NUDA: Non-Uniform Directory Architecture for Scalable Chip Multiprocessors

Wei Shu and Nian-Feng Tzeng

Abstract—Chip multiprocessors (CMPs) involve directory storage overhead if cache coherence is realized via sharer tracking. This work proposes a novel framework dubbed non-uniform directory architecture (NUDA), by leveraging our two insights in that the number of "active" directory entries required to stay on chip is usually small for a short execution time window due to high directory locality, and that the fraction of interrogated directory entries drops as the core count rises. Unlike earlier storage overhead reduction techniques that require all cached LLC blocks to have their directory entries fully on chip, NUDA dynamically buffers only most active directory vectors (DVs) on chip while keeping DVs of all LLC blocks in a backing store at low level storage. NUDA attains its superior efficiency via an inventive criticality-aware replacement policy (CARP) for on-chip buffer management and effective prefetching to pre-activate vectors (PAVE) for upcoming coherence interrogations. We have evaluated NUDA by gem5 simulation for 64-core CMPs under PARSEC and SPLASH benchmarks, demonstrating that CARP and PAVE enhance on-chip directory storage efficiency significantly. NUDA with a small on-chip buffer for DVs exhibits negligible performance degradation (to stay within 2.6 percent) compared to a full on-chip directory, while outperforming its previous counterparts for directory area reduction when on-chip directory budget is provisioned scarcely for high scalability.

Index Terms—Chip multi-processors, coherence protocols, hash tables, memory hierarchy, prefetching, sharer tracking directory

1 Introduction

Contemporary large scale chip multiprocessors (CMPs) usually resort to hardware-oriented cache coherence for easy programmability and good performance. Recent directory reduction methods [7], [15], [21] commonly lower the on-chip directory capacity to improve scalability while requiring all cached blocks in cores to have their directory entries fully on chip. However, such a directory can yield either poor private cache hit rates due to coherence-induced invalidations [7], [15] or significant network interference due to mounted coherence traffic [7], [21].

In this work, we present Non-Uniform Directory Architecture (NUDA), a novel directory scheme that greatly reduces on-chip directory storage. Unlike directory compression adopted conventionally, NUDA keeps the directory vectors (i.e., bit-map vectors) of all LLC blocks in off-chip memory and dynamically "buffering" the active vectors on chip in a small directory buffer, hence resulting in our layered directory resident in the multi-level memory hierarchy, so named "non-uniform directory architecture". NUDA builds upon our two newly observed insights: (1) the number of required directory entries (DEs) for a short execution time window is low and (2) the fraction of DEs which are interrogated decreases as the system scales up.

The first insight stems from the fact that during parallel program runs, the number of "active" DEs over a short time window is limited, as demonstrated in Fig. 1a. Active DEs may change from one time window to the next due to coherence interrogations on different LLC blocks. Given a cache block, its directory vector is altered under three coherence events: private cache misses (i.e., the data block is shared by a new core, resulting in adding the new sharer to the corresponding vector), private cache replacements (i.e., resulting in the corresponding bits in the vector reset, if the protocol is non-silent), or private cache write (i.e., with all the vector bits but the write missed one reset). If a block experiences none of the three events, its directory entry is inactive. Statically holding those inactive DEs fully on chip unnecessarily elevates chip area overhead.

The second insight results from the fact that the fraction of DEs that are active over a short time window decreases as the core count increases. For example, blackscholes in PARSEC benchmark suite [2] results in the active DEs ratio of 13.8 percent when run on a 4-core CMP, but the ratio drops to 2.4 percent on a 64-core CMP, as shown in Fig. 1b. Such increased concentration on active DEs offers a great opportunity for NUDA to contain the size of on-chip directory buffer, calling for small on-chip directory area budget in support of high scalability. Hence, the use of an on-chip directory buffer to cache only coherence-active DEs makes NUDA particularly attractive for large-scale CMPs.

NUDA keeps the vectors of *all LLC blocks* in DRAM memory for precise sharer tracking to avoid excessive directory-induced cache misses and mounted NoC traffic. Notice that memory storage provisioned for directory vectors is proportional to the LLC size, irrespective of the memory size. This is in sharp contrast to earlier multi-level directory approaches targeting shared-memory multiprocessors [1], [8], [9], [11], because such an earlier approach requires one directory entry for each memory block frame (of 64 bytes) statically. As a result, the earlier approach makes shared-memory multiprocessors suffer from the notorious directory memory-scaling problem as elaborated in Section 2.1. On the other hand, NUDA is designed for large scale on-chip cache-coherent CMPs to require extremely low chip directory area budget.

NUDA intelligently promotes vector transfer between on-chip and off-chip storage to sustain execution performance via two novel mechanisms. First, NUDA adopts a unique criticality-aware replacement policy (CARP) to help retain performance-critical directory vectors on chip, avoiding those vectors from being evicted undesirably. Second, NUDA reduces the average interrogation latency and increases the on-chip directory hit rate by means of an effective pre-activating vector (PAVE) technique, which prefetches the vectors that are likely to be involved in future directory interrogations, for sustained execution performance.

We use the gem5-simulated Linux environment [8] to evaluate NUDA under PARSEC [2] and SPLASH [19] benchmark suites for 64-core CMPs. Evaluation results reveal that NUDA soundly outperforms previous approaches for on-chip directory storage reduction when on-chip directory budget is kept fairly scarce for high system scalability. To sum up, this work makes the following contributions:

- We examine directory activities in many-core cache-coherent CMPs to show strong locality.
- We discover such locality intensifies as the core count rises.
- We develop NUDA as a multi-layer directory architecture to make it possible to have very low on-chip directory overhead for superior scalability.
- NUDA attains effective on-chip directory buffer management via novel CARP and PAVE techniques.
- NUDA is shown to outperform its earlier counterparts for on-chip area overhead reduction.

The NUDA framework is orthogonal to the plethora of prior compression-based area overhead reduction techniques, like

- *The authors are with Center for Advanced Computer Studies (CACS), University of Louisiana at Lafayette, Lafayette, LA 70503-2014.*
E-mail: {wxs0569, tzeng}@louisiana.edu.

