C^{3}D: Mitigating the NUMA Bottleneck via Coherent DRAM Caches

Cheng-Chieh Huang Rakesh Kumar Marco Elver
Boris Grot Vijay Nagarajan
Institute of Computing Systems Architecture
University of Edinburgh
{cheng-chieh.huang, rakesh.kumar, marco.elver, boris.grot, vijay.nagarajan}@ed.ac.uk

***Abstract*—Massive datasets prevalent in scale-out, enterprise, and high-performance computing are driving a trend toward ever-larger memory capacities per node. To satisfy the memory demands and maximize performance per unit cost, today's commodity HPC and server nodes tend to feature multi-socket shared memory NUMA organizations. An important problem in these designs is the high latency of accessing memory on a remote socket that results in degraded performance in workloads with large shared data working sets.**

This work shows that emerging DRAM caches can help mitigate the NUMA bottleneck by filtering up to 98% of remote memory accesses. To be effective, these DRAM caches must be private to each socket to allow caching of remote memory, which comes with the challenge of ensuring coherence across multiple sockets and GBs of DRAM cache capacity. Moreover, the high access latency of DRAM caches, combined with high inter-socket communication latencies, can make hits to remote DRAM caches slower than main memory accesses. These features challenge existing coherence protocols optimized for on-chip caches with fast hits and modest storage capacity. Our solution to these challenges relies on two insights. First, keeping DRAM caches *clean* avoids the need to ever access a remote DRAM cache on a read. Second, a non-inclusive on-chip directory that avoids tracking blocks in the DRAM cache enables a light-weight protocol for guaranteeing coherence without the staggering directory costs. Our design, called Clean Coherent DRAM Caches (C^{3}D), leverages these insights to improve performance by 6.4-50.7% in a quad-socket system versus a baseline without DRAM caches.

I. Introduction

High performance computers, enterprise workstations, and scale-out servers all operate on massive datasets, driving a trend in these market segments toward ever-larger memory capacities per node. Due to pin limitations and channel bandwidth constraints, a typical CPU socket is limited in the number of memory channels and DIMMs it can support. To enable high-capacity memory systems and achieve high throughput out of the expensive memory deployment, HPC, enterprise, and scale-out markets all favor multi-socket shared memory server configurations.

A well-known downside of NUMA servers is the so-called *NUMA bottleneck*, whereby an application's per-thread performance is diminished compared to a single-socket baseline due to a combination of inter-socket communication delays and bandwidth pressure on memory channels and inter-socket links. Existing OS and user-space techniques [1] aimed at reducing the NUMA bottleneck through memory and thread mapping policies are effective in avoiding load imbalance across the memory channels. However, they do not address the inherent lack of memory locality in applications with large shared data working sets. As a result, accesses to memory owned by remote sockets experience significantly higher latency than accesses to local socket's memory. Indeed, the engineering challenges associated with ensuring predictable performance in NUMA servers have recently led Facebook to abandon their dual-socket machines in favor of single-socket ones [2].

The central thesis of this paper is that DRAM caches can be effective in alleviating the NUMA bottleneck by exploiting their large capacities to uncover temporal locality beyond the reach of on-chip caches. This paper explores the design space of multi-socket DRAM caches[1] and answers two fundamental questions regarding their use in NUMA servers.

Question 1: private or shared? A *shared* DRAM cache maximizes the total cache capacity, thereby offering the largest reduction in main memory accesses (62.4-98% with 1GB of cache per socket), but does not provide a reduction in off-socket accesses. In contrast, a *private* DRAM cache per socket is able to filter 13.7-98% of all memory accesses while reducing off-socket trips by 35.9% on average. Because inter-socket communication delays can easily double the latency of a remote memory access compared to a local access [3], [4], we find that private DRAM caches have a fundamental advantage over shared designs by cutting the average memory access time (AMAT) despite the lower hit rate.

Question 2: how to provide effective coherence for private caches? Existing work on DRAM caches has solely looked at single-socket designs that, by their very nature, do not require a coherence solution. Meanwhile, we find that, compared to SRAM caches, DRAM caches have unique features that challenge conventional coherence schemes. Chief among these is the fact that access latencies of DRAM caches are roughly on par with main memory, meaning that a hit in a remote DRAM cache generally does not lower the AMAT compared to servicing the access by the main memory. Indeed,

978-1-5090-3508-3/16/$31.00

[1] In this work, we treat die-stacked and on-package designs similarly, without restricting ourselves to either one.

both snoopy and directory-based techniques have commonly-occurring pathologies whereby hits to remote sockets result in a *higher* AMAT than memory accesses. While directory-based techniques with precise sharing vectors have fewer such pathologies than snoopy schemes, we find that the massive capacity of DRAM caches incurs prohibitive directory storage costs. In principle, the directory itself could be stored in the DRAM cache; however, that would incur high access latency on the critical path.

In order to avoid the performance and cost overheads of existing coherence schemes, we exploit a critical insight that the high-latency on-critical-path accesses to remote DRAM caches can be avoided altogether by keeping the DRAM cache *clean*. The clean property guarantees that no DRAM cache has a modified copy of the block, allowing a socket's read misses to be serviced directly by the memory without needing to probe remote DRAM caches and/or their associated directory.

Our second insight is that allocating directory entries for (clean) blocks cached solely in the DRAM cache has limited value, as read requests that miss in the local socket bypass remote DRAM caches, while write requests are generally off the critical path.

Based on these insights, we propose *C^{3}D*, for *Clean Coherent DRAM Caches*. C^{3}D relies on clean private DRAM caches for fast hits in the local DRAM cache and, on a miss, complete bypassing of remote DRAM caches. Bypassing eliminates the pathologies that are inherent in dirty DRAM cache designs. C^{3}D uses a *non-inclusive* directory that avoids tracking DRAM cache blocks, thus avoiding the associated storage costs. To guarantee coherence upon write requests to untracked blocks, C^{3}D broadcasts invalidations to DRAM caches. In practice, broadcasts are relatively infrequent, adding just 5% additional inter-socket traffic versus a full directory in a quad-socket machine. To avoid broadcast traffic for writes to thread-private memory regions, C^{3}D uses a page table-based mechanism to track page ownership.

To summarize, we study contemporary parallel and server workloads in 2- and 4-socket NUMA deployments and make the following contributions:

- Identify a significant opportunity in reducing memory read accesses to remote sockets by up to 99% (71% average) via the use of private DRAM caches.
- Show that the pathologies in dirty DRAM cache designs result in an average slowdown of 12.9% for snoopy and 1.7% for directory-based coherence protocols.
- Introduce C^{3}D, which uses clean private DRAM caches to elide slow remote hits, while eschewing a huge directory at the DRAM cache level. In a 4-socket 32-core system, C^{3}D improves performance by 6.4-50.7% while reducing both inter-socket and memory traffic by an average of 35.9% and 49%, respectively, as compared to a baseline without DRAM caches.

II. Motivation

Two- and four-socket multi-core NUMA systems are the de-facto standard in today's high performance and datacenter computing to satisfy the large memory and computation requirements. Fig. 1 shows a typical multi-socket system, with each socket containing a portion of the node's physical memory and a multi-core CPU. The CPU's cache hierarchy consists of one or more per-core private cache levels and a last-level on-chip cache (LLC) shared between the cores within the socket. Globally, the LLC is private to each socket, thereby allowing local caching of L1 victims to minimize the incidence of high-latency inter-socket lookups.

