LDAC: Locality-Aware Data Access Control for Large-Scale Multicore Cache Hierarchies

QINGCHUAN SHI, University of Connecticut

GEORGE KURIAN, Massachusetts Institute of Technology

FARRUKH HIJAZ, University of Connecticut

SRINIVAS DEVADAS, Massachusetts Institute of Technology

OMER KHAN, University of Connecticut

The trend of increasing the number of cores to achieve higher performance has challenged efficient management of on-chip data. Moreover, many emerging applications process massive amounts of data with varying degrees of locality. Therefore, exploiting locality to improve on-chip traffic and resource utilization is of fundamental importance. Conventional multicore cache management schemes either manage the private cache (L1) or the Last-Level Cache (LLC), while ignoring the other. We propose a holistic *locality-aware* cache hierarchy management protocol for large-scale multicores. The proposed scheme improves on-chip data access latency and energy consumption by intelligently bypassing cache line replication in the L1 caches, and/or intelligently replicating cache lines in the LLC. The approach relies on low overhead yet highly accurate in-hardware runtime classification of data locality at both L1 cache and the LLC. The decision to bypass L1 and/or replicate in LLC is then based on the *measured reuse* at the fine granularity of cache lines. The locality tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in traditional cache coherence protocols. Moreover, the complexity of the protocol is low since no additional coherence states are created. However, the proposed classifier incurs a 5.6KB per-core storage overhead. On a set of parallel benchmarks, the locality-aware protocol reduces average energy consumption by 26% and completion time by 16%, when compared to the state-of-the-art Reactive-NUCA multicore cache management scheme.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: Multicore, cache, locality

ACM Reference Format:

Qingchuan Shi, George Kurian, Farrukh Hijaz, Srinivas Devadas, and Omer Khan. 2016. LDAC: Locality-aware data access control for large-scale multicore cache hierarchies. *ACM Trans. Archit. Code Optim.* 13, 4, Article 37 (November 2016), 28 pages.

DOI: <http://dx.doi.org/10.1145/2983632>

1. INTRODUCTION

Thermal and power limitations have halted the drive to higher core operating frequencies. Increasing the number of cores has become the preferred way of achieving higher performance. Computing trends indicate the integration of a large number of cores on a

This research was partially supported by the National Science Foundation under Grant No. CCF-1452327. This work was also supported in part by Semiconductor Research Corporation (SRC).

Authors' addresses: Q. Shi, Electrical and Computer Engineering, University of Connecticut, Storrs, CT; email: qingchuan.shi@uconn.edu; G. Kurian (current address), Google Inc., Mountain View, CA; email: gkurian@csail.mit.edu; F. Hijaz (current address), Qualcomm Inc., San Diego, CA; email: fhijaz@qti.qualcomm.com; S. Devadas, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA; email: devadas@mit.edu; O. Khan, Electrical and Computer Engineering, University of Connecticut, Storrs, CT; email: khan@uconn.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/11-ART37 \$15.00

DOI: <http://dx.doi.org/10.1145/2983632>

single chip. Since the diameter of on-chip networks increases with core count, the cost of moving on-chip data is becoming expensive. Furthermore, emerging technologies such as 3D die stacking [Dreslinski et al. 2013] make the “distance” to access data even more nonuniform and costly. To make things worse, the energy consumption of on-chip wires is not scaling at the same rate as transistors [Kaul et al. 2012]. Hence, the main performance bottleneck is shifting from computation capabilities to data access and communication.

A large monolithic on-chip cache that holds the application working set, does not scale beyond a small number of cores, and the only practical option is to physically distribute on-chip cache in pieces so that every core is near some portion of the cache [Bell et al. 2008]. In theory this provides a large amount of aggregate cache capacity and fast private cache hierarchy for each core. Unfortunately, it is difficult to manage the distributed cache and network resources effectively since they require architectural support for cache coherence and consistency under the ubiquitous shared memory model.

Popular directory-based coherence protocols enable fast local caching to exploit data locality, but scale poorly with increasing core count [Agarwal et al. 1988; Martin et al. 2012]. Many recent proposals have addressed directory scalability in single-chip multicores using sharer compression techniques or limited directories [Sanchez and Kozyrakis 2012; Zhao et al. 2010; Zebchuk et al. 2009; Eisley et al. 2006]. Yet, fast private caches still suffer from two major problems: (1) due to capacity constraints, they cannot hold the working set of applications that operate on massive data, and (2) due to frequent communication between cores, data is often displaced from them [Kurian et al. 2013]. This leads to an increase in the sensitivity to on-chip network bandwidth, latency, and energy efficiency since an increasingly higher fraction of memory access is spent in the network. Additionally, the request rate to the Last-Level Cache (LLC) is also increased. Hence, unnecessary data movement not only impacts memory access latency, but also incurs wasteful energy consumption of the network and cache resources [Borkar 2007; Kaul et al. 2012].

Keeping computation local can help reduce the on-chip traffic and improve the utilization of cache resources. In this regard, efficient use of local per-core cache hierarchy is of utmost importance. Since the working sets of applications rarely fit within the L1 cache of a two-level local cache hierarchy, bypassing the L1 cache for data that has low reuse (low spatiotemporal locality) frees up space for more useful and high locality data. Similarly, creating replicas for data with high reuse in the local LLC slice while bypassing for other data brings more useful data closer to the requesting core, thereby reducing on-chip traffic. Local computation and reduced on-chip communication can lead to a scalable and efficient multicore architecture.

1.1. Motivation for *Selective Caching* in Multicore Cache Hierarchy

Reuse-aware cache management schemes have been proposed for multicore cache hierarchies. The recently proposed locality-aware cache coherence protocol [Kurian et al. 2013] is motivated by the observation that cache lines exhibit varying degrees of reuse at the private (L1) caches. A traditional cache coherence scheme that replicates data in the private caches is employed for high-reuse data. Low-reuse data is handled efficiently using a word-level *remote access* [Fensch and Cintra 2008] mechanism that directs load and store requests to the core’s shared LLC cache slice without allocating data in the private cache levels.

The locality-aware data replication in the LLC [Kurian et al. 2014] also builds on the idea of cache lines exhibiting varying degrees of reuse at the LLC. A cache-line level classifier differentiates between low- and high-reuse cache lines. Low-reuse

cache lines are accessed at the LLC slices that are marked as their respective home location (determined by the on-chip data placement policy). High-reuse cache lines are replicated and accessed at the local LLC slice of the requesting core.

The proposed protocol is motivated by the observation that both previous cache management schemes enable variable performance and energy improvements for individual benchmarks, and to the best of our knowledge no scheme optimizes for data locality in all levels of a multicore cache hierarchy. Although the previously mentioned schemes efficiently manage the private-L1 and the shared-LLC caches separately, they leave the other one unmanaged. The need to intelligently manage the whole cache hierarchy exists to efficiently execute a wide range of workloads with varying access patterns, degree of sharing, and reuse at the L1 cache and the LLC.

1.2. Proposed Idea

We propose a *Locality-Aware Data Access Control* protocol (LDAC) to holistically manage all levels of the multicore cache hierarchy. LDAC adapts to cache line reuse at both L1 cache and the LLC to better utilize cache and network resources, and eliminate unnecessary on-chip traffic. When a core makes a memory request that misses the private L1 cache, the coherence protocol either brings the entire cache line using a traditional directory protocol, or just accesses the requested word at the LLC location. The LLC location can either be a replica at the local LLC slice of the requesting core, or the home location at a remote LLC slice. The decision to make a cache line replica or not at the local LLC slice is based on the *spatiotemporal locality* of that cache line. We propose a low-overhead (5.6KB storage per core) yet highly accurate in-hardware predictive mechanism to track and classify the *reuse* of each cache line in the L1 cache and the LLC. This reuse tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in traditional cache coherence protocols. Following are the key contributions of the proposed protocol:

- (1) Enables lower data access latency and energy consumption by selectively bypassing L1 cache for low-locality data, and/or selectively replicating high-locality cache lines in the LLC slice of the requesting core. LDAC improves utilization of each core's local cache hierarchy, and thereby eliminates unnecessary on-chip traffic due to eviction, write back, and invalidation of cache lines with low reuse.
- (2) Allows coherence complexity almost identical to that of a traditional nonhierarchical coherence protocol since replicas are only allowed to be placed at the local cache hierarchy of the requesting core. Moreover, memory consistency is guaranteed trivially for in-order cores. However, speculative out-of-order cores utilize a recently proposed timestamp-based memory consistency violation detection scheme that incurs a per-core storage overhead of 2.2KB [Kurian et al. 2015]. LDAC implements an in-hardware classifier that tracks and classifies reuse of cache lines. A novel classifier organization is proposed that ensures good performance with a per-core storage overhead of 5.6KB.

2. BASELINE SYSTEM

The baseline system is a tiled multicore with an electrical 2D mesh interconnection network, and a two-level cache hierarchy per core. Each core consists of a compute pipeline, private-L1 instruction and data caches, a physically distributed inclusive shared-LLC (L2 cache) with integrated directory, and a network router. Cache coherence is maintained using a directory-based MESI protocol. The coherence directory is integrated with the LLC slices by extending the tag arrays (in-cache directory organization [Censier and Feautrier 1978; Bell et al. 2008]) and tracks the sharing status

of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [Kurian et al. 2010]. Some cores have a connection to a memory controller as well.

The mesh network uses dimension-order X-Y routing and wormhole flow control. The shared LLC is managed using Reactive-NUCA's data placement, replication, and migration mechanisms [Hardavellas et al. 2009]. Private data is placed at the LLC slice of the requesting core, shared data is address interleaved across all LLC slices, and instructions are replicated at a single LLC slice for every cluster of four cores using a rotational interleaving mechanism. Private data is placed at the LLC slice of the requesting core and shared data is address interleaved across all LLC slices. Reactive-NUCA also proposed replicating instructions at a cluster level (e.g., four cores) using a rotational interleaving mechanism. However, this mechanism is not used since the proposed scheme can replicate all types of cache lines.

