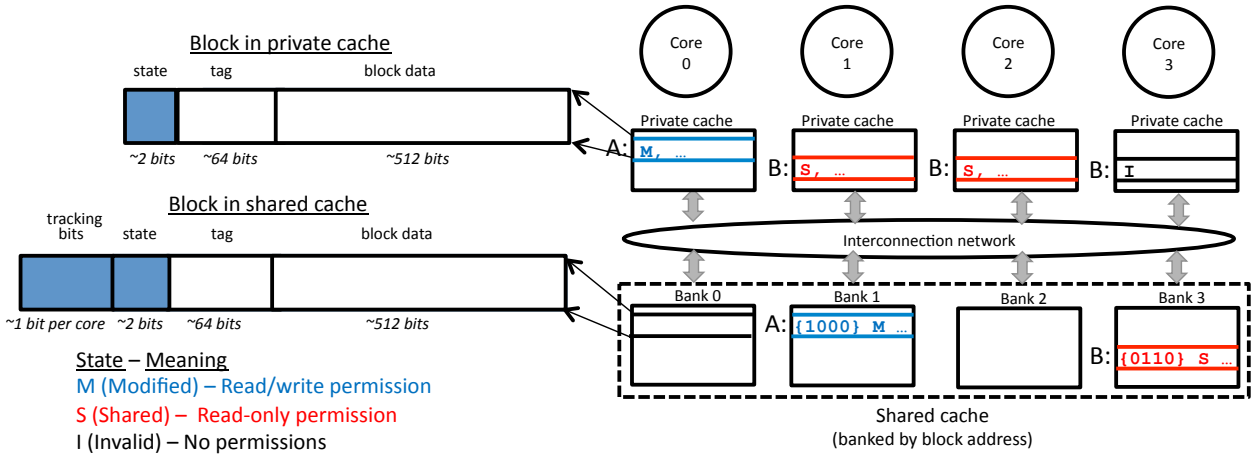


Figure 1: System Model. Additions for coherence are shaded.

scale to a moderate number of cores (*e.g.*, 16). In such a system, the cores on the chip communicate via loads and stores to the shared memory. Each core has its own private cache hierarchy (referred to hereafter simply as "private cache"). There is a single, shared last-level cache (referred to hereafter as "shared cache"). Such shared caches typically employ address-interleaved banking with one bank per core, thus proportionally scaling the bandwidth of the shared cache as the number of cores increases. Figure 1 illustrates this system model.

To make our analysis simpler and more concrete, we assume for now that the shared cache is *inclusive* with respect to all of the private caches. Inclusion means that, at all times, the shared cache contains a superset of the blocks in the private caches. Intel's Core i7 is one example of a chip with inclusive caches. Because inclusion is not a universally adopted design decision, we discuss extending our results to non-inclusive shared caches in Section 9.

With inclusion, cache coherence can be maintained with a coherence protocol that tracks copies of blocks in private caches using state that is embedded in the shared cache. That is, each block in the shared cache is augmented with a few bits of coherence state (*e.g.*, to denote if the block is writable by any core) and per-core tracking bits that denote which cores are privately caching the block (one bit per core). As depicted in Figure 1, inclusion requires that block *A* (cached by core 0) and block *B* (cached by cores 1 and 2) must be present in the shared cache with appropriate tracking bits (`{1000}` and `{0110}`, respectively. If the block size of the shared cache is larger than the private cache's block size, each entry in the shared cache maintains coherence state and tracking bits at the granularity of the private cache block size.

When a core issues a load or store that misses in its private cache, it issues a coherence request message to the shared cache. Based on the block's coherence state and the block's per-core tracking bits, the shared cache either responds directly or forwards the request to the one or more cores that need to respond to the request. For example, if the request is for read/write access and one or more cores are privately caching the block in a read-only state, then the shared cache forwards the request to all private caches in the tracking list and these private caches all invalidate their copies of the block. If no cores are caching the block, then a request has the negligible overhead of only looking up the coherence state bits in the shared cache. This protocol is essentially a simple directory protocol in which the directory entries are co-located with the tags of the shared cache. Inclusion ensures that each private block has a corresponding shared block to hold its coherence tracking bits.

To maintain inclusion, when the shared cache wishes to evict a block for which some per-core tracking bits are set, then the shared

cache issues a *recall* request (also known as a back-invalidation or notification) to any core that is currently caching that block (as determined by the per-core tracking state). Upon receipt of a recall message, the private cache is forced to evict the block.

This approach to coherence has many attractive features, which helps to explain why current Intel systems resemble it. This protocol avoids the need for a snooping bus and it avoids broadcasting; communication involves only point-to-point messages. Because it embeds the per-core tracking bits in the shared cache, it avoids adding additional structures dedicated solely to coherence. For small-scale systems (*e.g.*, 4-16 cores), the storage cost is negligible (a 16-core system adds just 16 bits for each 64-byte cache block in the shared cache, or approximately 3% more bits). For a miss to a block not cached by other private caches, the miss latency and energy consumed incur the negligible overhead of checking a couple of state bits in the shared cache rather than just a single valid bit. As we show later, even when blocks are shared, the traffic per miss is small and independent of the number of cores. Overall, this approach is reasonably low-cost in terms of traffic, storage, latency, and energy, and its design complexity is tractable. Nevertheless, the question is: does this system model scale to future many-core chips?

3. Scalability Concerns

Some prognosticators forecast that the era of cache coherence is nearing its end [5, 10, 13], primarily due to an alleged lack of scalability. However, when we examined state-of-the-art coherence mechanisms, we found them to be more scalable than one might expect.

