2017 IEEE International Symposium on High Performance Computer Architecture

A Split Cache Hierarchy for Enabling Data-oriented Optimizations

Andreas Sembrant, Erik Hagersten, and David Black-Schaffer
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05, Uppsala, Sweden
{andreas.sembrant, erik.hagersten, david.black-schaffer}@it.uu.se

***Abstract*—Today's caches tightly couple data with metadata (Address Tags) at the cache line granularity. The co-location of data and its identifying metadata means that they require multiple approaches to locate data (associative way searches and level-by-level searches), evict data (coherent writebacks buffers and associative level-by-level searches) and keep data coherent (directory indirections and associative level-by-level searches). This results in complex implementations with many corner cases, increased latency and energy, and limited flexibility for data optimizations.**

We propose *splitting* the metadata and data into two separate structures: a metadata hierarchy and a data hierarchy. The metadata hierarchy tracks the location of the data in the data hierarchy. This allows us to easily apply many different optimizations to the data hierarchy, including smart data placement, dynamic coherence, and direct accesses.

The new split cache hierarchy, *Direct-to-Master* (D2M), provides a unified mechanism for cache searching, eviction, and coherence, that eliminates level-by-level data movement and searches, associative cache address tags comparisons and about 90% of the indirections through a central directory. Optimizations such as moving LLC slices to the near-side of the network and private/shared data classification can easily be built on top off D2M to further improve its efficiency. This approach delivers a 54% improvement in cache hierarchy EDP vs. a mobile processor and 40% vs. a server processor, reduces network traffic by an average of 70%, reduces the L1 miss latency by 30% and is especially effective for workloads with high cache pressure.

I. Introduction

Today's cache hierarchies rely on many associative searches to locate data and ensure coherency: Within a node's private hierarchy outward searches look through tags and levels; At the shared level an associative search looks in the shared cache and directory; If coherence is required, an inward search proceeds down into the other nodes' hierarchies to find data and update coherence information. These search processes are a result of the explicit *coupling* of data and metadata (tags, sharing, state) in today's designs. This coupling forces the processor to look through layers of metadata to find the desired data, and incurs significant costs in energy and latency, while also limiting design flexibility.

In this work, we propose a cache hierarchy that *splits* data and metadata into two separate hierarchies. The *metadata hierarchy* tracks where data is located and how it is shared, and is implemented as an inclusive, coherent, and deterministic hierarchy with replacement policies tailored to the behavior of the metadata. This allows the *data hierarchy* to use a series of simple memory arrays, which need not enforce any particular inclusivity properties or even form an explicit hierarchy.

To separate the metadata and data hierarchies, we build upon the Direct-to-Data (D2D) [1] cache hierarchy, which replaces TLB lookups with access to location tracking information to identify the location (level and way) of the data in the private cache hierarchy. This work goes beyond the D2D design by 1) *extending* it to support coherent multicore processors, and, 2) *generalizing* it to develop an explicit separation of metadata and data hierarchies.

Our separate metadata hierarchy simplifies optimizations that build upon, or modify, data properties, behavior, or placement, as these are now tracked and controlled by a simpler, purpose-built metadata hierarchy. This enables many previously proposed cache optimizations under one common framework. These include:

- *Direct data access* [1] to lower latency by using the metadata information to skip level searches, way searches and directory indirections. (Guaranteeing determinism for metadata and extending it to support inter-node and shared level accesses for coherent multicore systems.)
- *Data placement and replication* to lower latency and reduce traffic by keeping data closer to where it is needed [2], [3], [4], [5], [6], [7]. (Leverage the metadata hierarchy's decoupling of cacheline addresses from placement, skipped directory indirections and providing an efficient mechanism to locate data.)
- *Dynamic coherence* to reduce coherence traffic through private/shared data classification [8], [9], [10], [11]. (Using classification from the sharing information in the metadata.)
- *Cache bypassing* to improve performance and energy by only installing a cacheline if it is likely to see reuse [12], [13], [14], [15], [16], [17], [18], [19]. (The metadata provides the functionality needed to bypassed some data while retaining the benefits of inclusion for other data.)
- *Dynamic indexing* to reduce conflict misses by changing the cache index functions [20], [21], [9]. (Using the metadata to store the "scrambling" value for the data.)

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.25

We begin with an overview of existing support for direct access to data in private cache hierarchies (Section II) and how we extend and generalize it to support coherent shared caches and a general split metadata/data hierarchy (Section III). We then demonstrate how we can use the split cache hierarchy by exploring optimizations for dynamic coherence, data placement, cooperative caching, and dynamic indexing (Section IV), and evaluate the overall effectiveness of the design (Section V). An overview of the coherence protocol is provided as an appendix.

The key contributions of this work are:

- A new cache hierarchy that splits metadata and data.
- The removal of directory indirections for all coherent reads as well as writes to private data.
- Demonstration of how multiple data-oriented optimizations can be integrated with a split hierarchy.

II. A Split Cache Hierarchy

The goal of this work is to show how splitting metadata and data provides an efficient framework for accessing data in a coherent multicore setting that supports a wide range of existing and future cache optimizations. As a first step, we extend the energy-efficient, but single-core only, Direct-to-Data (D2D) [1] cache framework to support multiple cores.

A. Private Hierarchies: Direct-to-Data (D2D)

The Direct-to-Data (D2D) design showed how private cache hierarchies could be made more efficient by replacing level-by-level searches with a single associative lookup that returns the cacheline's location. We generalize D2D as a hierarchy of *metadata stores* (MD1, MD2, etc.) which store cacheline *Location Information* (LI) separate from the data stores. With the Location Information from a metadata store, we can directly access the correct cache level and way for the cacheline, thereby avoiding the need for level-by-level searches, and improving efficiency.

To reduce overhead, the metadata information (MD) is organized into *regions*, each of which stores the Location Information (LI) for a number of adjacent cachelines, together with an address tag to identify the region. In D2D, each region contains 5 bits of Location Information per cacheline, which encodes the level and way (2-levels with 8-ways each, or memory). This encoding results in an implementation cost that is on par with the TLB and address tags it replaces, but still manages to deliver very effective tracking across the hierarchy: 99.7%, 87.2%, and 75.6% of L1, L2, and memory hits are tracked in the first level metadata (MD1), for a combined coverage of 98.8% of all memory accesses.

Figure 1 shows a D2D system with two metadata stores (MD1 and MD2). MD1 is virtually tagged, and can therefore provide the location of cachelines (A in L1, B in L2, C and D in memory) without the need for a separate TLB translation. MD2 is a larger backing store for MD1, and is

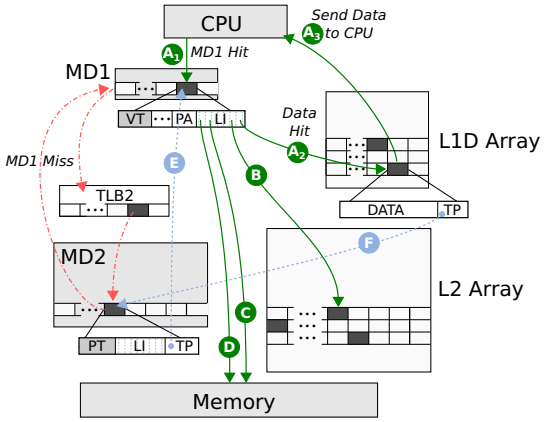

Figure 1. The basic D2D architecture with Metadata pointing to data in the L1, L2, and memory. MD1 is contains a Virtual Tag (VT) and the Physical Address (PA) for each region, as well as Location Information (LI) pointers for each cacheline in the region. MD2 has a Physical Tag (PT) and a Tracking Pointer (TP) that points to the active region.

physically tagged, thereby requiring a TLB2 lookup, which allows transparent handling of large pages.

