DOI:10.1145/2209249.2209269

On-chip hardware coherence can scale gracefully as the number of cores increases.

BY MILO M.K. MARTIN, MARK D. HILL, AND DANIEL J. SORIN

Why On-Chip Cache Coherence Is Here to Stay

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 reduce the average memory latency and memory traffic. Caches are thus beneficial, but private caches lead to the possibility of cache incoherence. The mainstream solution is to provide shared memory and prevent incoherence through a hardware cache coherence protocol, making caches functionally invisible to software. The incoherence problem and basic hardware coherence solution are outlined in the sidebar, "The Problem of Incoherence," page 86.

Cache-coherent shared memory is provided by mainstream servers, desktops, laptops, and mobile devices and is available from all major vendors, including AMD, ARM, IBM, Intel, and Oracle (Sun). Cache coherence has come to dominate the market for technical, as well as for legacy, reasons. Technically, hardware cache coherence provides performance generally superior to what is 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, written for cache-coherent shared-memory systems.

Although coherence delivers 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 arises from claims of unscalable storage and interconnection network traffic and concerns over latency and energy. Such claims lead to the conclusion that cores in future multicore chips will not employ coherence but instead communicate with software-managed coherence, explicitly managed scratchpad memories, and/or message passing (without shared memory).

Here, we seek to refute this conventional wisdom by presenting one way to scale on-chip cache coherence in which coherence overheads—traffic, storage, latency, and energy—grow slowly with core count and are similar to the overheads deemed acceptable in today's systems. To do this, we synergistically combine known techniques, including shared caches augmented

» key insights

- **The approach taken here scales on-chip hardware cache coherence to many cores with bounded traffic, storage, latency, and energy overheads.**
- **For the same reason system designers will not abandon compatibility for the sake of eliminating minor costs, they likewise will not abandon cache coherence.**
- **Continued coherence support lets programmers concentrate on what matters for parallel speedups: finding work to do in parallel with no undo communication and synchronization.**

