CANDY: Enabling Coherent DRAM Caches for Multi-Node Systems

Chiachen Chou† Aamer Jaleel‡ Moinuddin K. Qureshi†

†School of Electrical and Computer Engineering
Georgia Institute of Technology
{cc.chou, moin}@ece.gatech.edu

‡ NVIDIA
ajaleel@nvidia.com

***Abstract*—This paper investigates the use of DRAM caches for multi-node systems. Current systems architect the DRAM cache as *Memory-Side Cache (MSC)*, restricting the DRAM cache to cache only the local data, and relying on only the small on-die caches for the remote data. As MSC keeps only the local data, it is implicitly coherent and obviates the need of any coherence support. Unfortunately, as accessing the data in the remote node incurs a significant inter-node network latency, MSC suffers from such latency overhead on every on-die cache miss to the remote data. A desirable alternative is to allow the DRAM cache to cache both the local and the remote data. However, as data blocks can be cached in multiple DRAM caches, this design requires coherence support for DRAM caches to ensure correctness, and is termed *Coherent DRAM Cache (CDC)*.**

We identify two key challenges in architecting giga-scale CDC. First, the coherence directory can be as large as few tens of MB. Second, cache misses to the read-write shared data in CDC cause longer delay due to the need to access the DRAM cache. To address both problems, this paper proposes *CANDY*, a low-cost and scalable solution that consists of two techniques for these challenges. First, CANDY places the coherence directory in 3D DRAM to avoid SRAM storage overhead, and re-purposes the existing on-die coherence directory as a *DRAM-cache Coherence Buffer* to cache recently accessed directory entries. Second, we propose *Sharing-Aware Bypass*, which dynamically detects the read-write shared data at run-time and enforces such data to bypass the DRAM cache. Our experiment on a 4-node system with 1GB DRAM cache per node shows that CANDY outperforms MSC by 25%, while incurring a negligible overhead of 8KB per node. CANDY is within 5% of an impractical system that has a 64MB SRAM directory per node, and zero cache latency to access the read-write shared data.

I. Introduction

To mitigate the DRAM bandwidth wall, recent packaging advancements enable DRAM chips to be placed closer to the processors in the same package [1–4]. The DRAM chips are often stacked with multiple layers, and referred to as *3D-DRAM*, providing much higher bandwidth compared to commodity DIMM-based DRAM [5, 6]. Several leading industry vendors announce systems with 3D-DRAM, for example Intel's Xeon Phi, AMD's Radeon R9 and NVIDIA's Pascal [7–10]; such technology has advanced from prototype to commercial adoption. 3D-DRAM, however, cannot fully replace commodity DRAM in a cost-effective manner; therefore, future systems are likely to contain both 3D-DRAM (for high bandwidth) and commodity DRAM (for high capacity).

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

An attractive use of 3D-DRAM is to architect it as a hardware-managed cache that is an intermediate level between the on-die caches and the main memory. Although recently there are many research proposals in enabling large DRAM caches, these studies focus on only single-node systems [11–22]. In contrast, this paper studies DRAM caches for multi-node systems. Figure 1(a) shows a multi-node system, where each node has a 4-core multi-processor, an on-die L3 cache, a DRAM cache, and a DDR-based main memory.

To use DRAM caches in a multi-node system, one practical design is to restrict the DRAM cache in each node to store only the data that belongs to the local node (i.e., *Local Data*). Figure 1(b) shows such design, termed *Memory-Side Cache (MSC)* by the industry vendor [7, 8]. Node 0's DRAM cache holds only the data from Node 0 (⬡, and □ symbols in Figure 1); similar for Node 1's DRAM cache (◯, and △). Any data line is stored in at most one DRAM cache, so MSC is implicitly coherent and obviates the need of any coherence support for DRAM caches. However, MSC constraints the system to rely on the small on-die cache for the data from the remote node (i.e., *Remote Data*). As accessing the remote data incurs long network latency, MSC suffers from a significant latency overhead of on-die cache misses to the remote data.

A desirable alternative is to allow the DRAM cache to cache both the local and the remote data to mitigate the long network latency overhead. Given that the DRAM cache capacity is in the range of gigabytes, it is capable of holding a much larger working set, hence reducing the needs to access remote nodes. Figure 1(c) shows such design: Node 0's DRAM cache holds all request initiated from processors in Node 0 and caches the data from both Node 0 and Node 1 (□, ◯ and △); Node 1's DRAM cache does the same for requests from Node 1 (⬡, and ◯). As this type of DRAM cache stores data blocks from any node, a shared data block can be stored in multiple DRAM caches (e.g., ◯ stored in DRAM caches of both Node 0 and Node 1). Therefore, the DRAM caches must be kept coherent for correctness, and we term this design *Coherent DRAM Cache (CDC)*.

To realize CDC, we investigate how current multi-node systems maintained cache coherence: On-die L3 caches usually rely on a directory-based coherence protocol, which uses a *Coherence Directory* (also termed *Directory Cache*) to track all data blocks that are currently cached in the system [23–27].

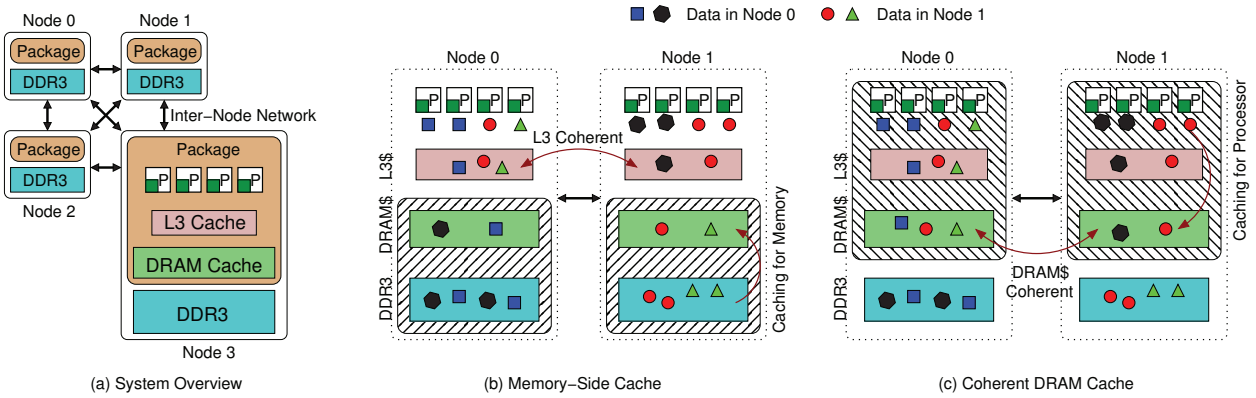

Fig. 1. (a) Overview of a multi-node System. Each node has a 4-core multi-processor, a shared on-die cache (L3 cache), a DRAM cache, and a DDR-based main memory. Figure (b) and (c) show different usage of DRAM caches in multi-node systems. Each symbol in (b) and (c) represents a data block in the memory. (b) Memory-Side Cache. The DRAM cache in node 0 is allowed to cache only the data that is in Node 0 (□ and ⬡ in this case); same for Node 1 (◯ and △). On-chip L3 caches must still be be kept coherent. (c) Coherent DRAM Cache. The DRAM cache stores data from both nodes. Node 0's DRAM cache stores data blocks requested by node 0 and caches data from both Node 0 and Node 1 (□, ◯ and △); same for Node 1 (⬡ , and ◯). In this case, DRAM caches must be kept coherent.

To serve a cache miss, the coherence protocol forms two steps: accessing the coherence directory for the current coherence state, and then retrieving the most recent copy of the data. To employ the directory-based protocol for CDC, we identify two key challenges.

First, the coherence directory for giga-scale DRAM cache is as large as few tens of megabytes. As the size of the coherence directory must be proportional to the cache capacity [28–31], a 1GB DRAM cache necessitates a 64MB storage for the coherence directory. Such large structure incurs a prohibitive overhead of both storage and latency. As accessing the coherence directory is on the critical path, a low-latency coherence directory is the key to the performance.

Second, to retrieve the most recent copy of data from a DRAM cache causes long latency in CDC. When a cache miss accesses a data block that is cached and modified by another cache, the protocol must request the owner to write back the most recent data to keep caches coherent. Such critical operation is termed *Request-For-Data*; in CDC, the cache access latency of Request-For-Data is a DRAM cache access latency and incurs a significant overhead.