Manuscript received 13 May 2017; revised 3 Nov. 2017; accepted 8 Nov. 2017. Date of publication 13 Nov. 2017; date of current version 13 Apr. 2018.
(Corresponding author: Wei Shu.)
Recommended for acceptance by R. F. DeMara.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2773061

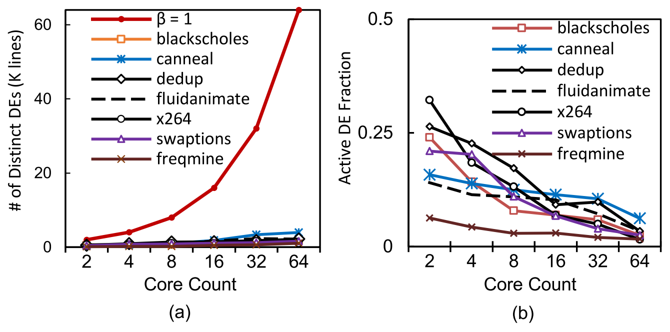


Fig. 1. (a) Maximum number of distinct DEs interrogated in a 1K-cycle time window, with the DE count under $\beta = 1.0$ shown, where β represents the coverage fraction. (b) Active DE fraction of various core counts, measured by the number of distinct DEs over the DE count under $\beta = 1.0$.

SPACE [21], SCD [15], and RECODE [16]. Hence, NUDA is readily applicable to those earlier compression techniques for further on-chip directory area overhead reduction. As each directory vector (DV) takes one DE to hold it, this article uses DV and DE interchangeably.

2 Related Work

This section first discusses how NUDA differs from directory overhead reduction methods for distributed shared memory (DSM) systems. Earlier approaches to achieving chip directory area overhead are then highlighted.

2.1 Memory Directories for DSM Systems

Distributed shared memory (DSM) systems may keep their shared memory directories either in DRAM, such as Origin [8], DASH [9], and FLASH [11], or in memory controllers for better performance [1], [13]. Each memory data block in such a system is associated with a directory entry. NUDA has no infamous directory memory-scaling problem the DSM system faces. It differs from the directory of a DSM system in three distinct aspects. First, NUDA avoids provisioning full directory entries to all LLC blocks on chip for storage reduction, whereas the DSM system equips every memory block with its directory information statically. Second, DRAM storage reserved for holding LLC directory vectors under NUDA is constant, irrespective of the memory size, whereas the directory of a DSM system grows linearly with its memory size. Third, NUDA explores the unique insight based on directory access locality, for effectively transferring sharer vectors across the storage hierarchy.

Multi-Level DSM Directory Design. A multi-level directory approach has been considered earlier [1], [8], [9], [11], [13] for the memory directory of DSM systems. It holds DEs of accessed cache blocks in an on-chip buffer. Each DE held in the buffer includes not only a directory vector but also the state bits (e.g., 5 bits under the MESI protocol) plus other control bits needed to manage the on-chip buffer. In contrast, our NUDA's on-chip buffer is for holding only directory vectors, which vastly dominate directory area overhead, but not the state bits. Every LLC block under NUDA is equipped with its state bits plus other control bits statically. The presence of state bits for *each LLC block avoids otherwise performance degradation* which will be caused by read misses to an LLC block whose directory vector is absent from the buffer.

2.2 Directory Area Overhead Reduction

Earlier methods for lowering cache-coherent directory area overhead generally fall into two classes: directory width reduction and directory height reduction. The first class of methods deals with various sharer representations, such as full bit-map [7], coarse-grained bit-map [8], imprecise bit-map [21], or limited pointers [8], [15], [16], [21]. The second class of methods mainly explores the fact that the amount of directory entries should be no more than the total number of private cache lines [7], [15], and it can be further reduced via directory entries encoded [15] or tracking-avoidance with software modifications [4]. Such further directory area reduction may either lead to imprecise tracking [21] or involve invalidating all cached copies for the victimized block [7], [15]. Meanwhile, the tiny directory [17] employs a small pool of entries to keep the directory elements of selected LLC blocks that experience high shared accesses. While achieving considerable area overhead reduction, such a directory limits its scalability to 512 (since it uses a 64-byte LLC data block to track sharers precisely), and it incurs one extra core-to-core trip for every interrogation on the data block converted for sharer tracking. Large-scale CMPs may avoid full-fledged implementation of cache coherence by domain-aware coherence [6], [12]. However, it adds hardware for on-the-fly logical-to-physical core mappings.

Based on relinquishment coherence and compressed sharer tracking, ReCoST [16] is able to achieve performance close to that of a full-fledged directory based CMP. However, it still keeps *all sharer patterns* non-redundantly in an *on-chip* table. Being hardware-light and complementary to ReCoST, NUDA may work in conjunction with ReCoST to further lift on-chip directory area efficiency.

To our best knowledge, this is the first work that employs DRAM to back up the CMP directory for efficient cache coherence, avoiding excessive directory-induced invalidation and surged network interference.

3 Motivation

NUDA is motivated by the newly discovered two insights: (1) the number of active DEs is small over a short time window, and (2) the fraction of interrogated directory entries shrinks as the number of cores rises. These insights offer a unique opportunity to keep only a very small number of active DEs on chip. By doing so would lower directory storage overhead substantially, promising superior CMP scalability.

To prop our insights, we have collected evaluation results via the gem5-simulated Linux environment, in which various multi-threaded PARSEC [2] and SPLASH [19] benchmarks are executed for a range of core counts. A two-level inclusive cache hierarchy (i.e., L1 and LLC) under the MESI coherence protocol is adopted. CMP cores are organized as meshed-NoC topologies, with each core assigned with one thread. Details of the simulated environment are provided in Section 5.

For easy description, the *coverage* (β) of a directory design is defined as the number of DEs provisioned over the number of aggregate private cache lines. The coverage of $\beta = 1.0$ signifies that each private cache line (be an instruction or data line in our two-level cache hierarchy detailed in Table 2) is provisioned with one DE.