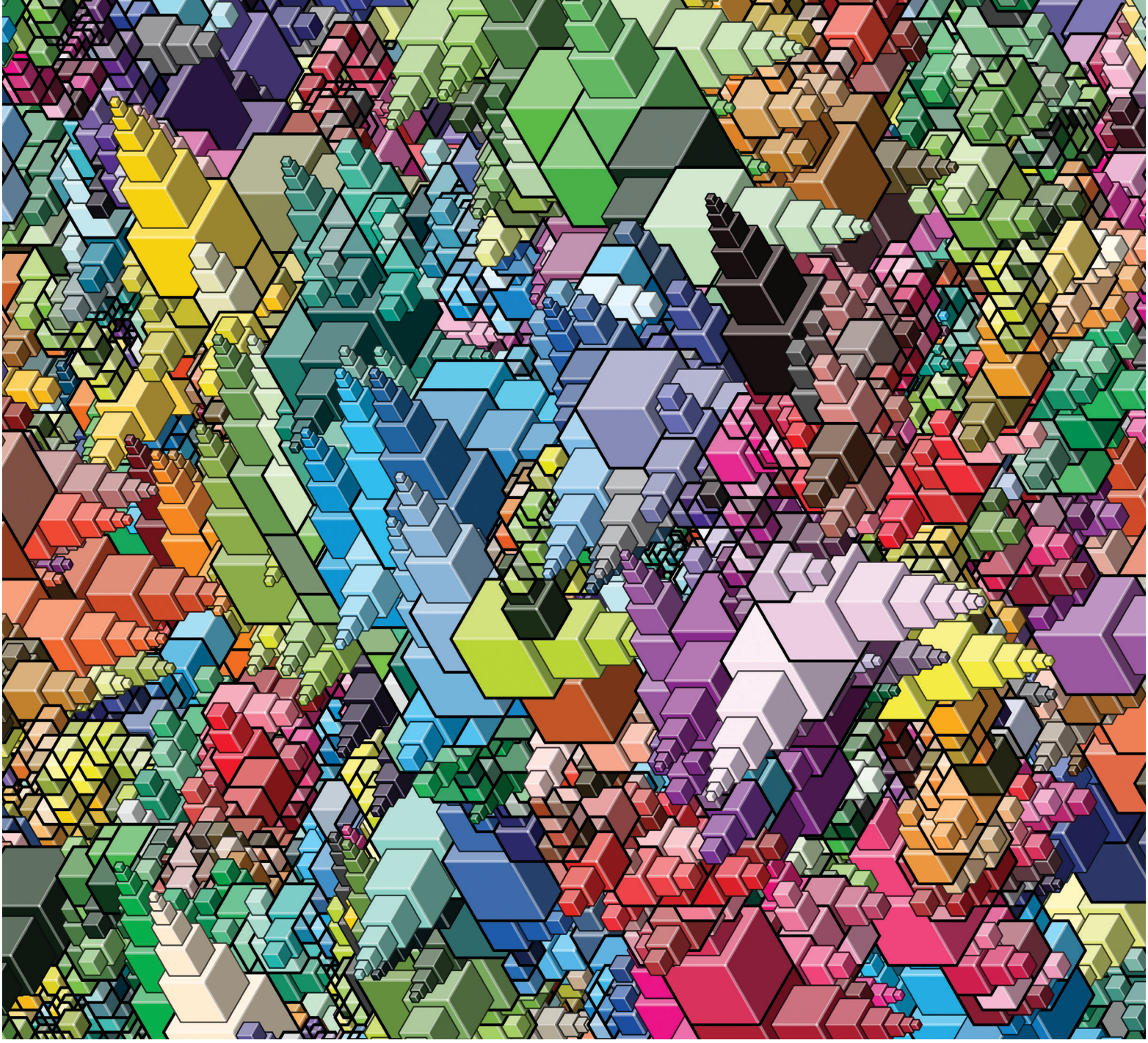


ILLUSTRATION BY DAVE BOLLINGER

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 (such as 2% of total cache size for even 512 cores). We find negligible energy and latency overheads for cache misses to data that is not actively shared; our analysis suggests the relative miss penalty and energy overheads of accessing shared data do not increase appreciably with increasing core count. Consequently, on-chip coherence is here to stay. Computer systems tend not to abandon compatibility to eliminate small costs (such as those found for scaling coherence). In particular, system designers will not likely replace conventional operating systems now that they have been shown to scale using cache coherence.[2,3] Boyd-Wickizer et al.[2] concur, writing "There is no scalability reason to give up on traditional operating system organizations just yet."

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

Our claim for the continued viability of on-chip cache coherence does not imply other communication paradigms will disappear. There will still be applications for which message pass-

ing is appropriate (such as to scale-out high-performance computing) or for which incoherent scratchpad memories are appropriate (such as real-time systems), and so those communication paradigms will persist. However, they are likely to continue to coexist with on-chip cache-coherent shared memory for the foreseeable future.

Cache Coherence Today

Before investigating the issues involved in coherence's future, we first describe today's cache coherence protocols. Rather than survey coherence protocol design, we focus on one concrete coherence protocol loosely based on the on-chip cache coherence protocol used by Intel's Core i7,[17] which represents the state of the art and can scale to a moderate number of cores (such as 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 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; this system model is outlined in the sidebar figure.

To make our analysis simpler and more concrete, we assume for now that the shared cache is inclusive with respect to all the private caches. Inclusion means, at all times, the shared cache contains a superset of the blocks in the private caches. Intel's Core i7 is an 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 later.

With inclusion, cache coherence can be maintained with a coherence protocol that tracks copies of blocks in private caches using state embedded in the shared cache; that is, each block in the shared cache is augmented with a few bits of coherence state (such as 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 outlined in the sidebar figure, 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.

We anticipate alternatives to cache-coherent shared memory will continue to exist and thrive in certain domains but that on-chip coherence will continue to dominate in mainstream multicore chips.

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 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 is 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 invalidate their copies of the block. If no cores are caching the block, then a request has the negligible overhead of looking up only the coherence state bits in the shared cache. This protocol is essentially a directory-based cache coherence 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 evicts a block for which some per-core tracking bits are set, the shared cache first issues a recall request (also known as a back-invalidation or notification) to any core 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, helping explain why current Intel systems resemble it. This protocol avoids the need for a snooping bus and avoids broadcasting; communication involves only point-to-point messages. Because the protocol embeds the per-core tracking bits in the shared cache, it avoids adding additional structures dedicated solely to coherence. For small-scale systems (such

as four cores to 16 cores), the storage cost is negligible; a 16-core system adds just 16b for each 64B 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 limited 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 for architects is: Does this system model scale to future manycore chips?

Scalability

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

We view a coherent system as "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 (such as memory and on-chip interconnection network) and software (operating system and applications) components.

Here, we examine five potential concerns when scaling on-chip coherence:

- Traffic on the on-chip interconnection network;
- Storage cost for tracking sharers;
- Inefficiencies caused by maintaining inclusion (as inclusion is assumed by our base system);
- Latency of cache misses; and
- Energy overheads.

The following five sections address these concerns in sequence and present our analysis, indicating that existing design approaches can be employed such that none of these concerns would present a fundamental barrier to scaling coherence. We then discuss extending the analysis to noninclusive caches and address some caveats and potential criticisms of this work.

Concern 1: Traffic

Here we tackle the concerns regarding the scalability of coherence traffic on the on-chip interconnection network. To perform a traffic analysis, we consider for each cache miss how many bytes must be transferred to obtain and relinquish the given block. We divide the analysis into two parts: in the absence of sharing and 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 relatively few cores, it will continue to be acceptable as the number of cores scales up. We conclude with a discussion of how coherence's per-miss traffic compares to that of a system without coherence.

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: a request from the private cache to the shared cache and 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 written explicitly 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 implementation of inclusive caching, as discussed later in the section on maintaining inclusion.) 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 (enough to, say, hold a 8B address) and can occur only subsequent to cache misses, transferring, say, a 64B cache block. Coherence's additional traffic per miss is thus modest and, most important, independent of the number of cores. Based on 64B cache blocks, the table here shows that coherence's traffic is 96B/miss for clean blocks and 160B/miss for dirty blocks.

With sharing. In a coherent system, when a core reads a block that is shared, the coherence protocol might need to forward the request but to at most only one core; 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 nonscalability 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 precisely, the overall traffic per miss of cache coherence is independent of the number of cores; the storage cost of precise tracking is addressed later, in the section on storage.

Consider an access pattern in which a block is read by all cores and then written by one core. The writer core is indeed forced to send an invalidation message to all cores, and each core will respond with an acknowledgment message, or a cost of $2N$ messages for N sharers. However, such an expensive write operation can occur only after a read miss by each of the cores.

Traffic cost of cache misses.