To overcome these challenges and enable CDC for multi-node systems with low overheads, this paper makes the following contribution:

1) To the best of our knowledge, this is the first paper that studies DRAM caches for multi-node systems, and quantifies the performance of different usages of DRAM caches. We show that an impractical CDC has a potential performance improvement of 30% over MSC.
2) To architect high-performing CDC, we identify two key challenges, and propose *DRAM Caches for Multi-Node Systems* (*CANDY*), composed of two orthogonal techniques to address the challenges, namely:
   a) We propose to place the coherence directory in the 3D-DRAM to avoid the SRAM storage overhead. To mitigate the 3D-DRAM access latency for the coherence directory, we propose to re-purpose the existing on-die coherence directory as a *DRAM-Cache Coherence Buffer (DCB)* to store the recently accessed coherence information. We further improve DCB by co-optimizing the organization of the DCB and the coherence directory. As the on-die coherence directory is already provisioned in systems without DRAM caches, DCB does not incur any SRAM storage overhead.
   b) We also propose *Sharing-Aware Bypass (SAB)* to mitigate the DRAM cache access latency of the Request-For-Data operation. Our insight is that Request-For-Data accesses only read-write shared data, which transition among nodes, and need not to be stored in DRAM caches. SAB dynamically detects read-write shared data at run-time and enforces such data to bypass DRAM caches. Thus, read-write shared data are stored only in L3 caches, and the cache access latency of Request-For-Data becomes an SRAM cache latency. SAB is effective and simple with a negligible storage overhead of 8KB SRAM per node (less than 0.2% L3 area).

We evaluate a 4-node system, where each node has a 1GB 3D-DRAM, architected as a Memory-Side Cache or as a Coherent DRAM Cache, and we run 16-core parallel applications from various benchmarks suites. Our experiment results show that DCB improves performance over the design without DCB by 10%, while SAB provides an additional 4% improvement over DCB. Overall, our proposed CANDY has 25% improvement over the baseline MSC with a negligible overhead of 8KB SRAM per node; CANDY is effective to obtain almost all the potential 30% improvement of an impractical CDC that incurs a prohibitive overhead of 64MB SRAM for coherence directory with zero cache access latency for the Request-For-Data operation.

II. Background and Motivation

In this section, we examine how current systems maintain cache coherence via coherence directory, and show two key problems of the giga-scale Coherent DRAM Caches: architecting the coherence directory and mitigating the Request-For-Data latency. We also show the potential performance improvement by an impractical CDC that overcomes the two problems without the overheads.

A. DRAM Caches in Multi-Node Systems

Prior studies focus on single-node systems with only one DRAM cache.[1] This paper investigates the use of DRAM caches for multi-node systems, where each node employs one DRAM cache. One practical design is *Memory-Side Cache (MSC)*, which allows the DRAM cache to store only the local data. For example, Intel's Xeon Phi features a direct-mapped, 64B line-size DRAM cache as a Memory-Side Cache [7, 8]. Although MSC does not need any coherence support, it constrains the system to use last-level caches (LLC) for the remote data; a LLC miss to the remote data incurs a significant latency overhead due to the inter-node network traversal.

Such constraint restricts the capability of DRAM caches. With a much larger capacity, DRAM cache is capable of holding a large working set including both the local and the remote data. If DRAM caches were able to keep the remote data, a LLC miss to the remote data could hit in the DRAM cache and avoid the latency overhead. To relax MSC's constraint, we allow DRAM caches to store both local and remote data. In this case, a cache block shared by the processors can be stored in multiple locations, and must be coherent for correctness. To distinguish, we refer to this kind of DRAM cache as *Coherent DRAM Cache (CDC)*. In CDC, the DRAM cache becomes the last-level cache (L4 in our configuration) and the point of cache coherence. For simplicity, we interchangeably use L4 caches and DRAM caches in CDC.

B. Directory-based Coherence Protocol

Current multi-node systems typically use directory-based coherence protocol because of its superior scalability [23–25]. One appealing design is to use *Sparse Directory* to track the cache blocks that are currently being cached (and need to be coherent) in the system. Commercial products implement sparse directory by dedicating part of the die area. For example, AMD's Magny-Cours has 1MB SRAM structure (named *Probe Filter* by AMD) for the sparse directory [32]. As sparse directory stores coherence information, we term such structure *Coherence Directory (CDir)*[2], to distinguish from *Tag Directory (TDir)*, which stores tag information for data blocks in the cache. Figure 2 shows how CDir and TDir are organized.

In Figure 2, the coherent cache stores the cache blocks requested by the local processors. TDir, the tag directory, is associated with one coherent cache, and has the state information

[1]With only one DRAM cache in the system, the DRAM cache is implicitly coherent, and needs no coherence support.

[2]Sparse Directory, also referred to as *Directory Cache* in other literatures [27–31], is not to be confused with full coherence directory [23].

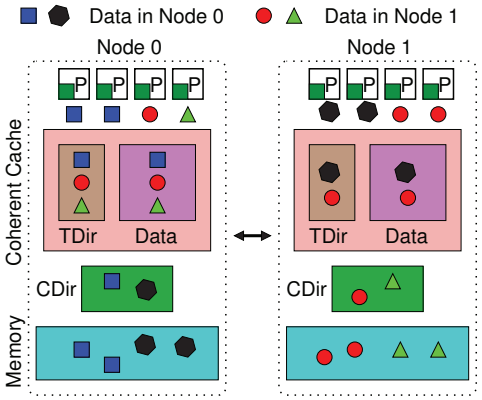

Fig. 2. Overview of Tag Directory (TDir) for cache, and Coherence Directory (CDir) for data. Notice that they are responsible for different data blocks that are currently being cached in the system.

(e.g., valid, dirty) for all lines that are currently stored in its associated cache. For example, in Node 0, processors request for three data blocks (□, ◯, and △). Therefore, the TDir in Node 0 has information of these three lines. In contrast, CDir, the coherence directory, is associated with the memory in a node, and has the coherence information (e.g., shared, exclusive, etc., depending on the coherence protocol) for all the cache blocks that belong to its memory and are currently being cached (by any node) in the system. For example, two blocks (△ and ◯) from Node 1 are currently being cached; the CDir in Node 1 keeps information of these two lines. Node 1 is referred to as the *Home Node* for these two lines.

In directory-based protocol, the home node is responsible for retrieving the most recent copy of the data on a L4 cache miss: First, the home node accesses the CDir for the coherence information of the requested data block. Based on the request type of the cache miss and the coherence state of the requested data block, the home node takes different operations. For instance, if the requested data block is uncached, the home node accesses the memory to retrieve the data. Other operations include invalidating a copy in others' caches, or requesting the owner to write back the most recent data. To apply the directory-based coherence protocol to CDC, we identify two key problems, described in the following sections.

C. The Need for Large Coherence Directory

To use directory-based coherence protocol for CDC, we need a CDir for the L4 DRAM cache. We examine whether the current architecture that keeps on-die L3 cache coherent can be applied to the L4 caches.

1) On-Die L3 Coherence Directory: In the baseline MSC, an on-die L3 coherence directory, termed *OnDie-CDir*, is responsible for maintaining L3 cache coherence. For example, the aforementioned Probe Filter by AMD. In CDC, the point of coherency is the L4 cache, and one simple way to organize the CDir for L4 caches is to use the same OnDie-CDir. Although this approach reuses the existing resources and does not incur extra overhead, we found that compared to MSC, using OnDie-CDir as the L4 coherence directory degrades performance by an average of 24%, with a maximum of 66% (detailed

methodology in Section III). Therefore, using existing OnDie-CDir for DRAM caches jeopardizes the use of CDC. The performance degradation stems from the insufficient coverage of OnDie-CDir, as we explain in the following paragraphs.

2) Coverage of Coherence Directory: A CDir entry includes the memory address, the state (e.g., modified, exclusive, etc.), and a sharer bit vector for the sharers or the owner. Every cached block must have a corresponding entry in the CDir; when a valid CDir entry is replaced, it invalidates the corresponding cache block in L4. Such invalidation is referred to as *Coherence-induced Invalidation*. To minimize coherence-induced invalidation, the number of CDir entries must be proportional to the cache capacity. The ratio of the number of the entries in CDir to the number of cache blocks in the cache is referred to as *Coverage* of the coherence directory. An 1X-coverage CDir, with as many entries as the number of cache blocks in the system, is the minimum coverage so that DRAM cache capacity is fully used.

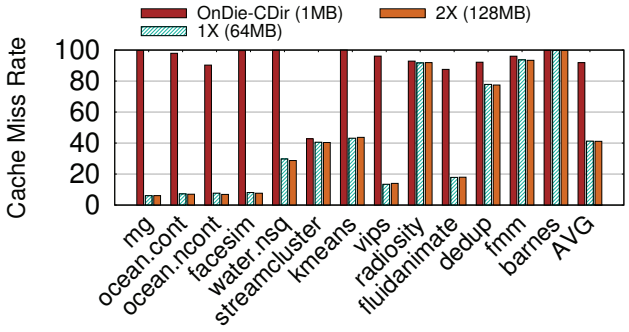

Fig. 3. Impact of Coherence Directory Coverage on DRAM Cache Miss Rate. We use a 1GB DRAM cache for this study, and show OnDie-CDir (equivalent to $\frac{1}{128}$X) and coverage of 1X, and 2X.