3.1 Locality of DE Accesses

The maximum number of active DEs over a short time window (e.g., 1000 cycles) stays very low for the range of core counts shown in Fig. 1a, where the number of DEs with directory coverage under $\beta = 1.0$ is also shown. The normalized results of active DE counts against that under $\beta = 1.0$ are depicted in Fig. 1b. As an example, in a 2-core system with each core holding a 64 KB private cache, its directory contains $2 \times (64\ \text{KB}/64\text{B}) = 2\text{K}$ entries for $\beta = 1.0$. However, when executing X264, it is found that only 644 entries are interrogated within the given time window (of 1K cycles), resulting in the active DE ratio of 32.2 percent. For a 64-core system, less than one thousand DEs are interrogated, signifying the active DEs to account for only 1.2 percent of the DE count under $\beta = 1.0$. Other benchmarks all exhibit limited percentages of active DEs, as depicted in Fig. 1b. The maximum percentage of active DEs is

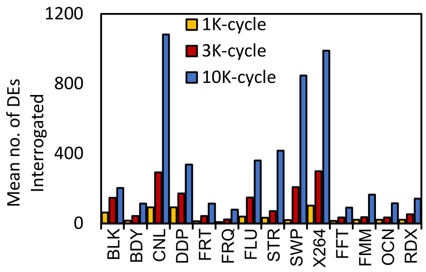


Fig. 2. Mean number of DEs interrogated within different time windows for 64 cores, where benchmarks along the X-axis are denoted by their abbreviations listed in Section 5.2.

below 6.2 percent (under canneal) for a 64-core CMP and is less than 33 percent (under X264) for a 2-core system.

Locality of DE accesses is also observed for longer time windows of 3K cycles and 10K cycles, as demonstrated in Fig. 2, where the results are averaged over multiple runs of each benchmark executed on the conventional 64-core CMP. The numbers creep up when the time window extends for a given benchmark, as expected. Nonetheless, they are far smaller than the total number of private cache lines in the CMP, even for the time window of 10K cycles. It thus suffices to hold on chip, those DEs interrogated for coherence enforcement (called "active DEs") in a short time window, with any other DE fulfilled from the backing store possibly in a few hundred cycles.

Our observed insight of directory locality may result from three facts. First, the multi-threaded runs are for single-tasking so that all concurrent threads of a task share the same data in their working sets within a short time window, with shared variables contained in a few cache blocks, thus involving a few DEs, irrespective of the number of sharers. Second, during typical task execution, a significant portion of data blocks is used privately without multiple sharers [4]. Hence, over an execution time window, most cache blocks are not involved in coherent events. Given a cache block, its directory vector is altered under one of the three coherent events: private cache miss (i.e., the data block is shared by a new core, resulting in its vector adding the new sharer), private cache replacement (i.e., resulting in resetting the corresponding bit in the vector, if the protocol is non-silent), or private cache write (i.e., with all but one bit corresponding to the write-missed core-reset). On the other hand, read/write hits to private blocks in any core involve no DE interrogation. Third, various cache placing/replacing policies [10] and OS locality explorations [14] all aim to improve private cache hit rates, thereby lowering the number of directory interrogation instances.

3.2 Decline Trend

The active DE fraction exhibits a decline trend when the system scales, as shown in Fig. 1b. Although the number of directory entries required to track *all private cache lines* grows linearly with the core count (as well-known previously [7], [15], [18]), the ratio of active DEs, in fact, drops monotonically as the core count rises. The most drastic decrease happens to *x264*, where its ratio drops from 32 percent for 2 cores to 1.6 percent for 64 cores. This is mainly because the aggregate private cache size of a 64-core CMP increases (by a factor of 32) while the number of referenced LLC lines rises much slowly. In a short execution time window, cores tend to share/synchronize a small set of data blocks, concentrating associated interrogations on a limited number of DEs. Additionally, each DV for a larger core count is longer, able to track more sharers and hence, making the ratio in decline.

Another reason is caused by an increase in the mean latency of communication for a larger NoC. The increased latencies prolong the read/write and replacement operations on private caches, lowering the number of data blocks referred within the given time

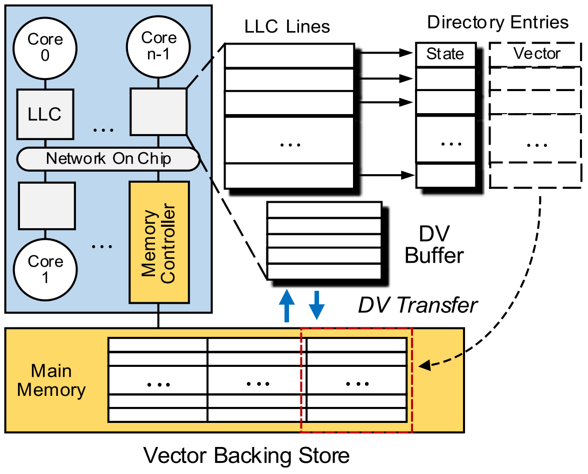


Fig. 3. Proposed NUDA architecture for inclusive cache hierarchy, with the full directory vectors of LLC (shown by the dashed box) kept in the backing store.

window. Further communication latency hikes can result, if the home of a data block happens to reside in a farther LLC tile while its sharers are close to each other [6], [12]. NUDA exploits this trend for chip directory area overhead reduction.

4 Design and Implementation Details

Fig. 3 illustrates our proposed NUDA architecture for a many-core CMP. Every LLC block under NUDA is equipped with its state bits plus other control bits statically. A small on-chip directory vector (DV) buffer is incorporated. A full-fledged backing store is provisioned in memory, with each store entry designated statically for one DV of a corresponding LLC block.

Advantages. In Fig. 3, the DVs which dominate storage overhead are resided in main memory instead of precious on-chip real estate. They will be brought to the on-chip DV buffer dynamically according to the coherence need. Such hierarchical design comes with other advantages. First, precise sharer-tracking can be achieved since a backing store is provisioned in memory to hold all LLC DVs. Second, privately cached data are tracked fully and explicitly. This is in contrast to prior work that requires switching between core pointers and vectors [15], [16], [18], [21] or using a software-managed mechanism [4] that triggers exceptions. Third, each DV is in one whole piece. Although breaking a DV into pieces as prior designs [15], [17] permits directory compression, but it may yield performance degradation which causing by lengthened directory interrogations.

Design Issues. NUDA faces three design issues critical to NUDA performance: (1) locating DEs in the backing store straightforwardly, (2) managing the DV buffer efficiently, and (3) prefetching DVs from the backing store appropriately.

4.1 On-Chip Buffer Management

It is critical to manage the buffer effectively for its scalability. For a given size, the buffer has a hit rate governed by such factors as the replacement policy and set-associativity (i.e., 16 for our current DV buffer design), among others. This work focuses on DV management and investigates a novel DV buffer replacement policy. It aims to keep as many active DVs in the on-chip buffer as possible according to DV criticality.