3. LOCALITY-AWARE CACHE HIERARCHY MANAGEMENT

3.1. System Overview

The key idea of locality-aware multicore cache management is to attempt to only replicate a cache line in the local cache hierarchy of the requesting core, where it can exploit *locality*. Cache lines heavily reused by a core's L1 and/or L2 cache should be replicated in the respective cache level. A core's local cache hierarchy may show cache access patterns where it might be beneficial to either replicate a cache line in both L1 and L2, or just one of the two levels. For example, if the working set thrashes the local L1, it may be still beneficial to cache that data in local L2 cache, granted it shows good reuse at that level. As another example, if the working set perfectly fits in the L1, then it is beneficial to not replicate cache lines in the L2 to save unnecessary insertions, or occupy L2 space that could be used for other useful data. Shared data (including instructions) also plays a key role in the selection of where in the local cache hierarchy it should be cached. For frequently written shared data, cache line invalidations or write-backs add significant latency and energy overheads. Therefore, for such type of data it is useful to not replicate in the requesting core's cache hierarchy, and just access the word at the home L2 cache (where the directory resides) using the auxiliary remote access protocol [Kurian et al. 2013, 2015]. On the other hand, shared data that is more frequently read by one or more cores benefits from replicating it in the requesting core's cache hierarchy. Again, based on the reuse of a cache line at the local cache hierarchy, it may be beneficial to either replicate it in the L1 and/or L2 cache locations.

To implement the locality-aware local cache hierarchy replication strategy, four replication modes are defined on top of the baseline system.

- (1) **L1-Private-L2-Remote:** Replicate cache line at requesting core's L1 cache only. This is the default mode that is also the access mode for traditional cache coherence protocols.
- (2) **L1-Private-L2-Private:** Replicate cache line at both L1 and L2 cache locations of the requesting core.
- (3) **L1-Remote-L2-Private:** Only replicate cache line at the L2 cache location of the requesting core. In this mode the cache line is not inserted in the L1 cache of the requesting core. If the core performs a load or store operation, the word is accessed at the local L2 cache.
- (4) **L1-Remote-L2-Remote:** Cache line is not inserted in the local cache hierarchy of the requesting core. If the core performs a load or store operation, the word is accessed at the home L2 cache location. In case the home L2 cache is on another core, a round-trip message is sent over the on-chip network.

Fig. 1. Protocol operations of the proposed system. L1 and L2 caches are extended with locality tracking and classification structures, which are shown in orange. The upper figure shows the various replication and data access operating modes. The red refers to the *L1-Private-L2-Remote* mode. The blue refers to the *L1-Private-L2-Private* mode. The violet refers to the *L1-Remote-L2-Remote* mode. The green refers to the *L1-Remote-L2-Private* mode.

The replication locations are also defined to illuminate the modes.

- (1) **L1 Replica Location:** A requesting core's private L1 cache.
- (2) **L2 Home Location:** The home L2 cache location of a cache line, where it is first brought on chip. It is determined by data placement policy, and maintains the directory information for cache coherence.
- (3) **L2 Replica Location:** A requesting core's L2 cache that is granted a replica of a cache line.

The top part of Figure 1 demonstrates different modes of a cache line in LDAC. The lower left core's L2 cache is the *L2 Home* location of that cache line. Core (A) has the cache line in *L1-Private-L2-Remote* mode. Thus, it is replicated at the *L1 Replica Location*. Core (A) accesses the cache line directly at its L1 cache to exploit locality. Core (B) has the cache line in *L1-Private-L2-Private* mode; it is replicated in both *L1 Replica Location* and *L2 Replica Location*. Core (B) accesses the cache line directly at its L1 cache. In case the cache line is evicted frequently (due to limited L1 capacity) from (B)'s L1 cache, it will hit in the local L2 cache. Core (C) has the cache line in *L1-Remote-L2-Remote* mode. The cache line is replicated neither in the L1 nor the L2 cache of (C). Core (C) accesses the requested word using the remote access protocol at the *L2 Home* location. Core (D) has the cache line in *L1-Remote-L2-Private* mode. The cache line is only replicated in

L2 Replica Location. Core ① accesses the cache line using word requests at its local L2 cache.

The bottom part of Figure 1 shows the microarchitecture modifications (in orange) needed on top of the baseline machine to support the four local cache hierarchy replication modes. LDAC adds in-hardware structures to track cache line level *reuse* at the L1 and L2 caches, as well as a *classifier* that intelligently utilizes the reuse information to adapt the replication decisions. The protocol details are explained in Section 3.2, however, the basic ideas are outlined first. Each cache line entry in the L1 and L2 cache tag is extended with *reuse bits* (L1-Private Reuse and L2-Replica Reuse) to track the reuse of the cache line at the L1 replica and L2 replica locations, respectively. This reuse information is updated on each access that hits in the L1 or L2 cache of the requesting core, and communicated to the home L2 cache when the replica cache line is removed. The purpose of this reuse information is to quantify whether a requesting core shows enough reuse at one or both cache levels at the local cache hierarchy. The home L2 location is equipped with a *classifier* that is responsible for making decisions about whether a cache line should be allowed a replica copy in the requesting core's cache hierarchy or not. In case the cache line is not allowed to be replicated in the requesting core's L1 or L2, or both L1 and L2 cache levels, the classifier maintains *reuse* information that tracks how often a core accesses the cache line at the L2 home location (L1-Remote Reuse and L2-Home Reuse are used to predict the potential to later promote the cache line for replication in the requesting core's L1 and L2 caches, respectively). The classifier also tracks the replication mode decision (L1-Mode and L2-Mode), which is updated on each access to the home L2 cache. A classifier is maintained for each core, such as for a p core system; the *Core ID* would be an index from 1 to p . This assumption will be removed in Section 3.3.1, where a storage efficient implementation of the classifier is presented.

3.2. Protocol Operation

In the baseline per-core private-shared inclusive cache hierarchy, after a core accesses data, it is placed in the home L2 cache determined by the data placement policy, and replicated in the requesting core's private L1 cache (L1 replica). The proposed LDAC protocol enables a flexible replication policy where each cache line is either replicated in the L1 and/or the L2 cache of the requesting core, or it is not allowed to be replicated at all. For each requesting core, the replication mode of a cache line switches between four modes based on the tracked reuse. The cache line reuse at various cache locations is maintained in hardware structures shown in Figure 1, and defined as follows.

- (1) **L1-Private Reuse:** The number of times a cache line is accessed (read or written) by a core in its L1 cache before it gets invalidated or evicted. As shown in Figure 1, it is tracked at the requester's L1 cache. The *L1-Mode* of the cache line is set to private ("1") when its reuse is above a threshold, called Private to Remote threshold for L1 **P2R-L1**. Otherwise, it is set to remote ("0"), which disallows replication of the cache line in the requester's L1 cache.
- (2) **L1-Remote Reuse:** The number of times a cache line is accessed by a core at the home L2 cache before it is brought into its L1 cache or gets written to by another core. As shown in Figure 1, it is tracked at the home L2 cache. This reuse counter is updated when the *L1-Mode* of the cache line is in remote mode, and the classifier uses this information to make the decision for transitioning from remote to private mode.
- (3) **L2-Replica Reuse:** The number of times a cache line is accessed at L2 replica location before it is invalidated or evicted. As shown in Figure 1, it is tracked at the classifier in the L2 replica location. The *L2-Mode* of the cache line is set to private

(“1”) when its reuse is above a threshold, called Private to Remote threshold for L2 **P2R-L2**. Otherwise, it is set to remote (“0”), which disallows replication of the cache line in the requester’s L2 cache.

- (4) **L2-Home Reuse:** The number of times a cache line is accessed by a core at its home L2 cache before it is brought into its L2 cache or gets written to by another core. As shown in Figure 1, it is tracked at the home L2 cache, and updated when the *L2-Mode* of the cache line is in remote mode. The classifier uses this reuse information to make the decision for transitioning from remote to private mode, in which case the cache line would be allowed to replicate in the requester’s L2 cache.

When a cache line is first brought on chip, it is inserted at the L2 home location. The L2 cache then hands out a private replica to the requesting core’s L1 cache. Note that when a cache line is brought in from off-chip memory, it is in the default *L1-Private-L2-Remote* mode. The classifier at the L2 home cache initializes the *L2-Home Reuse* counter to “1,” and hands out a copy of the cache line to the requesting core. The core then tracks the locality of the cache line by initializing the *L1-Private Reuse* counter in its L1 cache tag to “1” and incrementing this counter for every subsequent read or write. This reuse is communicated to the classifier through coherence messages due to L1 cache eviction or invalidation. On a subsequent access, the classifier determines whether the cache line should be replicated at the L1, or L2, or both L1 and L2 caches of the requesting core (L1 replica and L2 replica locations). Note that the classifier at the L2 replica location uses its *L2-Replica Reuse* counter to track the number of times the cache line at the L2 replica location has been reused. Based on this reuse, the classifier at the L2 replica location only determines whether the cache line should be replicated at the L1 of that core (L1 replica location). Again, this counter is communicated back to the classifier at L2 home location on an invalidation or eviction, which determines future replication decisions.

To understand the protocol operation, let us consider read requests first, write requests next, and finally evictions and invalidations.

3.2.1. Read Requests.

At L1 Cache: On a read request (includes an instruction cache access), first the L1 cache is looked up. If the cache line is present, it is returned to the compute pipeline and the *L1-Private Reuse* counter at the L1 cache (shown in Figure 1) is incremented. Otherwise, the request is sent to the L2 replica location.