3) Size of Coherence Directory: To understand coverage requirements for DRAM caches, we use a system with 1GB DRAM cache, which holds 16 million cache blocks; we vary the coverage and show the corresponding DRAM cache miss rate in Figure 3. OnDie-CDir, with 256K entries or an equivalent coverage of $\frac{1}{128}$X, limits the effective capacity of L4 caches. Therefore, it results in very high DRAM cache miss rate and degrades performance. For an 1X coverage, the CDir needs 16 million entries for a 1GB DRAM cache (16 million blocks). Even each entry is only 4 bytes, the size of the coherence directory would be 64MB.[3] Note that the required coverage is a function of application's working set and access behavior. While 1X coverage seems sufficient, certain application, suggested by prior studies [28–31], could require higher coverage (at even higher storage cost). We assume 1X coverage (64MB) for our study, but our proposal can also be extended to 2X coverage (128MB).

[3]Given 16 million entries, each CDir entry requires 22-bit tags (48-bit physical address, 6-bit line offset, 20-bit set indexing for 16-way associativity), 1 valid bit, 2 state bits, and 4 bits for the sharer vector, so the size of each CDir entry is 29 bits. We provision 32 bits (4 Byte) for one CDir entry.

D. The Need for Low-Latency Request-For-Data Operation

The other challenge to architect giga-scale CDC is a low-latency *Request-For-Data* operation. The Request-For-Data (also termed *Fwd-GetS*) is the operation that the home node asks the owner to write back the most recent copy of the data. Therefore, the Request-For-Data operation reads the most recent data via a cache access, and is on the critical path. To illustrate, we use Figure 4 that follows Figure 2 as an example. Now consider a read miss to a *Modified* data block (⬡) ❶. The CDir in the home node (Node 0) indicates that Node 1 is the owner of the block ❷. The home node requests Node 1 to write back the most recent data via a Request-For-Data (RFD) operation ❸. When the owner receives the request, it reads its copy of the data from the cache ❹ and replies to the home node. The home node then replies to the requester and makes the coherence state *Shared*.

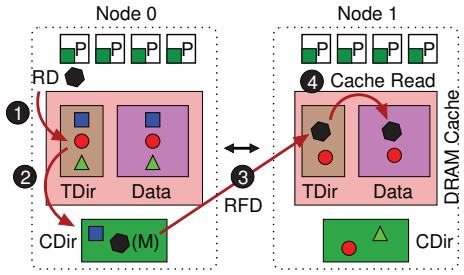

Fig. 4. Sequence for a Request-For-Data (RFD) Operation

Figure 5 shows a latency breakdown of the Request-For-Data operation that reads the data in the cache for both MSC and CDC. Figure 5(a) shows the case of MSC, where L3 caches are the point of coherency. Therefore, the cache access latency of Request-For-Data is a L3 cache latency. In contrast, such latency becomes a DRAM cache latency when it comes to CDC in Figure 5(b), because L4 caches are the point of coherency; therefore, the Request-For-Data operation in CDC incurs a significantly long latency.

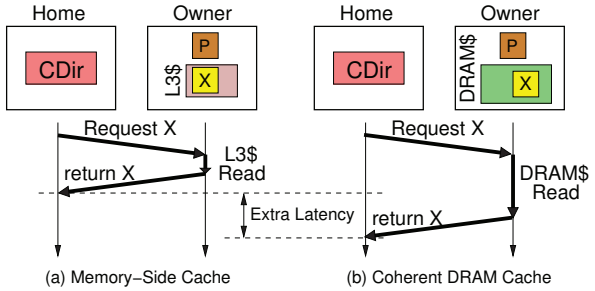

Fig. 5. Latency Breakdown of Request-For-Data Operation in Memory-Side Cache (coherent L3), and Coherent DRAM Cache (coherent L4). Latency not to scale.

E. Performance Potential of CDC

We identify two key challenges to architect giga-scale CDC, and show that existing architecture cannot be directly applied to CDC. To understand the performance potential of CDC, Figure 6 shows the performance improvement by CDC

TABLE I
WORKLOADS: BENCHMARK, SUITES, AND INPUT SIZE

Name	Suites	Input
mg	NPB	Set C
ocean.cont	splash2	2050x2050 Grid
ocean.ncont	splash2	2050x2050 Grid
facesim	parsec	80,598 particles, 372,126 tetrahedra, 1 frame
water.nsq	splash2	20^3 Molecules
streamcluster	parsec	16,384 input points, block size 16,384 points, 128 point dimensions
kmeans	NU-MineBench	10M elements, 9 dimension, 16 cluster
vips	parsec	2,662 x 5,500 pixels
radiosity	splash2	BF refinement =$1.5e^{-3}$
fluidanimate	parsec	300,000 particles, 5 frames
dedup	parsec	184 MB file size
fmm	splash2	256K Particles
barnes	splash2	256K Particles,Timestep= 0.25

with respect to MSC. We architect the CDC by using an impractical SRAM-based coherence directory of an 64MB SRAM overhead and zero L4 DRAM cache access latency for Request-For-Data operations (still incurs inter-node network latency). We term such design *Impractical-CDC*. On average, the Impractical-CDC outperforms MSC by 30%, with a maximum speedup of 2.8X from *ocean.cont*. Note that workloads with significant performance improvement tend to have large footprint of private data or read-only shared data (detailed workloads in Section III and Table I). Therefore, for such workloads, CDC avoids inter-node network latency and significantly improves performance.

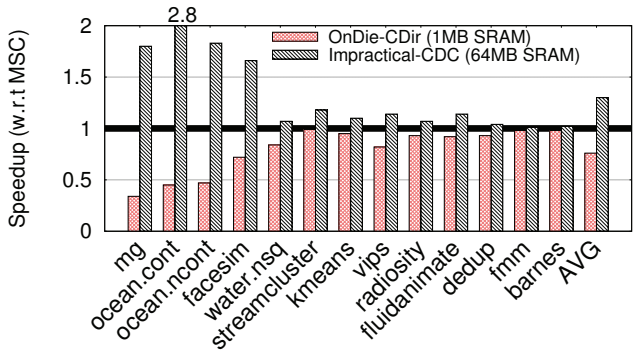

Fig. 6. Performance of CDC using OnDie-CDir and Impractical Coherent DRAM Cache (Impractical-CDC).

To architect high-performing giga-scale Coherence DRAM Cache, we propose *DRAM Caches for Multi-Node Systems* (*CANDY*), which has two orthogonal components to address the challenges. In Section IV, we investigate the coherence directory for giga-scale L4 caches, and show how to leverage existing resources for a low-latency coherence directory. In Section V, we provide further analysis on the Request-For-Data problem, and propose a technique to mitigate the latency by exploiting the characteristics of read-write shared data. Before we present our solutions, we first describe the methodology.

III. EXPERIMENTAL METHODOLOGY

A. System Configuration

We use Sniper [33] simulator to conduct our experiments, and configure the system using the parameters shown in Table II. We evaluate a 4-node system, where each node has 4 processors, a shared L3 on-die cache, a DRAM cache, and also DDR3 memory. The timing and bandwidth specification of DRAM cache are modeled after High Bandwidth Memory [2]. Each processor has private L1-D, L1-I, and L2 caches. Inter-node communication relies on high-speed links, modeled after Intel's QPI and AMD's HyperTransport [34, 35]. We use Alloy Cache to implement DRAM caches, including Memory-Side Cache and Coherent DRAM Cache [15], and also equip the DRAM caches with Hit-Miss predictor [14, 15]. MSC is the default baseline system, unless stated otherwise.