Fig. 4a depicts the DV buffer allocation policy upon two different state transitions, from an exclusive (E) or modified (M) state to a shared (S) state. Notice that each LLC block is statically associated with its state bits, in order to permit fast protocol reactions (e.g., fulfilling read miss requests from private caches) without interrogating the backing store. Fig. 4b illustrates the DV buffer line format, which involves four fields: tag, ever-written (EW) flag,

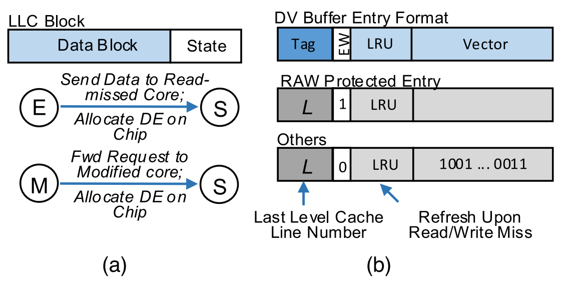


Fig. 4. (a) DV buffer allocation, and (b) proposed DV buffer entry format, where EW and LRU fields are for replacement decisions and the LRU field is refreshed upon read/write misses.

LRU control, and bit-map DV. The buffer entries are 16-way set-associative, with every entry tagged by its corresponding LLC line number (L).

DV Buffer Entry Allocation. Any DV denoting just a single sharer (i.e., in an E or M state) will not be allocated in the buffer. Instead, buffer entry allocation takes place when an LLC block exhibits its state transition from E or M State to S State. This means that any DV of the LLC line that is cached first time by a core is not allocated in the DV buffer. The affected DV in the backing store is written through, in order to track this first privately caching situation. Once a later coherent event from another core happens to the LLC line, its associated DV will be fetched from the backing store and then kept in the on-chip DV buffer, for accelerating subsequent directory interrogations.

This allocation method has two advantages. First, during program execution, those memory blocks cached only for private use [4], [17] require no on-chip DVs for them. Second, if a block is for private use, its DV is *never interrogated* by other cores. Our allocation strategy is preferred over employing encoded core pointers [15] and software-assisted tracking-avoidance approaches [4].

4.2 Criticality-Aware Replacement Policy (CARP)

To take criticality into consideration in DV buffer replacement, NUDA pursues a Criticality-Aware Replacement Policy (CARP) by identifying two categories of critical DVs: sharing-intensive and Read-After-Write (RAW) types. Sharing-intensive DVs are those frequently referred ones, and replacing them is likely to be followed soon by calls for their returns on chip. Hence, this type of DV is desired to be kept in the on-chip buffer. The LRU field, which is refreshed upon every interrogation that caused by a read/write miss, suffices to signify their activeness for them to stay on chip.

A DV that is interrogated due to a write miss will refresh its LRU field as well. In addition, the EW flag is raised to mark this ever-written (EW) history as shown in Fig. 4b, aiming to record the Read-After-Write (RAW) sequence. As the second critical type, RAW criticality is captured by the EW flag to prevent the entry from being evicted undesirably in a predefined time span. For a modified block with its DV not held on chip, the latest block owner can be identified only after its DV is brought from the backing store into the buffer. The read miss is subject to a prolonged latency because of the critical RAW sequence, which could be translated to performance degradation, since the read miss typically is on the critical path. The EW flag is reset periodically to purge out the stale write history, so that its associated DV entry can be replaced later by another critical DV. A suitable reset period for the EW bits is design-specific and is affected by the system size.

Selecting the replaced entry in a DV set starts with checking the EW flag to exclude those vectors which are potentially under critical RAW sequences. The victim is then selected according to the LRU fields of the remaining entries in the set. In an extremely rare situation for the 16-way set in our evaluation, if all entries in the set have their EW flags set, the victim is determined by the LRU field of all entries in the set.

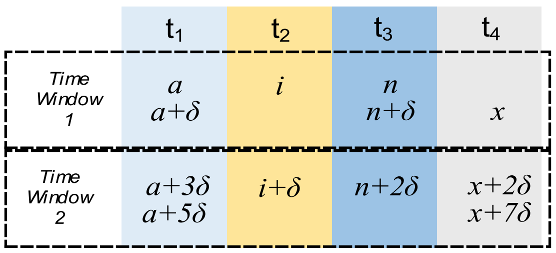


Fig. 5. Example memory block access patterns by 4 threads, with δ denoting the block size (i.e., 64 bytes typically).

The replaced DVs are temporarily stored in the write back buffer in memory controllers aside with other data blocks, and they are written back to backing store when the buffer is flushed. If later DV interrogations hit DVs in the buffer, they are served directly from the memory write buffer. A DV read request is treated as a memory read request, albeit with a different target address at the backing store that is a reserved DRAM region.

4.3 Prefetching for Pre-Activating Vectors (PAVE)

During program execution, active DVs may vary from one time window to the next, calling for prefetching DVs appropriately to boost the on-chip DV hit rate. Our proposed pre-activating vectors (PAVE) module predicts potential DVs to be required for future coherence. When a read/write miss interrogates a missing DV, the DV is fetched from the backing store, followed by prefetching DVs anticipated in future coherence interrogations.

Unlike data prefetching, PAVE must deal with the DV accesses of all threads globally in the CMP, as illustrated in Fig. 5. Thread t_1 accesses to memory blocks a and $a + \delta$ during the first given time window, and then to blocks $a + 3\delta$ and $a + 5\delta$ in the next time window, where δ is the block size. Likewise, Threads t_2, t_3, and t_4 access their respective sequences of memory blocks. Such memory block access sequences widely exist and provide an indication for PAVE to prefetch DVs anticipated by execution of all threads.

PAVE History Table. PAVE faces a unique issue in prefetching DVs from the backing store, since the LLC line number to which a given memory block resides, is not known by the memory controller. With set-associative LLCs, PAVE must know the way number of an LLC set to which the memory block is mapped, so as to prefetch its corresponding DV properly in the backing store through address translation performed in a memory controller. To this end, PAVE employs a hash table to keep the recent LLC line number allocation history.