We consider a coherent system to be "scalable" when the cost of providing coherence grows (at most) slowly as core count increases. We focus exclusively on the cache coherence aspects of multicore scaling, whereas a fully scalable system (coherent or otherwise) also requires scalability from other hardware (*e.g.*, memory and interconnect) and software (operating system and applications) components.

This paper examines five potential concerns raised when scaling on-chip coherence:

1. **traffic** on the on-chip interconnection network,
2. **storage** cost for tracking sharers,
3. inefficiencies caused by **maintaining inclusion** (as inclusion is assumed by our base system),
4. **latency** of cache misses, and
5. **energy** overheads.

The next five sections address these concerns in sequence and present our analysis which indicates that existing design approaches can be employed such that none of these concerns present a fundamental barrier to scaling coherence. After this analysis, we discuss extending the analysis to non-inclusive caches and address some caveats and potential criticisms of this work.

4. Concern #1: Traffic

We now tackle the concerns regarding the scalability of coherence's traffic on the on-chip interconnection network. To perform a traffic analysis, we consider for each cache miss how many bytes need to be transferred to obtain and relinquish the given block. We divide the analysis into two parts: in the absence of sharing and then with sharing. This analysis shows that when sharers are tracked precisely, the traffic per miss is independent of the number of cores. Thus, if coherence's traffic is acceptable for today's systems with small numbers of cores, it will continue to be acceptable as we scale up the number of cores. We conclude this section with a discussion of how coherence's per-miss traffic compares to that of a system without coherence.

4.1 Without Sharing

We first analyze the worst-case traffic in the absence of sharing. Each miss in a private cache requires at least two messages: (1) a request from the private cache to the shared cache and (2) a response from the shared cache to provide the data to the requestor. If the block is written during the time it is in the cache, the block is *dirty* and must be explicitly written back to the shared cache upon eviction.

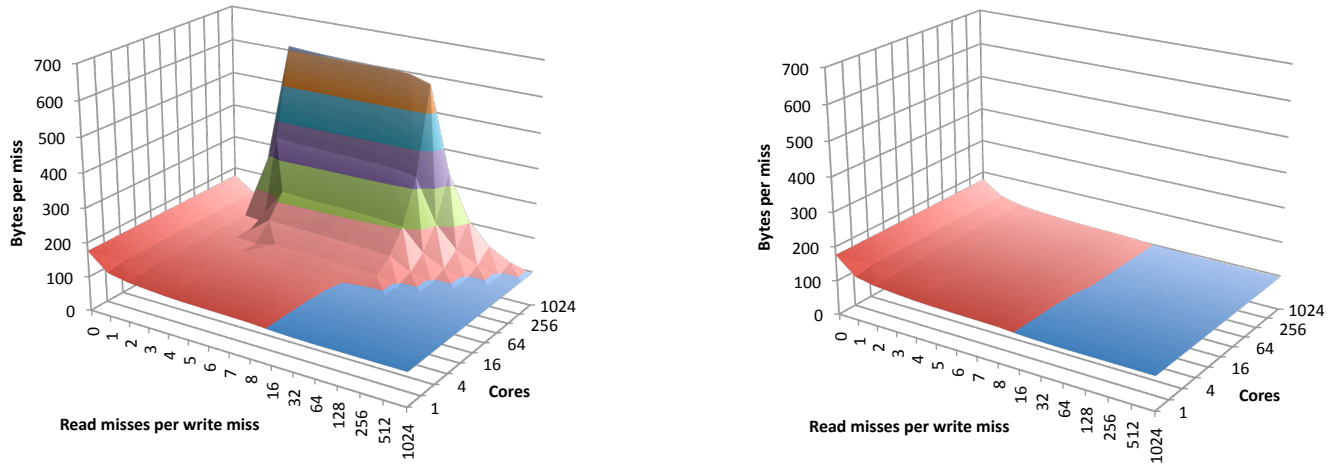
Even without sharing, the traffic depends on the specific coherence protocol implementation. In particular, we consider protocols that require a private cache to send an explicit eviction notification message to the shared cache whenever it evicts a block, even when evicting a clean block. (This decision to require explicit eviction notifications benefits the implementation of inclusive caching, as discussed in Section 6.) We also conservatively assume that coherence requires the shared cache to send an acknowledgment message in response to each eviction notification. Fortunately, clean eviction messages are small (say, enough to hold a 64-bit address) and can only occur subsequent to cache misses (which transfer, say, a 512-bit cache block). Thus coherence's additional traffic per miss is modest (we will compare it to a system without coherence later in this section) and, most importantly, independent of the number of cores. Based on 64-byte cache blocks, Table 1 shows that coherence's traffic is 96 bytes/miss for clean blocks and 160 bytes/miss for dirty blocks.

4.2 With Sharing

In a coherent system, when a core reads a block that is shared, the coherence protocol may need to forward the request, but to at most one core (and thus the traffic for each read miss is independent of the number of cores). However, when a core incurs a write miss to a block that is cached by one or more other cores, the coherence protocol generates extra messages to *invalidate* the block from the other cores. These invalidation messages are often used to argue for the non-scalability of cache coherence, because when all cores are sharing a block, a coherent system must send an invalidation message to all other cores. However, our analysis shows that when sharers are tracked exactly, the overall traffic per miss of cache coherence is independent of the number of cores (the storage cost of exact tracking is addressed in Section 5).