At L2 Replica: If the cache line L2 replica exists, the classifier is looked up to get the *L1-Mode*. Since the L2 replica exists, the *L2-Mode* can only be *L2-Private*. If the mode is *L1-Private-L2-Private*, then a read-only copy of the cache line is transferred back to the requesting core. It is inserted into the L1 cache with the *L1-Private Reuse* counter initialized to “1.” If the mode is *L1-Remote-L2-Private*, then the *L1-Remote Reuse* counter at the L2 replica location is incremented to indicate that the cache line was reused at its L2 replica location. As shown in Figure 2, if the *L1-Remote Reuse* counter is $\geq P2R-L1$, the cache line is promoted to *L1-Private-L2-Private* mode and a cache line copy is handed to the requesting core. Else, the requested **word** is returned to the core, without replicating the cache line in L1 cache. Finally, the *L2-Replica Reuse* counter at the L2 replica is also incremented on each word access to indicate that the cache line has been reused at the L2 replica location.

At L2 Home: If the cache line is not present at the L2 replica location, the request is forwarded to the L2 home location. If the cache line is present at the L2 home location, then the classifier is looked up to get the mode of the cache line. Otherwise, the cache line is brought from the off-chip memory. If the classifier indicates that the cache line is *L2-Private*, then the line is sent to the requesting core’s L2 cache (L2 replica location) and the L1 mode provided by the classifier (*L1-Private* or *L1-Remote*)

	Action	Triggering Event	Reuse Check	Possible Location
L1 Mode	Promote	Read Write	$L1\text{-Remote Reuse} \Rightarrow P2R\text{-}L1$	L2 Replica L2 Home
	Demote	Invalidation Eviction	$(L1\text{-Local Reuse} + L1\text{-Remote Reuse}) < P2R\text{-}L1$	L2 Replica L2 Home
L2 Mode	Promote	Read Write	$L1\text{-Home Reuse} \Rightarrow P2R\text{-}L2$	L2 Home
	Demote	Invalidation Eviction	$(L2\text{-Home Reuse} + L2\text{-ReplicaReuse}) < P2R\text{-}L1$	L2 Home

Fig. 2. The replication mode of a cache line is determined based on its reuse at the L1 and L2 cache levels. The mode can be *promoted* from *remote* to *private*, or demoted from *private* to *remote*.

at the L2 home is used to initialize the L1 mode at the L2 replica location. On the other hand, if the classifier returns L2 mode as *L2-Remote*, then the *L2-Home Reuse* counter at the classifier is incremented. If the *L2-Home Reuse* is $\geq P2R\text{-}L2$, then the cache line is promoted to *L2-Private* as shown in Figure 2, and it is sent to the requesting core's L2 cache. Otherwise, the L2 mode remains unchanged as *L2-Remote*, and L1 mode is checked at the L2 home location. If the L1 mode is *L1-Private*, the cache line is sent to the L1 cache, else the *L1-Remote Reuse* counter is incremented. In case the *L1-Remote Reuse* is $\geq P2R\text{-}L1$, the mode is transitioned from *L1-Remote-L2-Remote* to *L1-Private-L2-Remote* (as shown in Figure 2), and the cache line is sent to the L1 cache. Otherwise, the requested word is directly sent to the core, and the *L1-Remote Reuse* counter at the L2 home location is incremented.

At DRAM Controller: If the data is not present at the L2 home location, the request is forwarded to the DRAM controller. Once the cache line is returned from DRAM, the classifier is initialized to default *L1-Private-L2-Remote* mode. Consequently, the cache line is to the requesting core's L1 cache.

3.2.2. Write Requests.

At L1 Cache: On a write request the L1 cache is looked up. If the cache line is present and in *Modified (M)* or *Exclusive (E)* state, then the word (i.e., write data) is directly written to the L1 cache. The *L1-Private Reuse* counter at the L1 cache tag is incremented to indicate that the cache line has been reused. Otherwise, the request is sent to the L2 replica location.

At L2 Replica: If the cache line is present at the L2 replica location and in *M* or *E* state, then the classifier is looked up to get the mode of the cache line. If the mode is *L1-Private-L2-Private*, then it is inserted into the L1 cache with the *L1-Private Reuse*

counter initialized to “1.” If the mode is *L1-Remote-L2-Private*, then the *L1-Remote Reuse* at the L2 replica location is incremented to indicate that the cache line was reused at its remote location. If the *L1-Remote Reuse* counter is $\geq P2R-L1$, the sharer is promoted to *L1-Private-L2-Private* mode and a shared read-write copy of the cache line is handed to it. Else, the word is directly written to the L2 cache at L2 replica location. The *L2-Replica Reuse* counter is also incremented to indicate that the cache line has been reused at the L2 replica location. If the cache line is not present at the L2 replica location, or it is in Shared (S) state, the write request is sent to the L2 home location.

At L2 Home: If the cache line is present at the L2 home location, then the directory (colocated with the L2 home) performs the following actions: (1) It invalidates all the replicas of the cache line as described in Section 3.2.3, which includes triggering the mode transition as shown in Figure 2, and resetting the reuse counters. (2) Based on the updated mode, it handles the write request accordingly. After the invalidation process, if the classifier indicates that the L2 mode is *L2-Private*, a private read-write copy of the cache line is sent to the L2 replica location. Similar to read request, then the L1 mode in the L2 replica classifier is initialized accordingly. On the other hand, if the classifier returns an *L2-Remote* mode, then the *L2-Home Reuse* counter in the classifier is incremented and no L2 replica is sent. Then, if the classifier indicates that the *L1-Mode* is *L1-Private*, a private read-write copy of the cache line is sent to the L1 cache. However, if the cache line is in *L1-Remote* mode, the word is directly written to the L2 cache and the *L1-Remote Reuse* counter at the L2 home location is incremented.

At DRAM Controller: Same as read request, if the data is not present at the L2 home location, the request is forwarded to the DRAM controller. When the cache line is returned from DRAM, the classifier is initialized to *L1-Private-L2-Remote* mode.

3.2.3. Evictions and Invalidations.

Invalidations: When a core requests to get an exclusive copy of a cache line, invalidation requests are initiated from the directory at the L2 home location to all the sharer cores. (Note that evictions can also trigger invalidations, which is discussed in the next paragraph.) When a core gets an invalidation request, both L1 and L2 replica locations are probed and invalidated. An acknowledgment is sent back if a valid cache line is found. In the case in which the cache line is in *L1-Remote-L2-Remote* mode, no acknowledgment is sent. In the case in which the cache line is in *L1-Private-L2-Remote* mode, the *L1-Private Reuse* is sent back to the L2 home location. In the classifier at L2 home location, if $(L1-Remote\ Reuse + L1-Private\ Reuse) < P2R-L1$, the mode of the cache line is demoted to *L1-Remote-L2-Remote*, as shown in Figure 2. Otherwise the mode stays as is. In the case in which the cache line is in *L1-Private-L2-Private* or *L1-Remote-L2-Private* mode, *L2-Replica Reuse* in the classifier at L2 replica location is sent back to the L2 home location. In the classifier at L2 home location, if $(L2-Home\ Reuse + L2-Replica\ Reuse) < P2R-L2$, the mode of the cache line is demoted to *L1-Private-L2-Remote* mode or *L1-Remote-L2-Private* mode accordingly, as shown in Figure 2. Then *L1-Remote Reuse* and *L2-Home Reuse* counters of the sharer cores are reset to ‘0’ after they are invalidated.

Evictions: When a cache line is evicted from the L1 replica location, an acknowledgment is sent to L2 cache with the *L1-Private Reuse* count. In case the cache line is in *L1-Private-L2-Private* mode, the acknowledgment is sent to the L2 replica location. In the classifier at the L2 replica location, if $(L1-Remote\ Reuse + L1-Private\ Reuse) < P2R-L1$, the mode of the cache line is demoted to *L1-Remote-L2-Private*, as shown in Figure 2. If the cache line is in *L1-Private-L2-Remote* mode, the acknowledgment is sent to the L2 home location. Similarly, in the case of $(L1-Remote\ Reuse + L1-Private\ Reuse) > P2R-L1$, its mode is demoted to *L1-Remote-L2-Remote*. When a cache line is evicted from the L2 replica location, the cache line is invalidated from the L1 replica

Fig. 3. Locality classifier at L2 cache. It only tracks a limited number of cache lines, and limited cores for each cache line.

location of that core (if it exists), and the *L2-Replica Reuse* count is sent with an acknowledgment to the L2 home location. The mode in the classifier at the L2 home location is then updated based on $(L2\text{-Home Reuse} + L2\text{-Replica Reuse})$. When a cache line is evicted from the L2 home location, all L1 and L2 replicas are invalidated using the same process as described earlier.

3.3. Optimizations

3.3.1. Limited Entry Locality Classifier. The classifier organization described so far keeps track of locality information for all cache lines in the shared L2 cache. Moreover, for each tracked cache line it maintains storage bits for all cores in the system. This *Complete* locality classifier has a storage overhead of $>90\%$ at 64 cores, and over $15\times$ at 1,024 cores. LDAC implements a storage efficient classifier in hardware that builds on the idea of tracking a limited number of cache lines, as well as a limited number of cores per cache line.