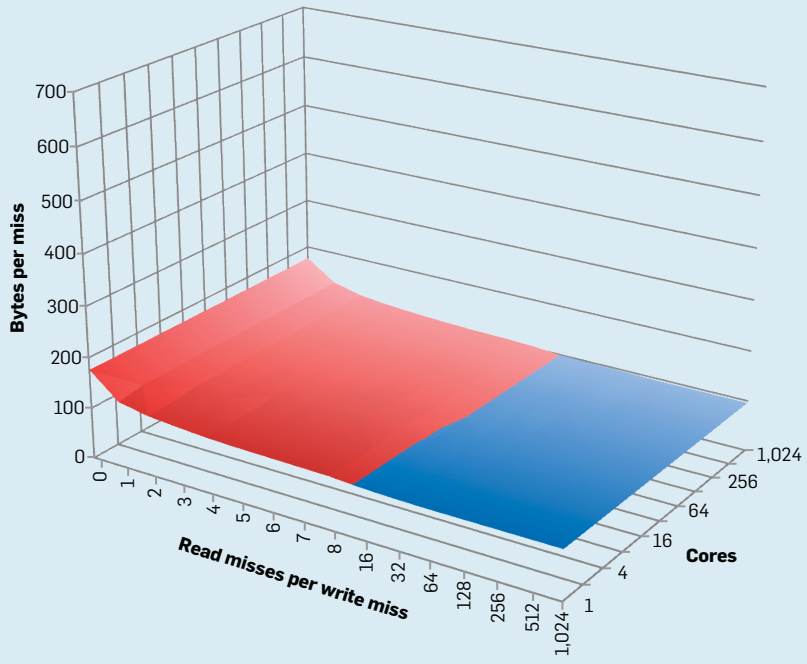
To calculate traffic, we must assume values for the size of addresses and cache blocks (such as 8B physical addresses and 64B cache blocks). Request and acknowledgment messages are typically short (such as 8B) because they contain mainly a block address and a message type field. A data message is significantly larger because it contains both an entire data block plus a block address (such as 64B + 8B = 72B).

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

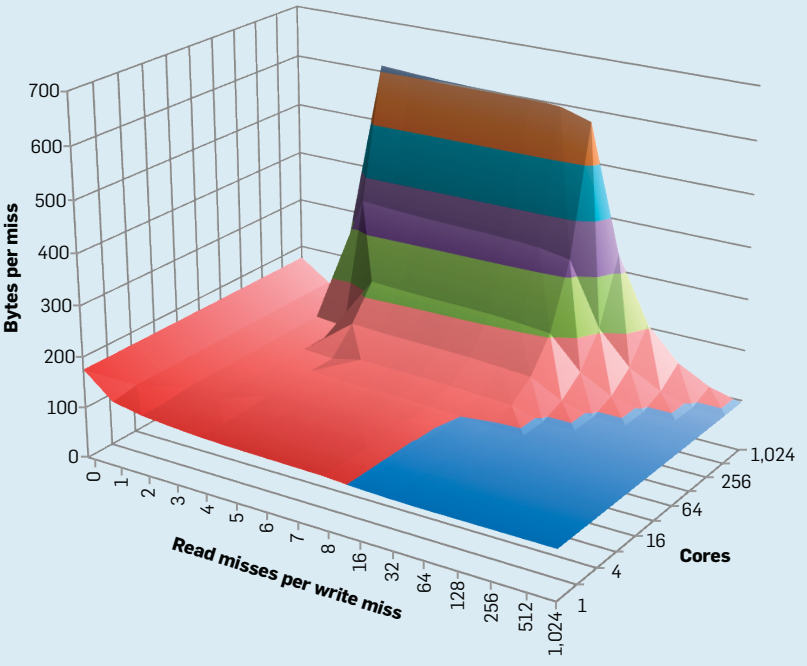
More generally, for every write that invalidates *N* caches, the write must have been preceded by *N* read misses. The traffic cost of a read miss is independent of the number of cores; a read miss is forwarded to a single core, at most. Thus, through amortized analysis, the overall average traffic per miss is constant; a write miss that causes *N* messages can occur at most only once every *N*th miss.

In support of this general analysis, Figure 1a 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 Figure 1a) has the highest traffic per miss because all blocks are dirty. Traffic per miss is independent of the number of cores because 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 in Figure 1a), the average traffic per miss actually decreases slightly because fewer writes lead to fewer dirty blocks. More important, 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.

Figure 1. Communication traffic for shared blocks.



(a) With exact tracking of sharers



(b) With inexact tracking of sharers

Traffic overhead of coherence. We have shown that coherence's per-miss traffic scales because it is independent of the number of cores. We now consider coherence's traffic overhead per miss with respect to a hypothetical design with caches but no hardware coherence (such as when software knows precisely 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 assuming that coherence does not change the number of cache misses. However, this assumption is potentially compromised by false sharing and inefficient synchronization, which can cause nonscalable increases in the number of cache misses. Both of these phenomena are well-known challenges with well-known techniques for their mitigation; we cannot completely eliminate their impact nor cleanly incorporate them into our intentionally simplistic models.

The table lists the traffic per miss for this system without coherence. We now compare this traffic 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 none of these three extra messages contain the data block, and such "control" messages are significantly smaller than "data" messages.

Coherence's overhead is small, bounded, and—most important—independent of the number of cores. Based on 64B cache blocks, the table shows that coherence adds a 20% traffic overhead for clean blocks and a 5% overhead for dirty blocks.