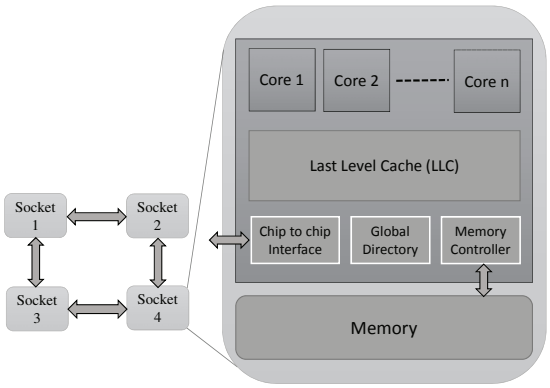


Figure 1. Contemporary NUMA System

A. The NUMA Bottleneck

A well-known problem in multi-socket shared memory systems is the so-called NUMA bottleneck, whereby memory traffic experiences degraded performance compared to a single-socket setup due to bandwidth and/or latency overheads. NUMA bandwidth overheads can arise if one or more memory interfaces become congested if a disproportionate fraction of the global memory traffic is directed toward them. Latency overheads are caused by the delay incurred in inter-socket communication on accesses to remote memory. For instance, on the Intel SandyBridge-E dual-socket system, the Intel MLC tool [5] reports a local memory latency of 60 to 70ns and remote latency of 120 to 130ns, corroborating earlier studies showing that accessing local memory is considerably faster than remote [3], [4].

Obviously, the NUMA bottleneck is not a concern if the majority of memory accesses stay within the socket issuing the request. To accomplish that, the OS community has proposed various memory mapping policies and thread placement techniques to improve the memory locality [6], [7]. However, such approaches are fundamentally limited in their effectiveness as large datasets require distribution and many applications share vast regions of data, thereby confounding locality-centric mapping strategies. For example, Dashti et al. [1] showed that in a 4-socket system, only 25-33% of memory accesses originating at a given socket are satisfied by local memory, regardless of whether the memory mapping is address-interleaved (memory pages are spread evenly across all sockets) or first touch (a memory page is placed on a socket where it is first accessed).

Our analysis of a range of contemporary parallel and server workloads on a simulated quad-socket system corroborate these

Fraction of memory accesses satisfied by remote memory					
facesim	76.6%	streamcluster	73.6%	freqmine	74.6%
fluidanimate	75.2%	canneal	75%	tunkrank	61.6%
nutch	75.2%	cassandra	75.2%	classification	75.2%

Table I
FRACTION OF MEMORY ACCESSES SATISFIED BY A REMOTE SOCKET MEMORY[2]

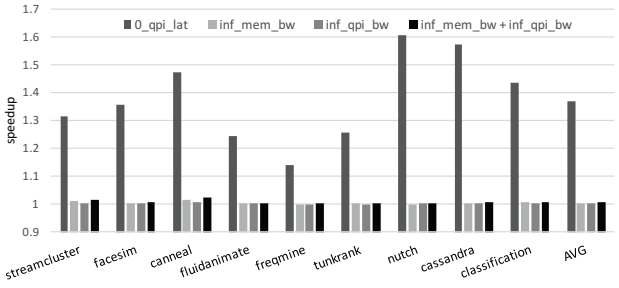


Figure 2. NUMA bottleneck analysis.

findings. As shown in Table I, the first-touch memory mapping policy is able to satisfy, on average, only 26.5%[2] of memory accesses with memory local to the requesting socket, with the majority of accesses entailing a trip to a remote socket.

B. Latency or Bandwidth?

Given the absence of affinity in the memory access streams of contemporary parallel and server workloads, we next study the root cause(s) of the NUMA bottleneck. We consider three sources of performance degradation: inter-socket communication latency (which includes QPI, on-chip interconnect, and coherence controller latencies), memory controller congestion, and QPI congestion. We model the effect that each of these has on performance by modeling configurations where the inter-socket communication latency is 0, memory bandwidth is infinite, or QPI bandwidth is infinite. Our studies are done on a simulated 32-core quad-socket system modeled after a modern AMD-like NUMA server (see §V for details of our simulation methodology) using a mix of contemporary parallel and server workloads. For each workload, we consider a first-touch and interleaved mapping policies and use the best performing one based on a separate profiling run [1].

Fig. 2 presents the results of the study by normalizing the performance of each of the three idealized configurations to that of the baseline. In general, we observe little benefit from infinite DRAM or QPI bandwidth, indicating that bandwidth is not a bottleneck in contemporary NUMA servers. In contrast, we see significant sensitivity to inter-socket communication delays, with the 0-QPI-latency configuration delivering speedups ranging from 14-60% across our workload spectrum. We thus conclude that minimizing off-socket accesses represents the most promising direction for improving performance of modern workloads on NUMA servers.

C. Avoiding Remote Accesses with Caching

Caching is a promising solution to the NUMA problem as local replicas avoid the need to go off-socket [8]. Intuitively, large cache capacities are needed to provide a significant miss reduction beyond today's multi-MB on-chip caches. Fig. 3 confirms the intuition by showing the reduction in remote memory accesses with larger cache sizes normalized to a 16MB baseline cache. We observe that even in applications operating on large datasets, there is significant temporal locality within each socket's memory access stream, making caching beneficial for NUMA system. However, uncovering this locality requires high-capacity caches (256MB and 1GB data points), which are able to eliminate 38.6-45.5% of remote memory accesses.

[2]We collect this statistic while executing the parallel region of these workloads.

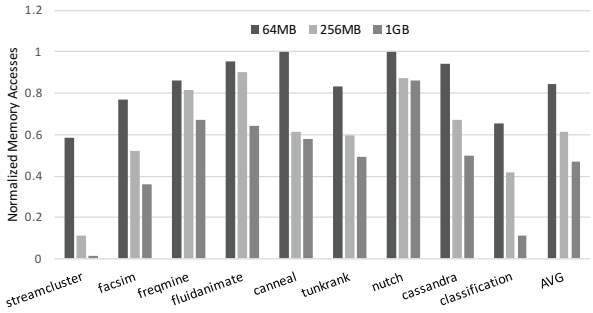


Figure 3. Memory accesses as a function of LLC size normalized to a system with a 16MB LLC.

Emerging die-stacked and on-package DRAM cache architectures present ideal candidates for such high capacity caches. However, existing work has not considered the use of DRAM caches in NUMA servers, which present two fundamental options for organizing the multiple DRAM caches:

1) *Shared organization.* This design treats each DRAM cache as a memory buffer, with the aggregate cache capacity scaling with the number of sockets. Because capacity is shared across sockets, this option provides the maximum hit rate and does not require a coherence solution, since each memory address can only reside in its home socket's DRAM cache. Alas, the shared organization can only help filter accesses to main memory, but does not help in reducing off-socket accesses.

2) *Private organization.* The private design turns each socket's DRAM cache into a massive victim cache that captures the socket's LLC evictions. As such, private DRAM caches can reduce off-socket traffic by exploiting temporal locality beyond the reach of on-chip caches. On the downside, because DRAM cache capacity is not shared, the private design may not be able to filter as much memory traffic as the shared design. Another concern for a private design is the need for a coherence solution, as multiple caches may have replicas.

Our results in Fig. 2 clearly point to private DRAM caches as the preferred design choice, since they directly attack the inter-socket delay problem by reducing off-socket traffic. The principal challenge for private DRAM cache designs is guaranteeing coherence. Existing work on DRAM caches has focused exclusively on single-socket systems, which are immune from the replica problem inherent in the multi-socket configuration with private caches. The rest of the paper is dedicated to solving this challenge.

III. Coherent DRAM Caches: The Naive Approaches

As observed in §II, large private caches can be effective in relieving the NUMA bottleneck by capturing distant temporal reuse in each socket's access stream. Emerging DRAM cache architectures are well-suited for this purpose due to their high capacity, reaching into hundreds of MBs or even GBs per socket. However, the data replication in private DRAM caches necessitates them to be coherent.