Across the MD levels only one entry can be *active* at a given time, to avoid having to update multiple LIs atomically. To identify the active MD1 entry, each MD2 entry has a tracking pointer (TP, E) which points to the active MD1 entry if it is not itself active[1].

Since there is no way to search through the cacheline data (cachelines no longer have tags and can only be found through the MD entries), D2D implements inclusion between MD2 and the lower-level structures MD1, L1 and L2 (and also with L1-I, not shown in Figure 1). A consequence of this is that when an MD2 entry (region) is evicted, all its tracked cacheline residing in any of these lower-level structures must be evicted, or there would be no way to locate them again. As a result, the MD has precise information: if it claims that the cacheline is in a certain way in a given level, it will indeed be found there. We refer to this guarantee as *deterministic* information. To limit the effect of the forced eviction, the number of cachelines tracked in the MD2 is larger than the size of the L2 (typically by a factor four) and the replacement policy can favor choosing regions with few cachelines present.

B. Towards Shared Hierarchies

The goal of this work is to retain D2D's efficient direct access to data in a coherent/shared multicore setting. To do so, we rely on two key invariants:

1. *Deterministic Location Information*: the data location pointed to by the metadata is guaranteed to contain valid data

[1]When an MD1 entry is evicted, its LI information is copied to MD2. When a cacheline is chosen for eviction, the corresponding active MD entry needs to be updated. This is handled by adding tracking pointers to each cacheline (e, f). More details on this can be found in [1].

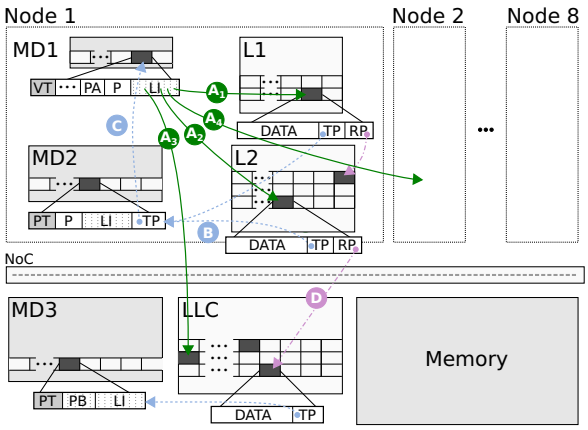

Figure 2. The generic architecture for D2M. TLB-2 and instruction cache not shown. The Cacheline in L2 stores metadata used for eviction: TP = Tracker Pointer; RP = Replacement Pointer. The MD2 entry is passive if its TP pointer identifies the active MD1 entry. The Presence Bits (PB) indicate which nodes track the region. The Private bit (P) indicates to the node that the region is private.

when the node accesses it. This guarantee means that as long as a node can access the metadata it can directly access the data, thereby enabling efficient direct access from the nodes without having to consult a centralized directory.

2. *Per-Region Private Classification*: when a region's private bit is set in a node, it is guaranteed that other nodes can neither make direct reads to the region data nor have private cached copies of its data. The Private Bits inform nodes as to when they can not make direct write accesses.

Taken together, these properties enable direct access for *all reads* (both to private and non-private regions) and *all writes to private regions*, regardless of where data is located in the hierarchy. This only leaves writes to shared regions not covered by direct accesses. The challenge is to ensure that the Location Information remains deterministic by ensuring that the metadata is updated with care.

III. Direct-to-Master (D2M)

A generic D2M configuration for up to eight nodes is shown in Figure 2. Each node contains a core (not shown), separate 8-way instruction and data caches (only L1-D is shown[2]), a unified 8-way L2, and 8-way first-level (MD1, virtually tagged) and second-level (MD2, physically tagged) metadata stores. All nodes are connected to each other and with the 32-way last-level cache and the metadata MD3 through the on-chip interconnect. D2M can also be applied to architectures with different numbers of levels and nodes.

As with traditional coherent cache designs, there is a single *master location* for each cacheline at any point in time, which is assigned to reply to coherent read requests (for example, cache lines in coherence states M, O, E or F). The master location may be a cache location, a remote node or memory. Valid cachelines that are not masters are referred to as *replicated cachelines* (e.g., in state S).

Another invariant maintained by D2M is *Metadata Inclusion*: as with D2D, we require that all data in the cache hierarchy be tracked in the metadata hierarchy, and, in particular, that data stored within a node must be tracked by metadata within that node. Each node in Figure 2 enforces inclusion between its MD2 and its MD1, L1, and L2. In order to not lose track of data, inclusion is also enforced between MD3 and the MD2s and LLC.

A. Extending Metadata Entries for Coherency

In order to be able to use the MD information directly in a coherent setting, and thus avoid a costly directory lookup, D2M needs the following extensions over D2D:

1. Track data in the (globally shared) LLC
2. Track master data in remote nodes
3. Track the set of sharing nodes (for coherent writes)
4. Classify regions
5. Handle eviction of shared data

The first two tasks are achieved by extending the location information encoding to cover four cases: 1) in a local cache level ("Level = 1 or 2"), 2) in the LLC ("Level = 3"), 3) in a specific remote Node ID ("NodeID"), or, 4) in memory ("MEM"). Compared with the 5 bit encoding for location information used in D2D, D2M needs one additional bit to add encodings for remote nodes. A possible encoding is shown in Table I. The 6 location information bits per cacheline is substantially smaller than a typical address tag ($\approx$ 30 bits).

000NNN	In NodeID: NNN
001WWW	In L1, way=WWW
010WWW	In L2, way=WWW
011SSS	Encoding of eight "symbols", where "MEM" is one of them.
1WWWWW	In LLC, way=WWWWW

Table I
Example 6-bit encodings of D2M location information corresponding to the architecture in Figure 2.

Note that D2M's location information identifies the *exact* location for data in its own L1 and L2 caches, and the LLC (level+way), but master cachelines in remote nodes are tracked only by their NodeID. This allows nodes to move their cachelines between their L1 and L2 without having to update metadata in other nodes. If a metadata lookup for a read access determines that a remote NodeID is the master location, a read request is sent directly to that node. The remote master node will perform a lookup in its MD2 (and possibly access its MD1 if it is active for this cacheline) in order to determine the location of the master cacheline.

The third task (tracking the set of sharing nodes) is handled by extending each region in MD3 with Presence Bits (PB bits), where each PB bit corresponds to one node.

[2]There are separate MD1-I and L1-I structures, not shown in the example. An additional field in MD2 indicates whether the region's active location information is in the MD1-I or MD1-D.

Private	$\#PB = 1$	Contains MD2 entries in exactly one node
Shared	$\#PB > 1$	Contains MD2 entries in more than one node
Untracked	$\#PB = 0$	Has an entry in MD3 but not in any nodes
Uncached	-	MD3 does not contain a metadata entry

Table II
METADATA REGION CLASSIFICATION.

A PB bit is set if the corresponding node has a valid MD2 entry. This implies that the node *may* have a valid copy of any cacheline in this region. If the bit is not set, it is *guaranteed* (thanks to MD2 inclusion) that the node does not have any valid cachelines for this region.