First, a classifier that tracks a limited number of cores per cache line is described. Figure 3 shows this classifier that maintains a list of the locality information for a limited number of cores (k), and is termed the *Limited_k* classifier. At startup, all entries in the limited locality list are free and this is denoted by marking all core IDs as *Invalid*. When a core makes a request to the L2 home location, the directory first checks if the core is already being tracked by the limited locality list. If so, the actions described previously are carried out. Else, the directory checks if a free entry exists. If it does exist, it allocates the entry to the core and the same actions are carried out. Otherwise, the directory checks if a currently tracked core can be replaced. An ideal candidate for replacement is a core that is currently not using the cache line. Such a core is termed an *inactive* sharer and should relinquish its entry to a core in need of it. A replica core becomes inactive on an L2 invalidation or an eviction. A nonreplica core becomes inactive on a write by another core. If such a replacement candidate exists, its entry is allocated to the requesting core. The initial replication mode of the core is obtained by taking a majority vote of the modes of the tracked cores. This is done so as to start off the requester in its most probable mode. Finally, if no replacement candidate exists, the mode for the requesting core is obtained by taking a majority vote of the modes of

all the tracked cores. The limited locality list is left unchanged. It should be noted that if the number of tracked sharers is an even number, the majority vote decision is biased toward the default *L1-Private-L2-Remote* mode. For example, in $k = 2$, both the sharers need to be in “remote” mode for L1 classification for the vote to be *L1-Remote*. Similarly, both the sharers need to be in “private” mode for the majority vote to be *L2-Private*. The intuition behind the *biased* majority voting is that the classifier tries to maintain the *L1-Private-L2-Remote* mode, unless it is proven that the other mode will benefit more. The storage overhead for the $Limited_k$ classifier is directly proportional to the number of cores (k) for which locality information is tracked. Section 3.5 outlines the storage overheads for the $Limited_k$ classifier, whereas Section 5.2 evaluates the performance and energy trade-offs for the $Limited_k$ classifier.

To further reduce the storage overhead, the classifier is modified to track a limited number of cache lines on top of the $Limited_k$ classifier. As shown in Figure 3, the limited cache line classifier is implemented as a set-associative cache-like structure that is accessed in parallel with the L2 cache. Each entry contains the $Limited_k$ locality information for one cache line. A new classifier entry is brought in when the cache line is placed in the L2 cache for the first time. When no empty slot is available for a new cache line, an existing entry is replaced based on the Least Recently Used (LRU) replacement policy. In this case, the storage overhead is proportional to the number of cache lines being tracked. Section 3.5 outlines the storage overheads for the limited classifier that tracks a limited number of cache lines, as well as a limited number of cores per cache line. The performance and energy of the limited classifier are evaluated in Section 5.2. Based on empirical observations, the $Limited_2$ classifier with 512 tracked cache lines is picked, which uses an eight-way set-associative hardware structure.

3.3.2. Remote Access Threshold for L1 Cache. Efficient management of the L1 cache capacity is of utmost importance for maintaining good performance. When a cache line is marked as *L1-remote*, it is accessed by the requesting core at the L2 replica or L2 home location. When a cache line is promoted from *L1-remote* to *L1-private* mode, it is brought into the L1 cache. Under this mode transition, it is important to make the decision such that other L1 cache lines that are equally or better utilized are not evicted, that is, the just promoted cache line does not cause L1 cache pollution. The protocol described so far allows a *L1-remote* cache line to be promoted to *L1-private* after a fixed number of accesses (P2R-L1). This can possibly result in cache lines evicting each other and ping-ponging between the L1 and L2 caches. In this section, a method is devised to mitigate this effect for the proposed locality classifier.

We introduce a new threshold, Remote Access Threshold (**L1-RAT**), for remote-to-private mode transition for the L1 cache, that is, the number of accesses at which a core transitions from *L1-Private* mode to *L1-Remote* mode. L1-RAT serves the following objectives: (1) it decouples the threshold for remote to private mode transition from that for private to remote transition of L1 cache; (2) it dynamically adjusts this threshold based on the observed L1 cache set pressure. Initially, L1-RAT is set equal to P2R-L1. On an invalidation, if the core is classified as remote for L1 cache, L1-RAT is unchanged. This is because the cache set has an invalid line immediately following an invalidation leading to low cache set pressure. However, on an eviction, if the core is demoted to “remote,” L1-RAT is increased to a higher level. This is because an eviction signifies higher cache set pressure. By increasing the L1-RAT to a higher level, it becomes harder for the core to be promoted and acquire an L1 replica, thereby counteracting the cache set pressure. If there are back-to-back evictions, with the core demoted to an *L1-Remote* mode on each of them, *L1-RAT-Level* is further increased to higher levels. However, L1-RAT is not increased beyond a certain value (RATmax) due to the following two reasons: (1) the core should be able to get an L1 replica if it

later shows good locality, and (2) the number of bits needed to track remote utilization should not be too high. The protocol is also equipped with a shortcut in case an invalid cache line exists in the L1 cache. In this case, if remote reuse reaches or rises above $P2R-L1$, the requesting core is promoted to have an L1 replica since it will not cause cache pollution. On the other hand, if the core is classified as an *L1-Private* on an eviction or invalidation, L1-RAT is reset to its starting value of $P2R-L1$. Doing this is essential because it provides the core an opportunity to relearn its classification.

In order to reduce the storage overhead, *L1-RAT-Level* is implemented in the classifier (as shown in Figure 3), to calculate the L1-RAT value. The number of RAT levels used is abbreviated as $nRATlevels$, which is the number of steps from $P2R-L1$ to $RATmax$. Based on empirical observations, $RATmax = 16$ and $nRATlevels = 2$ are used.

3.3.3. L2 Replica Locations. In LDAC, the location where a L2 replica is placed is always the L2 cache of the requesting core. An additional method by which one could explore the trade-off between L2 hit latency and L2 miss rate is by replicating at a cluster level. A cluster is defined as a group of neighboring cores where there is at most one L2 replica for a cache line. Increasing the size of a cluster would increase L2 hit latency and decrease L2 miss rate, and decreasing the cluster size would have the opposite effect. The optimal replication algorithm would optimize the cluster size so as to maximize the performance and energy benefits. Based on empirical observations (similar to the observations in Kurian et al. [2014]), cluster-level replication was not found to be beneficial in the evaluated 64-core system.

3.4. Discussion

3.4.1. Speculative Execution and Memory Consistency Violation Detection. The popular Total Store Order (TSO) memory model is considered in this article. The TSO model can be implemented by enforcing the following two constraints: (1) Load operations wait (i.e., without being issued to the cache subsystem) until all previous loads and fences complete; (2) store operations and fences wait until all previous loads, stores, and fences complete. Since instructions are issued, completed and committed serially in an in-order core, enforcing TSO consistency is trivial. However, a memory transaction that misses in the L1 cache takes several (~ 10 – 100) cycles to complete. Since in-order cores do not allow speculative execution, this latency cannot be overlapped as is possible in an Out-Of-Order (OOO) core that enables instruction and memory level parallelism. In an OOO core an instruction can complete before previous instructions; such as two loads can complete out of order. In case there are interleaving stores to the same address, the second load may consume the stale value and violate TSO ordering. This has been traditionally resolved by relying on cache coherence messages and performing a consistency violation detection check at the commit time of each load operation. If an invalidation is observed for the load address before committing the load operation, the core pipeline is rolled back and the execution is restarted from the load operation.

However, under LDAC the L1 cache allocation is disallowed in the *L1-Remote* mode. In this case, there would not be any invalidation messages for speculative OOO remote word accesses. Therefore, traditional coherence message based consistency violation detection becomes ineffective. Recently, Kurian et al. [2015] proposed a timestamp-based consistency violation detection scheme that enables speculative execution under invalidation-free data access protocols, similar to LDAC. In an OOO speculative core, each load and store operation is assigned an associated timestamp and a simple arithmetic check is done at commit time to ensure that memory consistency has not been violated. The timestamp mechanism is efficient due to the observation that consistency violations occur due to conflicting accesses that have temporal proximity (i.e., within a few cycles of each other), thus requiring timestamps to be stored only for a small

time window. This technique works completely in hardware and requires only 2.2KB of storage per core. Note that this timestamp consistency detection scheme is not required for in-order core type since serialization of data classified as remote does not hamper performance.

3.4.2. Coherence Complexity. The local L2 cache is always looked up on an L1 cache miss or eviction. Additionally, both the L1 and L2 caches are probed on every asynchronous coherence request (i.e., *invalidate*, *downgrade*, *flush*, or *write-back*). This is needed because the directory only has a single pointer to track the local cache hierarchy of each core. This method also allows the coherence complexity to be similar to that of a nonhierarchical (flat) coherence protocol.

To avoid the latency and energy overhead of searching the L2 replica, one may want to optimize the handling of asynchronous requests, or decide intelligently whether to look up the local L2 cache on a cache miss or eviction. In order to enable such an optimization, additional sharer tracking bits are needed at the directory and L1 cache. Moreover, additional network message types are needed to relay coherence information between the L2 home and other actors.

In order to evaluate whether this additional coherence complexity is worthwhile, the LDAC protocol is compared to a dynamic oracle that has perfect information about whether a cache line is present in the local L2 cache. The dynamic oracle avoids all unnecessary L2 lookups. The completion time and energy difference when compared to the dynamic oracle was less than 1%. Hence, in the interest of avoiding the additional complexity, the L2 replica is always looked up for the preceding coherence requests.

3.4.3. Classifier Organization. The classifier for the LDAC protocol is organized using a physically distributed set-associative cache tag-like structure alongside the associated home L2 cache. This makes the classifier decoupled from the directory and L2 cache. The storage overhead for the classifier is calculated in Section 3.5. A separate lookup is required for the classifier and the directory/L2 cache. Even though these lookups are performed in parallel with no latency overhead, the energy expended to look up the classifier needs to be paid.

3.4.4. Three-Level Cache Hierarchy per Core. LDAC can be extended for three-level cache hierarchy with similar benefits and overhead shown in this article. From the point of protocol operation, LDAC focuses on a private-shared cache organization, and it is not sensitive to the number of cache levels. For a private-L1–private-L2–shared-L3 cache organization, it can selectively bypass both private caches together for low-reuse data, and selectively replicate in the shared L3 cache for high-reuse data. From the point of performance, LDAC maintains its benefits from efficiently utilizing the caches. Note that a sharing miss does not depend on the private cache size, thus it can benefit the same way as converting costly sharing misses into word misses. From the point of overheads, LDAC does not require additional storage overhead due to more cache levels. The classifier size is proportional to the size of the last level cache (details in Section 3.5). For three-level cache, the classifier would be only implemented in the L3 level, while only reuse tracking bits with minimum overhead are needed in the private caches.