To ensure local (intra-socket) and global (inter-socket) coherence, today's multi-socket systems incorporate hierarchical coherence [9], [10]. This section explores extending the existing snoopy and directory based coherence protocols for maintaining inter-socket coherence in a commodity (2- and 4-socket) NUMA system with DRAM caches.

A. Snoopy Coherence Protocol

As the number of sockets in current multi-socket NUMA systems is small, it is tempting to use a snoopy protocol for inter-socket coherence. The snoopy protocol broadcasts every local DRAM cache miss. Upon receiving the broadcast, every socket checks if it has cached the requested block. If any of the sockets has a dirty copy, it is forwarded to the requesting socket. In case of write requests, all clean copies of the requested block must be invalidated. If none of the sockets has a dirty copy, the request is served from the memory.

In the presence of DRAM caches, they must be searched upon receiving a snoop request. Because DRAM caches and main memory employ the same DRAM technology, even the most aggressive latency projections for DRAM caches show them to be at most twice as fast as main memory [11]. Realistic expectations are that the difference in latency between the two will be negligible. Indeed, the recently-released Intel Phi Knight's Landing has an in-package DRAM cache whose access latency exceeds that of its memory [12]. As a result, we observe that obtaining a block from a remote DRAM cache via a snoop can be much slower than accessing main memory, since the slow DRAM cache problem is compounded by the inter-socket communication delays. We refer to this phenomenon as the *slow remote hit pathology*. Even if none of the sockets has a copy, all DRAM caches must still be snooped before the request can be served from memory. Therefore, the furthest socket's response latency is on the critical path and determines the AMAT, even if the block is uncached in any of the sockets.

One potential optimization to avoid the high DRAM cache access latency is to employ a missmap [11]. If the missmap indicates that the requested block is not present in the DRAM cache, the socket can respond without probing it. However, if the block is present, DRAM cache still has to be accessed 1) to check if it is dirty and 2) to invalidate it in case of a write request. Furthermore, the missmap does not help in addressing the high inter-socket communication delays.

B. Directory-based Coherence

Today's multi-socket systems tend to use a directory for inter-socket coherence. One could extend the directory protocol to additionally track blocks residing in DRAM caches in the global directory. Each socket contains a slice of the global directory that tracks the blocks belonging to the local memory of the socket that are cached in the system. In addition, each block in the DRAM caches carries coherence metadata.

As the global directory tracks all the blocks cached in the system, it eschews some of the inefficiencies of the snoopy protocol. For example, the high-latency remote DRAM cache accesses can be avoided on a DRAM cache miss for uncached blocks by directly accessing main memory. Similarly, if the requested block is cached clean in remote sockets, a read miss for this block can also be served from memory. Unfortunately, if the block is modified in a remote socket's DRAM cache, a high-latency access to the owner socket is unavoidable. In such a case, a multi-socket system with DRAM caches would actually be slower than the baseline without DRAM caches. The following example illustrates why.

Modified block in a remote DRAM cache: Suppose a dirty block is cached in the DRAM cache on one of the sockets other than the *home socket* (the socket where the directory slice resides). On a DRAM cache miss for this block, the *requesting socket* will forward the request to the *home socket* as shown by step ① in Fig. 4. The *home socket* will lookup its slice of global directory and forward the request to the *owner socket* (where the dirty block is cached), as shown by steps ② through ④. In turn, the *owner socket* will forward the dirty block to the *requesting socket*, as shown by steps ⑤ to ⑦.

We observe that the dirty block in the DRAM cache would have been in memory in the baseline system without DRAM caches. Therefore, the request would have been served from the memory. However, with DRAM caches, the memory cannot service the request as one of the DRAM caches contains a dirty copy. Since accessing memory from the home socket is faster than accessing a remote DRAM cache, the baseline system would provide a lower AMAT for these cases than a system with DRAM caches.

In addition to the slow remote hit pathology, the addition of DRAM caches to the baseline's coherence protocol (that only covers on-chip caches) introduces the following directory-related overheads.

Global Directory Size: Existing systems generally use a sparse global directory to track the cached blocks. As larger caches capture more blocks, the directory must be made proportionately larger. However, due to one to two orders-of-magnitude difference in capacity between DRAM caches and on-chip caches, directory storage requirements can become a significant burden in a practical system. For example, a 256MB DRAM cache, even with a minimally-provisioned (1x) sparse directory, would require 16MB of directory storage per socket. For a 2x-provisioned directory, as featured in AMD's Magni Cours Opteron processor [9], the storage costs increase to 32MB for a 256MB cache or a whopping 128MB for a 1GB DRAM cache.

Having such a huge on-chip directory in SRAM is clearly impractical. An alternative is to move the directory off-chip, potentially leveraging the DRAM cache for storage. Unfortunately, a DRAM directory introduces a new problem.

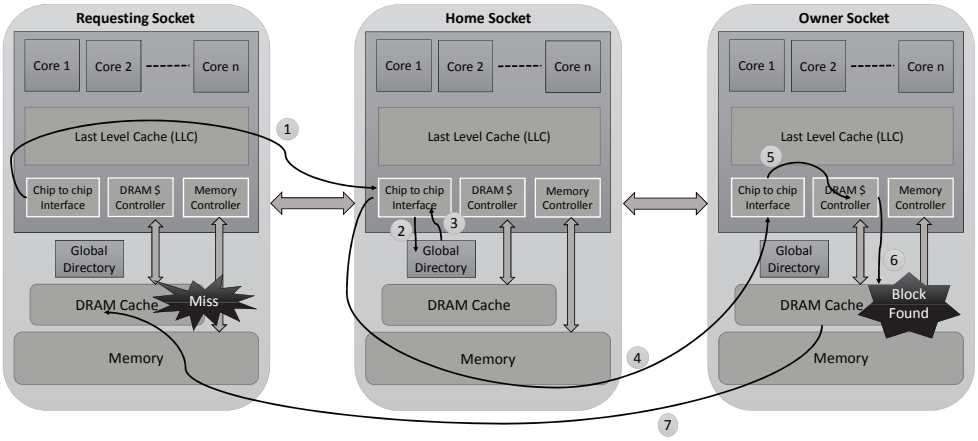


Figure 4. Accessing modified blocks from a remote DRAM Cache.

Global Directory Access Latency: Directory access latency is directly proportional to the directory size. For a larger directory, the additional access latency gets added to the critical path and the AMAT increases correspondingly. The access latency would be particularly high for a DRAM-resident directory, as DRAM access latency is up to an order-of-magnitude higher than that of SRAM.

Summary: Accesses to remote DRAM caches cause slow remote access pathologies in both snoopy and directory protocols. While a directory-based approach results in fewer remote DRAM cache accesses than snoopy, it incurs unaffordable directory storage and latency overheads.

IV. Clean Coherent DRAM Caches

In this section, we describe the proposed C^{3}D design and how it addresses the chief limitations of the naive designs: slow hits in remote sockets, global directory size and global directory access latency.

In §IV-A, we tackle the first two challenges while still relying on an inclusive global directory tracking DRAM cache blocks. We then show in §IV-B that tracking blocks in DRAM caches is not beneficial, thereby enabling a massive reduction in directory-related overheads. §IV-C describes our proposed coherence protocol that reflects the above ideas. Finally, §IV-D discusses an optimization for further reducing C^{3}D's inter-socket traffic.