The presence bits (PB bits) steer the multicast scope of messages. For example, invalidations need only be sent to the nodes whose PB bit is set. Each of these "PB nodes" will perform a lookup in their MDs and update the location information to point to the NodeID of the requesting node and also invalidate any local cached copy of the data. The presence bits allow us to trivially classify regions as private or shared (Table II), and set the regions' private bits (P in Figure 2) in MD1/MD2

Note that the PB bits are recorded *per region*, while the location information tracks cachelines individually. The PB bits therefore provide a course-grain representation of the set of sharing nodes. This may lead to invalidation messages being sent to nodes that have never cached the corresponding cacheline because they have cached other lines in the region, and, therefore, have an MD2 entry for that region. This can increase traffic and MD2 lookups, but the simple private/shared classification enabled by the PB bits allows us to implement several key optimizations, leading to a large overall reduction in network traffic. For tracking 16 cachelines in a region across 8 nodes, the number of bits needed (PB(8) + 16*LI(6)) is on-par with a traditional fully mapped directory (16 * 9).

While the D2M architecture discussed in this paper have tag-less caches at all cache levels, it is possible to design a D2M system that interfaces traditional caches at some levels, e.g., unmodified cores with traditional TLBs and L1 caches, and traditional coherence interfaces (e.g., ARM's ACE interface) while achieving most of the reported D2M advantages.

B. Extending Replacement Metadata

One way to solve the fifth task (tracking evictions) is to add a Replacement Pointer (RP) to each cacheline. The RP identifies the *victim location* (a place in a higher-level cache, or in memory) that will become the master location for a cacheline when it is evicted. The victim location is determined prior to eviction and is identified by the RP (6 bits, using the same encoding as MD). All master cachelines must have a valid RP at all times, and by default that location is simply set to memory. Upon eviction of a master cacheline, the victim location identified by its RP automatically becomes the new master location for the cacheline, thereby significantly simplifying evictions.

To see how the Replacement Pointer simplifies evictions, consider the cacheline (A2) in L2 of Figure 2. This is a clean cacheline tracked by a private region in MD1. Its Replacement Pointer (RP) points to a victim location in the LLC (D). To evict the cacheline, we need to update the location information in MD1 to point to this victim location. This requires first finding its active MD entry. To do so, we use the cacheline's tracking pointer (TP) to locate its MD2 entry (B) and then follow the MD2 entry's TP to the active MD1 entry (C). We can now update the Location Information in the MD1 entry to point to the victim location from the cacheline's Replacement Pointer (D). After this update, the L2 cacheline location can be reused. Note that in this process we never had to search for an eviction location or wait for other lines to be evicted before evicting our line.

If the cacheline in L2 had been dirty, the dirty data would have been copied to the victim location using the RP pointer before updating the Location Information (LI) in the active MD (case *E* in Appendix).

Replicated (i.e., non-master) cachelines in L2 can be replaced silently. Their RPs do not point to a victim location, but instead contain the location of the master cacheline, which may be in the LLC, memory, or another node. On replacement, the same steps are performed, however, the updated LI will point to the master location recorded in RP, and not to a victim location. L1 cachelines may have victim locations allocated for them in L2 (see the replication optimization in Section IV) and are handled similarly to the example above.

C. Unified Data and Metadata Coherence Protocol

To support deterministic location information, and thereby enable cores to use their local metadata information to access data without a directory access, we need to keep global information in the metadata hierarchy coherent. This implies that changes to master cacheline locations (seen by all sharers) or to the PBs (which tell sharers who else may have a copy) may require coherent updates to metadata. Fortunately, several of the metadata coherence updates that are required can be handled as part of the "normal" data coherence protocol (e.g., per-cacheline state changes) or completely avoided by private region classifications. An overview of the coherence protocol is presented in Appendix.

IV. DATA-ORIENTED OPTIMIZATIONS

The metadata hierarchy's ability to specify arbitrary locations for data placement and efficiently locate data opens up some nearly trivial, but very important optimization: dynamic coherence by classifying regions as (private/shared), smart data placement by moving slices of the LLC "adjacent" to the node, and cooperative caching by choosing to move or replicate data on a per-cacheline basis.

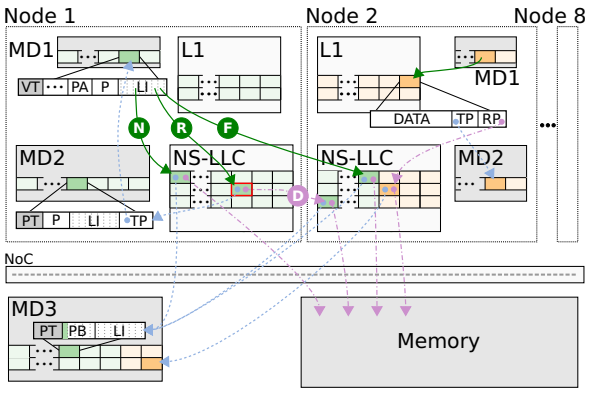

Figure 3. Near-side LLC (NS-LLC) with the LLC slices moved to the core side of the interconnect. Node 1 uses more data than Node 2, and has therefore more green cachelines in Node 2's NS-LLC slice.

A. Dynamic Coherence

By looking at the PB bits we can determine which other nodes may have cached copies of a cachelines in each region, which provides region-level data classification for free. We can classify regions into four types, as shown in Table II.

Private (80% of data[8]) and *untracked* regions allow for important optimizations. For private regions, we can completely avoid metadata and data coherence with respect to other nodes and allow for silent upgrades for writes. Any movements of a master cacheline (e.g., evictions) of a private region need only be recorded in the local MD1/MD2. When a region transitions from *private* to *shared* upon a first access from another node, the metadata information in the local MD2 is used to create an up-to-date MD3 entry. The region gets marked as not-private in MD1/MD2 (clear the P bit). These transitions are handled by the coherence protocol, as discussed in the Appendix.

Untracked regions can be evicted from LLC to memory without any metadata coherence updates, since they are only tracked in MD3. Depending on the MD2 and LLC sizes, most regions may become untracked before their cachelines are evicted from LLC.

The relatively large MD2 may limit the amount of data classified as private, since a region may remain in MD2 long after the node stops using a its data. A MD2 pruning heuristic may improve this situation by proactively evicting MD2 entries to make their region private or untracked. In the evaluation we use a very simplistic pruning heuristic that removes MD2 entries if a node receives an invalidate for a region for which its MD2 entry has no private L1/L2 copies and the TP pointer shows the MD1 entry is not active.

D2M implements data coherence, metadata determinism, region classification and dynamic coherence optimization (base on region classification) with one unified mechanism. In our evaluation, 68% of the accesses missing in the node's private caches are to private data.

B. Data Placement

Typically a request from a core has to first traverse the on-chip interconnect to reach the LLC. Such a *far-side LLC* is shown in Figure 2, while Figure 3 shows a *near-side LLC* (NS-LLC), where a slice of the LLC is co-located with each node. In this case each slice is a 4-way structure (for a total of 32-ways for all 8 nodes). The NS-LLC has the potential to reduce energy and latency if local data can be preferentially placed in its local slice and we can avoid associative searches across the full LLC. However, as with all NUCA-like arrangements, there are three challenges [5]: decoupling data placement and addressing, efficiently locating data, and choosing a placement policy. In this work we demonstrate how D2M elegantly addresses the first two of these, but we only explore extremely simple policies[3].