TABLE II
SYSTEM CONFIGURATION

Node	
Number of Nodes	4
Each Node Configuration	
Processors	
Number of Cores	4
Frequency	3.2GHz
Last Level Cache	
Shared L3 Cache	4MB, 16-way, 24 cycles
DRAM Cache	
Capacity	1GB
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	8
Banks	16 Banks per rank
Bus Width	64 bits per channel
Row Buffer Size	2048 Bytes
tCAS-tRCD-tRP	11-11-11 bus cycle
tRAS	45 bus cycle
Main Memory (DDR-based DRAM)	
Capacity	16GB
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	2
Banks	8 Banks per rank
Bus Width	64 bits per channel
Row Buffer Size	2048 Bytes
tCAS-tRCD-tRP	11-11-11 bus cycle
tRAS	45 bus cycle
Coherence Protocol	
Protocol	MESI
On-Die L3 Directory	1MB
Inter-Node Network	
Bandwidth	12.4GB/s
Latency	50 ns one-way latency

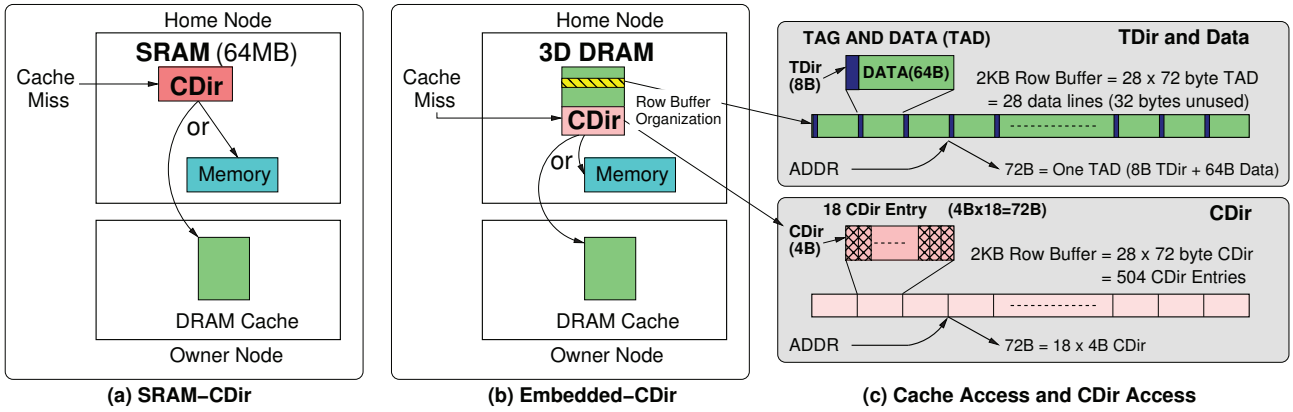

Fig. 7. Coherence Directory Organization. (a) SRAM-based Coherence Directory, (b) Embedded Coherence Directory and (c) Cache Access and CDir Access. One read access to Embedded-CDir in 3D DRAM gets 18 CDir entries (72 bytes). Note that our proposal is based on, but not limited to, Alloy Cache [15].

We use MESI coherence protocol as default [36]. The coherence directory (CDir) is distributed and associated with each node. Each entry in the CDir uses a full sharer vector to record the sharing information. For MSC, the coherence directory for L3 cache coherence is 1MB and located on-die. It tracks 256K cache blocks, same as AMD's Magny-Cours [32].

B. Workloads

We study parallel multi-threaded programs from various benchmarks suites, including splash2, parsec, NPB, and NU-MineBench [37–40]. We, unless stated otherwise, use *simlarge* input set shown in Table I. We simulate the workloads only during the parallel parts of the workloads (the region of interest, ROI); the sequential sections at the beginning is used to warm up the cache, and is not included in our timing evaluation [41]. We report the workloads that execute more than 1 billion instructions in ROI, and sort the workloads based on their memory intensity. The speedup of the workloads is normalized to the baseline system that uses MSC.

IV. DRAM-cache Coherence Buffer: Architecting Low-Latency Coherence Directory

In this section, we present a practical design to build a low-cost and low-latency coherence directory for giga-scale Coherent DRAM Caches.

A. Coherence Directory Organization

One way to build a coherence directory is to use a separate storage structure to keep the coherency information [28–32]. For giga-scale CDC, we first examine two simple ways that follow the same principle.

1) SRAM-Based Coherence Directory: The first approach is to use SRAM storage. However, given the size of the coherence directory (64MB, larger than L3 cache), putting it on die is prohibitively expensive. Figure 7(a) shows the design of SRAM-based CDir, termed *SRAM-CDir*. In SRAM-CDir, the latency to access the CDir is the SRAM access latency.

2) Embedding Coherence Directory in 3D DRAM: Alternatively, a practical design to accommodate such large structure is to place the coherence directory in the 3D DRAM to avoid SRAM storage overhead. A portion of the 3D DRAM capacity is reserved for the CDir, thus reducing the DRAM cache capacity. We refer to this embedded approach as *Embedded-CDir*, and show it in Figure 7(b). Figure 7(c) illustrates how to access DRAM cache and Embedded-CDir in 3D-DRAM, and shows an example of the state-of-the-art Alloy Cache [15]. In DRAM cache, each cache access gets a basic access unit of 72 bytes (8B TDir and 64B Data). For simplicity, we also use the same basic access unit for the Embedded-CDir. Thus, one access to the Embedded-CDir also gets 72 bytes, which includes 18 CDir entries (4B each). Note that we use Alloy Cache only for illustration purpose; our proposal can also be extended to other DRAM cache organizations.

To reduce coherence-induced invalidation, we can organize the Embedded-CDir as a highly set-associative structure. For example, 18-way set-associative (*High-Assoc*), given each access returns 18 CDir entries. Figure 8 shows the performance for Embedded-CDir that is High-Assoc, and Low-Assoc (direct-mapped, 18 sets for 18 CDir entries). On average, High-Assoc improves performance by 11%, while Low-Assoc is 10%. (We also conduct studies for other associativities, but show only two configurations due to the space constraint.) Although High-Assoc delivers the higher performance, we will discuss how lower set-associativity could further improve performance in later section.

In addition, Figure 8 also shows the performance improvement by SRAM-CDir. On average, SRAM-CDir improves performance by 25%, while Embedded-CDir is 11%. However, Embedded-CDir degrades performance for several benchmarks, such as *streamcluster* (42%), *radiosity* (16%), and *barnes* (15%). Therefore, although Embedded-CDir is a more practical design to reduce costly SRAM storage overhead, it does not perform as well as the SRAM-CDir, due to the latency overhead of accessing the Embedded-CDir in 3D-DRAM.

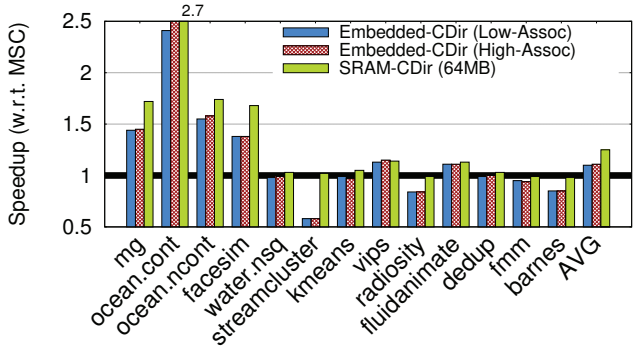

Fig. 8. Performance of Embedded-CDir (Low-Assoc and High-Assoc) and SRAM-CDir

B. Leveraging On-Die Coherence Directory

Ideally, we want the latency of SRAM-CDir with the lightweight SRAM storage overhead of the Embedded-CDir. To this end, we propose to re-purpose the on-die SRAM coherence directory (OnDie-CDir, meant for L3 cache coherence in MSC) as a buffer to store the recently accessed CDir entries from Embedded-CDir. Recall that OnDie-CDir is provisioned and used for the L3 cache coherence in MSC (described in Section II-C); in CDC, as L4 cache becomes the point of cache coherence, the OnDie-CDir is unused. We leverage such existing SRAM structure in CDC and term it *DRAM-cache Coherence Buffer* (*DCB*).

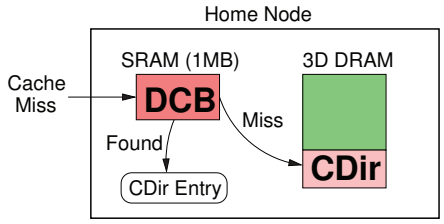

Fig. 9. Overview of DRAM-cache Coherence Buffer (DCB) and Embedded-CDir. On a cache miss, the home node first checks the DCB; if the entry misses in DCB, the home node checks the Embedded-CDir in 3D-DRAM.

Figure 9 shows the overview of DCB and also its interaction with Embedded-CDir. On a cache miss, the home node first checks the DCB to find the corresponding entry. If the entry is found in DCB (a *hit* in DCB), the latency to access the CDir entry is only the DCB latency, which is a SRAM access latency. On the other hand, if the entry is not found in DCB (a *miss* in DCB), the home node retrieves the entry from Embedded-CDir by issuing a 3D-DRAM read access. In this case, the latency to access the CDir entry is the sum of the DCB latency and the Embedded-CDir latency. After Embedded-CDir returns the entry, the home node inserts the entry into DCB for future references.

C. Design of DCB

The latency to retrieve the CDir entry is determined by whether the entry is found in DRAM-cache Coherence Buffer, or, in other words, the *Hit Rate* of DCB. Higher DCB hit rate leads to lower average latency. Therefore, the effectiveness of mitigating CDir access latency depends on the hit rate of DCB. The hit rate of DCB is a function of its size and its interaction with the Embedded-CDir, and we present two simple designs for DCB to maximize the hit rate.