A. Clean DRAM Cache

As explained in §III, local DRAM cache misses (with respect to the requesting socket) can be satisfied faster by memory than by a remote DRAM cache. This is easily accomplished for clean blocks, but maintaining coherence for dirty blocks residing in a remote socket's DRAM cache mandates an access to the owner cache.

Our critical insight is that the slow remote hit for modified blocks can be avoided by keeping DRAM caches clean. This requires also writing back dirty LLC evictions to memory, while retaining the clean block in the local DRAM cache. Keeping the DRAM cache clean allows a greater fraction of DRAM cache misses (in other sockets) to be satisfied by memory, avoiding the slow path. At the same time, the hit rate of the local DRAM cache is not diminished as a result of writing through to memory, since a subsequent read request from the same socket can still be served from its local DRAM cache without having to probe the global directory and accessing memory.

B. Non-inclusive directory

While clean DRAM caches overcome the problem of slow hits to remote DRAM caches, they are still encumbered by excessive directory sizes, leading to significant storage and latency overheads. The directory size requirement of the naive design is high because it is an inclusive directory that is guaranteed to track all blocks cached by the large DRAM caches. However, with clean DRAM caches, the only modified blocks are in LLC or higher-level caches. Thus, the bulk of the global directory's capacity is expended on tracking clean blocks in DRAM caches. Is it worth it?

As far as read requests that miss in the local DRAM cache are concerned, there is little benefit in tracking the coherence state of DRAM caches. Indeed, with clean DRAM caches, a block cannot be in modified state in a remote DRAM cache; it can only be in modified state in an on-chip cache. Therefore, tracking modified blocks in just the on-chip caches suffices.

The only benefit to tracking the coherence state of clean DRAM caches would be for write requests. By tracking the clean blocks in DRAM caches, the directory would be able to precisely identify sockets to which it needs to send invalidation messages. In the absence of this information, invalidation messages must be broadcast to all DRAM caches, possibly resulting in wasted inter-socket and DRAM cache bandwidth whenever none or only a subset of the sockets are caching the block. However, since repeated write requests would likely hit in the LLC (and be tracked by the directory), broadcasting invalidations for just the remaining write misses results in minimal additional traffic. Moreover, the additional latency (due to the broadcast) incurred by write requests is off the critical path, as stores are already out of the critical path in modern processors, all of which have a store buffer. Maintaining a huge directory to avoid the minimal additional traffic incurred by write requests that miss in the requesting socket's on-chip

cache hierarchy does not appear to be a cost-effective trade-off. Therefore, we propose a non-inclusive directory in which we avoid tracking the blocks solely residing in clean DRAM caches; instead we employ the directory for tracking the blocks cached by LLC or higher. Doing so minimizes directory costs while still providing fast hits for frequently communicating writes, which tend to stay on chip and are precisely identified via the global directory.

In summary, our design (C^3D) features:

- Fast read hit in the local socket's DRAM cache with no messages sent to remote sockets.
- No accesses to remote sockets' DRAM caches on a read miss in a local socket. Local read misses are serviced either by a remote socket's on-chip cache or by memory. This avoids the slow remote hit pathology.
- Minimal directory overhead for blocks held solely in DRAM caches.
- Writes to blocks not tracked by the directory trigger a broadcast to invalidate any copies in DRAM caches. Since writes are generally not in the critical path, the performance impact of broadcasts is minimal and the extra traffic modest. §IV-D presents a further optimization to elide broadcasts for thread-private data.

C. Coherence Protocol for C^3D

In this section, we flesh out the coherence protocol that incorporates our two ideas: (a) clean DRAM caches and (b) non-inclusive directory. But first, we analyze the implications of a non-inclusive directory that avoids tracking cache blocks solely held in the DRAM caches.

Since there may be a block in one of the DRAM caches unbeknownst to the directory, a directory miss does not *imply* that the block is uncached in any of the sockets as in a classical directory protocol. Therefore, a write miss for a block without a directory entry requires a broadcast to invalidate any potential DRAM cache sharers. Second, we observe that there is little benefit to allocating a directory entry (and to start tracking sharers) on a read miss that is previously untracked at the directory. This is because, without an expensive broadcast that checks other DRAM caches, there is no way to arrive at a valid sharing vector. This also explains why there is little benefit to maintaining an *exclusive* state, as confirming that there are no other sharers requires a broadcast that checks other DRAM caches.

Based on these observations, C^3D's global directory can have a block in one of three stable states: Modified, Shared and Invalid. The following examines in more detail the cases handled by each of these stable states, whose state transition diagram is depicted in Fig. 5.

Invalid: The meaning of the Invalid state is slightly different from the norm. In a conventional directory-based protocol, the Invalid state implies that the block is not present in the cache hierarchy (inclusivity). However, C^3D's protocol allows the block to be present in one or more DRAM caches (and/or higher) without the directory tracking this information (non-inclusivity). In other words, the only invariant that the Invalid state guarantees is that the value in memory is not stale (because of clean DRAM caches). Consequently, a read (GetS) request to a block in Invalid state is satisfied by memory, and the block is *not* inserted into the global directory. A Write (GetX) request,[3] with a block in Invalid state requires broadcasting invalidations to all other DRAM caches before the response can be sent to the requester and the directory transitions to Modified.

Modified: The Modified state is most similar to that in a conventional MSI protocol: a block in Modified satisfies the invariant that the block is present in exactly one socket; the *clean* property implies that a DRAM cache cannot exclusively hold a Modified block. However, note that it is possible for a DRAM cache to hold a (stale copy of a) block held in Modified state in its LLC or higher (to deal with LLC evictions correctly).

Evicting an entry in Modified state from the directory requires either: (a) simply invalidating the copy in the owning socket (DRAM cache and/or higher) *or* (b) issuing a downgrade request to the owner allowing the owner to retain the copy in Shared state. We choose (a) as it requires fewer transient states and message exchanges.

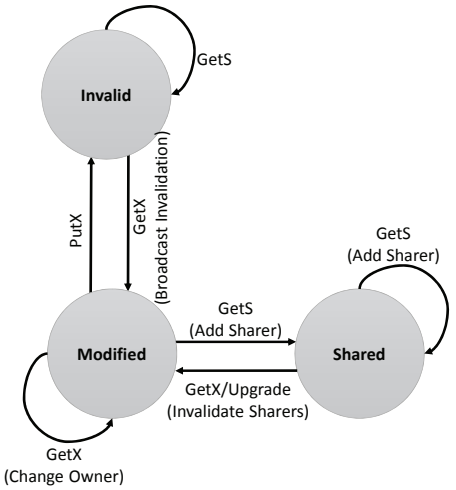
Evictions by the LLC in Modified state cause a write-back (PutX) message first to be sent to the DRAM cache which then transitions to Shared and updates its copy of the data. The DRAM cache then forwards the PutX to the global directory; finally, the global directory acknowledges receipt directly to the LLC and transitions to Invalid.

Downgrade requests from the directory are sent directly to the LLC, and are then responded to like a LLC eviction (but with the LLC transitioning to Shared). Invalidation requests from the directory take the opposite path of downgrades, with them being sent to the DRAM cache first, which acts by transitioning to Invalid. The DRAM cache then forwards the invalidation to the LLC (which transitions to Invalid) and directly responds to the global directory with a PutX.

Shared: In a conventional MSI protocol, the Shared state provides two invariants: first, that the value in memory is not stale; second, a block in Shared state has a valid sharing vector, which helps in limiting invalidation messages on a subsequent GetX requests to that block. The Shared state in the C^3D protocol continues to satisfy these invariants, in spite of the non-inclusive directory. This follows from the fact that, unlike a conventional MSI protocol, there does not exist a direct transition from Invalid to Shared (GetS requesters in Invalid are not tracked, with directory staying in Invalid).