Consider the access pattern in which a block is read by all cores and then written by one core. The writer core will indeed be forced to send an invalidation message to all cores, and each core will respond with an acknowledgment message (*i.e.*, a cost of $2N$ messages for N sharers). However, such an expensive write operation can occur only after a read miss by each of those cores. More generally, for every write that invalidates N caches, it must have been preceded by N read misses. The traffic cost of a read miss is independent of the number of cores (a read miss is forwarded to a single core at most). Thus, by amortized analysis, the overall average traffic per miss is constant. (A write miss that causes N messages can occur at most once every Nth miss).

In support of this general analysis, Figure 2(b) shows the traffic (in average bytes per miss) over a range of core counts for an access pattern parameterized by the number of read misses to a block between each write miss to the block. A workload consisting of all write misses (zero read misses per write miss; far left of the graph) has the highest traffic per miss because all blocks are dirty. The traffic per miss is independent of the number of cores because



(a) With inexact tracking of sharers

(b) With exact tracking of sharers

Figure 2: Communication traffic for shared blocks

	Clean block	Dirty Block
Without coherence	(Req+Data)+0 = 80 B/miss	(Req+Data) + Data = 152 B/miss
With coherence	(Req+Data) + (Evict+Ack) = 96 B/miss	(Req+Data) + (Data+Ack) = 160 B/miss
Per-miss traffic overhead	20%	5%

Table 1: To calculate the traffic, we must assume values for the size of addresses and cache blocks (say, 64-bit physical addresses and 64-byte cache blocks). Request and acknowledgment messages are typically short (*e.g.*, 8 bytes) because they contain mainly a block address and message type field. A data message is significantly larger because it contains both an entire data block plus a block address (*e.g.*, 64 + 8 = 72 bytes).

the shared cache forwards the write misses to at most one core (the most recent writer). With an increasing number of read misses per write miss (moving to the right on the graph), the average traffic per miss actually decreases slightly because fewer writes lead to fewer dirty blocks. More importantly, as predicted, the traffic is independent of the number of cores in the system, because each write miss that causes N messages is offset by N previous read misses.

4.3 Traffic Overhead of Coherence

We have already shown in this section that coherence's per-miss traffic scales, because it is independent of the number of cores. We now examine coherence's traffic overhead per miss with respect to a hypothetical design with caches but no hardware coherence (*e.g.*, software knows exactly when cached data is stale without extra traffic). We continue to measure traffic in terms of bytes of traffic on the interconnection network per cache miss, thus making the assumption that coherence does not change the number of cache misses. This assumption is, however, potentially compromised by false sharing and inefficient synchronization, which can cause non-scalable increases in the number of cache misses. Both of these phenomena are well-known challenges with well-known techniques for mitigating them; we cannot completely eliminate their impact nor can we cleanly incorporate them into our intentionally simplistic models.

In Table 1, we show the traffic per miss for this system without coherence, and we now compare it to the system with coherence. For a clean block, the system without coherence eliminates the need for the eviction notification and the acknowledgment of this notification. For a dirty block, the system without coherence avoids the acknowledgment of the dirty eviction message. The key is that all three of these messages are small. Thus, coherence's overhead is small, bounded, and—most importantly—independent of the number of cores. Based on 64-byte cache blocks, Table 1 shows that coherence adds a 20% traffic overhead for clean blocks and a 5% overhead for dirty blocks.

Conclusion: *Coherence's interconnection network traffic per miss scales when exactly tracking sharers (one bit per core).*

5. Concern #2: Storage

The scalable per-miss traffic result in the previous section assumed an exact tracking of sharing state in the shared cache, which requires N bits of state for a system with N cores. This assumption leads to the understandable concern that such an increase in tracking state for systems with more cores could pose a fundamental barrier to scalability. In this section, we show that the storage cost scales gracefully by quantifying the storage cost and describing two approaches for bounding this cost: (1) the traditional approach of using inexact encoding of sharers [1, 8], which we discard in favor of (2) an often-overlooked approach of using on-chip hierarchy to efficiently maintain an exact encoding of sharers. The storage cost at the private caches is negligible—supporting coherence in the private caches adds just a few state bits for each cache block, which is less than 1% storage overhead and independent of the number of cores—so our analysis in this section focuses on additional storage in the shared cache.

5.1 Conventional Approach: Inexact Encoding of Sharers

The conventional approach to limit the storage—inexact encoding of sharers—can work well but has poor worst-case behavior. This technique represents a conservative superset of sharers using fewer bits than one bit per potential sharer, and it was well-studied in the early 1990s [1, 8]. As a concrete example, the Origin