1) Exploiting Temporal Locality: After a DCB miss, the home node retrieves the corresponding CDir entry by accessing the Embedded-CDir and performing tag-matching. To exploit temporal locality, the home node inserts the demand missing CDir entry into DCB so that future cache misses are likely to hit the same entry in the DCB. As this design inserts CDir entries into DCB on demand misses, we refer to this as *DCB-Demand*. To minimize the misses of DCB, we architect the DCB to maximize the associativity; therefore, we organize the DCB as a 16-way set-associative structure, indexed by the memory address.

2) Exploiting Spatial Locality: Besides temporal locality, we also exploit spatial locality to improve DCB hit rate. For example, for a streaming workload that sequentially accesses the memory, a cache miss to memory address *X* implies a high likelihood that the subsequent cache miss would go to memory address close to *X* [42–44]. In the context of CDir entries, it means that the next requested coherence directory entry is spatially correlated to the currently requested CDir entry. Based on this, we propose to organize the DCB and the Embedded-CDir for spatial locality by exploiting the access granularity of 3D-DRAM.

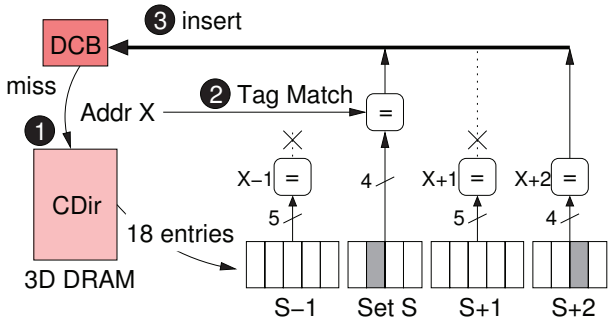

Fig. 10. Optimizing DCB and Embedded-CDir for Spatial Locality

To exploit spatial locality, we want to install both demand and spatially correlated CDir entries by fetching multiple CDir entries (across sets). In High-Assoc Embedded-CDir, one 3D-DRAM access returns only one set; fetching additional sets incurs extra 3D-DRAM read. To avoid such bandwidth overhead, we propose to use lower set-associative Embedded-CDir such that the CDir entries of continuous memory addresses are placed in one 3D-DRAM access unit and will be fetched in one 3D-DRAM access. In Figure 10, we show an example that one 3D DRAM access returns 4 sets (medium set-associativity). (One access gives 18 entries, so we implement this by using 5-way for odd sets, and 4-way for even sets.) CDir entries for up to 4 continuous memory addresses from different sets are fetched together in one access. Therefore, every access to the

Embedded-CDir returns not only the requested entry but also entries of the continuous addresses. In Figure 10, the address finds an extra match and inserts one additional CDir entry (address *X+2*). As this design is for spatial locality, we term this design *DCB-SpaLoc*.

D. Effectiveness of DCB

1) DCB Hit Rate: Figure 11 shows the hit rate of DCB for both DCB-Demand and DCB-SpaLoc designs. Recall that DCB is re-purposed from OnDie-CDir, and has a fixed area budget of 1MB SRAM.[4] In such allowance, DCB-Demand has an average DCB hit rate of 75%, while DCB-SpaLoc has an average DCB hit rate of 81%.

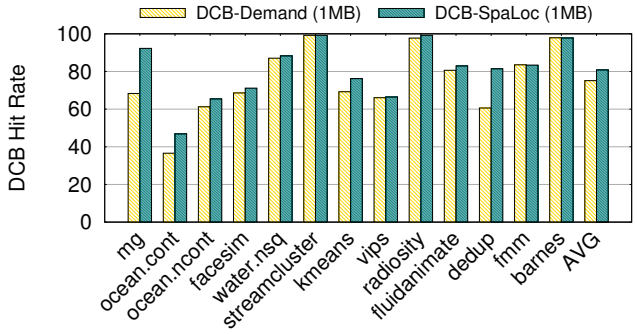

Fig. 11. DCB Hit Rate: DCB-Demand and DCB-SpaLoc. Both are allowed for 1MB SRAM in size.

2) Performance: Figure 12 compares the performance for the Embedded-CDir, DCB-Demand, and DCB-SpaLoc, as well as a case where DCB is perfect with 100% hit rate (termed *DCB-Perfect*). On average, DCB-Demand outperforms the baseline MSC by 18%, while DCB-SpaLoc improve performance by 21%, and DCB-Perfect has a performance improvement of 25%. DCB-Demand and DCB-SpaLoc mitigate the performance degradation introduced by the Embedded-CDir (e.g., mg, streamcluster, and radiosity), and outperform the Embedded-CDir by 7% and 10%, respectively. Also, the improvement of DCB hit rate by DCB-SpaLoc reflects on the performance improvement (3% over DCB-Demand). For the rest of the paper, we use DCB-SpaLoc.

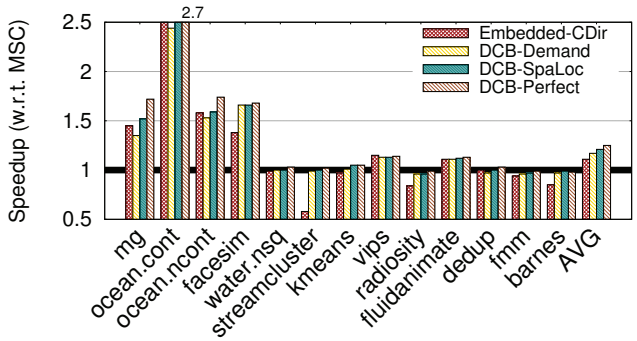

Fig. 12. Performance Comparison of Embedded-CDir, DCB-Demand, DCB-SpaLoc and DCB-Perfect

[4]We also conduct a sensitivity study of the DCB size, and find that the DCB hit rate of 1MB DCB-SpaLoc is as good as 8MB DCB-Demand.

V. Sharing-Aware Bypass: Architecting Low-Latency Request-For-Data

Request-For-Data is a critical and necessary operation that reads the most recent data from a cache. Unfortunately, in CDC, such operation incurs a 3D-DRAM access latency, which is much higher than the latency in its counterpart MSC. We investigate the Request-For-Data problem in CDC, and propose a technique to mitigate the cache access latency of Request-For-Data by exploiting the characteristics of read-write shared data.

A. Request-For-Data: What and Why?

To maintain coherent caches, coherence protocol relies on different operations based on the type of the request and the coherence state of the requesting block. Such operations include accessing the memory and *Coherence Operations*. While the memory access fetches data from the memory, the coherence operations are dedicated to keeping the cached data up-to-date. Without loss of generality, we can further classify coherence operation into three categories: (1) *Request For Data* (*RFD*), which asks the owner to write back the most recent data; (2) *Invalidate* (*INV*), which invalidates the copy in a cache; and (3) *Flush*, which is a combination of RFD and INV) [24, 25].[5] Table III shows the detailed classification.

TABLE III
Classification of coherence operations based on request type and the coherence state of the data block

State	Read	Exclusive/Write
Modified	RFD	Flush
Exclusive	RFD	Flush
Shared	Memory Read	INV
Invalid	Memory Read	Memory Read

An INV updates the TDir (valid bit) by a write access to TDir, while an RFD and a flush must access the data in the cache via a read access to the cache. In CDC, the RFD operation reads the data from the DRAM cache, and incurs a 3D-DRAM read access. Therefore, the RFD latency in CDC is much longer than in MSC, where RFD incurs only a L3 SRAM cache access latency (Illustration shown in Figure 5). Also, the RFD is on the critical path, and such RFD latency overhead in CDC penalizes the performance. To understand the impact of RFD, we conduct an analysis to break down the percentage of different operations in CDC.

Figure 13 shows the percentage of four operations with respect to DRAM cache accesses: Request-For-Data (RFD), Invalidate (INV), Memory (Mem), and Hit. (Flush is counted as RFD, as a Flush includes an RFD.) On average, the L4 cache hit rate is 59%; RFD contributes to 20% of the L4 cache accesses, while 12%, and 9% of the L4 accesses are INV and Mem, respectively. Therefore, 49% of the cache misses results in RFD operations. Mitigating the latency for RFD is a key challenge of high-performing CDC.

[5]Reuqest-For-Data is termed Fwd-GetS, and Flush Fwd-GetM in MESI protocol [27]. Specific coherence operations depend on the coherence protocol. We use MESI as an example for explanation, but all protocol require similar operations to maintain coherence.