In the directory, the only possible transition leading to Shared is from Modified. As we know that a block in Modified is exclusive to exactly one socket, a subsequent GetS request in Modified will transition to the Shared state with the sharing vector being updated to now contain the previous owner as well as the requester. Any further GetS requests can immediately

[3]Note that the protocol handles Upgrade requests (requester already has data in Shared) similarly to GetX requests, but Upgrade responses do not carry data.



GetS	Read request. Sender: DRAM cache controller, forwarding from LLC due to miss in DRAM cache.
GetX, Upgrade	Write request. Sender: DRAM cache controller, forwarding from LLC.
PutX	Write-back. Sender: DRAM cache controller (if due to LLC replacement, or downgrade request) or LLC (if responding to invalidation).

Figure 5. C^{3}D global directory stable states.

be satisfied with the sharing vector being updated with the corresponding requester.

As we have a valid sharing vector in the Shared state, upon a GetX request, invalidations only need to be sent to those sockets which are tracked in the sharing vector. We note, however, that the sharing vector is only precise until one socket evicts the block from both LLC (and higher) and DRAM cache, as evictions in Shared are silent; it follows that the sharing vector is a superset of sharers after complete eviction of a block from a socket, therefore still valid. Note that silent evictions in Shared from the DRAM cache imply that the DRAM cache itself is also *non-inclusive* with respect to LLC and higher.

Verification: We verified the coherence protocol using the model checker Murφ [13], proving absence of deadlock and race conditions. Furthermore, we also verified that the Single-Writer-Multiple-Reader (SWMR) invariant and SC per memory location [14] are not violated. We modeled the global directory (3+10 states), DRAM cache (3+5 states), LLC (3+7 states), with a total of 15 message types – assuming no ordering constraints on interconnect. A detailed state transition table as well as the Murφ model is available online[4].

D. Avoiding Broadcasts for Private Data

One downside of the C^{3}D design is its use of broadcasts to invalidate remote DRAM caches upon a write to a block with an untracked directory entry. While write-related coherence traffic is generally not on the critical path, it may contribute to QPI congestion and extra energy expenditure. In this section, we describe an optimization to the C^{3}D protocol to completely elide broadcasts on write accesses to thread-private data. The optimization is particularly useful in the context of single-threaded applications whose write working-set size is larger that the LLC; in such cases, sharing is generally not expected (except for user/kernel interactions), and much of the broadcast traffic can be suppressed.

Our approach relies on a simple private/shared page classification mechanism to identify thread-private memory regions. Similar to prior work [15], we extend the page table entries with a field that stores the owner thread's id and a bit that denotes the current classification. On the first access, the processor will trigger a TLB miss that will be handled by the OS. The OS will first mark the page as private and store its core id (CID) in the owner field. On a subsequent TLB miss, if the triggering thread's CID does not match the owner's CID, the OS will take an action depending on the cause of the mismatch. If the mismatch is caused by thread migration, the OS will update the CID field with the new CID and shoot down the page in the memory hierarchy. If the mismatch is due to active sharing (different thread ID), the page will be re-classified as a shared page. During this private-to-shared transition, the OS will need to trap the owner thread to guarantee all pending writes to that page are flushed. Note that the page does not need to be shot down in the memory hierarchy in the event of re-classification.

Given the shared/private classification bit in the TLB, a GetX request due to a miss in the on-chip cache hierarchy will carry the classification as part of the request to the directory. If the directory is in Invalid state and receives a GetX request with a private bit set, the directory can transition the block to Modified state without broadcasting invalidations.

V. Methodology

Simulation Environment and Parameters: Table II lists the system parameters used in our studies. We model two 32-core NUMA configurations: with two sockets (16 cores/socket) and four sockets (8 cores/socket). Both setups have a 16MB LLC per socket. For each DRAM cache, we use a direct-mapped organization with a 4K-entry miss predictor [16]. Without loss of generality, we assume a die-stacked DRAM design with an access latency 50% lower than that of main memory [11] and supplement our evaluation with a sensitivity study to this parameter in §VI-D.

For the inter-socket latency, we measure the latency difference between local and remote memory on a dual-socket Intel SandyBridge-E machine, which is around 50-60ns. Note that this latency is compound and includes on-chip routing to system agent, QPI packet translation, inter-socket link, on-chip routing to the directory controller, and the directory access itself. Based on this, we model a 40ns round-trip delay per hop, excluding the directory access. This is a conservative estimate, as higher inter-socket latencies improve the relative benefits of DRAM caches in general and our scheme in particular. We also study the sensitivity to this parameter in §VI-D.

Simulating many-core NUMA systems with high-capacity DRAM caches and massive application working set sizes is challenging with existing publically-available toolsets. To overcome this challenge, we build a custom trace-driven

[4]https://github.com/icsa-caps/c3d-protocol

Processor	32-core, 32-entry store queue, TSO, 3GHz width = 1 IPC [11], 64B line buffer 8-core/socket in 4-socket; 16-core/socket in 2-socket
Memory Model	Total-Store-Order (TSO)
L1 I/D	64KB/8way, 3-cycle, private
L2 cache (LLC)	16MB/16-way, 7-cycle for tag, 13-cycle for data
L3 cache (DRAM)	1GB, block-based, direct-mapped, 40ns 12.8GB/s per channel, 8 channels region-based miss predictor [16], 4K-entry, 2-cycle
Global Directory	10-cycle, sparse 2x/32-way, socket-grain sharing vector
Local Directory	7-cycle, embedded in L2, full sharing vector
Inter-socket interconnect	Ring (4-Socket), P2P (2-Socket) 20 ns per hop, 25.6GB/s 16-byte control / 80-byte data packet
Main Memory	50 ns, DDR3-1600 (12.8GB/s), 2 channels

Table II
SYSTEM PARAMETERS

simulator with a simple timing processor model and a cycle-accurate memory subsystem that, among other things, models the latency and bandwidth of inter-chip communication, DRAM memory and cache, and coherence protocol actions including transient states. The simulated parameters are listed in Table II. Our traces are collect from workloads described in §V using Pin [17] and Simics [18] as described next.

Workloads: We evaluate the various designs using contemporary parallel and server workloads consisting of PARSEC 3.0 [19] and CloudSuite [20] benchmarks. For PARSEC, we use all of the benchmarks which have large working set sizes (over 100MB) in their native input [21]. We first fast-forward to the parallel region and warm-up the DRAM caches with 100 million accesses. Then, we collect the results for half a billion instructions per core.

For CloudSuite, we use the three workloads from CloudSuite 1.0 that have 32-core Simics checkpoints available online and run them on Simics 3.0 [18]. To collect the traces, we execute 100 million instructions for each core as warm-up and collect the results for a billion instructions per core. We also use the Graph Analytics (tunkrank) benchmark from CloudSuite 2.0 (for which Simics checkpoints are not available) as it is readily runnable on our Pin-based infrastructure. We use the same methodology for Graph Analytics as we do for the PARSEC suite.

Memory Allocation Policy: We studied three memory allocation policies:

- Interleave (INT): Adjacent pages are interleaved across memory controllers in a round-robin fashion.
- First-touch-1 (FT1): The first touch to the page from application start determines the page's location in memory. In many cases, we found this policy to perform poorly as large regions of memory get mapped to one node before the application enters the parallel phase. Therefore, we use an alternative approach (FT-2).
- First-touch-2 (FT2): We first fast-forward to the parallel region and only then start to allocate memory to sockets based on first touch. The warm-up period is extended into the parallel phase and measurements are performed in steady state.

For the evaluation, we do profiling runs with all three policies and select the best-performing one for each application.