3.5. Overheads

3.5.1. Storage. The LDAC protocol requires extra bits at the L1 and L2 tag arrays to track cache line reuse information. Each L1 cache tag requires 2 bits for private reuse counter (assuming an optimal *P2R-L1* of 4). The storage overhead per core for L1 cache tags, for an L1 I-Cache of 16KB and L1 D-Cache of 32KB, is $2 \times \frac{(16+32)}{512} = 0.19KB$. We neglect this in future calculations since it is a small overhead.

The *Complete* classifier at the L2 cache is evaluated first. Tracking one core requires 1 bit to store the *L2-Mode*, 2 bits for the *L2-Home Reuse* counter, 1 bit to store the *L1-Mode*, 4 bits for the *L1-Remote Reuse*, and 1 bit to store the *L1-RAT-Level*. Hence, a *Complete* classifier for a 64-core processor requires an additional 576 ($= 64 \times 9$) bits of storage per L2 tag entry.

The *Limited₂* classifier tracks the reuse information for two cores. Tracking one core requires 6 bits to store the core ID (for a 64-core processor), 1 bit to store the *L2-Mode*, 2 bits for the *L2-Home Reuse* counter, 1 bit to store the *L1-Mode*, 4 bits for the *L1-Remote Reuse*, and 1 bit to store the *L1-RAT-level*. Hence, the *Limited₂* classifier requires an additional 30 ($= 2 \times 15$) bits of storage per L2 tag entry.

The *Limited-entry Limited₂* classifier requires 30 bits of storage per tracked L2 cache line, similar to *Limited₂* classifier. The reduction comes from tracking a limited number of cache lines. Each tracked cache line now needs 40 bits for the tag, 1 bit for the valid bit, 3 bits for the LRU along with 30 bits for *Limited₂* classifier. The total overhead for each tracked cache line is 74 bits.

All the following calculations are for one core but they are applicable for the entire processor since all the cores are identical. The sizes of the per-core L1 and L2 caches used in the system are shown in Table I. Each directory entry requires 2 bits for the *L2-Replica Reuse* counter (assuming an optimal *P2R-L2* of 3). The storage overhead of the replica reuse bit is $2 \times \frac{256}{512} = 1KB$. The storage overhead of 512-entry *Limited₂* classifier is $\frac{74 \times 512}{1024 \times 8} = 4.625KB$ per core. The total per-core storage overhead for LDAC classifier is 5.625KB. Compared to baseline, this is an increase of only 1.64% in storage per core. In comparison, for the *Complete* classifier, the overhead is a massive $576 \times \frac{256}{512} = 288KB$ per core. We propose the 512-entry *Limited₂* classifier for LDAC.

3.5.2. Cache Accesses. Updating the private reuse counter in a cache requires a read-modify-write operation on every cache hit. This is true even if the cache access is a read. However, the reuse counter, being just 2 bits in length, can be stored in the tag array. Since the tag array already needs to be written on every cache hit to update the replacement policy (e.g., LRU) counters, the LDAC protocol does not incur any additional cache accesses.

Updating the replica reuse counter in the local L2 slice requires a read-modify-write operation on each replica hit. However, since the replica reuse counter (being 2 bits) is stored in the L2 tag array that needs to be written on each L2 cache lookup to update the LRU counters, the LDAC protocol does not add any additional tag accesses.

At the home location, the lookup/update of the reuse information in the classifier is performed concurrently with the lookup/update of the sharer list for a cache line. This additional expense is accounted in the evaluation.

3.5.3. Network Traffic. The LDAC protocol could create network traffic overhead due to the following three reasons:

- (1) The private reuse counter has to be sent along with the acknowledgment to the directory on an invalidation or an eviction.
- (2) In addition to the cache line address, the cache line offset and the memory access length has to be communicated during every cache miss. This is because the requester does not know whether it is a private or remote sharer (only the directory maintains this information as explained previously).
- (3) The data word(s) to be written has (have) to be communicated on every cache miss due to the same reason.

Some of these overheads can be hidden, while others are accounted for during evaluation.

Table I. Architectural Parameters for Evaluation

Architectural Parameter	Value
Number of Cores	64 @ 1 GHz
Physical Address Length	48 bits
Core	
Type	{In-order, Out-of-order}; Single-issue
Reorder Buffer Size	128 (Out-of-order only)
Load Queue Size	64 (Out-of-order only)
Store Queue Size	48 (In-order and Out-of-order)
Memory Consistency Violation Detection	Timestamps (Out-of-order); Serialization (In-order) [Kurian et al. 2015]
Memory Subsystem	
L1-I Cache per core	16KB, four-way set associative, one cycle
L1-D Cache per core	32KB, four-way set associative, one cycle
L2 Cache per core	256KB, eight-way set associative, six cycles; Inclusive, R-NUCA
Cache Line Size	64 bytes
Directory Coherence Protocol	Invalidation-based MESI; ACKwise ₄ [Kurian et al. 2010]
Number of Memory Controllers	8
DRAM Bandwidth and Latency	5GBps per Controller; 75ns
2D Mesh with XY Routing	
Hop Latency	Two cycles (one-router, one-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits
Header	1 flit (Src, Dest, Addr, MsgType)
Word Length	1 flit (64 bits)
Cache Line Length	8 flits (512 bits)
LDAC Protocol	
Private to Remote Threshold for L1	$P2R-L1 = 4$
Private to Remote Threshold for L2	$P2R-L2 = 3$

- (1) Sending back the reuse counter can be accomplished without creating additional network flits. For a 48-bit physical address and 64-bit flit size, an invalidation message requires 42 bits for the physical cache line address, 12 bits for the sender and receiver core IDs, and 2 bits for the reuse counter. The remaining 8 bits suffice for storing the message type.
- (2) The cache line offset needs to be communicated but not the memory access length. We profiled the memory access lengths for the benchmarks evaluated and found it to be 64 bits in the common case. Memory accesses that are ≤ 64 bits in length are rounded-up to 64 bits, while those > 64 bits always fetch an entire cache line. Only 1 bit is needed to indicate this difference.
- (3) The data word to be written (64 bits in length) is always communicated to the directory on a write miss in the L1 cache. This overhead is accounted for in our evaluation.

4. EVALUATION METHODOLOGY

We evaluate a 64-core simulated multicore processor. The default architectural parameters used for the evaluation are shown in Table I.

4.1. Performance Models

Detailed experiments are performed using single-issue in-order core type, a two-level per-core cache hierarchy, directory coherence protocol, on-chip mesh interconnection network, and memory system models implemented within the Graphite multicore simulator [Miller et al. 2010]. For sensitivity to core type, a single-issue out-of-order core based multicore system is also evaluated. All mechanisms and protocol overheads discussed in Sections 3 are modeled. The electrical mesh interconnection network uses

XY routing. The network delay consists of three parts: router latency, link latency, and contention delay. Since modern network-on-chip routers are pipelined [Dally and Towles 2004], and two- or even one-cycle per hop router latency [Park et al. 2012] has been demonstrated, we model a two-cycle per hop delay (one-cycle for router and link delay each). The contention delay accumulates from the router at each hop. Each router has five output queues (north, east, south, west, and self) that model the contention delay using a history-tree queue model. Similar contention delay modeling is in place for other multicore shared hardware resources, such as the shared cache slices and the memory controllers.

We measure the completion time, that is, the time in a parallel region of the benchmark; this is further broken down into the following categories:

- (1) *Instructions*: Time spent retiring instructions.
- (2) *L1-I Fetch Stalls*: Stall time due to instruction cache misses.
- (3) *Compute Stalls*: Stall time due to waiting for functional unit (ALU, FPU, Multiplier, etc.) results.
- (4) *Memory Stalls*: Stall time due to load/store queue capacity limits, fences, and waiting for load completion. (Note that network-on-chip delay of a memory message is part of this stall.)
- (5) *Load Speculation*: Stall time due to memory consistency violations caused by speculative loads.
- (6) *Branch Speculation*: Stall time due to mispredicted branch instructions.
- (7) *Synchronization*: Stall time due to waiting on locks, barriers, and condition variables.
- (8) *Idle*: Initial time spent waiting for parallel worker threads to be spawned.

For on-chip private L1 cache misses, we analyze the following types of misses:

- (1) *Cold misses* occur to a cache line that has never been previously brought into the L1 cache.
- (2) *Capacity misses* occur to a cache line that was brought in previously but later evicted to make room for another cache line.
- (3) *Sharing misses* occur to a cache line that was brought in previously but was invalidated or downgraded due to a read/write request by another core.
- (4) *Word misses* occur to a cache line that is remotely accessed.

Each miss type is further divided into *Local* and *Remote*: *Local* refers to a hit in the local L2 slice (same core); *Remote* refers to a hit in a remote L2 slice (another core).

4.2. Energy Models

We evaluate just dynamic energy of the memory system components including the cache hierarchy, on-chip network, and DRAM. For energy evaluations of on-chip network routers and links, we use the DSENT [Sun et al. 2012] tool. Energy estimates for the L1-I, L1-D, L2 (with integrated directory) caches, LDAC classifier, and DRAM are obtained using McPAT/CACCTI [Li et al. 2009; Thoziyoor et al. 2008]. The energy evaluation is performed at the 11nm technology node to account for future scaling trends. The DSENT and McPAT tools are extended with models for a trigate 11nm electrical technology node [Kurian et al. 2012].