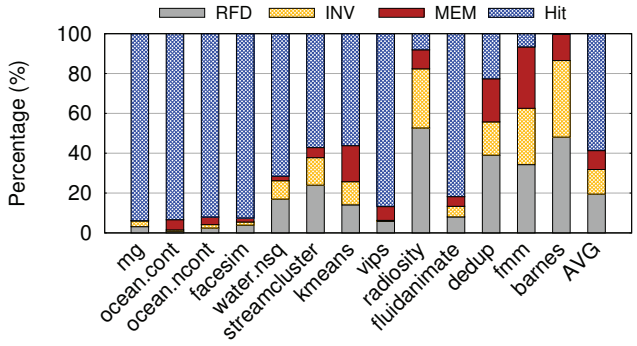

Fig. 13. Percentage of DRAM Cache Requests: Request For Data (RFD), Invalidate (INV), Memory (MEM) and Hits

B. Sharing-Aware Bypass: Mitigating DRAM Cache Access Latency of Request-For-Data

Although RFD reads data from a cache, not all data in the cache are accessed by RFD operations: As RFD maintains cache coherence for *read-write shared data*, only read-write shared data are accessed by RFD [45]. Therefore, if such data are not read from the L4 DRAM caches, but from the L3 caches, the cache access latency of RFD reduces significantly. One simple way to achieve this goal is to use select caching or cache bypassing [46]. In other words, the read-write shared data *bypass* L4 caches, and are stored only in L3 caches.[6]

To this end, we propose *Sharing-Aware Bypass* (*SAB*), which enforces read-write shared data to bypass L4 caches at run-time. Figure 14 shows the overview of Sharing-Aware Bypass. SAB is composed of two parts: dynamically detecting read-write shared data, and enforcing the bypassing policy at L4 DRAM caches.

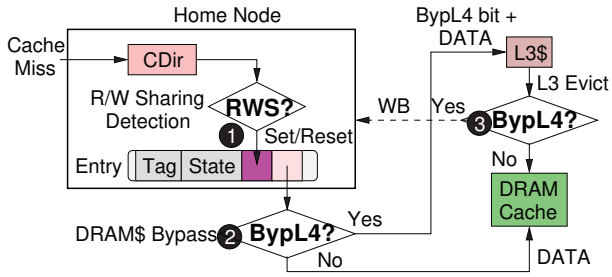

Fig. 14. Overview of Sharing-Aware Bypass: (1) Read-write Shared Data Detection, (2) Cache Miss and DRAM Cache Bypass, and (3) L3 Dirty Eviction and Bypassing DRAM Cache

C. Detecting R/W Shared Data

To detect read-write shared data, we employ a simple mechanism that is based on the events of coherence operation [29, 31]. The read-write shared data is the necessary and sufficient condition for coherence operations, so read-write shared data can be easily identified when a given request incurs such operations. (otherwise, it is read-only shared data or private data.) Thus, on the event of any coherence operations, we can classify the data block into read-write shared data or the other. On invalidate, RFD, and flush operations, the data block is marked *Read-Write Shared*. We use one bit in the CDir entry to keep track of this classification. On a memory read operation, we reset the bit to mark the data as not read-write shared data. As a data block can transition into one mode from the other, our detection mechanism dynamically identifies read-write shared data at run-time. We refer to the bit as *Read-Write Shared (RWS)* bit.

[6]Another way to mitigate the DRAM cache access latency is to hide the L4 cache access latency by accessing the L3 cache in parallel. However, if the data requested by RFD are not in L3, but in L4, the latency is determined by the L4 cache access latency.

D. Enforcing Sharing-Aware Bypass

To enforce the bypassing policy for respective types of data, we discuss when to determine the bypassing decision, and how to enforce the bypassing decision in the system.

1) When to Decide: Once SAB identifies a read-write shared data, it can use such information to decide the bypassing policy. The bypassing decision is determined only on the event of ownership change, because ownership change implies that only one node, the requesting node, in the system will have the copy of the data. This avoids unnecessary invalidation to memory address that is shared by multiple sharers in the system. The bypassing decision is stored in the CDir entry associated with the data block by using another bit in the CDir entry. To distinguish from Read-Write Shared bit, we term this bit *Bypass L4 (BypL4)* bit.

2) How to Enforce: For simplicity, SAB maintains a uniform bypassing decision for all nodes in the system. This means if one data block bypasses DRAM caches, such block cannot be stored in any DRAM caches in any case. However, a data block would attempt to be stored in DRAM caches in two cases: (1) a L4 cache miss (to be installed to L4 caches), and (2) a L3 dirty eviction (to be written back to L4 caches). We describe how SAB handles these two cases.

L4 Cache Miss. A cache miss goes to the home node to request the data. When replying to the requester, the home node communicates the BypL4 bit in the message with data. The requesting node examines the BypL4 bit, and the data bypass the DRAM cache if the BypL4 bit is set.

L3 dirty Eviction. On a L3 dirty eviction that attempts to write the data to DRAM caches, the L4 DRAM cache must know the bypassing decision. L4 caches can choose to consult the home node for such information; however, this significantly and unnecessarily increases the latency as well as the network traffic. Alternatively, we propose to store the BypL4 information in the L3 cache by adding one bit in the tag directory of L3 cache. A L3 dirty eviction uses the BypL4 bit to decide whether the evicted data block should bypass the L4 cache. If the bit is set, meaning bypassing L4 caches, the L3 cache writes the modified data back to the home node.

The storage overhead of Sharing-Aware Bypass includes two bits in the CDir entry, and one bit per line in L3 caches. As the CDir entry is provisioned to be 32 bits (4 bytes), and only 29 bits are used, two bits in CDir entry do not incur any

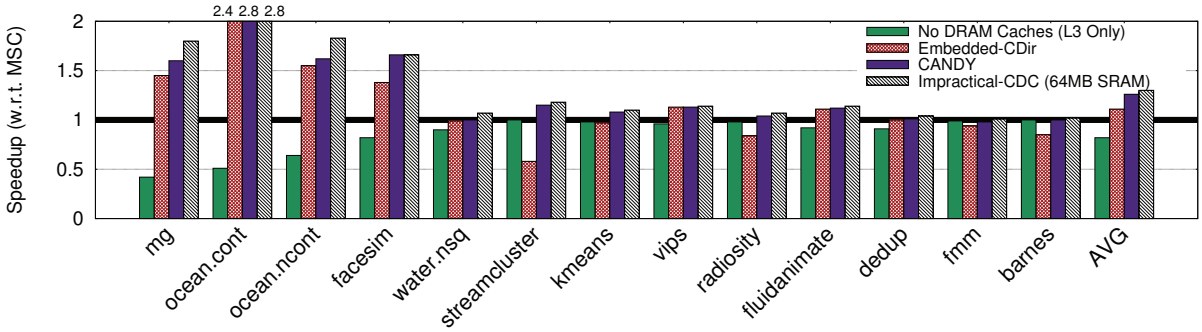

Fig. 15. Performance of No DRAM Caches (L3 Only), Embedded-CDir, our proposed CANDY, and impractical CDC with 64MB SRAM overhead and zero L4 cache read latency for RFD operation (Impractical-CDC). Note that the performance is normalized to the baseline Memory-Side Cache. We report the geometric mean as an *AVG* in the right most bar.

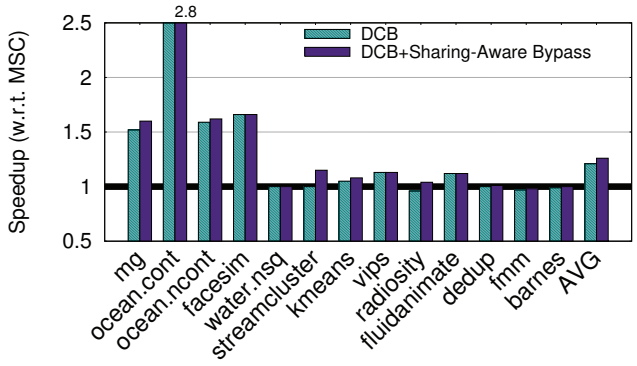

Fig. 16. Performance of Sharing-Aware Bypass

storage overhead. For a 4MB L3 cache (64K lines), one bit per line incurs an SRAM overhead of 8KB per node. Therefore, our proposed Sharing-Aware Bypass incurs negligible storage overhead (less than 0.2% L3 area).

E. Effectiveness of Sharing-Aware Bypass

We conduct a study to understand the effectiveness of SAB by counting the number of RFD serviced by the L3 caches. Compared to a bypassing scheme that has the oracle knowledge of the read-write shared data and uses L3 caches for all the RFD, SAB uses L3 cache for 78% of the RFD. Also, the effectiveness of SAB reflects on the performance improvement, shown in Figure 16. On average, SAB provides a speedup of 4% in addition to the improvement by DCB, and overall outperforms MSC by 25%.