A. Evaluated Designs

Baseline: The baseline has no DRAM cache. Caches across sockets are kept coherent using a global directory and local caches within a socket are kept coherent using a local directory (parameters are shown in Table II). The local directory settings are the same in all evaluated designs.

Snoopy: The *snoopy* design, introduced in §III-A, ensures inter-socket coherence by snooping all remote caches on a miss. In this design, the global directory in the baseline is retained and used instead as a block level snoop filter. To avoid serializing the memory access in case of a miss in remote caches, we access the memory in parallel with probing remote caches.

Full directory (*full-dir*): This is the design presented in §III-B. We model a full global directory (i.e., no recalls) with an inclusive cache hierarchy. Despite the massive capacity required by the directory, we optimistically assume a 10-cycle directory access latency, which is the same latency incurred by the baseline's global directory.

C^{3}D (*c3d*): This is our proposed design as described in §IV. It features clean DRAM caches, a non-inclusive global directory that does not track blocks cached solely in DRAM caches (same parameters at *baseline*), and the coherence protocol elaborated in §IV-C. The DRAM caches are non-inclusive of the higher levels of the cache hierarchy, as our coherence protocol does not force inclusion (§IV-C).

C^{3}D + Full directory (*c3d-full-dir*): This combines the proposed C^{3}D with an idealized full global directory (no recalls, 10 cycle access latency). We made a small modification to the C^{3}D protocol to make modified blocks transition to the shared state after receiving a writeback. This idealized design eliminates all broadcasts incurred in the C^{3}D protocol, thus allowing us to study the effect of broadcasts in the base C^{3}D design on performance.

VI. EVALUATION

The goals of our evaluation are as follows. First and foremost, we want to evaluate how our proposal performs in comparison with the baseline (no DRAM cache) and alternative techniques for ensuring coherence of DRAM caches. Second, we want to measure how much our proposal (and the alternatives) are able to reduce inter-socket traffic in comparison with the baseline, which basically explains our results. Third, we evaluate the TLB technique we introduced in §IV-D to understand the necessity of this optimization. Finally, we study the sensitivity of C^{3}D to DRAM cache and inter-socket latencies to evaluate how robust our gains are.

A. Performance Analysis

4-socket: Fig. 6 shows the performance of the evaluated designs in quad-socket systems featuring 1GB of DRAM cache per socket. We observe that C^{3}D is, on average, 19.2% better than the baseline, which is the key result. This validates our approach: DRAM caches are effective in filtering out remote accesses, while clean caches with C^{3}D's coherence protocol and LLC directory deliver an efficient coherence substrate.

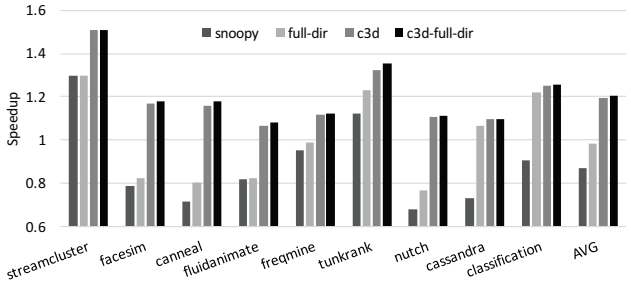


Figure 6. 4-socket (8-core/socket) performance comparison

One major winner with C^{3}D is *streamcluster* (50.7% speedup), as its working set fully fits in the DRAM caches. Other benchmarks, like *nutch*, whose working sets don't fit in the DRAM caches also benefit from C^{3}D as private DRAM caches filter out a portion of remote accesses, thus reducing AMAT, without incurring performance pathologies that afflict other designs.

The *snoopy* design is generally ineffective, slowing down most workloads. The reason is that it relies on checking all remote DRAM caches on each local miss to guarantee coherence, exposing both inter-socket and DRAM cache access latencies on the critical path.

The *full-dir* design improves performance over *snoopy* by enabling a fast path (to memory) in the event of a local miss with the block clean or untracked in the global directory. Nonetheless, *full-dir* hurts performance for the majority of PARSEC workloads. The workloads that suffer have a high degree of inter-thread communication, which exposes the *slow remote hit* pathology in the *full-dir* design (§III). In contrast, server applications, which are known to have less inter-thread communication [20], benefit from *full-dir*, outperforming the baseline by 6.4% to 22.9%. The exception is *nutch*, which sees a significant slowdown. The reason is that the thread that handles the request is usually different from the thread that handles the processing. The high communication cost between these threads hurts performance whenever the threads are not on the same socket[5].

Finally, we observe a small difference (19.2% versus 20.3%) between C^{3}D and the idealized *c3d-full-dir* designs, indicating that the broadcast invalidation messages in C^{3}D do not hamper performance significantly.

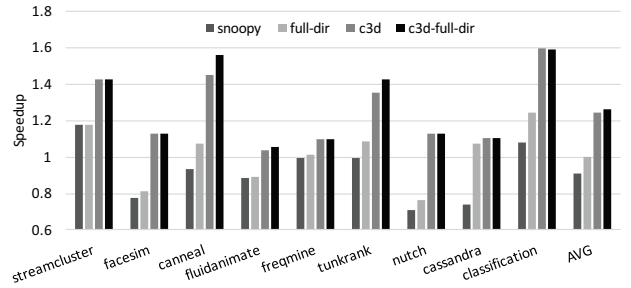


Figure 7. 2-socket (16-core/socket) performance comparison

[5]We believe this particular case can be handled by a better NUMA-aware thread scheduler. However, this is beyond the scope of this paper.

2-socket: The general trends in the two-socket system, shown in Fig. 7, closely follow those in the quad-socket configuration. Overall, we observe higher speedups for C^{3}D in the dual-socket system. This is because LLC has a higher miss rate as each socket has more cores (16) sharing the LLC as compared to the 4-socket system (8). This provides a higher opportunity for the DRAM cache to be beneficial by filtering expensive off-socket accesses. On average, C^{3}D achieves a 24.1% performance gain with a 1GB DRAM cache per socket, which is within 3% of the idealized *c3d-full-dir* design (26.3%).

B. Memory and Inter-Socket Traffic

In this section, we study the effect of the various designs on the memory and inter-socket traffic. For this analysis, we use a 4-socket system with 1GB DRAM cache. Fig. 8 shows the reduction in inter-socket (i.e., remote) memory traffic for the C^{3}D design. As shown in the figure, DRAM caching can reduce up to 98% of memory accesses (in *streamcluster*) and 49% on average compared to the baseline system without DRAM caches. As expected, there is no reduction (but also no increase) in write traffic as compared to the baseline, as the DRAM caches in C^{3}D are write through.

In addition to overall memory accesses, we also look into the average memory read accesses as most of them are on the critical path of system performance. Fig. 8 shows the remote memory reads over the baseline. We can see up to 99% (70.9% on average) of remote reads are avoided in our workload set. This result clearly indicates that having a private DRAM cache is key to mitigating the NUMA bottleneck.