Handling the NUCA properties of a NS-LLC simply requires reinterpreting the LI encoding for LLC data (Table I) as 1NNNWW, where NNN identifies the node where the NS-LLC resides and WW identifies its way information. With this change, the MD1 of Node-1 can track data in its own NS-LLC slice (N) and in remote slices (Node-2, F, in Figure 3). Both can be accessed directly and efficiently using MD1 location information. Accesses to the remote NS-LLC require interconnect traffic, adding extra latency and energy equal to those of a regular access for a far-side LLC. The latency and energy for accessing a local NS-LLC, however, is much lower. Note that there is no inclusion enforced between a node's MD2 and its near-side LLC.

If the NS-LLC placement policy can promote local NS-LLC accesses over remote NS-LLC accesses, we can reduce interconnect traffic, energy, and average latency compared to a traditional far-side LLC. In the evaluation section we use a very simplistic heuristic based on NS-LLC *cache pressure*. For this simple policy, the local cache pressure is simply the number of replacements in each NS-LLC every 10k cycles, and is periodically shared with the other NS-LLCs. If the cache pressure for the local NS-LLC is lower than the other NS-LLCs, then the node allocates data in its local NS-LLC. (There would be no point in using the other NS-LLCs, since the survival time for data would be lower due to the higher pressure.) However, if the local NS-LLC has a higher pressure, 80% of the allocations are made locally and 20% are made remotely. In spite of this simple heuristic, our evaluation (Section V) shows that 58% of NS-LLC data accesses are to the local NS-LLC.

C. Cooperative Caching

In Figure 3, the NS-LLC slices (with 1/N the LLC size) take the place of the L2 caches in Figure 2. Yet our basic NS-LLC cannot replicate data (such as code) across each node's NS-LLC slice for lower latency, as a private L2

[3]Most proposed NUCA policies could readily be applied. D2M's contribution is in the mechanism, not the policy.

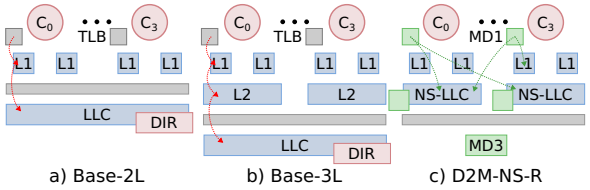

Figure 4. Processor configurations. Base-2L and D2M-NS-R have similar implementation costs while the cost of Base-3L is substantially higher due to its large L2 caches.

would. However, as already demonstrated, D2M is capable of handling replicated of data (e.g., the L2 cacheline in Figure 2 may also be replicated in some of the other nodes' L2s). The same technique can be used to support replicated data in the NS-LLCs, which gives us the potential to dynamically adjust whether the NS-LLC capacity is used as a private L2 (replicate) or a shared LLC. Figure 3 shows such a replicated cacheline (R) with its RP (D) pointing to the master location in the LLC of node 2. Inclusion is enforced between a node's MD2 and its replicated data in its near-side LLC.

This raises a classic question: when to replicate vs. preserve capacity. This questions has been addressed many time in many contexts (e.g., cooperative caching [6], [22], victim replication [23], RNUCA [5] and commercial NUMA/COMA designs [24]). Replication reduces overall system shared cache capacity, but can improve local latency. In this work we demonstrate how D2M can provide the infrastructure for supporting this trade-off efficiently. To that end, we evaluate only the following simple heuristic: Instructions are always replicated and data read from the MRU (Most Recently Used) position of a remote NS-LLC are replicated. In spite of this simplistic heuristic, our evaluation shows that NS-LLC data accesses to the local NS-LLC increase to 76% when both the NS-LLC allocation policy above and the replication heuristics are applied.

D. Dynamic Indexing

Another example of the flexibility that the MD provides is in using the Location Information to assign random (*scrambled*) indices to each region, thereby eliminating conflict misses caused by regular address patterns. To do so, we simply store a few bits of random index value with each region's metadata when it is loaded into the MD3 and use them to index into the data hierarchy's caches. This results in a dramatic energy reduction for a few application's with malicious access patterns, such as LU[4]. This feature demonstrates the flexibility of our general metadata hierarchy for attaching properties to each region, which can be easily extended to record cache bypass policies, prefetch statistics, read-only predictions, etc.

[4]We have not evaluated this scrambling solution compared to other alternatives [20], [25], [21], [9].

Frequency / Cores	2.5 GHz / 4 cores
Width: F / D / R / I / W / C	3 / 3 / 3 / 8 / 8 / 8
Size: ROB / IQ / LQ / SQ / IR. / FPR.	128 / 64 / 16 / 16 / 128 / 128
L1 Instruction / Data Caches	32kB, 64B, 8-way
– L1 Prefetcher	Stride prefetcher, degree 8
L2 Unified Cache *(Base-3L only)*	256kB, 64B, 8-way
L3 Shared Cache	4MB, 64B, 16-way
MD1 Instruction / Data *(D2M-only)*	128e, 1kB regions, 8-way
MD2 Unified *(D2M-only)*	4k, 1kB regions, 8-way
MD3 Shared *(D2M-only)*	16k, 1kB regions, 8-way

Table III
BASELINE PROCESSOR CONFIGURATION.

V. EVALUATION

A. Methodology

We use the gem5 ARM full-system simulator [26] configured to be similar to a contemporary energy-efficient processor as shown in Table III. We use CACTI [27] and McPAT [28] with 22nm technology, and published performance numbers to estimate latency and energy.

To evaluate D2M, we look at 5 categories of workloads, Parallel (Parsec [29]), HPC (Splash2x [30], [31]), Mobile (Chrome browser with Telemetry [32]), Server (mixes of SPEC CPU-2006 [33]) and Database (TPC-C [34] with MySQL/InnoDB). For Parsec, Splash2x, TPC-C and Spec-Mix, we use Ubuntu 14.04. We use region-of-interest for Parsec and Splash2x, sampling for the Spec-Mix and TPC-C. For Chrome, we use Telemetry on ChromeOS/Veyron with 14 of the more commonly visited websites, and start simulating when the page begins to scroll. We use the Freon graphics stack to avoid X11 overhead and NoMali GPU [35], [36] emulator to avoid measuring software rendering for the mobile workloads.

We compare with baseline systems modeled on contemporary mobile and server processors (see Figure 4). Base-2L is a 2-level baseline system with L1 caches and a shared LLC cache, similar to ARM's A57, but with an 8-way L1 with perfect way prediction (i.e., without the energy penalty for a parallel lookup). Our Base-3L system models a three-level cache hierarchy, with an additional 256kB L2 cache between the L1 and the LLC for each core.

For D2M we evaluate: L1 caches and a far-side LLC (D2M-FS), L1 caches and a near-side LLC with the simple policy for allocating near-side LLC space (D2M-NS), and D2M-NS with the heuristics for replicating data and instructions in near-side LLC and dynamic indexing (D2M-NS-R). All modeled systems have the same total LLC size, but the Base-3L systems have the highest implementation cost due to their large L2 caches.

To evaluate these systems we provide high-level metrics including coherency traffic, data traffic, locality in the NS-LLC, and average latency for memory accesses, as well as low-level (e.g., implementation-dependent) metrics for speedup (normalized to Base-2L) and energy.