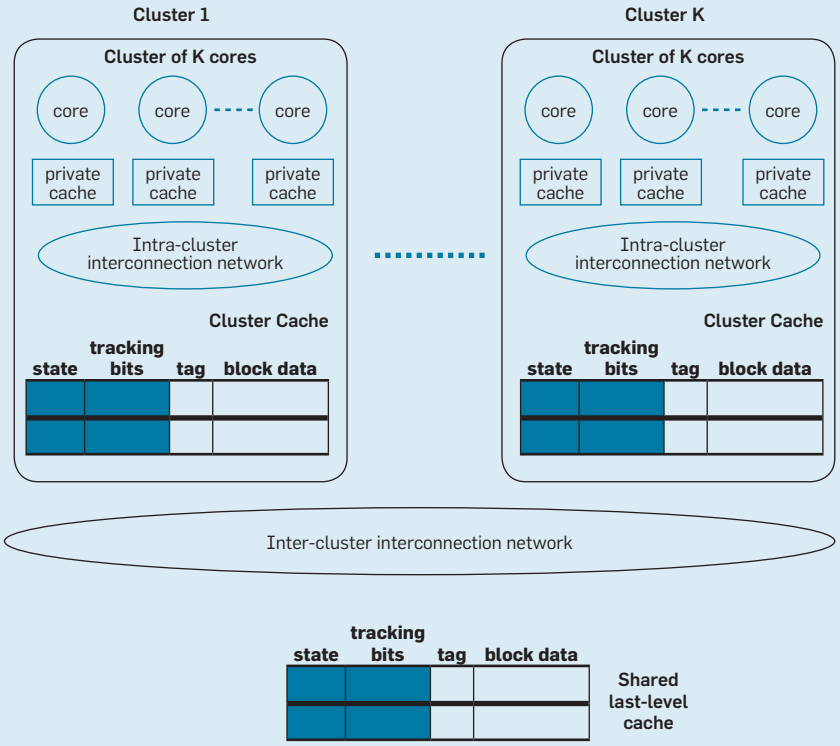
Conclusion. Coherence's interconnection network traffic per miss scales when precisely tracking sharers.

Concern 2: Storage

The scalable per-miss traffic result assumed a precise tracking of sharing state in the shared cache, requiring N bits of state for a system with N cores. This assumption leads to the reasonable concern that such an increase in tracking state for systems with more cores could pose a fundamental barrier to scalability. Here, we show that the storage cost scales gracefully by quantifying the storage cost and describing two approaches for bounding this cost: the first is the traditional use of inexact encoding of sharers,[1,8] which we discard in favor of often-overlooked use of 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 focuses on additional storage in the shared cache.

Conventional approach: Inexact encoding of sharers. The conventional approach to limiting storage—inexact encoding of sharers—can work well but has poor worst-case behavior. It represents a conservative superset of sharers using fewer bits than one bit per potential sharer and was well-studied in the early 1990s.[1,8] As a concrete example, the SGI Origin 2000[14] used a fixed number of bits per block, regardless of the number of cores. For small systems, the Origin used these bits as a bit-vector that tracks sharers exactly. For larger systems, the Origin alternated between two uses of these tracking bits. If there were only a few sharers, the bits would be used as a limited number of pointers (each of which requires log_2N bits to encode) that can exactly track sharers. If the number of sharers exceeds this limited number of pointers, the Origin would use the bits as an inexact, coarse-vector encoding, in which each bit represents multiple cores. Though the storage can be bounded, the traffic of such schemes could suffer due to unnecessary invalidations.

Figure 2. Hierarchical system model; additions for coherence are shaded.



To quantify the traffic impact of such inexact encodings, Figure 1b shows the result of applying the analysis from the previous section on traffic when using the Origin's inexact encoding scheme to bound the storage at 32b per block in the shared cache (approximately 6% overhead for 64B blocks). When the 32b 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 Figure 1b), the sharers are encoded exactly, resulting in the same traffic-per-miss as the exact encoding. When the number of sharers is large (far right of Figure 1b), the write invalidations must be sent to all cores (independent of 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 1,024 cores, the spike reaches almost six times the traffic of the exact encoding cases. Though conventional wisdom might have predicted an even larger traffic spike for 1,024 cores, we next describe an alternative design that eliminates any such spike in traffic.

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 through 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 chipwide shared cache is likely large. As such, many-core systems,[4,16] GPUs,[15] and proposed manycore 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 between two pipelines, each with four threads. NVIDIA's Fermi GPU[15] clusters 32 execution pipelines into a "shared multiprocessor." In AMD's Bulldozer architecture,[4] each pair of cores has per-core private L0 caches and shares an L1 cache. Such

systems fill the gap between a private cache and a large, distributed shared cache, allowing the cluster cache to deliver faster access to data shared within the cluster. An additional benefit is that coherence requests may be satisfied entirely within the cluster (such as by a sibling node caching the block) that can be significant if the software is aware of the hierarchy.

The same techniques described earlier—inclusion, integrating tracking state with caches, recall messages, and explicit eviction notifications—are straightforward to apply recursively to provide coherence across a hierarchical system. Rather than just embed 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 (see Figure 2) 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 nonhierarchical systems; 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 layers of cache lookups—but also 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 this block, the cluster cache lacks write permission for the block, so it forwards it to the shared last-level cache. The shared last-level cache then sends an invalidation message to each cluster (not to each core), triggering 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 the protocol 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 the previous section on traffic 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) consisting of K clusters of K cores each ($K^2 = C$ total cores). Each cluster cache is inclusive with respect to all private caches within the cluster, and the shared last-level cache is inclusive with respect to all cluster caches. Each cluster cache block uses one bit for each of the K private caches it includes, plus a few bits of state. Likewise, each shared last-level cache block consumes a 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 consisting of K level-1 clusters, with each level-1 cluster consisting of K cores. This system has $C = K^3$ cores and a storage cost proportional to cube root of C.