As depicted in Fig. 6, each memory controller is equipped with one PAVE hash table and a collection of queues to serve as the prefetching buffer. Every entry of the hash table tracks a sequence of blocks existing in the LLC. It includes a tag filed, a block address present-bit vector (Ω-bit) to mark the memory blocks that have

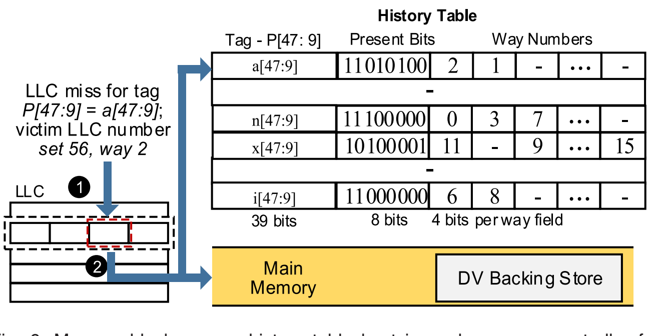


Fig. 6. Memory block access history table kept in each memory controller for PAVE. Each history table entry includes a tag field, an 8-bit presence field, and eight way-fields, where the tag field keeps the first 39 bits of the 48-bit physical address, aligned at (8×64) for the cache block size of 64 bytes.

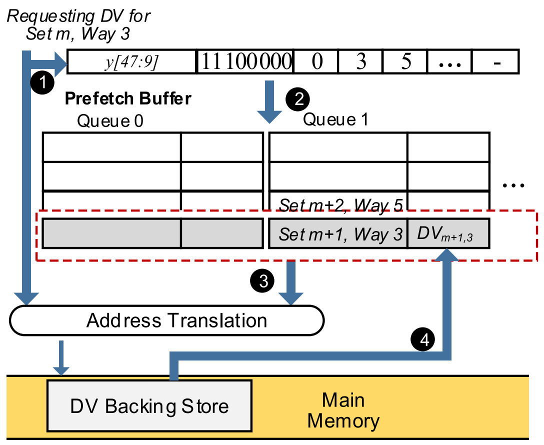


Fig. 7. DV transfer with effective prefetching by a memory controller, realized through the PAVE history table.

been accessed by the memory controller, and Ω way-fields to indicate the way numbers in LLC sets. Each way-field takes $\log_2 \alpha$ bits, if the LLC is α-way set-associative.

History Recording. PAVE conducts history recording whenever a memory block is requested due to an LLC miss, as demonstrated in Fig. 6. Upon receiving the request for a missing block (❶), say Block $\boldsymbol{a}$, the memory controller handles the memory block access and records this access in the PAVE history table (❷), hashed with the key of $P = \boldsymbol{a}[47:9]$. The LLC controller provides the way number (i.e., 2, as stated in ❶, based on the LLC block mapping policy) in the LLC set for Block $\boldsymbol{a}$. Any block access within the eight consecutive memory blocks recorded in the same table entry. Their presences are encoded by the present-bit vector. Likewise, the block access patterns of the other three threads shown in Fig. 5 are recorded respectively by three history table entries in Fig. 6.

The PAVE history table records the recent history of memory block accesses during thread execution. It addresses the unique need of the LLC line number (L) in PAVE, by letting every entry track both the memory blocks fetched and their corresponding way numbers after brought into LLC. There are relatively few entries (say, 128) in the history table, which is the basis to enable PAVE for effective DV prefetching from the backing store.

Vector Pre-Activation. When a DV under interrogation is not in the DV buffer, the buffer initiates an access to the DV from the backing store through a memory controller as marked by ❶ in Fig. 7. Meanwhile, PAVE prefetching is triggered and carried out by the same memory controller. Prefetching is based on the memory block access history recorded in the PAVE hash table at the entry hashed by $P[47:9]$.

Given the DV of LLC line number L (obtained using *Set m and Way 3* shown in Fig. 7) is not on chip, a request is created by a memory controller for fetching the DV from the backing store. Simultaneously, the PAVE history table is checked by hashing $P[47:9]$. If any subsequent block exists in LLC, its way number will be employed to generate a prefetching DV request, as denoted by ❷ in Fig. 7. The DV requests are kept in prefetching queues and all prefetching queues together constitute the prefetch buffer, as depicted in Fig. 7. Each element in the prefetch buffer contains two fields, one for the determined prefetch address of a DV and the other for storing the DV from the backing store.

The requests are placed in the prefetching queues determined by the last q bits of the tag stored in the hit hash table entry, if the prefetch buffer consists of 2^q queues. Each memory controller then issues prefetching to the addresses specified by the head elements of all non-empty queues at a time, as shown by ❸ in Fig. 7. Multiple DV prefetch requests are issued at the same time for supporting multiple threads coherence execution. The prefetching results are stored in their corresponding queue entries, as shown by ❹.

TABLE 1
Directory Configurations Parameters

	DV length	Control field	Major augmented logics

NUDA	64 bits	1-bit EW;
		15-bit LRU;
		22-bit tag
			PAVE assisted directory prefetching in memory controller

Sparse	64 bits	15-bit LRU; 22-bit tag	Invalidation-based precise tracking
SPACE	64 bits	20-bit counter	Pattern table pointer for each LLC line; imprecise tracking

SCD	8 bits	15-bit LRU;
		21-bit Tag;
		4-bit index;
		2-bit type
			2-D root-leave hierarchy; invalidation-based precise tracking

When an interrogated DV is not in the DV buffer, it is checked in the prefetch buffer that the DV might be resided. If the DV is ready in the prefetch queue, it is promoted to the DV buffer in support of future directory interrogation.

It is expected that PAVE delivers better performance for a larger prefetch buffer size. Further, higher performance results if more queues are involved in the prefetch buffer of a given size, because fewer threads are then to share a prefetch queue. The evaluation results of PAVE are given in Section 6.

5 Evaluation Methodology

We evaluate our proposed NUDA under the inclusive cache hierarchy, presenting evaluation results on the storage efficiency of NUDA and on the impacts of CARP and PAVE under various prefetch buffer sizes.