Commercial multicores deploy various mechanisms to reduce static energy consumption, such as high threshold transistors (HVT), and aggressive power gating techniques [Powell et al. 2000]. Without evaluating such schemes, static energy evaluation would not be insightful. A thorough static energy analysis is thus outside the scope of this article. However, we note that the proposed classifier size is reasonably small, and

Table II. Problem Sizes for the Parallel Benchmarks

Application	Problem Size
SPLASH-2 [Woo et al. 1995]	
FFT	4M complex data points
RADIX	4M Integers, radix 1,024
VOLREND	head
LU-NC	512×512 matrix, 16×16 blocks
BARNES	64K particles
OCEAN-NC	514×514 ocean
WATER-SP	512 molecules
PARSEC [Bienia et al. 2008]	
BLACKSCHOLES	64K options
DEDUP	31MB data
BODYTRACK	two frames, 2,000 particles
FLUIDANIMATE	five frames, 100,000 particles
FACESIM	one frame, 372,126 tetrahedrons
Parallel MI Bench [Iqbal et al. 2010]	
PATRICIA	5,000 IP address queries
CRONO [Ahmad et al. 2015]	
BFS, DFS, PAGERANK, SSSP DIJKSTRA	Graph: 2^{18} nodes, 16 edges/node
TRIANGLE COUNTING, CONNECTED COMPONENTS	
TSP	16 cities
OLTP Database Management System [Yu et al. 2014]	
TPCC	1GB database

can also be power gated. Thus, LDAC would have a similar static power profile as the baseline system, which is expected to follow the trends of completion time.

4.3. Application Benchmarks

We simulate seven SPLASH-2 [Woo et al. 1995] benchmarks: five PARSEC [Bienia et al. 2008] benchmarks, one Parallel MI Bench [Iqbal et al. 2010] benchmark, seven CRONO [Ahmad et al. 2015] benchmarks, and one database management [Yu et al. 2014] benchmark. Each multithreaded benchmark is run to completion using the input sets from Table II.

5. RESULTS

5.1. Comparison of Multicore Cache Management Schemes

In this section, we perform an exhaustive comparison between LDAC and various state-of-art multicore cache hierarchy management schemes.

- (1) **Reactive-NUCA (R-NUCA):** This is the baseline scheme that implements the data placement and migration techniques of R-NUCA [Hardavellas et al. 2009].
- (2) **Private Caching Thread (PCT) scheme:** This is the Locality-Aware Adaptive Cache Coherence scheme that selectively enables replication of cache lines in the private L1 cache [Kurian et al. 2013]. However, the L2 cache is logically shared, that is, replication in the requesting core's L2 cache is not allowed.
- (3) **Replica Thread (RT) scheme:** This protocol implements the Locality-Aware Data Replication in the last-level L2 cache [Kurian et al. 2014]. However, the private L1 cache is *always* allocated on a data access by the requesting core.

Fig. 4. Completion time breakdown for the cache management schemes evaluated. Results are normalized to that of R-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

(4) **LDAC:** This is the proposed LDAC protocol that implements locality-aware replication of cache lines in the requesting core's cache hierarchy (L1 and/or L2 cache).

Completion Time: As shown in Figure 4, the completion time varies distinctly for different benchmarks, even in R-NUCA baseline. This is mainly due to the inherent characteristics of different benchmarks. Such as, for *BODYTRACK* over 80% of the completion time is synchronization and only around 5% is memory stalls. This leaves less space to improve the overall performance through more efficient data accesses. On the other hand, benchmarks like *SSSP-DIJK*, spend a major amount of time in data accesses (>90%), which gives the LDAC greater potential to improve performance.

In order to illuminate the magnitude of completion time breakdown for the R-NUCA baseline, let us consider a qualitative analysis of the Breadth First Search (BFS) benchmark. BFS shows that >90% of the completion time breakdown is attributed to memory stalls. Assuming one micro-operation per cycle throughput for nonmemory micro-operations, the percentage of Memory Stalls in the completion time can be calculated by dividing the memory stall by the completion time. Memory stall can be computed by multiplying the number of memory micro-operations by the average memory access latency. The completion time can be computed by adding the memory stall with the number of nonmemory operations (assuming each nonmemory operation observes an ideal throughput of one cycle). From the simulation, the average memory latency of BFS is ~ 38 cycles, while every third micro-operation is a memory operation. Thus, BFS observes a memory stall of 95%. This is inline with the memory stall breakdown observed in Figure 4. The reason why BFS has such a high average memory access latency is mainly due to its 6% sharing miss rate. Each sharing miss generates at least four messages on the network-on-chip: two for request and reply to the directory and at least one invalidation request and response message to the sharer of the cache line. Note that invalidations to multiple sharers can be sent in parallel. Due to a high degree of sharing, BFS observes the effect of $2\text{--}3\times$ higher number of messages on the network for each sharing miss. The simulation results indicate ~ 76 average network-on-chip delay per message for BFS, and ~ 60 of these cycles originate from contention delay in the network. Given an approximate eight messages and 76 cycles per message, the cost of each sharing miss can be calculated as 600 cycles. For a 6% sharing miss rate, the average memory latency would then match the observed value of ~ 38 cycles.

Figure 4 shows that completion time of the LDAC scheme tracks the best of PCT and RT cache management schemes, and in some cases outperforms both of them. The main

Fig. 5. L1 cache miss type breakdown for the cache management schemes evaluated. Note that in this graph the miss rate increases from left to right, as well as from top to bottom.

reason is that LDAC successfully classifies cache-line level reuse, and either allows or disallows cache line replication at all levels (L1 and L2) of the requesting core's cache hierarchy. As a result, cache lines with high reuse exploit locality at L1 or L2 or both L1 and L2 cache levels. Moreover, low-locality cache lines avoid polluting the local cache hierarchy, and also remove costly invalidations and write-backs for shared data. The result for BFS specifically stands out, primarily because LDAC converts the expensive sharing misses into much cheaper word misses, reducing the memory stall component by a large magnitude. The completion time for the evaluated schemes is dependent primarily on the per-core L1 cache misses. Hence, the L1 cache miss rate is plotted with miss type breakdowns, as shown in Figure 5.

For most benchmarks, we observe a combined behavior in LDAC, which is shown in Figure 5. The RT scheme replicates high-reuse cache lines in the local L2 cache of the requesting core, thereby converting capacity misses to remote L2 cache into local L2 replica hits. Moreover, the PCT scheme also successfully classifies a significant number of the cache lines as low reuse at the L1. Hence, it converts many capacity misses into cheaper word misses. The LDAC scheme combines the advantages of both PCT and RT schemes. As shown in Figure 5, many cache lines in these benchmarks show high reuse at the local L2 replica, however, a significant number of cache lines do not exhibit enough reuse at the L1 cache level. Hence, LDAC observes a significant number of word misses that are now word accesses to the local L2 cache of the requesting core. For example, for BARNES, almost all L1 cache misses under LDAC are classified as local capacity misses or local word misses. This translates into completion time benefits for almost all these benchmarks. The BARNES, BLACKSCHOLES, FACESIM, AND PATRICIA are the largest beneficiaries since they not only take advantage of the PCT scheme, but they also convert a significant number of the L1 misses into local capacity or remote accesses. CONCOMP performance is also improved in PCT and RT schemes individually, however, it is in-between in LDAC. This is because word misses are less costly than capacity misses, as the PCT scheme shows more improvement. When combined, the capacity misses introduced by the RT scheme limit the performance gain, although it is able to convert certain remote capacity misses into local ones. As a result, LDAC outperforms all evaluated schemes for these benchmarks. The VOLREND, WATER-SP, DEDUP, AND BODYTRACK benchmarks perform at par with all evaluated schemes. The OCEAN-NC, FLUIDANIMATE benchmarks do not perform well under the PCT scheme. The reason for their underperformance is that converting capacity misses into word misses may not

Fig. 6. Dynamic energy breakdown for the cache management schemes evaluated. Results are normalized to that of R-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

always result in latency benefits since word misses to remote L2 cache require round-trip accesses over the on-chip network. This overhead is mitigated with the LDAC scheme that converts many word misses into local L2 cache accesses, therefore avoiding the overheads of the on-chip network. Therefore, the completion time of these benchmarks is observed to improve over the PCT scheme. This is justified by the observed statistics (not plotted in the article) for the network-on-chip. For FLUIDANIMATE, in R-NUCA baseline there are ~ 35 million packets (~ 147 million flits) injected in the network-on-chip. Under LDAC, the number of packets injected in the network increases to ~ 50 million (~ 160 million flits). This is because the number of L1 misses is higher, which can also be observed in Figure 5.

The LU-NC benchmark shows an outlier scenario where the PCT scheme significantly hurts performance since converting capacity misses into multiple word misses results in higher memory stalls. Moreover, these memory stalls occur within the parallel critical sections, resulting in significant synchronization delays. The RT scheme mitigates this overhead and in fact shows an overall performance advantage by converting capacity misses to remote L2 cache into local L2 replica hits. The LDAC scheme attempts to take advantage of the RT scheme but the overheads of PCT still cause memory stalls and synchronization delays.

The TSP, BFS, SSSP-DIJK, TRI-CNT, PAGERANK, and DFS benchmarks observe a significant portion of their L1 cache misses for shared data. Moreover, these shared cache lines observe low reuse at both the L1 and L2 caches of the requesting cores. Therefore, the LDAC scheme follows the PCT trends and converts these sharing misses into much cheaper word misses. The completion time advantage of the PCT scheme for these benchmarks is carried over to the LDAC scheme as well. TPCC also has a large sharing misses portion in its miss rate breakdown. However, they are converted into more word misses in LDAC, which makes its performance follow a similar trend as FLUIDANIMATE.

The RADIX and FFT benchmarks follow PCT, however, since the capacity misses that are converted into word misses do not result in completion time benefits, the same trends are observed for LDAC.

Overall, the LDAC scheme performs well for all evaluated benchmarks. The LDAC scheme improves performance by a geometric mean of 24% (average of 16%), 4%, and 20% over R-NUCA, PCT, and RT schemes, respectively.