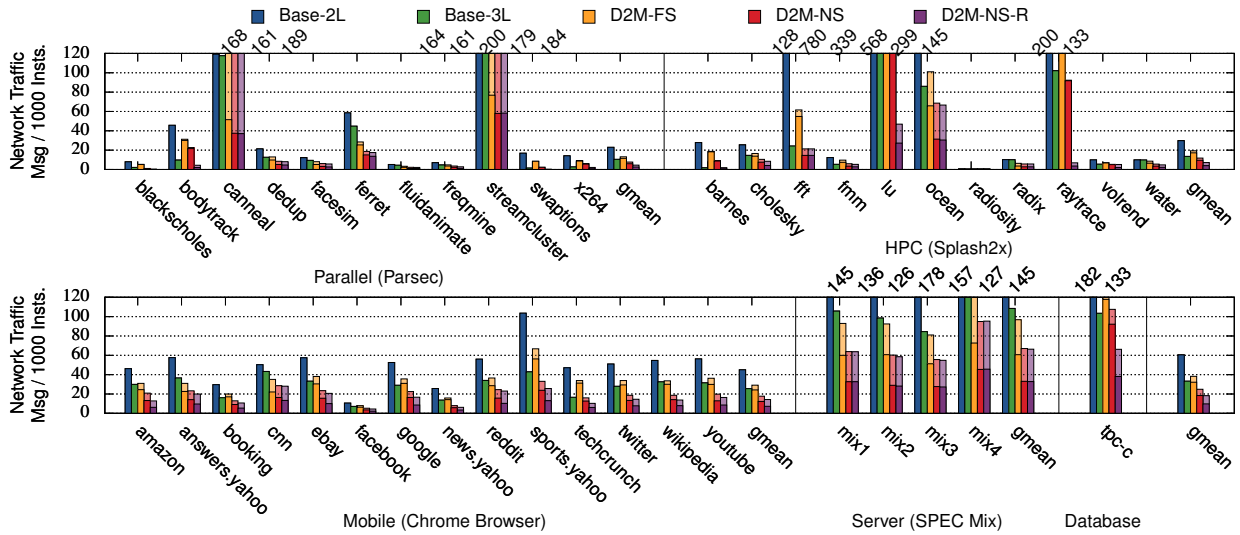

Figure 5. Network traffic in Messages per thousand instructions. The darker bars show basic network traffic and lighter bars show D2M-only traffic (e.g., new master updates). D2M reduces the amount of traffic by distributing the directory information (i.e., fewer network hops) and by placing the LLC on the near-side of the interconnect (i.e., no network traffic on near-side LLC hits).

	L1 Cache (%)				Near-Side Hit Ratio (%)							
	Misses		Late Hits		B-2L	B-3L		FS	NS		NS-R	
	L1I	L1D	L1I	L1D	I/D	I	D	I/D	I	D	I	D
Parallel	0.2	1.9	0.1	2.9	-	67	57	-	28	51	82	71
HPC	0.0	2.2	0.0	4.6	-	27	69	-	17	54	44	79
Server	0.4	3.6	0.3	9.5	-	100	78	-	82	83	95	83
Mobile	2.2	1.3	1.8	3.0	-	76	59	-	56	66	96	73
Database	8.8	3.3	6.2	4.2	-	59	41	-	26	34	97	72
Average	2.3	2.5	1.7	4.8	-	66	61	-	42	57	83	76

(L2 hits)

Table IV
PERCENT OF L2 AND LLC ACCESSES THAT HIT ON THE NEAR-SIDE OF THE INTERCONNECT (L2 FOR BASE-3L, AND NEAR-SIDE LLC FOR D2M) FOR INSTRUCTION AND DATA ACCESSES (I / D). ON AVERAGE, 67% (INSTRUCTIONS) AND 68% (DATA), OF THE LLC ACCESSES HIT ON THE NEAR-SIDE FOR D2M-NS-R.

	Invalidations			Private	
	Base-2L	Base-3L	NS-R	L1D	L1I
Parallel	100	100	119	70	5
HPC	100	107	71	46	4
Server	100	101	0	99	72
Mobile	100	109	16	89	62
Database	100	108	205	36	0
Average	100	105	82	68	29

Table V
NUMBER OF RECEIVED INVALIDATIONS (INCLUDING FALSE INVALIDATIONS) NORMALIZED TO BASE-2L (IN %), AND PERCENT OF MISSES TO PRIVATE MD REGIONS. ON AVERAGE, 68% OF THE DATA MISSES ARE TO PRIVATE REGIONS, WHICH REQUIRE NO COHERENCE TRAFFIC.

B. Traffic

Figure 5 shows the interconnect network traffic as number of sent interconnect messages. D2M-NS-R reduces the amount of network traffic by 70% on average largely due to reduced far-side LLC requests and private regions requiring no coherence traffic. The average data-only traffic (number of bytes moved) is reduced by 65% (graphs omitted due to limited space).

D2M-specific traffic (e.g., MD2 spill/fill) is shown in the lighter portion of the bars, while data coherence traffic is the darker portion. Canneal and streamcluster are outliers. Canneal is suffering from a exceptionally large number of MD2 misses while streamcluster is dominated by L1 misses going to memory where D2M can offer a latency advantage, but no traffic advantage. All other applications show a traffic advantage for D2M-NS-R.

Table V shows number of received invalidations relative to Base-2L, and the percent of misses that are to private MD regions. For the Server workloads (SPEC mixes) we see that all misses are to private MD regions, since the programs do not share any data. Across all workloads, 68% of the misses are to private pages. D2M also puts a lower pressure on the different SRAM structures (not shown). D2M accesses to MD3 are 11% as frequent as directory accesses of Base-2L and 27% of Base-3L. MD2 is accessed 58% as often as the L2-tags in Base 3-L.

C. Energy

Figure 6 shows EDP (static and dynamic) for the cache hierarchy normalized to Base-2L. D2M-NS generally performs better than D2M-FS. One exception is the cnn mobile benchmark, where EDP increases. This indicates that the simple heuristic for where to place cache lines in the NS-LLC made a non-optimal decision, but with replication (D2M-NS-R) this effect is reduced.

Across all benchmarks suites, D2M-NS-R increases performance and reduces network traffic and number of accesses to the far-side LLC. Combined, these improvements reduce EDP by 54% compared to Base-2L and by 40% compared to Base-3L.

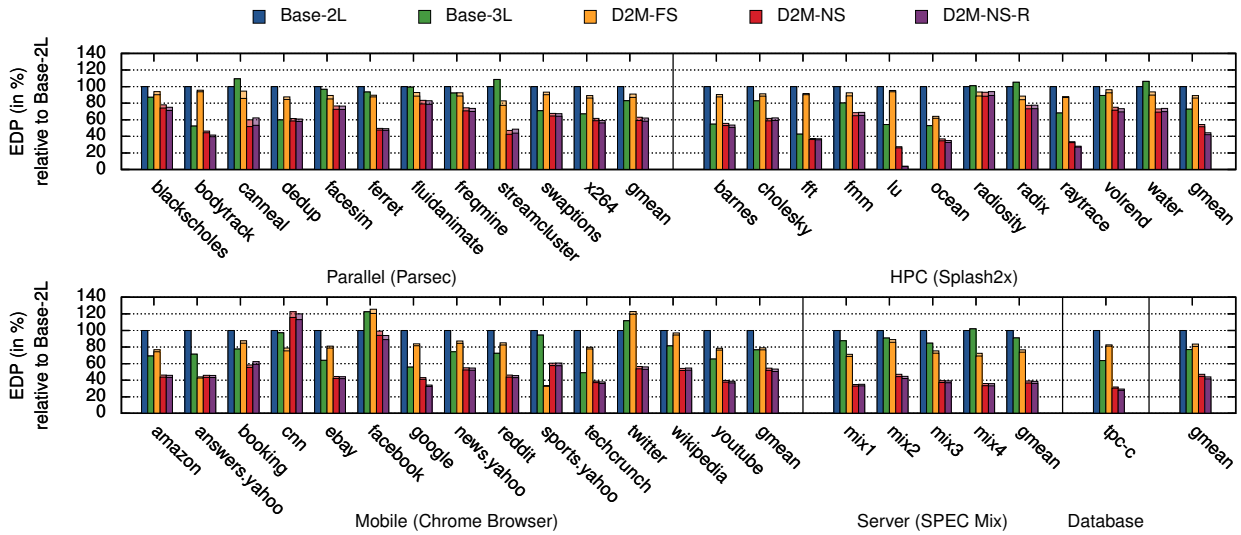

Figure 6. EDP of the cache hierarchy energy normalized to Base-2L. The darker bars show EDP contribution from standard structures and lighter bars show D2M-only structures (e.g., location trackers). Most energy is spent searching levels and moving data over the interconnect and between cache levels. D2M eliminates cache searches and reduces data movement by placing data on the near-side of the interconnect. This results in a average EDP reduction of 54% (D2M-NS-R).