VI. Results and Analysis

A. Overall Performance

We evaluate our proposal using a 16-core system running parallel benchmark suites. Figure 15 compares our proposal CANDY to Memory-Side Cache (MSC), and also an impractical Coherent DRAM Cache that uses a 64MB SRAM storage for the coherence directory and has zero L4 cache read latency for RFD operation (Impractical-CDC). Figure 15 also shows the performance for a system that has no DRAM cache (termed *L3-Only*). The baseline is Memory-Side Cache, and we report the geometric mean in the right most bar, labeled *AVG*. On average, L3-Only (i.e., no DRAM caches) degrades performance by 18%, with a maximum loss of 58% from *mg*. Embedded-CDir outperforms MSC by an average of 11%, but degrades performance for a couple of workloads (e.g., *streamcluster* and *radiosity*). CANDY not only mitigates such performance degradation but also improves average performance by 14% over Embedded-CDir. Overall, CANDY outperforms MSC by an average of 25% with a maximum improvement of 1.8X from *ocean.cont*. Also, CANDY gets almost all the potential performance improvement of Impractical-CDC, which has an average performance improvement of 30%.

B. Sensitivity Studies: Scalability and Network Latency

We assume 4 nodes in our default system, and we conduct a sensitivity study to test the scalability of our proposal. We vary the number of nodes by changing the total number of processors while keeping the number of processors per node constant: We vary the number of nodes from 2 nodes to 8 nodes (8 cores to 32 cores), and show the performance for MSC and Candy in Figure 17(a). Note that the speedup is with respect to each own baseline. CANDY outperforms MSC consistently across the spectrum, and improves performance by 41%, 25%, 32% for 2-node, 4-node, and 8-node systems, respectively. We also conduct a study that varies the inter-node network latency from 0.5X (25ns) to 2X (100ns). As shown in Figure 17(b). CANDY consistently outperforms MSC by 25%, 25%, and 29% speedup for 0.5X, 1X, and 2X inter-node network latency, respectively.

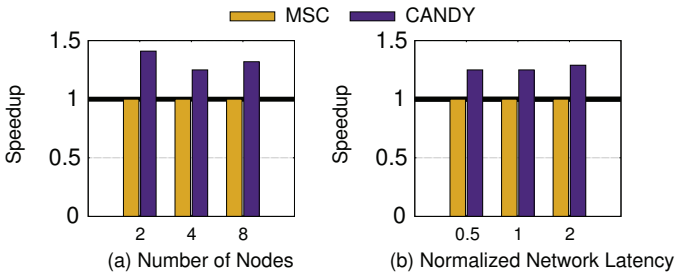

Fig. 17. Sensitivity Studies. (a) Scalability with Number of Nodes, and (b) Normalized Network Latency.

C. Savings of Inter-Node Network Traffic

CDC enables the DRAM cache to keep the remote data; such capability not only avoids the inter-node network latency, but also reduces inter-node traffic and alleviates the bandwidth pressure on the inter-node network. Figure 18 shows the inter-node network traffic reduction by CANDY. On average, CANDY reduces the traffic by 63%, meaning CANDY incurs only $\frac{1}{3}$ the traffic compared to MSC. Notice that workloads with significant traffic reduction tend to have significant performance improvement, as CANDY is able to cache the private data or read-only shared data of such workloads.

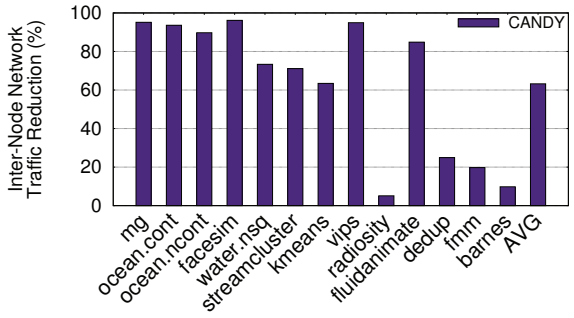

Fig. 18. Inter-node Network Traffic Reduction by CANDY (The higher the better)

D. Data Placement in Multi-Node Systems

To avoid inter-node network latency, prior work exploits thread-local data in Non-Uniform Memory Access (NUMA) systems [47]. We use interleaved memory address as default, but conduct a study for NUMA-aware systems [48–50]. In such systems, the operating system maps pages using NUMA-aware data placement policy (e.g., *First-Touch*) [51, 52]. Figure 19 shows such systems for MSC and CANDY. Note that *CANDY (NUMA-Aware)* is normalized to MSC that also uses NUMA-aware policy. On average, CANDY (NUMA-Aware) outperforms MSC by 26%. *Streamcluster* prefers interleaved data placement, because its programming model appoints the master thread to initiate data structures, and centralizes all data in one node [53]. Figure 19 also shows a configuration where OS optimizes the page placement policy for individual workload based on the performance (termed *SW-Opt*). The OS chooses the best-performing page mapping policy for MSC, and uses the same policy for both MSC and CANDY. Even with such highly optimized data placement policy, CANDY (SW-Opt) still outperforms MSC by an average of 15%.

VII. Related Works

A. DRAM Caches

Recently, there are many studies on DRAM caches. These proposals focus on the architecture of DRAM caches, and optimize the DRAM cache for performance [11–22]. However, all studies implicitly consider one DRAM cache in single-node systems. In contrast, our paper aims to use DRAM caches for multi-node systems.

There are also several studies optimizing the latency for DRAM caches. One optimization is to use a small SRAM structure to avoid the tag look-up latency [11, 12, 16, 22]. In

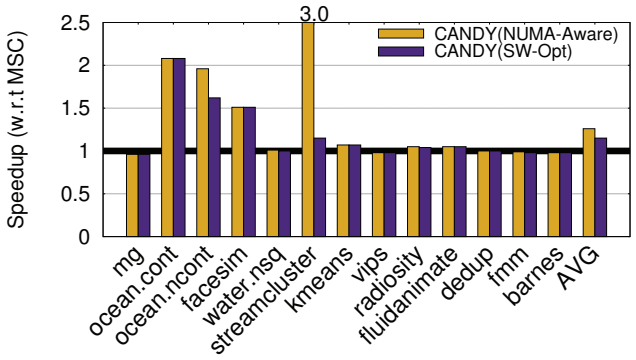

Fig. 19. Performance of CANDY with NUMA-aware and software-optimized (SW-Opt) policy

contrast, our DCB re-purposes the existing on-die coherence directory that is provisioned for L3 caches, and does not incur additional SRAM overheads. Note that DCB is orthogonal to the tag look-up optimization.

B. Coherence Directory and Protocols

Coherence protocols is a popular research area in the past few decades [54]. For scalability, multiprocessor and multi-node systems typically adopt directory-based protocols [55–58]. For directory-based protocols, the key challenge is to design a low-cost coherence directory [23–25]. Duplicate-Tag directories incur low area cost, but its high associativity makes it energy inefficient and hard to scale [59, 60].

Sparse Directory is one of the appealing directory-based protocol, because of its scalability and energy-efficiency. Prior studies focus on reducing the directory storage overhead by reducing the sharer vector length or by reducing the conflict misses [28–31]. However, in our paper, we focus to design the coherence directory for a giga-scale DRAM cache, which is two orders of magnitude larger than the on-die cache. Even with the most aggressive technique that reduces the coverage of coherence directory to $\frac{1}{4}$X, the coherence directory still needs 16MB of storage. Nevertheless, all these optimizations are orthogonal to our proposal, and can be combined to further improve the performance.

Like the relationship between DCB and the Embedded-CDir, other studies propose similar idea to construct a two-level coherence directory architecture to improve space complexity or to improve energy consumption [61–63]. However, these proposals incur additional storage overhead to implement the two-level structure, while our proposed DCB re-purposes the existing on-die coherence directory and does not incur extra storage overhead.

C. Shared Memory Systems

For a shared-memory computer system that has non-uniform memory access latency, Cache-Only Memory Architecture (COMA) and Reactive-NUMA are proposed to mitigate the long latency across nodes [48–50]. As opposed to line-granularity coherent caches in a NUMA system (Cache-Coherent Non-Uniform Memory Architecture, or ccNUMA), COMA operates at a page granularity, and relies on the operating system to maintain coherence. These works study

the use of the memory, not cache. However, our work is independent of these proposals, as caches are still coherent, regardless a NUMA or COMA system.

Since the network is a scarce resource, many prior works, such as Remote Memory Operations, focus on how to reduce the network traffic by either hardware-based or software-based and by either delegation or privatization [45, 64–66]. However, these proposals come with significant change in the whole computing stacks (i.e., program, compiler, ISA, and hardware). We address the Request-For-Data problem for DRAM caches with negligible overhead and change.

VIII. Summary

Recent technology advancement makes DRAM cache a promising candidate to transparently provide high bandwidth to the applications. In this paper, we study DRAM caches for multi-node systems. To the best of our knowledge, this is the first paper to investigate coherence DRAM caches and also quantify the performance of DRAM caches. To architect giga-scale coherence DRAM caches, we discover two key challenges: (1) the coherence directory, whose size is as large as tens of MB, thus incurring prohibitive overheads of storage and latency, and (2) the Request-For-Data operation, which is critical to access the most recent copy of data. To enable high-performing CDC in multi-node systems, we propose CANDY, a scalable and low-cost solution to address both issues.