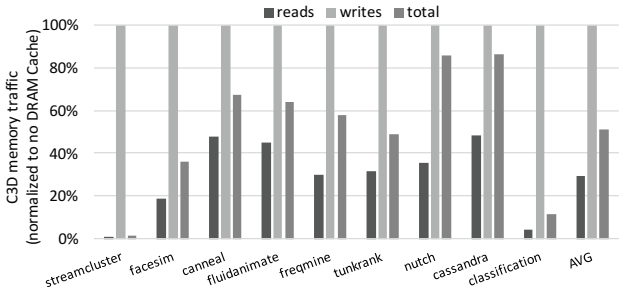


Figure 8. C^{3}D Memory Traffic

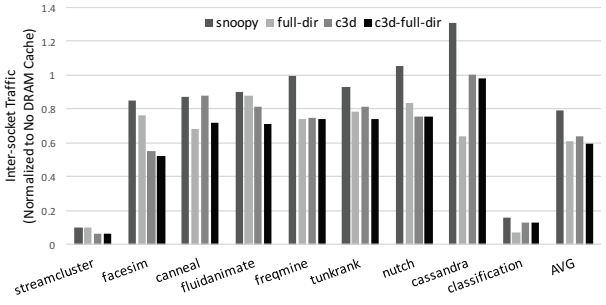


Figure 9. Inter-socket traffic comparison

With regard to the inter-socket traffic (Fig. 9), C^{3}D generates 35.9% less traffic compared to baseline due to a reduction in remote memory accesses. Furthermore, we can see C^{3}D has

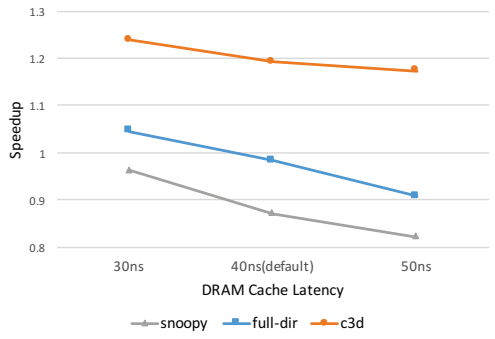


Figure 10. C^{3}D performance improvement over different DRAM cache latencies

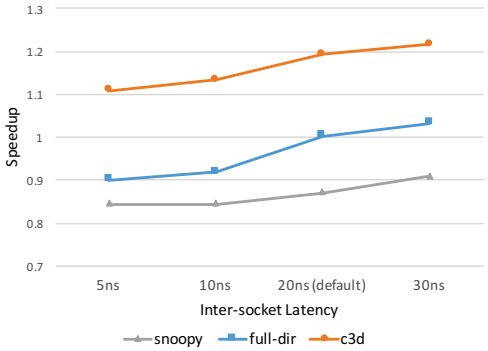


Figure 11. C^{3}D performance improvement over different inter-socket latencies

only about 5% more traffic compared to both *full-dir* and *c3d-full-dir* designs. This can be explained by the fact that any additional traffic in C^{3}D can only arise as a result of writes. However, the bulk of the traffic is read data; writebacks are only nominally higher with C^{3}D, and the additional broadcast-related control messages constitute a small fraction of the overall bytes transferred due to their small size compared to the data packets. It is worth noting that C^{3}D actually reduces inter-socket traffic over *full-dir* in several workloads (e.g., *facesim*). This is because of the additional overhead inherent in dirty DRAM caches as described in §III-B.

C. Reducing Broadcast Traffic

In §IV-D, we presented a TLB-based technique to filter broadcast invalidation messages incurred by the C^{3}D protocol. We found that filtering broadcasts to pages classified as private can avoid about 5% broadcast messages in our workloads. However, the reduction in the overall inter-socket traffic is almost negligible (less than 0.1% traffic), as broadcast traffic constitutes a small fraction of the overall inter-socket traffic dominated by data-carrying messages.

However, the optimization is useful in the context of single-threaded workloads that have no shared data component. Since such workloads are a realistic part of many practical server deployments, we evaluate the proposed optimization using the memory-intensive *mcf* benchmark from SPEC'06. Our study confirms that the write-related inter-socket traffic incurred in *mcf* can be completely removed by using our TLB classification. However, even in the case of the single-threaded workload, the reduction in overall traffic is small as reads dominate. We thus conclude that the TLB classification is a useful but non-essential component of C^{3}D.

D. Sensitivity Studies

In this section, we study the effect of DRAM cache latencies and different inter-socket latencies on the efficacy of C^{3}D with 1GB DRAM cache.

DRAM cache latency: As shown in Fig. 10, when the DRAM cache latency is equal to memory latency (50 ns), C^{3}D continues to deliver over 17.3% performance gain versus the baseline. The high efficacy can be attributed to C^{3}D's avoidance of remote DRAM cache accesses on a read. On the other hand, if the DRAM cache latency is much faster than DRAM, local DRAM hits deliver a further performance boost (compared to memory accesses even on the local socket), leading to a 24% performance improvement. It is worth noting that snoopy and full-dir designs also follow a similar trend in our study.

Inter-Socket latency: Generally, C^{3}D's speedup is coming from two sources: (a) the removal of remote memory accesses (i.e., avoiding the NUMA bottleneck) and (b) the latency/bandwidth difference between die-stacked DRAM accesses and off-chip memory. Therefore, in this experiment, we vary the inter-socket latency and see how it will impact C^{3}D's performance.

As we can see from Fig.11, even with an unrealistically small 5ns (15-cycle) inter-socket communication latency (remember that the latency is compound and includes multiple on-chip interactions; see §V), C^{3}D delivers a 10% performance improvement over the baseline. As expected, we see the C^{3}D's speedup increases as the communication latency increases. In this figure, we can again see C^{3}D consistently outperforming full-dir and snoopy across different communication latencies.

E. Summary

To summarize, C^{3}D improves NUMA system performance by 6.4-50.7%. Our results also show that C^{3}D can consistently outperform the baseline even with slow DRAM caches or fast inter-socket communication. Finally, our results demonstrate that there is no benefit in a huge directory for DRAM caches, thus avoiding a potential storage and latency bottleneck.

VII. Related Work

A. DRAM Caches

A large body of recent work has proposed using die-stacked DRAM as a cache [11], [16], [22], [23], [24], [25], [26], [27], [28]. Prior work on this topic primarily focus on reducing the access latency of DRAM caches, avoiding miss penalty, and reducing the tag overhead. Among prior work, Sim et al. [26] propose *mostly-clean cache* for load-balancing the memory access stream between a die-stacked DRAM cache and off-chip memory. In contrast, our work advocates a *completely* clean cache to ensure fast coherence in NUMA systems with DRAM caches. In general, prior work has not considered multi-socket DRAM caches and its associated cache coherence problem, although Lee et al. [29] allude to this challenge.

B. Cache Coherence

Cache coherence techniques can be broadly divided into snooping based techniques [30] and directory based techniques [31].

Snooping based: In snooping protocols, writes to non-exclusive cache lines need to be broadcast. Pior work has looked at reducing the number of such broadcasts using filtering techniques that track coherence state for large regions [32], [33], [34]. However, the efficacy of such mechanisms is fundamentally limited by false sharing which increases with region size [34]; using smaller regions, on the other hand, would require correspondingly larger tracking storage.

Directory based: Scalable multiprocessors use directory based protocols. Although avoiding the costly broadcasts, the size of the directories is a problem that a number of researchers have attempted to optimize. One way to reduce the size of the directory is to reduce the total number of directory entries or the directory *height*. Sparse directories [35] use the idea of caching to reduce the directory height. More recent techniques, such as Multigrain coherence directory [36], attack the same problem. While these works are quite effective – for example, multi-grain directory can reduce 66% space overhead of a 2x sparse directory – the size of the directory will still be around 38 MB per socket for our situation. In contrast to the above work which optimize over the LLC directory, we propose a new approach in which we simply do not track the clean DRAM cache blocks for massive storage savings.

Another way to reduce the size of the directory is to reduce the size of each directory entry or the directory *width*. Limited pointer directories [30], for instance, reduces directory width by maintaining a coarse-grained sharing vector. Prior work has also suggested eliminating individual tags in the directory by maintaining tags per cache sets using Bloom filters [37]. The directory width is not a major issue in our setting, as we focus on systems with only a few sockets. For higher socket counts, these techniques can be used.