In Figure 3, we plot coherence's storage overhead (coherence's storage as a fraction of the total cache storage) in terms of the bits needed to provide precise tracking of sharers, for conventional flat (nonhierarchical) systems, 2-level systems, and 3-level systems. As a very large example, a 1,024-core 2-level system might have 32 clusters of 32 cores, thus 32b per 64B cache block at each level, which is just 6%. An extreme 4,096-core 3-level system would have 16 clusters, each with 16 subclusters of 16 cores, with storage overhead of only 3%.

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

Concern 3: Maintaining Inclusion

In the system model covered here, we initially choose to 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, and is why we use this design point. Inclusion also simplified our earlier analysis of communication and storage.

Inclusion requires that if a block is cached in any private cache, it must also be cached in the shared cache. When the shared cache evicts a block with nonempty 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 makes recall messages vanishingly rare.

Recalls occur when the shared cache is forced to evict a block with one or more sharers. To reduce the number of recalls, the shared cache always chooses to evict nonshared blocks over shared blocks. Because the capacity of an inclusive shared cache often exceeds the aggregate capacity of the private caches (for example, the ratio is 8 for the four-core Intel Core i7 with 8MB shared cache and four 256KB second-level private caches), it is highly likely that a nonshared block will be available 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 in some systems does not always know when a private cache has evicted the block. In such systems, clean blocks (those not written during their lifetime in the cache) are evicted

silently from the private caches, 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 protocol uses explicit eviction notifications on clean-exclusive block replacements to improve sharer state encoding.[6] If such eviction notifications occur on every cache eviction, the protocol enables the shared cache to maintain precise up-to-date tracking of private caches that hold each block, transforming the tracking information from conservative to exact. When an eviction decision does occur, the shared cache thus knows which blocks are no longer being cached and likely have a choice to evict a nonshared block to avoid a recall. However, this precision comes at a cost in the form of increased traffic for evictions of clean blocks, the overhead of which was already included in the traffic analysis.

Explicit eviction notifications can potentially eliminate all recalls, but only if the associativity, or 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 nonshared block to evict, if it has exact sharing information, it is guaranteed to find a nonshared 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 is prohibitive, as reported by Ferdman et al.[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.

Rather than eliminate all recalls, we focus on a system in which recalls are possible but rare. To estimate the effect 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 that prior work found accurately approximates conflict rates.[9] We also pessimistically assumed no data sharing among the cores that would reduce the inclusive capacity pressure on the shared cache.

Fortunately, recalls can be made rare in the expected design space. Figure 4 shows the recall rate, or 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 underprovisioned shared cache. As the size of the shared cache increases, the recall rate drops quickly. When the capacity ratio reaches four times, even an eight-way set-associative shared cache keeps

Figure 3. Storage overhead in shared caches.

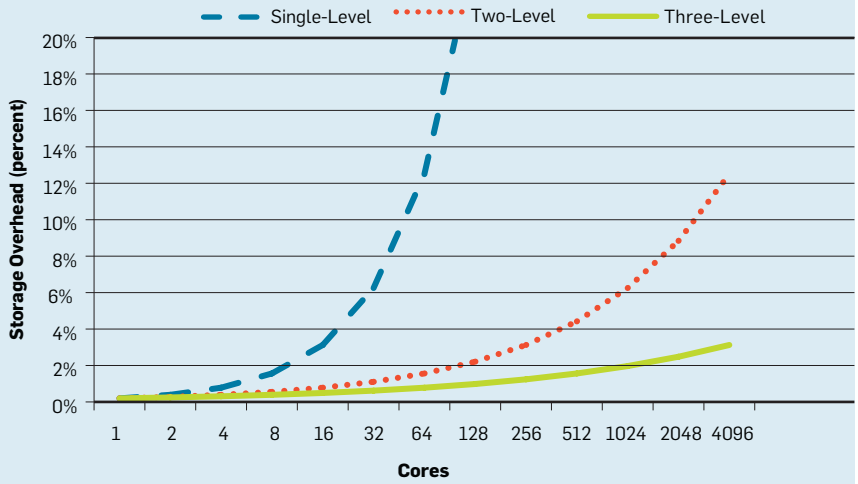
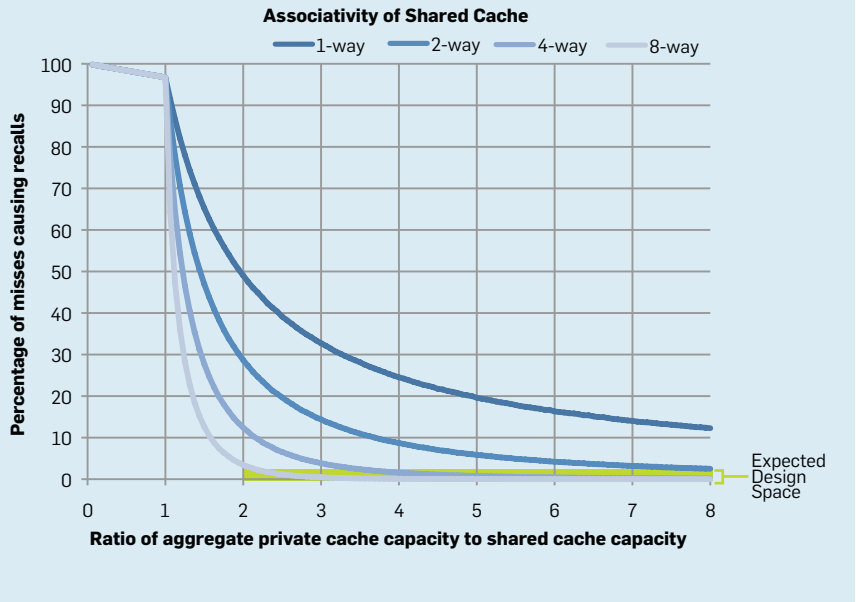


Figure 4. Likelihood a shared cache miss triggers a recall.