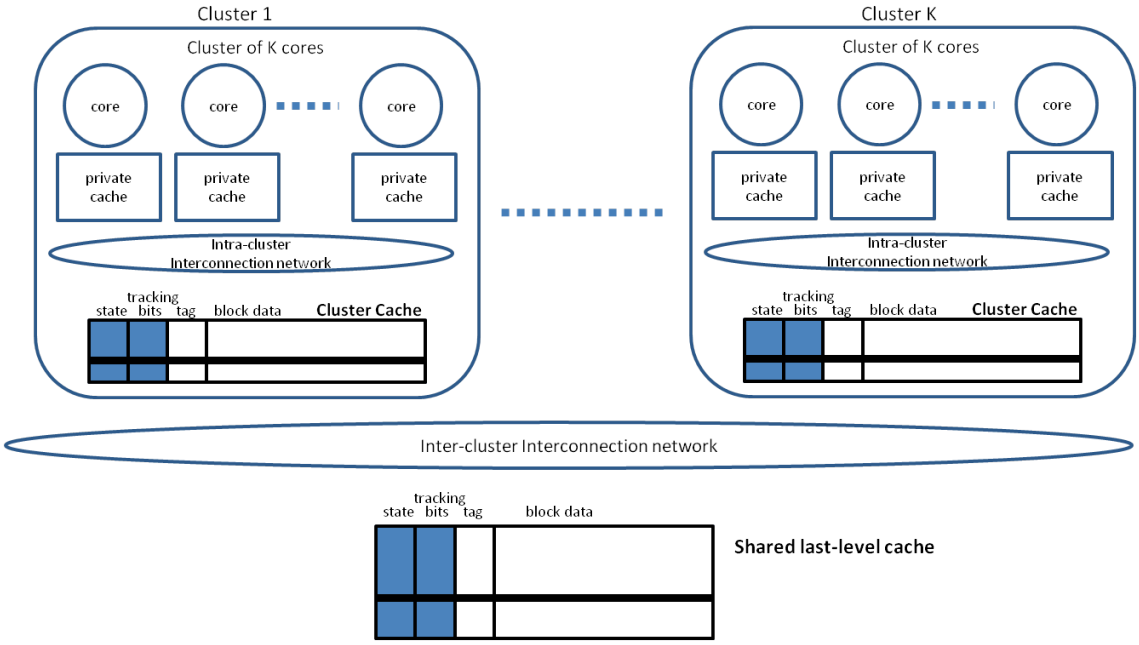


Figure 3: Hierarchical System Model. Additions for coherence are shaded.

2000 [14] uses a fixed number of bits per block, regardless of the number of cores. For small systems, the Origin uses these bits as a bit-vector than tracks sharers exactly. For larger systems, the Origin alternates between two uses of these tracking bits. If there are only a few sharers, the bits are used as a limited number of pointers (each of which requires $log_2 N$ bits to encode) that can exactly track sharers. If the number of sharers exceeds this limited number of pointers, the Origin uses the bits as an inexact, coarse-vector encoding, in which each bit represents multiple cores. Although the storage can be bounded, the traffic of such schemes suffers due to unnecessary invalidations.

To quantify the traffic impact of such inexact encodings, Figure 2(a) shows the result of applying the analysis from the previous section when using the Origin's inexact encoding scheme to bound the storage at 32 bits per block in the shared cache (approximately 6% overhead for 64-byte blocks). When the 32 bits is enough for exact tracking (up to 32 cores) or when the number of sharers is smaller than the number of limited pointers (far left of the graph), the sharers are encoded exactly, which results in the same traffic-per-miss as the exact encoding. When the number of sharers is large (far right of the graph), the write invalidations must be sent to all cores (independent of the encoding), so the inexact encoding incurs no traffic penalty. However, when the number of cores grows and the number of sharers is in the middle of the range, the traffic overheads spike. With 1024 cores, the spike reaches almost 6x the traffic of the exact encoding cases. Although conventional wisdom might have predicted an even larger traffic spike for 1024 cores, we next describe an alternative that eliminates any such spike in traffic.

5.2 Less Conventional Approach: On-Chip Hierarchy for Exact Tracking

To avoid a spike in traffic for some sharing patterns, an alternative is to overcome this scalability problem using an on-chip hierarchy of inclusive caches. Hierarchy is a natural design methodology for scalable systems. With many cores, the size of private caches is limited and the miss latency from a private cache to the chip-wide shared cache is likely large. As such, many-core systems [16, 4], GPUs [15], and proposed many-core architectures [12] cluster some number of cores/threads to share an intermediate level of cache. For example, Sun/Oracle's T2 systems [16] share a small L1 cache among two pipelines each with four threads. NVIDIA's Fermi GPU [15] clusters 32 execution pipelines into a "shared multiprocessor." In AMD's forthcoming Bulldozer architecture [4], each pair of cores has per-core private L0 caches and shares a L1 cache. Such systems fill the gap between a private cache and a large, distributed shared cache, allowing the cluster cache to provide faster access to data shared within the cluster. An additional benefit is that coherence requests may be satisfied entirely within the cluster (*e.g.*, by a sibling node that is caching the block), which can be significant if the software is aware of the hierarchy.

The same techniques described in the previous sections (inclusion, integrating tracking state with caches, recall messages, and explicit eviction notifications) are straightforward to apply recursively to provide coherence across a hierarchical system. Instead of just embedding tracking state at a single shared cache, each intermediate shared cache also tracks sharers—but just for the caches included by it in the hierarchy. Consider a chip, illustrated in Figure 3, in which each core has its own private cache, each cluster of cores has a cluster cache, and the chip has a single shared last-level cache. Each cluster cache is shared among the cores in the cluster and serves the same role for coherence as the shared cache in the non-hierarchical systems we have discussed previously. That is, the cluster cache tracks which private caches within the cluster have the block. The shared last-level cache tracks which cluster caches are caching the block, but not which specific private cache(s) within the cluster are caching it. For example, a balanced 256-core system might consist of 16 clusters of 16 cores each with a 16KB first-level cache, a 512KB second-level shared cluster cache, and a 16MB third-level (last-level) cache shared among all clusters.