Energy Consumption: Figure 6 shows the memory system’s dynamic energy consumption breakdown for the evaluated schemes. The PCT and RT schemes incur their

Fig. 7. Energy and completion time for the Limited _{k} classifier as a function of the number of tracked sharers (k). The results are normalized to that of the complete (= Limited₆₄) classifier.

classifier overhead as part of the “Directory” component. However, the LDAC implements a stand-alone classifier structure and hence its energy consumption is reported as a separate component. On average, the LDAC scheme performs better than both the PCT and RT schemes, mainly due to reductions in the L1-D cache lookups, as well as reduced on-chip network traffic. Since the word access is cheaper than cache line access, the L2 cache energy component also improves. In most benchmarks, LDAC tracks the best energy consumption among the PCT and RT schemes, and in some benchmarks it even improves upon the best. LDAC can significantly reduce cache and network energy mainly due to the following factors:

- (1) Fetching an entire cache line on cache miss is replaced by cheaper word accesses.
- (2) Since the caching of low-locality data in L1 cache is eliminated, its space is more effectively used for high-locality data, thereby decreasing the evictions.
- (3) High-locality data can be replicated at local L2 cache, thereby reducing the costly remote accesses to L2 home location on L1 misses.
- (4) Data that has low locality in L1 cache can be replicated at local L2 cache, which makes word accesses incur even lower cost.

Overall, LDAC improves the energy consumption by a geometric mean of 29% (average of 26%), 6%, and 20% over R-NUCA, PCT, and RT schemes, respectively.

Next, we perform sensitivity studies to evaluate how the limited locality classifier and P2R thresholds are configured for LDAC. Finally, we show completion time and energy consumption analysis of LDAC when out-of-order cores are used to configure the multicore system.

5.2. Limited Locality Classifier

Figure 7 plots the completion time and energy consumption of the benchmarks with the Limited _{k} classifier when k is varied as (1, 3, 5, 7, 64). $k = 64$ corresponds to the complete classifier. The results are normalized to Limited₁. The experiments are run with the same schemes as in Section 5.1. We observe that the average completion time and energy are not quite sensitive to k values. The Limited₁ has the smallest storage

Fig. 8. Energy and completion time for the Limited₂ classifier with different numbers of tracked cache lines.

Fig. 9. Average completion time and energy for different P2R thresholds. The P2R values are shown as P2R_L1_L2. Darker color refers to better completion time or energy.

overhead, however, it is more unstable than the other classifiers, due to the fact that only one core is tracked, and its behavior may not be representative. While it performs better than the complete classifier for FLUIDANIMATE, it performs the worst for the BARNES and LU-NC benchmarks. The better energy consumption in FLUIDANIMATE is due to the fact that the Limited₁ classifier starts off new sharers in replica mode as soon as the first sharer acquires replica status. On the other hand, the other classifiers have to learn the mode independently for each sharer leading to a longer training period. Overall, to trade off the storage overhead of our classifier with the energy and completion time improvements, we choose $k = 2$ as the default for the limited classifier.

On top of the Limited₂ classifier, Figure 8 plots the completion time and energy consumption of different configurations of the classifier. First, we evaluate different numbers of cache line entries using the same associativity (eight-way). As shown in the left figure in Figure 8, as the number of entries increases, the results approach that of the full classifier. Since the improvement in performance and energy is not significant going from 512 to 1,024 entries, while the storage overhead is doubled, we select the classifier with 512 entries. For similar reasons, we select the eight-way set-associative structure. As a result of this analysis, we choose the 512-entry Limited₂ classifier for LDAC whose overhead is quantified as 5.625KB per core in Section 3.5.

5.3. P2R Thresholds

The P2R-L1 and P2R-L2 parameters of the LDAC scheme are individually varied from 1 to 10 and the completion time and energy consumption results are plotted in Figure 9. The plots show a color gradient normalized to the case with $P2R-L1 = 1$ and $P2R-L2 = 1$. The darker color refers to better completion time or energy consumption. The system performs worse when $P2R-L2 = 1$, since it makes unnecessary L2 replications for all

Fig. 10. Completion time breakdown for the cache management schemes evaluated in out-of-order core setup. Results are normalized to that of R-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

Fig. 11. Dynamic energy breakdown for the cache management schemes evaluated in out-of-order core setup. Results are normalized to that of R-NUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

the cache lines. When $P2P-L2$ is greater than 1, $P2R-L1$ tends to have a major impact. Overall, we observe that the completion time and energy consumption is high when both $P2R-L1$ and $P2R-L2$ values are high. At high thresholds, LDAC makes minimal replicas in both L1 and L2 caches of the requesting cores, which results in low cache utilization and worse performance. When the $P2R$ values are both in middle range, the completion time and energy consumption tend to be optimal. The $P2R-L1$ of 4 and $P2R-L2$ of 3 are selected because they provide the best Energy $\times$ Delay product among the possible $\langle P2R-L1, P2R-L2 \rangle$ combinations.

5.4. Out-of-Order Core Type

In this section, the impact of the OOO core type used to configure a multicore system is evaluated. As discussed in Section 3.4.1, an OOO core implements a timestamp-based consistency detection protocol to detect memory consistency violations due to speculative execution.

Figures 10 and 11 show the completion time and energy consumption results for the OOO core based multicore, using the R-NUCA, PCT, RT, and LDAC cache management schemes. The results are for a subset of benchmarks evaluated for the in-order core based system. These benchmarks are chosen to show the various trends in performance and energy consumption. It is immediately apparent that the percentage of time spent in memory and compute stalls is much lower in OOO cores due to their ability to exploit instruction level parallelism through dynamic scheduling. Since an OOO core executes

instructions speculatively, long latency operations such as L1 cache misses can be hidden without stalling the pipeline. Hence, all evaluated cache management schemes benefit less for performance when compared to the R-NUCA baseline. However, the energy consumption trends are quite disparate, and on average the LDAC scheme outperforms R-NUCA, PCT, and RT schemes. The timestamp-based memory consistency detection scheme requires additional hardware structures (History Queues) that incur additional overhead for both PCT and LDAC schemes. With the exception of FLUIDANIMATE, all evaluated benchmarks overcome this overhead and improve the energy consumption of LDAC over the R-NUCA baseline. Overall, the LDAC scheme performs well for all evaluated benchmarks. The LDAC scheme improves the completion time by a geometric mean of 20% (average of 18%), 2%, and 7%, and energy by a geometric mean of 19% (average of 21%), 7%, and 10%, over R-NUCA, PCT, and RT schemes, respectively, for the OOO core based multicore.

6. RELATED WORK

Data access bottlenecks in multicores can be alleviated using intelligent cache hierarchy management. Better LLC partitioning [Qureshi and Patt 2006; Beckmann and Sanchez 2013] and replacement schemes [Qureshi et al. 2007; Jaleel et al. 2010] have been proposed to reduce memory pressure. Better cache placement [Hardavellas et al. 2009; Beckmann and Sanchez 2013] and allocation [Kurian et al. 2013] schemes have been proposed to exploit application data locality and reduce network traffic.

CMP-NuRAPID [Chishti et al. 2005] only replicates shared read-only data and adds an additional coherence state to maintain coherence for shared read-write data. *Adaptive Selective Replication (ASR)* [Beckmann et al. 2006] and Enhanced Shared-Private NUCA (ESP-NUCA) [Merino et al. 2010] replicate cache lines in the requester's local LLC slice based on a cost-benefit analysis. *Victim replication* starts out with a private-L1 and shared-L2 organization and uses the local L2 slice as a victim cache for data that is evicted from the L1 cache. By only replicating the L1 capacity victims, this scheme attempts to combine the low hit latency of private designs with the low off-chip miss rates of shared LLCs. These proposals either do not quickly adapt their policies to dynamic program changes, or replicate cache lines in the LLC without paying attention to their locality. POPS [Hossain et al. 2011] proposes a cache coherence design that adapts coherence activities to data sharing patterns. It enables shifting the coherence management of cache lines to L1 or L2 cache, which lowers the cost of certain coherence misses. POPS replicates private cache lines into local LLC slices when they are evicted. This LLC replication decision is based on classification of cache line sharing patterns, not their reuse. Moreover, POPS does not allow selective bypassing of cache lines at the L1 cache. It relies on coherence management at the L1 cache to improve the cost of sharing misses. Therefore, it does not remove costly sharing misses (and cache line ping pong), but just attempts to minimize their cost. In addition, some of these prior works significantly complicate coherence and do not scale to large core counts.

Proposals that start with a private-L1, private-L2 organization have been proposed [Chang and Sohi 2006; Herrero et al. 2010; Qureshi 2009; Srikantaiah et al. 2011; Lee et al. 2011]. These proposals attempt to improve the negative characteristics of private caches, that is, the low hit rates, and at the same time try to capitalize on the low LLC access latency. They start with a private-private organization and then either spill data into other private caches, or reconfigure the private cache size to reduce the off-chip misses and efficiently utilize the on-chip capacity. However, keeping track of and figuring out where to place the spilled data is challenging by itself. These schemes generally rely on coarse-grain information that is not very accurate. Furthermore, they all suffer from the complex hardware structures required to keep the private caches coherent.

Previously, researchers have studied techniques for management of private caches in the context of uniprocessors. In Cache Bursts [Liu et al. 2008], the private cache fetches the entire cache line on every miss and evicts it as soon as it is detected to be a dead block. This does not accrue the network traffic or memory access latency benefits that our protocol enables by just fetching a single-use word for low-locality data. Selective caching has been suggested in the context of single processors to selectively cache data in the on-chip caches based on its locality [Tyson et al. 1995; Johnson and Hwu 1997]. Remotely accessing a word without allocating a cache line for it in the private cache has been explored in the past [Fensch and Cintra 2008].