D. Performance

As seen in Section V-C, D2M-NS-R cuts the number of transactions and the data traffic both by more than a factor of two. Thus, simulating a bandwidth-constraint system could potentially result in a 2x speedup. To avoid mixing the performance effects of traffic reduction and latency reduction, we have simulated a system with infinite bandwidth.

D2M-NS-R reduces the average L1 miss latency by 30% compared with Base-2L. Since we are simulating a fairly aggressive OoO CPU, not all of this latency reduction will translated directly into performance improvement. Figure 7 shows speedups over Base-2L with the L1 miss ratios and NS-LLC (or L2 for Base-3L) hit ratios in Table IV. D2M-FS improves performance through lower latency by an average of 5.7%. The near-side LLC (D2M-NS) reduces latency further by avoiding the performance cost of going over the interconnect on LLC hits for a 7% speedup. D2D-NS-R replicates shared data and instructions, and can therefore access the replicated data from the core's near-side LLC instead of going to the far-side cacheline copy. This increases the near-side hit ratio from 43% to 84% (instructions) and 58% to 76% (data) resulting in a 8.5% average speedup[5].

The Mobile and Database workloads have by far the most instruction misses. As a result, we see larger performance gains for them as the out-of-order processor cannot hide instruction misses. This is especially visible in Database, which has an 8.8% L1-I miss ratio on Base-2L. While D2M-NS significantly improves performance by 21% over Base-2L, only 26% the LLC instructions access hit in the near-side LLC. With D2M-NS-R, we are able to service 97% of the L1-I misses from the NS-LLC with our simple replication heuristic. This gives a net speedup of 28% over Base-2L by automatically using the near side LLC slice as a private L2 cache for instructions.

Base-3L has an L2 cache that filters LLC access resulting in a 4% average speedup over Base-2L. However, Mobile and Database still experience significant L2 misses, with only 59% of L2 and LLC accesses hitting in the L2 for Database. Base-3L must still access the LLC 41% of the time. In such cases, the L2 cache may hurt performance due to the latency of searching it first. D2M-NS-R does not have a L2 cache, but it can use the core's slice of the NS-LLC for instructions (1MB NS-LLC vs. 256kB L2). This results in 6.8% (vs. 4% for Base-3L) and 28% (vs. 13% for Base-3L) speedups over Base-2L for Mobile and Database, respectively.

[5]We evaluated scaling the MD1, MD2, and MD3 entries from 1x (128, 4k, 16k) to 2x and 4x and found the average speedup went from 8.5% (1x) to 9.5% (2x) while the number of direct accesses to the NS-LLC (MD1 and NS-LLC hits) increased from 78% to 86%.

VI. Related Work

NUCA caches have been proposed to address the wire delays in LLCs [2], [3], [4], [5], [6], [7]. To optimize for varying access latencies, many policies have been suggested to migrate or replicate data closer to where it is needed. Our NS-LLC topology is reminiscent of NUCA caches in that each node has a slice of the LLC cache residing close to it. Our approach decouples the cacheline address from its placement, provides an efficient cacheline location mechanism, and can replicate cachelines. All these features fall out "for free" from our data tracking. These features simplify

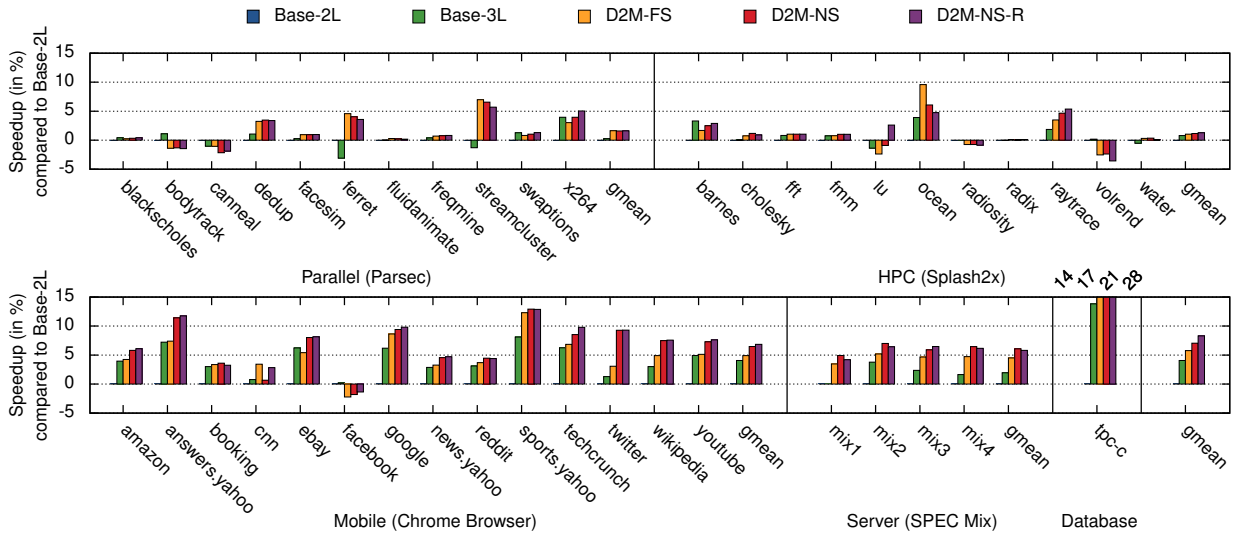

Figure 7. Speedup in percent compared to Base-2L for systems with infinite bandwidth. From left to right: Base-2L (always zero), Base-3L, D2M-FS, D2M-NS, D2M-NS-R. On average, D2M-NS-R improves performance by 8.5% (max 28%) over Base-2L

efficient NUCA implementations, while the placement and replication policies we investigated are simple, with much room for improvement.

DDCache [7] provides data replication, but adds a sharer list to each L1 cacheline whereas D2M has one sharer list per region. When replicating data in L2/NS-LLC DDCache increases the latency since the local slice need to be searched first before accesses the slave. In D2M, most accesses are directed to the slave node immediately after the MD1, which results in shorter latency since D2M does not have to search for the data. DDCache partially decouples data and metadata by keeping extra tags, but when the data is local they are still tightly coupled. D2M provides separate hierarchies for tracking metadata (MD) and data (cache), whereas DDCache uses tag over-provision.

Directory indirection avoidance has been proposed for pages classified as private [8]. Our approach also avoids directory indirection for shared data by storing the necessary information in MD1 and enabling direct read accesses to LLC data classified as shared, without the involvement of other nodes. This is a key component that enables NS-LLC.