The Problem of Incoherence

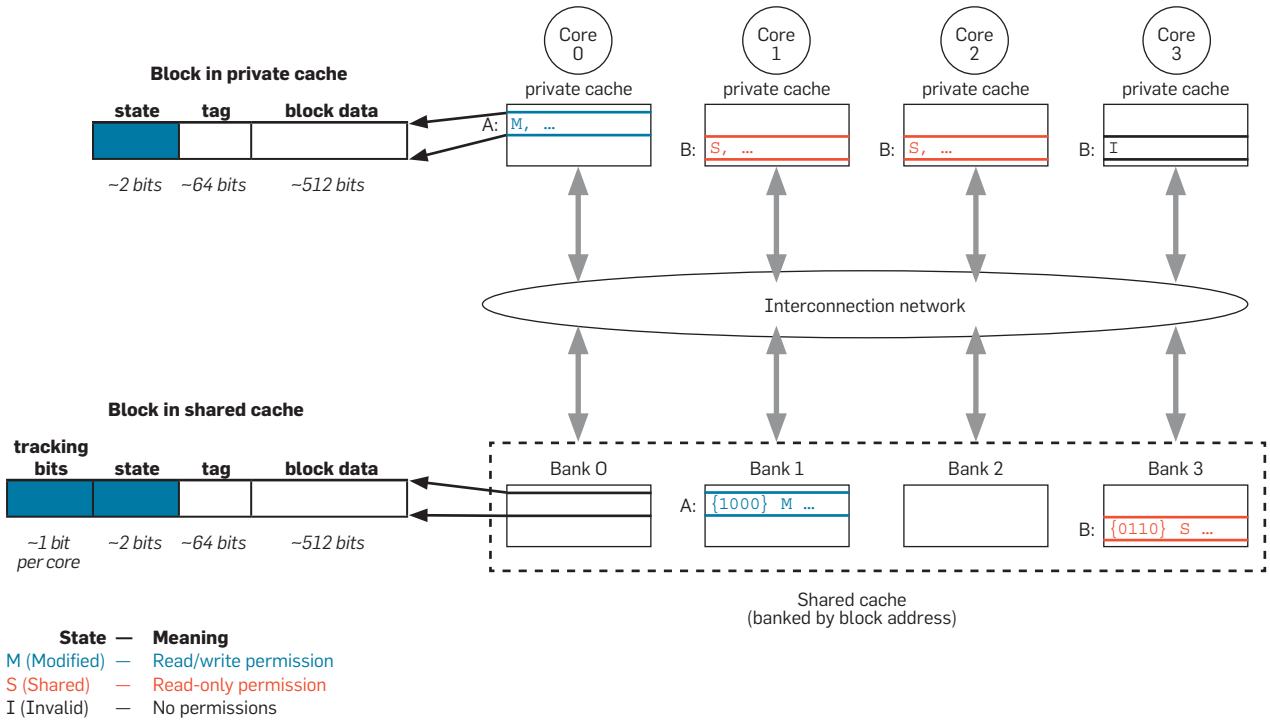
Incoherence. To illustrate the problem of incoherence, consider the multiple cores and corresponding private caches in the upper-right of the figure here. If core 1 writes the block labeled *B* by updating its private cache only, subsequent reads by core 2 would see the old value indefinitely. This incoherence 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. Though there are many possible coherence protocols, all maintain coherence by ensuring the single-writer, multiple-reader 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 figure here outlines 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 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 integrated into the cache controllers. The section on cache coherence today presents a protocol (and describes the rest of the diagram); for more, see Sorin et al.[18]

System model; additions for coherence are shaded.



the recall rate below 0.1%. For comparison, the Intel Core i7 has a 16-way set-associative cache with eight times capacity ratio. 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. Chip architects can design a system with an inclusive shared cache with negligible recall rate, and thus can efficiently embed the tracking state in the shared cache.

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 (see the sidebar figure), 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 (such as off-chip memory).

In a coherent system with private caches and a shared cache, four cases are worth consideration with regard to miss latency: a hit in the private cache; a direct miss in which the shared cache can fully satisfy the request, that is, to a block not cached in other private caches; an indirect miss, in which the shared cache must contact one or more other caches; and a miss in the shared cache, incurring a long-latency access to off-chip memory. Coherence adds no latency to perhaps the two most performance-critical cases: private cache hits (the first case) and off-chip misses (the fourth case). 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.