BIP [Qureshi et al. 2007] and DRRIP [Jaleel et al. 2010] optimize the insertion position of a cache line at the home location in the LLC, and do not replicate cache lines. These schemes focus on mitigating the scanning and thrashing effects within each LLC slice. LDAC considers data locality, coherence, and NUCA effect, and has a “global” view of the cache line behavior. At LLC level, it replicates the high-reuse data in local LLC slice and reduces the access latency. At L1 cache level, LDAC bypasses caching in the private L1 cache based on the locality observed at a per cache line granularity, whereas the insertion policies do not support that. The insertion policies are orthogonal to the proposed LDAC scheme. These can work together in a system for better performance.

The locality-aware cache coherence protocol has been recently proposed to improve on-chip memory access latency and energy efficiency in large-scale multicores [Kurian et al. 2013]. This protocol is motivated by the observation that cache lines exhibit varying degrees of reuse (i.e., variable spatiotemporal locality) at the private cache levels. A cache-line level classifier is introduced to distinguish between low- and high-reuse cache lines. A traditional cache coherence scheme that replicates data in the private caches is employed for high-reuse data. Low-reuse data is handled efficiently using a remote access [Fensch and Cintra 2008] mechanism that does not allocate data in the private cache levels. Instead, it allocates only a single copy in the designated core’s shared cache slice and directs load and store requests made by all other cores toward it. Data access is performed at the word level and requires a round-trip message between the requesting core and the remote cache slice. This improves the utilization of private cache resources by removing unnecessary data replication. In addition, it reduces network traffic by transferring only those words in a cache line that are accessed on-demand. Consequently, unnecessary invalidations and write-back requests are removed that reduce network traffic even further.

The locality-aware data replication in the LLC also builds on the idea of cache lines exhibiting varying degrees of reuse [Kurian et al. 2014]. A cache-line level classifier differentiates between low- and high-reuse cache lines. Low-reuse cache lines are accessed in the LLC from the home location. On the other hand, high-reuse cache lines are replicated in the local LLC slice of the requesting core. This enables local access to cache lines that are accessed frequently in the LLC, reducing network traffic and improving access latency.

All the schemes discussed either manage the L1 cache or the LLC separately, leaving the other cache unmanaged. In contrast, the proposed LDAC scheme holistically manages the complete cache hierarchy. It enables replicating data anywhere in the local cache hierarchy (L1 or L2 or both caches) of the requesting core. This decision is based on the locality demonstrated by a cache line. This way, it can seamlessly adapt to a wide range of workloads with varying access patterns, degree of sharing, and reuse at the L1 and the LLC.

7. CONCLUSION

We have proposed an intelligent locality-aware data access control (LDAC) protocol for efficiently managing the cache hierarchy of large-scale multicores. The cache-line level

reuse is profiled at runtime using a low-overhead yet highly accurate in-hardware classifier. Consequently, high-reuse cache lines are allowed to be replicated in the local cache hierarchy of the requesting core. However, if a particular cache level shows low reuse for a particular cache line, it is disallowed from allocation at that cache level. On a set of parallel benchmarks, LDAC reduces the overall memory system energy consumption by geometric mean of 29% (average of 26%), 6%, and 20%, and the completion time by geometric mean of 24% (average of 16%), 4%, and 20% when compared to the previously proposed R-NUCA, PCT, and RT schemes. The coherence complexity of the LDAC protocol is almost identical to that of a traditional nonhierarchical (flat) coherence protocol since replicas are only allowed to be created at the cache hierarchy of the requesting core. The LDAC classifier is implemented with 5.625KB storage overhead over 304KB cache hierarchy per core.

REFERENCES

- Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark Horowitz. 1988. An evaluation of directory schemes for cache coherence. In *International Symposium on Computer Architecture*.
- Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *IEEE International Symposium on Workload Characterization (IISWC)*.
- Bradford M. Beckmann, Michael R. Marty, and David A. Wood. 2006. ASR: Adaptive selective replication for CMP caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, 443–454. DOI: <http://dx.doi.org/10.1109/MICRO.2006.10>
- Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlav Khan, Froilan Montenegro, Jay Stickney, and John Zook. 2008. TILE64 - Processor: A 64-core SoC with mesh interconnect. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC 2008). Digest of Technical Papers*. 88–598. DOI: <http://dx.doi.org/10.1109/ISSCC.2008.4523070>
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*.
- Shekhar Borkar. 2007. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. ACM, New York, NY, 746–749. DOI: <http://dx.doi.org/10.1145/1278480.1278667>
- Lucien M. Censier and Paul Feautrier. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.* 27, 12 (Dec. 1978), 1112–1118. DOI: <http://dx.doi.org/10.1109/TC.1978.1675013>
- Jichuan Chang and G. S. Sohi. 2006. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*. 264–276. DOI: <http://dx.doi.org/10.1109/ISCA.2006.17>
- Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Washington, DC, 357–368. DOI: <http://dx.doi.org/10.1109/ISCA.2005.39>
- William J. Dally and Brian Towles. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.
- Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, Michael Wieckowski, Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. 2013. Centip3De: A 64-core, 3D stacked near-threshold system. *IEEE Micro* 33, 2 (2013), 8–16. DOI: <http://dx.doi.org/10.1109/MM.2013.4>
- Noel Easley, Li-Shiuan Peh, and Li Shang. 2006. In-network cache coherence. In *IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. 321–332. DOI: <http://dx.doi.org/10.1109/MICRO.2006.27>
- Christian Fensch and Marcelo Cintra. 2008. An OS-based alternative to full hardware coherence on tiled CMPs. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA 2008)*. 355–366. DOI: <http://dx.doi.org/10.1109/HPCA.2008.4658652>

- Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *International Symposium on Computer Architecture*.
- Enric Herrero, José González, and Ramon Canal. 2010. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 419–428. DOI: <http://dx.doi.org/10.1145/1815961.1816018>
- Hemet Hossain, Sandhya Dwarkadas, and Michael C. Huang. 2011. POPS: Coherence protocol optimization for both private and shared data. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. 2010. ParMiBench - An open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters* (2010).
- Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*.
- Teresa L. Johnson and Wen-Mei W. Hwu. 1997. Run-time adaptive cache hierarchy management via reference analysis. In *International Symposium on Computer Architecture*.
- Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. 2012. Near-threshold voltage (NTV) design: Opportunities and challenges. In *Design Automation Conference*.
- George Kurian, Srinivas Devadas, and Omer Khan. 2014. Locality-aware data replication in the last-level cache. In *Proceedings of the 2014 IEEE 120th International Symposium on High Performance Computer Architecture (HPCA 2014)*.
- George Kurian, Omer Khan, and Srinivas Devadas. 2013. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 523–534. DOI: <http://dx.doi.org/10.1145/2485922.2485967>
- George Kurian, Jason Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgén Michel, Lionel Kimerling, and Anant Agarwal. 2010. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *International Conference on Parallel Architectures and Compilation Techniques*.
- George Kurian, Qingchuan Shi, Srinivas Devadas, and Omer Khan. 2015. OSPREY: Implementation of memory consistency models for cache coherence protocols involving invalidation-free data access. In *International Conference on Parallel Architectures and Compilation Techniques*.
- George Kurian, Chen Sun, Chia-Hsin Owen Chen, Jason E. Miller, Jurgén Michel, Lan Wei, Dimitri A. Antoniadis, Li-Shiuan Peh, Lionel Kimerling, Vladimir Stojanovic, and Anant Agarwal. 2012. Cross-layer energy and performance evaluation of a nanophotonic manycore processor system using real application workloads. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*. 1117–1130. DOI: <http://dx.doi.org/10.1109/IPDPS.2012.103>
- Hyunjin Lee, Sangyeun Cho, and B. R. Childers. 2011. CloudCache: Expanding and shrinking private caches. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. 219–230. DOI: <http://dx.doi.org/10.1109/HPCA.2011.5749731>
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*.
- Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *International Symposium on Microarchitecture*.
- Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7 (2012).
- Javier Merino, Valentin Puente, and Jose A. Gregorio. 2010. ESP-NUCA: A low-cost adaptive non-uniform cache architecture. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. 1–10. DOI: <http://dx.doi.org/10.1109/HPCA.2010.5416641>
- Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA*. 1–12.
- Sunghyun Park, Tushar Krishna, Chia-Hsin Chen, Bhavya Daya, Anantha Chandrakasan, and Li-Shiuan Peh. 2012. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI. In *Design Automation Conference*.
- Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. 2000. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000*

- International Symposium on Low Power Electronics and Design (ISLPED'00)*. ACM, New York, NY, 90–95. DOI : <http://dx.doi.org/10.1145/344166.344526>
- Moinuddin K. Qureshi. 2009. Adaptive spill-receive for robust high-performance caching in CMPs. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA 2009)*. 45–54. DOI : <http://dx.doi.org/10.1109/HPCA.2009.4798236>
- Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*.
- Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *International Symposium on Microarchitecture (MICRO)*.
- Daniel Sanchez and Christos Kozyrakis. 2012. SCD: A scalable coherence directory with flexible sharer set encoding. In *International Symposium on High-Performance Computer Architecture*.
- Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut Kandemir, Mary Jane Irwin, and Yuan Xie. 2011. MorphCache: A reconfigurable adaptive multi-level cache hierarchy. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. 231–242. DOI : <http://dx.doi.org/10.1109/HPCA.2011.5749732>
- Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT - A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *International Symposium on Networks-on-Chip*.
- Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*.
- Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. 1995. A modified approach to data cache management. In *International Symposium on Microarchitecture*.
- Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *International Conference on Computer Architecture*.
- Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. DOI : <http://dx.doi.org/10.14778/2735508.2735511>
- Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. 2009. A tagless coherence directory. In *International Symposium on Microarchitecture*.
- Hongzhou Zhao, Arrvinth Shriraman, and Sandhya Dwarkadas. 2010. SPACE: Sharing pattern-based directory coherence for multicore scalability. In *International Conference on Parallel Architectures and Compilation Techniques*. 135–146.

Received December 2015; revised June 2016; accepted August 2016