First, to accommodate such large structure, we dedicate a portion of the 3D DRAM capacity to avoid SRAM storage overhead. To mitigate the latency to access the coherence directory in 3D DRAM, we propose DRAM-cache Coherence Buffer, which re-purposes the existing on-die coherence directory to cache recently accessed coherence directory entries. As the on-die coherence directory is already provisioned for L3 cache coherence, our proposal does not incur any SRAM storage overhead. To improve DCB hit rate, we further exploit spatial locality to co-organize DCB and Embedded-CDir.

Second, to mitigate the Request-For-Data latency for read-write shared data, we propose Sharing-Aware Bypass, which dynamically identifies read-write shared data, and enforces such data to bypass DRAM caches. Our insight is that we can mitigate the latency if read-write shared data is stored only in L3 caches. We develop a simple mechanism to identify the read-write shared data at run time and also enforce the bypassing decision for the system. Sharing-Aware Bypass incurs negligible overheads of 8KB per node, but is effective to mitigate the Request-For-Data latency.

We evaluate parallel workloads in a 4-node system. Our proposed CANDY outperforms Memory-Side Cache by 25%. CANDY has negligible overhead of 8KB per node; still, it provides within 5% of the potential performance improvement from an impractical coherent DRAM cache that incurs a storage overhead of 64MB SRAM with idealized Request-For-Data latency. We believe that enabling coherence DRAM caches not only improves performance for multi-node systems but also explores a new avenue of cache coherence studies.

Acknowledgements

We thank Swamit Tannu, other group members of CARET at Georgia Tech, and anonymous reviewers for their comments and feedbacks. We also thank Ching-Kai Liang for the feedbacks on an earlier version of the paper. This work was supported in part by Intel, NSF grant 1319587 and by the Center for Future Architecture Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] *HMC Specification 1.0*, 2013.

[2] JEDEC, *High Bandwidth Memory (HBM) DRAM (JESD235)*, JEDEC, 2013.

[3] Micron, *HMC Gen2*, Micron, 2014.

[4] JEDEC, *WIDE I/O SINGLE DATA RATE (WIDE I/O SDR)*, JEDEC, 2011.

[5] *1Gb_DDR3_SDRAM*, Micron, 2010.

[6] *DDR4 SPEC (JESD79-4)*, JEDEC, 2013.

[7] *Intel Xeon Phi*, Intel, 2014.

[8] A. Sodani *et al.*, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.

[9] *AMD Radeon R9*, AMD, 2015.

[10] *NVIDIA Updates GPU Roadmap; Announces Pascal*, NVIDIA, 2015.

[11] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual International Symposium on Microarchitecture*, 2011.

[12] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[13] G. H. Loh *et al.*, "Challenges in heterogeneous die-stacked and off-chip memory systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads*, 2012.

[14] J. Sim *et al.*, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*, 2012.

[15] 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 *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*, 2012.

[16] D. Jevdjic *et al.*, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[17] ——, "Unison cache: A scalable and effective die-stacked dram cache," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 25–37.

[18] C. Chou *et al.*, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE Computer Society, 2014, pp. 1–12.

[19] J. Sim *et al.*, "Transparent hardware management of stacked dram as part of memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE Computer Society, 2014, pp. 13–24.

[20] C.-C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[21] N. Gulur *et al.*, "Bi-modal dram cache: Improving hit rate, hit latency and bandwidth," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 38–50.

[22] C. Chou *et al.*, "Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches," in *Proceedings of the 42Nd Annual*

International Symposium on Computer Architecture, ser. ISCA '15. ACM, 2015, pp. 198–210.

[23] A. Gupta *et al.*, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *In International Conference on Parallel Processing*, 1990, pp. 312–321.

[24] A. Agarwal *et al.*, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. IEEE Computer Society Press, 1988, pp. 280–298.

[25] D. Lenoski *et al.*, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. ACM, 1990, pp. 148–159.

[26] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the amd opteron processor," *Micro, IEEE*, vol. 30, no. 2, pp. 16–29, March 2010.

[27] D. J. Sorin *et al.*, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[28] M. Ferdman *et al.*, "Cuckoo directory: A scalable directory for many-core systems," pp. 169–180, Feb 2011.

[29] B. A. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. ACM, 2011, pp. 93–104.

[30] D. Sanchez and C. Kozyrakis, "Scd: A scalable coherence directory with flexible sharer set encoding," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. IEEE Computer Society, 2012.

[31] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014.

[32] *Introduction to Magny-Cours*, AMD, 2010.

[33] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[34] *Intel QuickPath Interconnect*, Intel, 2009.

[35] *AMD HyperTransport*, AMD, 2001.

[36] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ser. ISCA '84. ACM, 1984, pp. 348–354.

[37] S. C. Woo *et al.*, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. ACM, 1995, pp. 24–36.

[38] C. Bienia *et al.*, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[39] D. H. Bailey *et al.*, "The nas parallel benchmarks; summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. ACM, 1991, pp. 158–165.

[40] J. Pisharath *et al.*, "Nu-minebench 2.0," Center for Ultra-Scale Computing and Information Security, Northwestern University, Tech. Rep. CUCIS-2005-08-01, August 2005.

[41] W. Heirman *et al.*, "Power-aware multi-core simulation for early design stage hardware/software co-optimization," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012.

[42] P. J. Denning and S. C. Schwartz, "Properties of the working-set model," *Commun. ACM*, vol. 15, no. 3, pp. 191–198, Mar. 1972.

[43] T. L. Johnson *et al.*, "Run-time spatial locality detection and optimization," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. IEEE Computer Society, 1997, pp. 57–64.

[44] J. Weinberg *et al.*, "Quantifying locality in the memory access patterns of hpc applications," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. IEEE Computer Society, 2005, pp. 50–.

[45] G. Zhang *et al.*, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. ACM, 2015, pp. 13–25.

[46] C. H. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, vol. 1, Jan 1989, pp. 277–285 vol.1.

[47] A. Roy and T. M. Jones, "Allarm: Optimizing sparse directories for thread-local data," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. European Design and Automation Association, 2014, pp. 78:1–78:6.

[48] P. Stenström *et al.*, "Comparative performance evaluation of cache-coherent numa and coma architectures," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. ACM, 1992, pp. 80–91.

[49] B. Falsafi and D. A. Wood, "Reactive numa: A design for unifying s-coma and cc-numa," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. ACM, 1997, pp. 229–240.

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

[51] *Linux 3.8 Automatic NUMA balancing*, Linux, 2013.

[52] *Microsoft Windows NUMA Support*, Microsoft.

[53] F. Gaud *et al.*, "Challenges of memory management on modern numa systems," *Commun. ACM*, vol. 58, no. 12, pp. 59–66, Nov. 2015.

[54] M. M. K. Martin *et al.*, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[55] A. Moshovos, "Regionscout: exploiting coarse grain sharing in snoop-based coherence," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, June 2005, pp. 234–245.

[56] J. F. Cantin *et al.*, "Improving multiprocessor performance with coarse-grain coherence tracking," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. IEEE Computer Society, 2005, pp. 246–257.

[57] V. Salapura *et al.*, "Design and implementation of the blue gene/p snoop filter," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, Feb 2008, pp. 5–14.

[58] J. Zebchuk *et al.*, "A tagless coherence directory," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. ACM, 2009, pp. 423–434.

[59] L. A. Barroso *et al.*, "Piranha: A scalable architecture based on single-chip multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. ACM, 2000, pp. 282–293.

[60] *OpenSPARC T2 Overview*, Oracle, 2013.

[61] M. E. Acacio *et al.*, "A two-level directory architecture for highly scalable cc-numa multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 1, pp. 67–79, Jan. 2005.

[62] G. Pan *et al.*, "A two-level directory organization solution for cc-numa systems," in *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*, ser. ICA3PP'07. Springer-Verlag, 2007, pp. 142–152.

[63] J. J. Valls *et al.*, "Ps directory: A scalable multilevel directory cache for cmps," *J. Supercomput.*, vol. 71, no. 8, pp. 2847–2876, Aug. 2015.

[64] L. Zhang *et al.*, "Highly efficient synchronization based on active memory operations," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 58–.

[65] J. H. Ahn *et al.*, "Scatter-add in data parallel architectures," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, Feb 2005, pp. 132–142.

[66] H. Hoffmann *et al.*, "Remote store programming: A memory model for embedded multicore," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Springer-Verlag, 2010, pp. 3–17.