C. Tertiary Caching and COMA

The idea of employing caches to mitigate NUMA effects has been proposed previously in cache-only-memory architecture (COMA) [8], [38], [39], [40] and tertiary caching [41], [42], [43]. In tertiary caching, a fixed size of the local node's memory is used as a hardware managed cache that caches only *remotely* allocated blocks. In COMA, *all* of the local memory is used as a cache. In both approaches, the caches are kept coherent. At a conceptual level, our work is similar: we also use private DRAM caches to mitigate NUMA, thus exposing the need for coherence. However, there are important differences that motivate our specific approach (clean DRAM caches, non-inclusive directory).

First, back then, the latency to access a remote node was significantly higher than the latency to access a home node. Indeed, for the Sequent NUMA-Q processor – a real machine which employed a 32 MB DRAM tertiary cache – accessing a remote node was 10x slower than accessing a local node [42]. Therefore, the addition of DRAM cache accesses in the critical path of coherence transactions did not significantly affect performance; in contrast, in current systems, because of faster inter-socket interconnect, accessing a remote node is "only" about 2x slower. As a result, we find that the addition of a DRAM cache access adversely impacts the critical path in modern systems, thereby motivating our clean DRAM cache approach. Having said this, even in Sequent-style tertiary caches, the clean cache approach could be beneficial to avoid slow remote accesses to dirty blocks whose home is the local node.

Secondly, although prior approaches required a huge directory – the Sequent NUMA-Q required a 128 MB directory per node (for 4 GB of memory) – the latency to access the directory paled in comparison with inter-node latency. In contrast, accessing such a huge directory in today's systems, with faster inter-node communication delays as compared to the Sequent NUMA-Q, would significantly affect the critical path. This observation motivates our decision to not track clean DRAM cache blocks in the directory.

VIII. Conclusion

This work exploits emerging DRAM cache architectures to tackle the NUMA bottleneck. Our insight is that the large storage capacities offered by DRAM caches allow effective caching of remote memory, thereby reducing high-latency inter-socket accesses. However, to be effective, DRAM caches must replicate remote data – an observation that echoes earlier software work on mitigating the NUMA bottleneck that explicitly called for "hardware support for replication" [1]. The principal challenge of replication is ensuring coherence across the large DRAM caches.

To the best of our knowledge, this is the first work to study coherence for multi-socket die-stacked DRAM caches. We identify the high directory storage costs and the high latency of remote hits as two essential problems that a practical design must address. Our solution to both is C^3D, a clean coherent DRAM cache organization. C^3D avoids slow remote hits and maintains a non-inclusive directory that avoids tracking DRAM cache blocks, thereby solving the directory cost bottleneck. In quad-socket system with 1 GB per-socket DRAM caches, C^3D reduces remote accesses by an average of 49% (70.9% for reads), improving system performance by 19.2%.

IX. Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by EPSRC grants EP/M001202/1 and EP/M027317/1 to the University of Edinburgh.

References

[1] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth, "Traffic management: a holistic approach to memory placement on NUMA systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 381–394, 2013.

[2] C. Williams, "Intel gives Facebook the D – Xeons thrust web pages at the masses," 2015. http://www.theregister.co.uk/2015/03/10/facebook_open_compute_yosemite/.

[3] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 261–270, 2009.

[4] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *International Symposium on Microarchitecture (MICRO)*, pp. 413–422, 2009.

[5] Intel, "Intel Memory Latency Checker." https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[6] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," 2012. http://lwn.net/Articles/488709.

[7] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, "Simple but effective techniques for NUMA memory management," in *Symposium on Operating System Principles (SOSP)*, pp. 19–31, 1989.

[8] E. Hagersten, A. Landin, and S. Haridi, "DDM - A cache-only memory architecture," *IEEE Computer*, vol. 25, no. 9, pp. 44–54, 1992.

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[10] Intel, "Intel Xeon Processor E7 - 8800/4800/2800 Product Families." http://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/xeon-e7-8800-4800-2800-families-vol-2-datasheet.pdf.

[11] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *International Symposium on Microarchitecture (MICRO)*, pp. 454–464, 2011.

[12] A. Sodani, "Knights landing intel xeon phi cpu: Path to parallelism with general purpose programming," 2016. Keynote at International Symposium on High-Performance Computer Architecture (HPCA).

[13] D. L. Dill, "The mur*phi* verification system," in *International Conference on Computer-Aided Verification (CAV)*, pp. 390–393, 1996.

[14] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

[15] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *International Symposium on Computer Architecture (ISCA)*, pp. 184–195, 2009.

[16] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical sram-tags with a simple and practical design," in *International Symposium on Microarchitecture (MICRO)*, pp. 235–246, 2012.

[17] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200, 2005.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[19] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[20] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 37–48, 2012.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 72–81, 2008.

[22] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bimodal DRAM cache: Improving hit rate, hit latency and bandwidth," in *International Symposium on Microarchitecture (MICRO)*, pp. 38–50, 2014.

[23] C. Huang and V. Nagarajan, "ATCache: reducing DRAM cache latency via a small SRAM tag cache," in *International Conference on Parallel Architectures and Compilation (PACT)*, pp. 51–60, 2014.

[24] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *International Symposium on Microarchitecture (MICRO)*, pp. 25–37, 2014.

[25] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches," in *International Symposium on Computer Architecture (ISCA)*, pp. 198–210, 2015.

[26] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch," in *International Symposium on Microarchitecture (MICRO)*, pp. 247–257, 2012.

[27] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: adaptive filter-based DRAM caching for CMP server platforms," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12, 2010.

[28] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *International Symposium on Computer Architecture (ISCA)*, pp. 404–415, 2013.

[29] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *International Symposium on Computer Architecture (ISCA)*, pp. 211–222, 2015.

[30] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *International Symposium on Computer Architecture (ISCA)*, pp. 280–289, 1988.

[31] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1112–1118, 1978.

[32] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary, "JETTY: filtering snoops for reduced energy consumption in SMP servers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 85–96, 2001.

[33] A. Moshovos, "RegionScout: Exploiting coarse grain sharing in snoop-based coherence," in *International Symposium on Computer Architecture (ISCA)*, pp. 234–245, 2005.

[34] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi, "Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays," *IEEE Micro*, vol. 26, no. 1, pp. 70–79, 2006.

[35] A. Gupta, W. Weber, and T. C. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing. Volume 1: Architecture.*, pp. 312–321, 1990.

[36] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *International Symposium on Microarchitecture (MICRO)*, pp. 359–370, 2013.

[37] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *International Symposium on Microarchitecture (MICRO)*, pp. 423–434, 2009.

[38] A. Saulsbury, T. Wilkinson, J. B. Carter, and A. Landin, "An argument for simple COMA," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 276–285, 1995.

[39] S. Basu and J. Torrellas, "Enhancing memory use in simple coma: Multiplexed simple coma," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 152–161, 1998.

[40] F. Dahlgren and J. Torrellas, "Cache-only memory architectures," *IEEE Computer*, vol. 32, no. 6, pp. 72–79, 1999.

[41] R. Thekkath, A. P. Singh, J. P. Singh, S. John, and J. L. Hennessy, "An evaluation of a commercial CC-NUMA architecture - the CONVEX exemplar SPP1200," in *International Parallel Processing Symposium (IPPS)*, pp. 8–17, 1997.

[42] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture - a hardware / software approach*. Morgan Kaufmann, 1999.

[43] Z. Zhang and J. Torrellas, "Reducing remote conflict misses: NUMA with remote cache versus COMA," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 272–281, 1997.