However, indirect misses do incur the extra latency of sending a message on the on-chip interconnection network to the specified private cores. Such messages are sent in parallel, and responses are typically sent directly to the original requester, resulting in 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 from 1.5 to two times 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, in part because indirect misses are generally in the minority for well-tuned software. The ratio also indicates scalability, as the 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 all misses (even in a non-coherent system) roughly proportionally, keeping the ratio largely unchanged. Moreover, if latency is still deemed 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 latency ratio for a system with more layers of hierarchy. Though the effect of hierarchy may hurt absolute latency (such as due to additional layers of lookup), we see no reason why hierarchy should significantly affect 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 providing a closer mid-size cache that allows faster sharing within a cluster and reducing the number of longer-latency accesses to the chipwide distributed shared last-level cache. Modeling the full effect of hierarchy on latency (and traffic) is beyond the reach of the simple models we use here.

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

Concern 5: Energy

Though a detailed energy analysis is perhaps not as straightforward as the analyses we have reported here, we can use these analyses to support the conclusion that the energy cost of coherence is also not a barrier to scalability. Energy overheads generally 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 shown that interconnect traffic and message count per-miss do not increase with the number of cores, indicating the protocol state transitions and number of extra cache lookups are likewise bounded and scalable.

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

Furthermore, many energy-intensive parts of the system—the cores themselves, the cache data arrays, off-chip DRAM, and storage—are largely unaffected by coherence, so energy overheads incurred by coherence are relatively smaller when weighed against the context of the overall system.

Conclusion. Based on these traffic and storage scalability analyses, we find no reason the energy overheads of coherence must increase with the number of cores.

Non-Inclusive Shared Caches

So far we have assumed an inclusive shared cache, like that of Intel's Core i7, but this choice is not universal. Rather than require 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 for coherence.

A non-inclusive system can retain the coherence benefits of an inclusive shared cache by morphing it into two structures: a new noninclusive shared cache that holds tags and data, but not tracking state, and is free to be of any size and associativity; and 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 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. Though the directory-tracking state is the same as with an inclusive shared cache (total directory size is proportional to the sum of private cache sizes), the storage effect is more significant because the directory must also include tags (there for free

in the original inclusive shared cache), and the relative overhead becomes larger if the hardware designer opts for a smaller shared cache.

To be concrete, let $S1$ be the sum of private cache sizes, $S2$ the shared cache size, D the directory entry size relative to the size of a private cache block and tag, and R, 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, as discussed earlier in the section on maintaining inclusion. Directory storage adds $R \times S1 \times D$ to cache storage $S1+S2$ for a relative overhead of $(R \times D)/(1+S2/S1)$. Assume that $R=2$ and $D=64b/(48b+512b)$. If $S2/S1$ is 8, as in Core i7, then directory storage overhead is only 2.5%. Shrinking $S2/S1$ to 4, 2, and 1 increases relative overhead to 4.6%, 7.6%, and 11%, 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 in its co-located directory. For cache size ratio $Z = S3/S2 = S2/S1 = 8$, the storage overhead for reaching 256 cores is 3.1%. Shrinking Z to 4, 2, or 1 at most doubles the relative overhead to 6.5%, 13%, or 23%, respectively. Furthermore, such storage overheads translate into relatively lower overheads in terms of overall chip area, as caches are only part of the chip area. Overall, we find that directory storage is still reasonable when the cache size ratio $Z > 1$.

Forcing software to use software-managed coherence or explicit message passing does not remove complexity but rather shifts complexity from hardware to software.

Caveats and Criticisms

We have described a coherence protocol based on known ideas to show the costs of on-chip coherence grow slowly with core count. Our 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 have shown 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; for example, 512 cores can be supported with 5% extra cache area with two cache levels or 2% with three levels. Coherence adds negligible 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 is 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 expect on-chip coherence is here to stay due to the programmability and compatibility benefits it delivers.

Nevertheless, this work has limitations and potential criticisms. First, we did not include detailed architectural simulations with specific benchmarks or consider difficult-to-model queuing effects due to cache and interconnect contention. Instead, we showed that coherence's per-miss traffic is independent of the miss pattern and number of cores. Though less precise than detailed simulation, our results are more robust, as they are not limited to the specific benchmarks studied. Furthermore, we described our protocol as an existence proof of a scalable coherence protocol but do not claim it is the best. To this more modest end, less precision is required.

Second, we did not compare our protocol against multicore chips without caches or without a shared address space. Though these approaches have been successful in high-performance computing, they are not common in mainstream multicore systems. Given that coherence's costs can be kept low and that some operating systems use hardware coherence to scale to many cores,[2,3] we find no compelling reason to abandon coherence. We thus anticipate alternatives to cache-coherent shared memory will continue to exist and thrive in certain domains but that on-chip coherence will continue to dominate in mainstream multicore chips. Furthermore, coherent systems can support legacy algorithms from these other domains, as any program that works for scratchpad systems (such as the Cell processor) or message passing systems (such as an MPI clus-

ter) maps easily to a shared memory system with caches.

Third, we are aware of the complexity challenge posed by coherence and do not underestimate the importance of managing complexity but also that the chip-design industry has a long history of managing complexity. Many companies have sold many systems with hardware cache coherence. Designing and validating the coherence protocols in them is not easy, but industry 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 effect on the conceptual complexity of a coherence protocol, though it may increase the amount of time necessary to validate the protocol.

However, even the validation effort may not pose a scalability problem; research shows 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—software implemented coherence—is non-zero. As when assessing hardware coherence's overheads—storage, traffic, latency, and energy—chip architects must be careful not to implicitly assume 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 in today's multisocket servers. The same sort of tagged tracking structures can be applied to small-scale multisocket systems,[6] essentially adding one more level to the coherence hierarchy. Moreover, providing coherence across multisocket systems may become less important, because single-chip solutions solve more needs, and "scale out" solutions are required in any case (such as for data centers), but that is an argument for another article.