Such a hierarchical organization has some disadvantages (extra complexity and additional layers of cache lookups), but it has two key benefits for coherence. First, the hierarchy naturally provides a simple form of fan-out invalidation and acknowledgment combining. For example, consider a block cached by all cores. When a core issues a write miss to that block, the cluster cache lacks write permission for the block, so it forwards it on to the shared last-

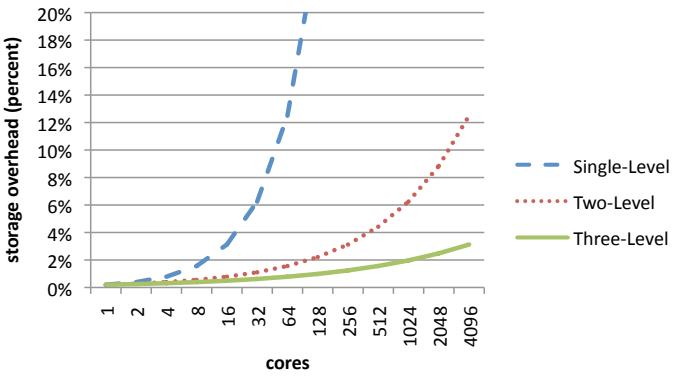


Figure 4: Storage overhead in shared caches

level cache. The shared last-level cache then sends an invalidation message to each cluster (not to each core), which triggers the cluster cache to perform an analogous invalidation operation within the cluster. The cluster then sends a single invalidation acknowledgment (independent of the number of cores in the cluster that were caching the block). Compared to a flat protocol, which must send acknowledgments to every requestor, the total cross-chip traffic is reduced, and it avoids the bottleneck of sequentially injecting hundreds or thousands of invalidation messages and later sequentially processing the same number of acknowledgments.

The second benefit is that a hierarchical system that enforces inclusion at each level reduces the storage cost of coherence. Recall from Section 4 that using an exact encoding of sharers allows for scalable communication for coherence, but that we deferred the seeming problem of the storage cost of exact encoding. Now we show that, by using hierarchy, we can also make the storage cost scale gracefully. Consider first a two-level system (three levels of cache) comprised of K clusters of K cores each ($K^2 = C$ total cores). Each cluster cache is inclusive with respect to all of the private caches within the cluster, and the shared last-level cache is inclusive with respect to all of the cluster caches. Each cluster cache block uses one bit for each of the K private caches it includes, plus a few bits of state. Similarly, each shared last-level cache block consumes one bit for each of the K cluster caches it includes, plus a few bits of state. Importantly, these storage costs grow as a linear function of K and thus proportional to $\sqrt{C}$. Even if C increases greatly, $\sqrt{C}$ grows more slowly.

This storage cost at the cluster caches and last-level cache could be reduced even further by extending the hierarchy by one level. Consider a system with K level-2 clusters, each of which consists of K level-1 clusters; each level-1 cluster consists of K cores. This system has $C = K^3$ cores and a storage cost proportional to $\sqrt[3]{C}$

In Figure 4, we plot coherence's storage overhead (i.e., coherence's storage as a fraction of the total cache storage) in terms of the bits needed to provide exact tracking of sharers, for conventional flat (non-hierarchical) systems, 2-level systems, and 3-level systems. For a very large example, a 1024-core two-level system would have 32 clusters of 32 cores and thus 32 bits per 64-byte cache block at each level, which is just 6%. An extreme (absurd?) 4096-core three-level system would have 16 clusters, each with 16 sub-clusters of 16 cores, with a storage overhead of only 3%.

Conclusion: *Hierarchy combined with inclusion enables efficient scaling of the storage cost for exact encoding of sharers.*

6. Concern #3: Maintaining Inclusion

In the system model we focus on in this paper, we have chosen to initially require that the shared cache maintain inclusion with respect to the private caches. Maintaining an inclusive shared cache allows efficient tracking of blocks in private caches by embedding the tracking information in the tags of the shared cache, which is why we choose to use this design point. Inclusion also simplified the analysis of communication and storage in the previous sections.

Inclusion requires that if a block is cached in any private cache, it is also cached in the shared cache. Thus, when the shared cache evicts a block with non-empty tracking bits, it is required to send a recall message to each private cache that is caching the block, adding to system traffic. More insidiously, such recalls can increase the cache miss rate by forcing cores to evict hot blocks they are actively using [11]. To ensure scalability, we seek a system that make recalls vanishingly rare, the design of which first requires understanding the reasons why recalls occur.

Recalls occur when the shared cache is forced to evict a block with one or more sharers. To reduce recalls, the shared cache always chooses to evict non-shared blocks over shared blocks. Because the capacity of an inclusive shared cache will often exceed the aggregate capacity of the private caches (for example, the ratio is 8 for the four-core Intel Core i7 with a 8MB shared cache and four 256KB second-level private caches), it is highly likely that there will be a non-shared block to evict whenever an eviction occurs.

Unfortunately, the shared cache sometimes lacks sufficient information to differentiate between a block *possibly* being cached and *certainly* being cached by a core. That is, the tracking bits in the shared cache are updated when a block is requested, but the shared cache does not always know when a private cache has evicted the block. Why? In most systems, *clean* blocks (those not written during their lifetime in the cache) are evicted silently from the private caches, thus introducing ambiguity at the shared cache as to what is still being cached and what has already been evicted. This lack of information manifests as poor replacement decisions at the shared cache.

To remedy this lack of information, a system can instead require the private caches to send explicit notification messages whenever a block is evicted (even when evicting clean blocks). For example, AMD's HT-Assist uses explicit eviction notifications on clean-exclusive block replacements to improve sharer state encoding [6]. If such eviction notifications occur on every cache eviction, it enables the shared cache to maintain a precise, up-to-date tracking of private caches that hold each block, transforming the tracking information from being conservative to exact. Thus, when an eviction decision occurs, the shared cache will know which blocks are no longer being cached, and thus likely have a choice to evict a non-shared block to avoid a recall. This precision, of course, comes with a cost: increased traffic for evictions of clean blocks (previously analyzed in Section 4).