5.1 Directory Organizations

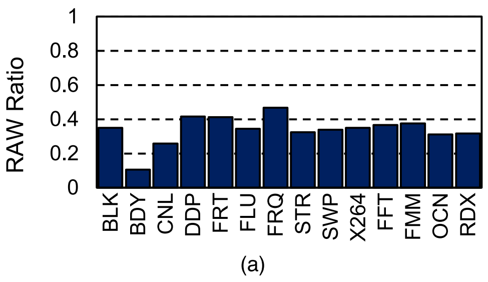
We compare NUDA against representative prior directory area reduction designs, including Sparse [7], SPACE [21], and SCD [15]. We evaluate all the designs across a range of on-chip directory area budget, with their results normalized with respect to that of the conventional full bit-mapped (FBM) directory, in which each LLC block is associated with one DV statically. For meaningful comparison, NUDA (including history tables and prefetch queues) and its prior counterpart are allocated with exactly identical on-chip directory area budget. Area budget signifies the numbers of directory elements for Sparse and SCD, and it dictates the sharer pattern table entry count for SPACE. The configuration of prior counterpart is partially listed in Table 1.

5.2 Simulation Setup and Workloads

We perform full system simulation using the cycle-accurate simulator of gem5 [3] to model a 64-core CMP. Table 2 lists the key system configuration parameters. We select ten applications from the PARSEC benchmark suite [2], i.e., blackscholes (BLK), bodytrack (BDY), canneal (CNL), dedup (DDP), ferret (FRT), fluidanimate

TABLE 2
System Configuration Parameters

Processor	x86-64, 64 out-of-order cores, 3 GHz
Private I/D L1 $	32 KB, 2-way, Pseudo-LRU, 2-cycle latency
Shared L2 $	2 MB per core, 16-way, Pseudo-LRU, inclusive, 16-cycle latency
Cache block size	64 bytes
Network	8×8 2D mesh, VC router, X-Y routing, 2-cycle per hop latency, 32-byte link
Coherence Protocol	Directory-based MESI
Memory	DDR3-1600, 4 GB



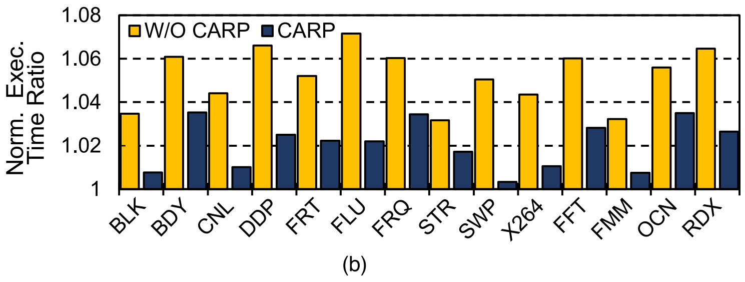


Fig. 8. (a) Mean RAW ratio of on-chip directory. (b) Normalized execution time of NUDA-1/16 for different configurations w.r.t. that of FBM.

(FLU), freqmine (FRQ), streamcluster (STR), swaptions (SWP) and X264, and four applications from the SPLASH benchmark suite [19], i.e., FFT, FMM, ocean_ncp (OCN), and radix (RDX), for multi-threaded execution. Each application spawns 64 threads, with each thread mapped to one core.

6 Results and Discussion

To ease explanation, NUDA with the on-chip directory area coverage of β is denoted by $\mathrm{NUDA} - \beta$. We evaluate NUDA with a range of β from 1 to $1/32$, to examine its performance for various coverage levels. It is found that NUDA-β exhibits no performance degradation for $\beta \leq 1/8$, when compared to the conventional FBM (full bit-mapped) directory. With β shrinks from $1/16$ to $1/32$, NUDA starts to degrade its performance gradually (usually within 3~5 percent slowdowns). Those coverage levels coincide with the fractions of active directory elements, as discussed in Section 3. In this article, we demonstrate NUDA performance results under $\beta = 1/16$ and $1/32$, which signify fairly scarce on-chip directory area budget. The results shown in subsequent figures are normalized with respect to that of FBM, and they confirm that NUDA indeed is subject to negligible performance degradation even under fairly low coverage. We illustrate evaluation results of NUDA under $\beta = 1/16$ subsequently.

6.1 Benefit Potentials of CARP

The mean fraction of directory entries initiating the RAW sequence (which will be marked by the EW bit, see Fig. 4b) is depicted in Fig. 8a. It reveals that on an average of 33 percent of the total directory entries are RAW-critical, to possibly prolong later read misses. Therefore, it is beneficial to keep RAW-critical entries on chip, avoiding detrimental DV replacement. Comparative performance results of NUDA with and without CARP (which follows only the naïve LRU-based replacement policy) for $\beta = 1/16$ are shown in Fig. 8b. They clearly demonstrate the marked gains of CARP by containing performance slowdowns up to 15× (from 5.0 percent down to 0.33 percent for SWP). While its counterpart (without CARP) suffers from moderate performance degradation (5.2 percent on an average), NUDA cuts mean performance degradation down to 2.0 percent). Without CARP, FLU has the highest performance degradation (of 7.1 percent), possibly because it involves frequent synchronization operations [2]. With CARP to track potential RAW

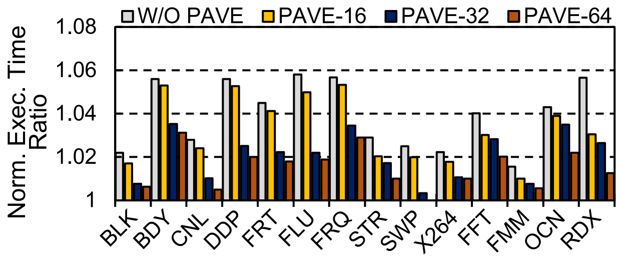


Fig. 9. Execution time ratio with PAVE under various prefetching buffer sizes for NUDA-1/16.

sequences, NUDA lowers the performance degradation of FLU down to 2.0 percent. Note that for $\beta = 1/32$, the performance gain due to CARP is expected to expand. CARP indeed offers substantial performance benefits to NUDA, indispensable for improving directory efficiency. All following results for NUDA are with its CARP enabled.

6.2 Impacts of PAVE

The impacts of PAVE on the overall performance of NUDA are evaluated under various prefetch buffer sizes, with PAVE-x denoting x prefetch entries in every memory controller for 64-core CMPs. The x entries are organized into x/4 queues, with prefetch addresses indicated by queue head elements issued by a memory controller at a time. When the buffer size grows, the number of queues involved rises accordingly so that fewer threads are to share a given queue for higher performance.