Finally, even if coherence itself scales, we did not address other issues that might prevent practical multicore scaling, such as die-area limitations, scalability of the on-chip interconnect, and critical problems of software non-scalability. Despite advances in scaling operating systems and applications, many applications do not (yet) effectively scale to many cores. This article does not improve that situation. Nevertheless, we have shown that on-chip hardware coherence can be made to scale gracefully, freeing application and system software developers from having to reimplement coherence (such as knowing when to flush and refetch data) or orchestrating explicit communication via message passing.

Conclusion. On-chip coherence can be made to scale gracefully, enabling programmers to concentrate on what matters for parallel speedups—finding work to do in parallel without undo communication and synchronization.

Acknowledgments

We thank James Balfour, Colin Blundell, Derek Hower, Steve Keckler, Alvy Lebeck, Steve Lumetta, Steve Reinhardt, Mike Swift, and David Wood. This material is based on work supported by the National Science Foundation (CNS-0720565, CNS-0916725, CNS-1117280, CCF-0644197, CCF-0905464, CCF-0811290, and CCF-1017650); Sandia/Department of Energy (MSN123960/DOE890426); and the Semiconductor Research Corporation (2009-HJ-1881). Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation, Sandia/DOE, or SRC. The authors have also received research funding from AMD, Intel, and NVIDIA. Hill has a significant financial interest in AMD.

References

1. Agarwal, A., Simoni, R., Horowitz, M., and Hennessy, J. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu, May). IEEE Computer Society Press, Los Alamitos, CA, 1988, 280–298.
2. Boyd-Wickizer, S. Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., and Zeldovich, N. An analysis of Linux scalability to many cores. In *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, Oct. 4–6). USENIX Association, Berkeley, CA, 2010, 1–8.
3. Bryant, R. Scaling Linux to the extreme. In *Proceedings of the Linux Symposium* (Boston, June 27–July 2, 2004), 133–148.
4. Butler, M., Barnes, L., Sarma, D.D., and Gelinas, B. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro 31*, 2 (Mar./Apr. 2011), 6–15.
5. Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S.V., Adve, V.S., Carter, N.P., and Chou, C.-T. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques* (Galveston Island, TX, Oct. 10–14). IEEE Computer Society, Washington, D.C., 2011, 155–166.
6. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., and Hughes, B. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro 30*, 2 (Mar./Apr. 2010), 16–29.
7. Ferdman, M., Lotfi-Kamran, P., Balet, K., and Falsafi, B. Cuckoo directory: Efficient and scalable CMP coherence. In *Proceedings of the 17th Symposium on High-Performance Computer Architecture* (San Antonio, TX, Feb. 12–16). IEEE Computer Society, Washington, D.C., 2011, 169–180.
8. Hill, M.D., Larus, J.R., Reinhardt, S.K., and Wood, D.A. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems 11*, 4 (Nov. 1993), 300–318.
9. Hill, M.D. and Smith, A.J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38*, 12 (Dec. 1989), 1612–1630.
10. Howard, J. et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference* (San Francisco, Feb. 7–11, 2010), 108–109.
11. Jaleel, A., Borch, E., Bhandaru, M., Steely Jr., S.C., and Emer, J. Achieving noninclusive cache performance with inclusive caches: Temporal locality-aware cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Atlanta, Dec. 4–8). IEEE Computer Society, Washington, D.C., 2010, 151–162.
12. Kelm, J.H., Johnson, D.R., Johnson, M.R., Crago, N.C., Tuohy, W., Mahesri, A., Lumetta, S.S., Frank, M.I., and Patel, S.J. Rigel: An architecture and scalable programming interface for a 1,000-core accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, June 20–24). ACM Press, New York, 2009, 140–151.
13. Kelm, J.H., Johnson, D.R., Tuohy, W., Lumetta, S.S., and Patel, S.J. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro 31*, 1 (Jan./Feb. 2011), 42–55.
14. Laudon, J. and Lenoski, D. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, June 2–4). ACM Press, New York, 1997, 241–251.
15. Nickolls, J. and Dally, W.J. The GPU computing era. *IEEE Micro 30*, 2 (Mar./Apr. 2010), 56–69.
16. Shah, M., Barren, J., Brooks, J., Golla, R., Grohoski, G., Gura, N., Hetherington, R., Jordan, P., Luttrell, M., Olson, C., Sana, B., Sheahan, D., Spracklen, L., and Wynn, W. UltraSPARC T2: A highly treaded, power-efficient SPARC SOC. In *Proceedings of the IEEE Asian Solid-State Circuits Conference* (Jeju, Korea, Nov. 12–14, 2007), 22–25.
17. Singhal, R. Inside Intel next-generation Nehalem microarchitecture. Hot Chips 20 (Stanford, CA, Aug. 24–26, 2008).
18. Sorin, D.J., Hill, M.D., and Wood, D.A. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.
19. Zhang, M., Lebeck, A.R., and Sorin, D.J. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Atlanta, Dec. 4–8). IEEE Computer Society, Washington, D.C., 2010, 471–482.

Milo M.K. Martin (milom@cis.upenn.edu) is an associate professor in the Computer and Information Science Department of the University of Pennsylvania, Philadelphia, PA.

Mark D. Hill (markhill@cs.wisc.edu) is a professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department of the University of Wisconsin-Madison.

Daniel J. Sorin (sorin@ee.duke.edu) is an associate professor in the Electrical and Computer Engineering and Computer Science Departments of Duke University, Durham, NC.