Explicit eviction notifications can potentially eliminate all recalls, but only if the associativity (i.e., the number of places in which a specific block may be cached) of the shared cache exceeds the aggregate associativity of the private caches. With sufficient associativity, whenever the shared cache looks for a non-shared block to evict, if it has exact sharing information, it is guaranteed to find a non-shared block and thus avoid a recall. Without this worst-case associativity, a pathological cluster of misses could lead to a situation in which all blocks in a set of the shared cache are truly shared. Unfortunately, even with a modest number of cores, the required associativity becomes prohibitive, as has been previously reported [7]. For example, eight cores with eight-way set-associative private caches require a 64-way set-associative shared cache and the required associativity doubles for each doubling of the number of cores.

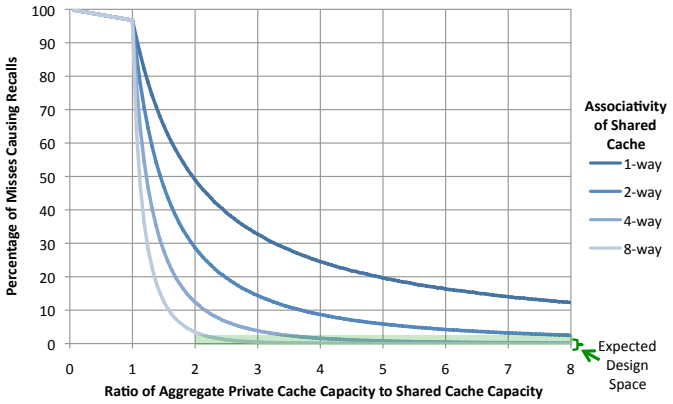


Figure 5: Likelihood that a shared cache miss triggers a recall.

Thus, instead of eliminating all recalls, we instead focus on a system in which recalls are possible but rare. To estimate the impact of limited shared cache associativity on recall rate, we performed a simulation modeling recalls due to enforcing inclusion in such a system. We pessimistically configured the private caches to be fully associative. To factor out the effect of any particular benchmark, we generated a miss address stream to random sets of the shared cache, which prior work has found accurately approximates conflict rates [9]. We also pessimistically assumed no data sharing among the cores, which would reduce the inclusive capacity pressure on the shared cache.

Fortunately, recalls can be made rare in the expected design space. Figure 5 shows the recall rate (the percentage of misses that cause a recall) for shared caches of various sizes (as a ratio of aggregate per-core capacity) for several shared cache associativities. When the capacity of the shared cache is less than the aggregate per-core capacity (ratio < 1.0), almost every request causes a recall because the private caches are constantly contending for an unrealistically under-provisioned shared cache. As the shared cache size increases, the recall rate quickly drops. Once the capacity ratio reaches 4×, even an eight-way set-associative shared cache keeps the recall rate below 0.1%. For comparison, the Intel Core i7 has a 16-way set-associative cache with an 8× capacity ratio. That 8-way shared caches have few recalls at capacity ratios of 2× and above is analogous to the behavior of hash tables that also work well when sized at least twice as large as the data stored. Based on this analysis, we conclude that the traffic overhead of enforcing inclusion is negligible for systems with explicit eviction notifications and reasonable shared cache sizes and associativities.

Conclusion: *We can design a system with an inclusive shared cache with a negligible recall rate, and thus we can efficiently embed the tracking state in the shared cache.*

7. Concern #4: Latency

In a non-coherent system, a miss in a private cache sends a request to the shared cache. As discussed earlier, to provide sufficient bandwidth, shared caches are typically interleaved by addresses with banks physically distributed across the chip (Figure 1), so the expected best-case latency of a miss that hits in the shared cache is the access latency of the cache bank plus the round-trip traversal of the on-chip interconnect to reach the appropriate bank of the shared cache. Requests that miss in the shared cache are in turn routed to the next level of the memory hierarchy (*e.g.*, DRAM).

In a coherent system with private caches and a shared cache, there are four cases to consider with regard to miss latency: (1) a hit in the private cache, (2) a *direct miss*, in which the shared cache can fully satisfy the request (*i.e.*, to a block not cached in other private caches), (3) an *indirect miss*, in which the shared cache must contact one or more other caches, and (4) a miss in the shared cache, which incurs a long-latency access to off-chip DRAM. Coherence adds no latency to perhaps the two most performance-critical cases: private cache hits (case 1) and off-chip misses (case 4). Coherence also adds no appreciable latency to direct misses, because the coherence state bits in the tags of the shared cache can be extended to unambiguously distinguish between direct and indirect misses.

Indirect misses, however, do incur the extra latency of sending a message on the on-chip interconnect to the specified private cores. Such messages are sent in parallel, and responses are typically sent directly to the original requester (resulting in what is known as a three-hop protocol). Thus, the critical path latency of direct and indirect misses can be approximated by the following formulas:

Non-coherent:

$$t_{noncoherent} = t_{interconnect} + t_{cache} + t_{interconnect}$$

Coherent:

$$t_{direct} = t_{interconnect} + t_{cache} + t_{interconnect}$$
$$t_{indirect} = t_{interconnect} + t_{cache} + t_{interconnect} + t_{cache} + t_{interconnect}$$

The indirect miss latency for coherence is between 1.5× and 2× larger than the latency of a non-coherent miss (the exact ratio depends on the relative latencies of cache lookup (t_{cache}) and interconnect traversal ($t_{interconnect}$). This ratio is considered acceptable in today's multicore systems (partly because indirect misses are generally in the minority for well-tuned software). The ratio also indicates scalability, as this ratio is independent of the number of cores. Even if the absolute interconnect latency increases with more cores, such increases will generally increase the latency of misses (even in a non-coherent system) roughly proportionally, thus keeping the ratio largely unchanged. Moreover, if latency is still deemed to be too great, for either coherent or non-coherent systems, these systems can use prefetching to hide the latency of anticipated accesses.