The runtime ratio results of NUDA-$1/16$ under various prefetching buffer sizes for NUDA-$1/16$ are demonstrated in Fig. 9. The outcomes when no prefetching exists are also included for comparison. NUDA without PAVE shows noticeable runtime increases (by 3.9 percent on an average). With PAVE-16, NUDA experiences performance improvement, with its runtime ratios drop by various degrees for the benchmarks evaluated (by up to 2.5 percent for RDX). Further, PAVE-32 boosts NUDA performance markedly in many benchmarks. For example, it shrinks the runtime slowdown to 2.2 percent (or 2.0 percent) for FLU (or DDP) from 5.8 percent (or 5.3 percent) when no prefetching exists. Overall, PAVE-32 cuts the runtime by 2.0 percent on an average for all benchmarks examined, when compared to the situation without PAVE. Further expanding the prefetch buffer size to PAVE-64 benefits a few benchmarks modestly (e.g., OCN and RDX by some 1.3 percent) but has little improvement for most benchmarks. As a result, PAVE-32 is adequate for NUDA, and it is chosen as the appropriate prefetch buffer size for our following evaluation.

Execution performance is expected to be correlated tightly with the hit rate to the DV buffer. Even with very scarce on-chip budget, NUDA-$1/16$ exhibits respectable hit rates (that exceed 60 percent) for seven benchmarks even without PAVE, as demonstrated in Fig. 10. The mean hit rate over all benchmarks without PAVE actually stands at 58.4 percent, signifying that directory interrogations indeed concentrate on hot directory entries. This result validates our insight that the small number of active directory entries is usually small within a short time window. The mean hit rate across all benchmarks under PAVE-16 rises to exceed 65 percent, yielding the runtime slowdowns of most benchmarks to shrink noticeably as shown in Fig. 10. PAVE-32 makes the mean hit rate rise beyond 73 percent, as can be found in Fig. 10, so that all but three benchmarks (i.e., BDY, FRQ, OCN) have their runtimes extended by less than 3 percent, as can be found in Fig. 9.

NUDA involves DV transfer between on-chip LLC and the backing store, consuming memory bandwidth. For a quantitative measure, bandwidth utilization for $\beta = 1/16$ under NUDA with different prefetching configurations is plotted in Fig. 12. In general,

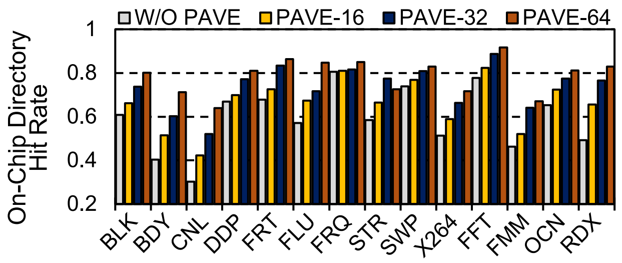


Fig. 10. On-chip directory hit rates under various prefetching buffer sizes for NUDA-1/16.

normalized memory bandwidth utilization increases as the number of prefetch requests grows (i.e., for more prefetch buffer queues). Without PAVE, NUDA contains its average extra bandwidth utilization within 1.3 percent, with the largest utilization rise by up to 6.7 percent (for BDY), since BDY possibly having the largest active DE fraction to involve the biggest memory bandwidth overhead. Several other sharing-intensive benchmarks, such as CNL, FLU, FRQ, and SWP, increase bandwidth consumption by 1.2 to 4.5 percent. The sharing degrees of CNL, FLU, and SWP are relatively higher as stated earlier [2] and so do their directory activity as shown in Fig. 1b.

It can be seen in Fig. 11 that PAVE-32 tends to have markedly lower memory bandwidth utilization than PAVE-64 for four of the benchmarks (i.e., BDY, CNL, DDP, and FRT). Meanwhile, those four benchmarks under PAVE-64 have insignificant reduction in their normalized runtimes when compared with those under PAVE-32, as demonstrated in Fig. 9. This indicates that PAVE-32 is more cost-effective, because aggressive prefetching can hike bandwidth utilization (up to 14.0 percent for BDY) without noticeably lowering runtimes. Subsequent NUDA evaluation results for comparing with those of prior directory storage reduction techniques are gathered under the prefetch buffer sized at PAVE-32.

6.3 Comparative Outcomes

The prior designs of SCD [15], SPACE [21], and Sparse [7] are found to exhibit limited performance degradation when the on-chip directory area is richly provisioned, i.e., with the coverage level of $\beta \geq 1/2$, under which NUDA presents negligible performance degradation. As the coverage level drops for a large core count, however, prior designs suffer from marked performance degradation while NUDA still retains performance. Our comparison here focuses on very scarce directory area budget (under $\beta = 1/16$ or even $\beta = 1/32$) for high CMP scalability.

The normalized performance outcomes of Sparse, SCD, SPACE, and NUDA for $\beta = 1/16$ are illustrated in Fig. 12, where on-chip directory area budget for NUDA is kept exactly identical to those for Sparse, SCD, and SPACE. In addition, the outcomes of NUDA under $\beta = 1/32$ are also included for comparison. It is observed that NUDA incurs negligible (or limited) performance degradation for a (or an extremely) scarce on-chip directory area of $\beta = 1/16$ (or $1/32$), as compared to the conventional FBM design, which provides each LLC block with a DV statically to track all privately cached contents precisely on chip. While achieving the best

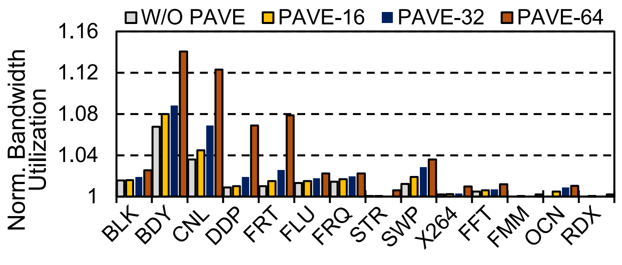


Fig. 11. Normalized memory bandwidth utilization for NUDA-1/16.