Coarse-grained tracking and indirect lookup is used by Zebchuk et al. [37] in the L2 and last-level caches to reduce tag-area, eliminate snoops and improve prefetching. Seznec [38] uses pointers to reduce tag storage by replacing the page number in the tags with a pointer to the page in a page number cache. This reduces the tag storage since the pointer is much smaller than the page number. However, both still use tag-based L1 caches and do not address multi-core issues. Boettcher et al. [39] and Sembrant et al. [40], [1] extend the TLB with cacheline way information to reduce L1 cache energy. Boettcher et al. [39] keep the tag array and treat the TLB way information as hints. Sembrant et al. [40] provide precise tracking and can therefore eliminate the tag array. Their work was later extended to provide direct accesses to data in multiple cache levels and memory (D2D), as discussed in Section II. Hallnor [41] suggests an indirect index cache to locate data in a managed cache.

Private/shared classification has been proposed based on hardware detection mechanisms [42], [43] and OS support [8], [5], [44]. The OS can classify data at page-granularity with minimal hardware support, but requires a complex shootdown and purging of cachelines when classification changes. Hardware classification can provide a finer granularity, but may have high storage requirements. Data classification has been used to steer NUCA placement [2], [3], [4], [5] and for filtering coherence activities [37] in broadcast protocols, among other things.

Our region classification solution is similar to earlier hardware mechanisms, even though the D2M detection supports region granularity and utilizes its existing coherence hardware to facilitate classification. D2M does not require shootdown or purging on re-classification.

Framework for optimization. While each of these related works addressed important problems, none of them provide a general framework that can enable the wide range of optimizations that D2M supports. D2M's split metadata and data hierarchies and efficient data location lookup enable us to implement and combine solutions to each of these problems.

VII. Conclusions

In this work we have introduced the Direct-to-Master (D2M) cache design that splits data and metadata into two

separate cache hierarchies. To do so we provided *deterministic* metadata information across a coherent system. We accomplished this by extending D2D's data tracking infrastructure to be able to index remote nodes, track sharers, and store explicit replacement locations. This allows us to keep location information coherent, while providing for efficient (direct, coherence-free) access to private data, low-overhead (multicast vs. broadcast) coherence updates, and low-latency (no directory access with local sharing information) access to shared data.

The separation of the metadata and data hierarchies is not only efficient, but also provides a flexible framework for implementing many data-oriented optimizations. We illustrated this by combining optimizations for data placement, coherence, and indexing within the D2M framework. In particular, we demonstrated how a near-side LLC, wherein we moved the LLC slice to the core-side of the interconnect, can significantly lower latency and energy. This optimization comes for "free" as D2M allows us to arbitrarily direct and replicate a node's data to across the LLC slices. While the heuristics we used for this were decidedly simple, D2M's flexibility has addressed the underlying NUCA problems of decoupling placement and addressing and of efficiently accessing the data.

D2M was evaluated against server and mobile processor configurations and a wide range of server, database, mobile, and parallel workloads. Across these configurations, D2M provided significant gains in cache hierarchy EDP (54% vs. mobile and 40% vs. server) and reduced network traffic by an average of 70%. D2M is especially valuable for workloads with large instruction footprints (database and mobile) as the near-side LLC can automatically serve as a large private L2, thereby reducing instruction misses by up to 97%.

ACKNOWLEDGEMENT

This work was supported in part by the Swedish Foundation for Strategic Research (grant FFL12-0051), the Uppsala Programming for Multicore Architectures Research Center, and the Swedish National Infrastructure for Computing (SNIC) at NSC (Linköping) and UPPMAX.

APPENDIX

We explain the unified coherence protocol for data and metadata at a high level through nine important coherence examples. Most examples are also shown graphically for clarity in Figure 8.

The D2M coherence protocol relies in part on an atomic update mechanism, such as those used previously to implement deterministic directories, for example Sun Microsystem's WildFire, SunFire 10k and 15k servers [24], [45]. These server's coherence solution scaled to about 100 CPUs and relied on a blocking mechanism at the directory that only allowed for one outstanding coherence transaction for each cacheline. This ensured that the directory state always reflected the correct location for the "master cacheline" (deterministic). A similar mechanism is implemented at the MD3 level of D2M, guaranteeing that there can be at most one ongoing operation *per region* that could make changes to the region's metadata. The blocking mechanism can be implemented as a set of hashed lock bits that are explicitly blocked and unblocked by the coherence protocol. Simulation show that 1K lock bits result in a negligible collision rate.

D2M is optimized to handle the most important L1 miss event efficiently: read misses. To put the significance of each event in perspective, the events per kilo memory operation (PKMO) are provided in the text with each event. On average, there are 13.3 read misses and 2.4 write misses (PKMO) and an additional 19 and 21 late hits (PKMO), respectively[6].

A: Read Miss, MD1/MD2 Hit (12.5): For read misses where the location information is present in the MD1 (9.2) or MD2 (3.3), the information is guaranteed to be deterministic with respect to a cacheline's master location. This allows D2M to satisfy a read miss with a read request directly to the master (hence the name: D2M), without the need to interact with MD3. A master location in LLC (8.9) or memory (2.7) can be accessed directly, similarly to D2D, while reading a master cacheline in a remote slave node (0.8), e.g. in a remote L1, requires an indirection through that node's active MD1 or MD2 to determine its location in that node. Finally, the cacheline's LI in the requesting node is updated to point to the new local location (e.g. in L1), while the global master location stays unchanged.

B: Write Miss, Private Region, MD1/MD2 hit (1.7): The private region is not tracked by any other node and its data is not cached in any other node. Thus, the requesting node can read data directly from the master location, similar to example A, and update the local MD1 to point to the new local location, which now becomes the new writable master location. This action makes the LI in MD3 invalid for private regions.

C: Write Miss, Shared Region, MD1/MD2 hit (0.72): The requesting node sends a blocking ReadEx request to MD3; Once the region has been blocked[7], a DirectReadEx is sent to the master and Invs are sent to the slave nodes with PB bits set (excluding master node); The slaves send[8] Acks to the requesting node and set their LIs to point to the requesting node's NodeID (i.e., the new master location); The requesting node sends Done to MD3 when all Acks[9] have been received, which unblocks the region, and points

[6]PKMO are average occurrences per 1,000 memory operations across all application categories for the basic D2M-FS architecture in the evaluation.

[7]This guarantees that there are no other ongoing blocking operations for the region.

[8]The sending is delayed until the slave node has no outstanding direct read requests (examples A and B) for the region in order to guarantee that there are no outstanding request for the old master location.

[9]Inv and Ack carry the number of Acks to expect.