Similar reasoning can be applied recursively to calculate the above latency ratio for a system with more layers of hierarchy. Although the impact of hierarchy may hurt absolute latency (*e.g.*, due to additional layers of lookup), overall, we see no reason why hierarchy should significantly impact the ratio of the latencies of direct to indirect misses. Furthermore, the cluster caches introduced by hierarchy may help mitigate the growing cross-chip latency by (1) providing a closer medium-sized cache that both allows faster sharing within a cluster and (2) reducing the number of longer-latency accesses to the chip-wide distributed shared last-level cache. Modeling the full impact of hierarchy on latency (and traffic) is beyond the reach of the simple models used in this paper.

Conclusion: *Although misses to actively shared blocks have greater latency than other misses, the latency ratio is tolerated today and the ratio need not grow as the number of cores increases.*

8. Concern #5: Energy

Although a detailed energy analysis is perhaps not as straightforward as the analyses we have performed thus far, we can use those prior analyses to support the conclusion that the energy cost of coherence is also not a barrier to scalability. In general, energy overheads come from both doing more work (dynamic/switching energy) and from additional transistors (static/leakage energy).

For dynamic energy, the primary concerns are extra messages and additional cache lookups. However, we have already shown that interconnect traffic and message count per-miss do not increase with the number of cores, which in turn indicates that the protocol state transitions and number of extra cache lookups are similarly bounded and scalable.

For static energy, the primary concerns are the extra tracking state, which we have also already shown scales gracefully, and leakage due to any extra logic for protocol processing. The protocol processing logic is added per core and/or per cache bank, and thus also should add at most a fixed per-core, and thus scalable, leakage energy overhead.

Furthermore, many energy-intensive parts of the system are largely unaffected by coherence (the cores themselves, the cache data arrays, off-chip DRAM, and storage), and thus energy overheads incurred by coherence will be relatively smaller when put into the context of the whole system.

Conclusion: *Based upon the aforementioned traffic and storage scalability analyses, we find no reason that the energy overheads of coherence will increase with the number of cores.*

9. Non-Inclusive Shared Caches

So far this paper assumes an inclusive shared cache, like that of Intel's Core i7, but this choice is not universal. Instead of requiring a private cache block to be present in the shared cache (inclusion), a system can forbid it from being present (exclusion) or allow but not require it to be present (neither inclusion nor exclusion). Not enforcing inclusion reduces redundant caching (less important for the Core i7 whose shared cache size is eight times the sum of its private cache sizes), but has implications on coherence.

A non-inclusive system can retain the coherence benefits of an inclusive shared cache by morphing it into two structures: (a) a new non-inclusive shared cache that holds tags and data, but not tracking state, and is free to be of any size and associativity, and (b) a "directory" that holds tags and per-core tracking state, but not data blocks, and uses inclusion to operate like a dataless version of the previous inclusive shared cache. This design roughly resembles some systems from AMD [6].

To the first order, the communication between the directory and its private caches is the same as with the original inclusive shared cache, provided that the directory continues to be large enough to keep recalls rare. Moreover, designers now have more freedom in setting the new non-inclusive shared cache configuration to trade off cost and memory traffic. Although the directory tracking state is the same as with an inclusive shared cache (the total directory size is proportional to the sum of private cache sizes), the storage impact is more significant because: (1) the directory must now also include tags (that were there for free in the original inclusive shared cache), and (2) the relative overhead gets larger if hardware designers opt for a smaller shared cache.

To be concrete, let S1 be the sum of private cache sizes, S2 be the shared cache size, D be the directory entry size relative to the size of a private cache block and tag, and R be the ratio of the number of directory entries to the total number of private cache blocks. R should be greater than 1 to keep recalls rare (Section 6). Directory storage adds R×S1×D to cache storage S1+S2 for a relative overhead of (R×D)/(1+S2/S1). Assume that R=2 and D=(16b+32b)/(32b+512b) for sharing bits for 16 cores and 32b tag+state (*e.g.*, 8MB, 16-way associative, 64-byte blocks, 48b physical addresses). If S2/S1 is 8, as in Core i7, then directory storage overhead is only 2%. Shrinking S2/S1 to 4, 2, and 1 increases relative overhead to 4%, 6%, and 9%, respectively.

The use of hierarchy adds another level of directory and an L3 cache. Without inclusion, the new directory level must point to an L2 bank if a block is either in the L2 bank or its co-located directory. For cache size ratio Z = S3/S2 = S2/S1 = 8, the storage overhead for reaching 256 cores is 2%. Shrinking Z to 4, 2, or 1 at most doubles the relative overhead to 5%, 10%, and 18%, respectively. Furthermore, such overheads translate into relatively lower overheads in terms of overall chip area, because caches are only part of the chip area. Overall, we find that directory storage is still reasonable when the cache size ratio Z > 1.

10. Summary, Caveats, and Criticisms

In this paper, we describe a coherence protocol based on a combination of known ideas to show that the costs of on-chip coherence grow slowly with core count. The design uses a hierarchy of inclusive caches with embedded coherence state whose tracking information is kept precise with explicit cache replacement messages. Using amortized analysis, we show that for every cache miss request and data response, the interconnection network traffic per miss is independent of the number of cores and thus scales. Embedding coherence state in an inclusive cache hierarchy keeps coherence's storage costs small (*e.g.*, 512 cores can be supported with 5% extra cache area with two cache levels or 2% with three levels). Coherence adds no latency to cache hits, off-chip accesses, and misses to blocks not actively shared; miss latency for actively shared blocks is higher, but the ratio of the latencies for these misses are tolerable today and independent of the number of cores. Energy overheads of coherence are correlated with traffic and storage, so we find no reason for energy overheads to limit the scalability of coherence. Extensions to a non-inclusive shared cache show larger but manageable storage costs when shared cache size is larger than the sum of private cache size. With coherence's costs shown to scale, we see on-chip coherence as here to stay for the programmability and compatibility benefits it provides.