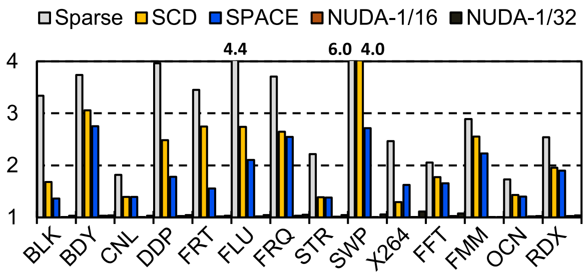


Fig. 12. Normalized execution times of NUDA and its earlier counterparts with respect to those of FBM, with the coverage of Sparse, SCD, and SPACE all equal to 1/16.

execution performance possible, FBM exceedingly over-provisions its on-chip directory and hence limits its scalability. In sharp contrast, NUDA-1/16 and NUDA-1/32 budget only 6.7 and 3.3 percent as much directory area, respectively, and yet exhibit just 2.1 and 5.9 percent execution slowdowns, on an average, when compared with its FBM counterpart.

NUDA is seen in Fig. 12 to contain execution slowdowns far better than Sparse, SPACE, and SCD for all benchmarks examined, under $\beta = 1/16$. For example, Sparse, SPACE, and SCD then degrade mean execution performance by $3.2\times$, $2.2\times$, and $1.9\times$, respectively, as opposed to only 2.1 percent for NUDA-1/16. Hence, NUDA lowers the mean execution slowdowns by $3.7\times$ (or $1.9\times$) in comparison to SCD (or SPACE) under the scarce directory area of $\beta = 1/16$.

NUDA has the largest gains in slowdown restraints for swaptions, with NUDA-1/16 yielding the gains of $5.9\times$, $4.0\times$, and $2.7\times$ when compared respectively with Sparse, SPACE, and SCD. The reasons are two-fold. First, swaptions often involves repetitive accesses to the privately-cached "coherence-active" data blocks. Under Sparse and SCD, the directory elements of those active LLC blocks may be evicted prematurely, to cause substantial execution slowdowns. Second, threads of swaptions frequently compete for shared lock variables to enter the critical section but the directory elements of those lock variables (without CARP to track their RAW sequences) can be evicted prematurely, to extend the execution times significantly. On the other hand, NUDA always keeps DVs of such synchronization-related data blocks on chip to achieve fast execution.

7 Conclusion

We have investigated the non-uniform directory architecture (NUDA) framework, aiming at CMP scalability improvement by drastically reducing on-chip directory area ratio. NUDA employs a multi-level directory configuration in CMP, equipping a small on-chip buffer to keep only "active" directory vectors (DVs) for sustained execution performance. It employs a full-fledged backing store in a low storage level to hold all LLC's DVs and relies on novel CARP (criticality-aware replacement policy) and PAVE (pre-activating vectors) to effectively manage DV transfer between the DV buffer and the backing store. Evaluation results reveal that NUDA experiences negligible performance degradation under a scarcely provisioned on-chip directory area, when compared with the one that provisions every LLC with an on-chip DV statically for best possible performance. NUDA is also shown to outperform earlier compression-based techniques for on-chip directory area reduction, especially if directory area budget is held very low for high scalability.

Acknowledgment

The authors wish to thank Xiyue Xiang for his constructive discussion. This work was supported in part by National Science Foundation with Award Numbers: CCF-1423302 and CNS-1527051.

REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, “A two-level directory architecture for highly scalable cc-NUMA multiprocessors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 1, pp. 67–79, Jan. 2005.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2008, pp. 72–81.

[3] N. Binkert, et al., “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, 2011.

[4] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. F. Duato, “Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks,” in *Proc. 38th Int. Symp. Comput. Archit.*, Jun. 2011, pp. 93–104.

[5] N. Eisley, L.-S. Peh, and L. Shang, “In-network cache coherence,” in *Proc. 39th IEEE/ACM Int. Symp. Microarchit.*, Dec. 2006, pp. 321–332.

[6] Y. Fu, T. M. Nguyen, and D. Wentzlaff, “Coherence domain restriction on large scale systems,” in *Proc. 48th ACM/IEEE Int. Symp. Microarchit.*, Dec. 2015, pp. 686–698.

[7] A. Gupta, et al., “Reducing memory and traffic requirements for scalable directory-based cache coherence schemes,” in *Proc. Int. Conf. Parallel Process.*, 1990, pp. 312–321.

[8] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA highly scalable server,” in *Proc. 24th Annu. Int. Symp. Comput. Archit.*, Jun. 1997, pp. 241–251.

[9] D. Lenoski, et al., “The Stanford DASH multiprocessor,” *IEEE Comput.*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[10] A. Jain and C. Lin, “Back to the future: Leveraging Belady’s algorithm for improved cache replacement,” in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 78–89.

[11] J. Kuskin, et al., “The Stanford FLASH multiprocessor,” in *Proc. Int. Symp. Comput. Archit.*, 1994, pp. 302–313.

[12] M. Marty and M. Hill, “Virtual hierarchies,” *IEEE Micro*, Vol. 28, no. 1, pp. 99–109, Jan. 2008.

[13] M. Michael and A. K. Nanda, “Design and performance of directory caches for scalable shared memory multiprocessors,” in *Proc. IEEE 5th Int. Symp. High Performance Comput. Archit.*, Jan. 1999, pp. 142–151.

[14] J. Philbin, et al., “Thread scheduling for cache locality,” in *Proc. 7th Int. Conf. Archit. Support Programming Languages Operating Syst.*, Oct. 1996, pp. 60–71.

[15] D. Sanchez and C. Kozyrakis, “SCD: A scalable coherence directory with flexible sharer set encoding,” in *Proc. IEEE 18th Int. Symp. High Performance Comput. Archit.*, Feb. 2012, pp. 1–12.

[16] W. Shu and N. Tzeng, “Superior directory efficiency via relinquishment coherence and compressed sharer tracking for chip multiprocessors,” *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1975–1981, Nov. 2017.

[17] S. Shukla and M. Chaudhuri, “Tiny directory, efficient shared memory in many-core systems with ultra-low-overhead coherence tracking,” in *Proc. IEEE 23th Int. Symp. High Performance Comput. Archit.*, Feb. 2017, pp. 205–216.

[18] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. San Rafael, CA, USA: Morgan and Claypool, 2011, pp. 161–163.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *Proc. 22nd Int. Symp. Comput. Archit.*, Jun. 1995, pp. 24–36.

[20] G. Zhang, W. Horn, and D. Sanchez, “Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems,” in *Proc. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2015, pp. 13–25.

[21] H. Zhao, et al., “SPACE: Sharing pattern-based directory coherence for multicore scalability,” in *Proc. 19th Int. Conf. Parallel Archit. Compilation Tech.*, Oct. 2010, pp. 135–146.