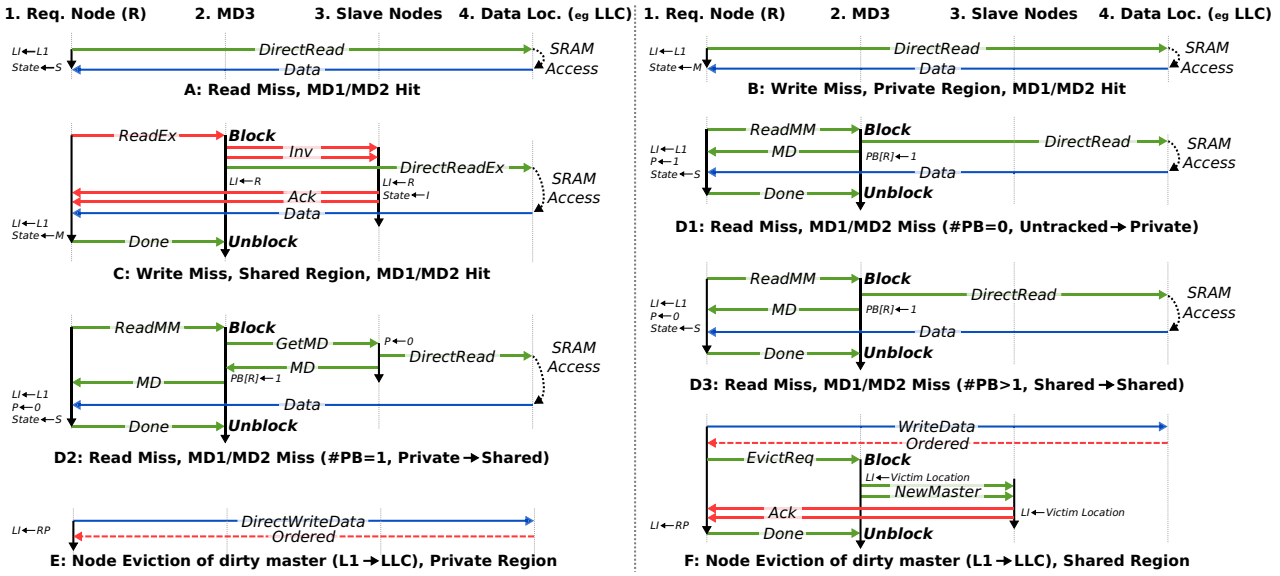

Figure 8. Coherence examples.

its LI to the local location in L1. The location in L1 is now the new master location.

D: Read Miss, MD1/MD2 Miss (0.82): This requires that we retrieve both the region metadata and the requested data. A blocking ReadMM request (Read Metadata Miss) is sent to MD3. Once the region has been blocked there are four possible outcomes depending on the classification of the region (i.e., the number of PB bits set):

1) **#PB == 0: Untracked→Private** (0.32): MD3 replies to the requesting node with metadata with the private bit set.
2) **#PB == 1: Private→Shared** (0.02): Note that MD3 LIs for private regions are invalid. MD3 sends an GetMD to the single slave node; It clears its private bit and sends[8] metadata to MD3; LIs pointing to local location in the slave node (e.g. in L1D) are converted to point to the slave NodeID and stored in MD3 metadata; MD3 Replies with metadata with the private bit cleared to the requesting node.
3) **#PB > 1: Shared→Shared** (0.14): MD3 sends metadata with the private bit cleared to the requesting node.
4) **MD3 miss: Uncached→Private** (0.34): Create a new MD3 entry with all LI set to "MEM"; Reply with metadata with the private bit set to the requesting node.

For all of the above cases (D1-D4), MD3 also sets the PB bit for the requesting node and sends a *direct read* request to the master on behalf of the requesting node; When the requesting node has received the data and metadata, it send Done to MD3, which unblocks the region.

E: Node Eviction of dirty master, Private Region: Copy data to the victim location (e.g., in LLC or MEM) stored in RP; Change LI of the active MD1/MD2 in the requesting node to point to the victim location.

F: Node Eviction of dirty master, Shared Region: The requesting node copies data to the victim location stored in RP; The requesting node sends[10] EvictReq to MD3; Once the region has been blocked[12], MD3 sends NewMaster messages to slave nodes with PB bits set; They send[8] Acks to the requesting node and point their LI[13] to the new master location; The requesting node sends Done to MD3 when all Acks[14] have been received, which unblocks the region.

We can observe that the most common cases, A and B (14.2 PKMO, or 90% of all misses) do not require any "directory" lookups in D2M, while the baseline[15] implementation does. The cost of the C and D cases are on-par with the baseline (assuming the MD3 lookup is as expensive as a traditional directory lookup), with the exception for case D2, which is fortunately quite rare (0.02 PKMO).

[10]The sending can be done once it can be guaranteed that all subsequent reads to the victim location will return the copied data.

[11]If MD3's LI does not point to the requesting node, it is no longer the master and the eviction is Nack-ed.

[12]The sending can be done once it can be guaranteed that all subsequent reads to the victim location will return the copied data.

[13]In the case of shared cachelines, their RP is instead updated.

[14]NewMaster and Ack carry the number of Acks to expect.

[15]Base-2L architecture in the Evaluation section, with the same implementation cost as D2M-NS-R.

References

[1] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The Direct-to-Data (D2D) Cache: Navigating the Cache Hierarchy with a Single Lookup," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2014.

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[3] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2003.

[4] B. M. Beckmann and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2004.

[5] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2009.

[6] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2006.

[7] H. Hossain, S. Dwarkadas, and M. C. Huang, "DDCache: Decoupled and Delegable Cache Data and Metadata," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[8] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2011.

[9] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García, "ASCIB: Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses," in *Proc. Int. Symposium on Low Power Electronics and Design (ISLPED)*, 2012.

[10] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-Aware Mechanism to Detect Private Data in Chip Multiprocessors," in *Proc. Int. Conference on Parallel Processing (ICPP)*, 2013.

[11] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The Effects of Granularity and Adaptivity on Private/Shared Classification for Coherence," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, pp. 26:1–26:21, Aug. 2015.

[12] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-level Caches," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2011.

[13] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal Bypass Monitor for High Performance Last-level Caches," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[14] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[15] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2012.

[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 *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2012.

[17] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[18] S. Gupta, H. Gao, and H. Zhou, "Adaptive Cache Bypassing for Inclusive Last Level Caches," 2013.

[19] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "The Reuse Cache: Downsizing the Shared Last-level Cache," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2013.

[20] A. Agarwal and S. D. Pudar, "Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 1993.

[21] C. Zhang, "Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2006.

[22] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2006.

[23] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2005.

[24] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," in *Proc. Int. Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[25] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses," in *Proc. Int. Symposium on High-Performance Computer Architecture (HPCA)*, 2004.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[27] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," Hewlett Packard Labs, Tech. Rep., 2009.

[28] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2009.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 1995.

[31] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Proc. Int. Symposium on Workload Characterization (IISWC)*, 2008.

[32] Telemetry, *https://www.chromium.org/developers/telemetry*.

[33] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006.

[34] Transaction Processing Performance Council, *http://www.tpc.org/*.

[35] A. Sandberg, "NoMali: Understanding the Impact of Software Rendering Using a Stub GPU," in *Second gem5 User Workshop*, 2015.

[36] R. de Jong and A. Sandberg, "Mobile Benchmarking Done Right: Understanding the System Impact of Mobile GPUs," in *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2016.

[37] J. Zebchuk, E. Safi, and A. Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2007.

[38] A. Seznec, "Don'T Use the Page Number, but a Pointer to It," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 1996.

[39] M. Boettcher, G. Gabrielli, B. M. Al-Hashimi, and D. Kershaw, "MALEC: A Multiple Access Low Energy Cache," 2013.

[40] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "TLC: A Tag-Less Cache for Reducing Dynamic First Level Cache Energy," in *Proc. Int. Symposium on Microarchitecture (MICRO)*, 2013.

[41] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2000.

[42] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence Protocol Optimization for Both Private and Shared Data," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[43] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[44] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace Snooping: Filtering Snoops with Operating System Support," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[45] A. Charlesworth, "The sun fireplane system interconnect," in *Supercomputing, ACM/IEEE 2001 Conference*, Nov 2001, pp. 2–2.