Nevertheless, there are limitations and potential criticisms of this work. First, this paper does not perform detailed architectural simulations with specific benchmarks or consider difficult-to-model queuing impacts due to cache and interconnect contention. We instead show that coherence's per-miss traffic is independent of the miss pattern and number of cores. Although less precise than detailed simulation, our results are actually more robust as they are not limited to the specific benchmarks studied. Furthermore, we describe our protocol as an existence proof of a scalable coherence protocol, but we do not claim it to be the best. To this more modest end, less precision is required.

Second, we did not compare against multicore chips without caches or without a shared address space. Although these latter approaches have been successful in high-performance computing, etc., they have not been common in mainstream multicore systems. Given that coherence's costs can be kept low and that we already have operating systems that use hardware coherence to scale to many cores [2, 3], there does not appear to be a need to abandon coherence. Thus, we anticipate that alternatives to cache-coherent shared memory will continue to exist and thrive in certain domains, but that on-chip coherence will continue to dominate mainstream multicore chips. Furthermore, coherent systems can support legacy algorithms from these other domains, because any program that works for scratchpad systems (*e.g.*, Cell processor) or message passing systems (*e.g.*, MPI cluster) can easily map to a shared memory system with caches.

Third, we are aware of the complexity challenge posed by coherence. We do not underestimate the importance of managing com-

plexity, but the chip design industry has a long history of successfully managing complexity. Many companies have sold many systems with hardware cache coherence. Designing and validating the coherence protocols in these systems is not easy, but industry has overcome—and continues to overcome—these challenges. Moreover, the complexity of coherence protocols does not necessarily scale up with increasing numbers of cores. Adding more cores to an existing multicore design has little impact on the conceptual complexity of a coherence protocol, although it may increase the amount of time necessary to validate the protocol. However, even the validation effort may not pose a scalability problem; research has shown that it is possible to design hierarchical coherence protocols that can be formally verified with an amount of effort that is independent of the number of cores [19]. Furthermore, the complexity of the alternative to hardware coherence—that is, software implemented coherence—is non-zero. As when assessing hardware coherence's overheads (storage, traffic, latency, and energy) one must be careful not to implicitly assume that the alternative to coherence is free. Forcing software to use software-managed coherence or explicit message passing does not remove the complexity, but rather shifts the complexity from hardware to software.

Fourth, we assumed a single-chip (socket) system and did not explicitly address chip-to-chip coherence used in today's multi-socket servers. The same sort of tagged tracking structures can be applied to small-scale multi-socket systems [6], essentially adding one more level to the coherence hierarchy. Moreover, providing coherence across multi-socket systems may become less important, because single-chip solutions solve more needs and "scale out" solutions are required in any case, e.g., for data centers, but that is an argument for a different paper.

Fifth, even if coherence itself scales, we do not address other issues that may prevent practical multi-core scaling, such as the scalability of the on-chip interconnect or the critical problem of software non-scalability. Despite advances in scaling operating systems and applications, there remain many applications that do not (yet) effectively scale to many cores, and this paper does not directly improve the situation. Nevertheless, we show that on-chip hardware coherence can scale gracefully, thereby freeing application and system software developers from re-implementing coherence (*e.g.*, knowing when to flush and re-fetch data) or orchestrating explicit communication via message passing.

Conclusion: *On-chip coherence can scale gracefully and it enables programmers to concentrate on what matters for parallel speedups: finding work to do in parallel without undo communication and synchronization.*

Acknowledgments

For feedback, we thank the anonymous referees, James Balfour, Colin Blundell, Derek Hower, Steve Keckler, Alvy Lebeck, Steve Lumetta, Steve Reinhardt, Mike Swift, and David Wood. This material is based upon work supported by the National Science Foundation (CNS-0720565, CNS-0916725, CNS-1117280, CCF-0644197, CCF-0905464, and CCF-0811290), Sandia/DOE (MSN123960/DOE890426), and SRC (2009-HJ-1881). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Sandia/DOE, or SRC. Hill has a significant financial interest in AMD.

References

[1] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[3] R. Bryant. Scaling Linux to the Extreme. In *Proceedings of the Linux Symposium*, 2004.

[4] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *IEEE Micro*, 31(2), March/April 2011.

[5] B. Choi, R. Komuravelli, H. Sung, R. Bocchino, S. Adve, and V. Adve. DeNovo: Rethinking Hardware for Disciplined Parallelism. In *Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30:16–29, 2010.

[7] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: Efficient and Scalable CMP Coherence. In *Proceedings of the 17th Symposium on High-Performance Computer Architecture*, Feb. 2011.

[8] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993.

[9] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[10] J. Howard *et al.* A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*, 2010.

[11] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010.

[12] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[13] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: An Adaptive Hybrid Memory Model for Accelerators. *IEEE Micro*, 31(1), January/February 2011.

[14] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[15] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, March/April 2010.

[16] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-threaded, power-efficient, SPARC SOC. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, Nov. 2007.

[17] R. Singhal. Inside Intel Next Generation Nehalem Microarchitecture. In *Hot Chips 20*, 2008.

[18] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool, 2011.

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