Pool Directory: Efficient Coherence Tracking with Dynamic Directory Allocation in Many-core Systems

Sudhanshu Shukla and Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur 208016, INDIA
{sudhan, mainakc}@cse.iitk.ac.in

Abstract— **The coherence directory in a chip-multiprocessor keeps track of each memory block inside the cache hierarchy and plays a significant role in offering a scalable shared memory abstraction in many-core systems. Multi-threaded applications typically require two types of directory entries, namely, limited pointer entries tracking a few sharers of a block and bitvector entries tracking larger number of sharers for widely shared blocks. Recent proposals aiming to optimize the average number of bits per directory entry have organized the directory as either a static mix of these two types of entries or a collection of relatively short bitvector entries that can encode either a limited number of sharer pointers or a larger number of sharers hierarchically. In this paper, we present a directory organization that facilitates allocation of two different types of directory entries dynamically. Our design maintains a pool of limited pointer entries, where each entry can also double as a segment directory entry encoding the sharers in a cluster of cores. Each tag in the primary sparse directory array has a pointer that can either represent a sharer or point to an entry in the pool. When multiple segment directory entries are needed to encode all the sharers of a block, our pool management protocol guarantees that all these entries are allocated contiguously so that maintaining a pointer to the head entry is enough. Such a design offers significant flexibility in sharer encoding and allows us to independently size the sparse directory array and the pool. Detailed simulation results show that our proposal incorporated in a 128-core system running multi-threaded applications drawn from scientific, general-purpose, and commercial computing domains can offer, on average, 5% improvement in performance and 20% savings in interconnect traffic compared to the state-of-the-art scalable coherence directory (SCD) proposal when using a $\frac{1}{16}\times$ sparse directory.**

Keywords*—*Many-core coherence, dynamic directory allocation, directory scalability.

I. Introduction

Cache coherence protocols are central to the correctness of shared memory abstractions in distributed parallel environments. An important storage structure used by these protocols is the coherence directory, which is responsible for keeping track of the current locations of the memory blocks in the cache hierarchy. In a single-chip many-core system, the coherence directory maintains information about the blocks resident in the private cache hierarchy of each processing core. As the on-chip cores grow in number, the design of the coherence directory needs to be space-efficient so that these systems can continue to support the shared memory abstraction with acceptable directory storage budget [25].

The sparse directory organization [17], [28] has become popular due to its simplicity and space-efficiency. The sparse directory organizes the coherence tracking information in the form of a cache, which can track only a limited number of blocks at a time. For example, in a three-level cache hierarchy with the first two levels being private, a sparse directory could be sized to track a fraction of the blocks that the last level (L2) of the private cache hierarchy across all the cores can accommodate. If the last level of the private caches aggregated over all the cores can accommodate N blocks, a $\frac{1}{16}\times$ sparse directory would track at most $N/16$ unique blocks at a time. A replacement from the sparse directory invalidates or retrieves (if dirty) the block the replaced directory entry corresponds to from all the private caches having a copy of the block. While the sparse directory provides an attractive starting point for optimizing the number of directory entries, the width of a sparse directory entry, which is typically a bitvector, still needs to scale linearly with the core count. In this paper, we focus on the problem of optimizing the average number of bits devoted to a sparse directory entry.

The attempts to optimize the width of a sparse directory entry exploit the observation that at a given point in time, not all blocks need a full-map bitvector. The degree of sharing varies across blocks [38] and over time within the same application. The left panel of Figure 1 shows the percentage of the allocated directory entries that experience a maximum of k sharers where k falls in four possible sharer count bins: 2 to 4, 5 to 8, 9 to 16, and 17 to 128 (end-points inclusive). These data are collected on a 128-core system for fourteen multi-threaded applications spanning the PARSEC suite [5], the SPLASH-2 suite [39], SPEC JBB, SPEC Web running on the Apache server, TPC running on the MySQL server, and the SPEC JVM suite. For these measurements we use a 2x sparse directory so that premature directory evictions do not hamper the amount of sharing. These data show that, on average, only 10% directory entries observe any sharing, while the rest of the allocated directory entries track only private blocks. Most of the sharing instances are limited to at most four sharers, while only two applications (swaptions and barnes) show noticeable number of directory entries experiencing more than sixteen sharers. These data indicate that, on average, a large number of bits in a full-map directory entry are wasted due to lack of high volume of sharing. This is further confirmed in the right panel of Figure 1 which shows the average percentage of set bits in an allocated directory entry. Across the fourteen applications, on average, only 2.4% bits in a full-map directory entry are set. These data clearly indicate the importance of optimizing the directory entry width.

The proposals aiming to optimize the sparse directory width while preserving the preciseness of sharing information and not resorting to broadcast, overflow-induced early invalidations, or software solutions organize the directory in one of the following three forms: 1) a statically designed mix of different types of entries [14], 2) a hierarchical organization of the sharing bitvector by decomposing the system into a hierarchy of clusters [31], and 3) dynamic allocation of tracking information [32], [33], [40]. Based on the observation that the private blocks are often more in number than the shared blocks, it has been proposed that the sparse directory sets be designed to have a static mix of pointer and bitvector ways [14]. The pointer ways track private blocks and have width that grows logarithmically with core count. The bitvector ways track shared blocks and their width grows linearly with

978-1-4673-7166-7/15/$31.00

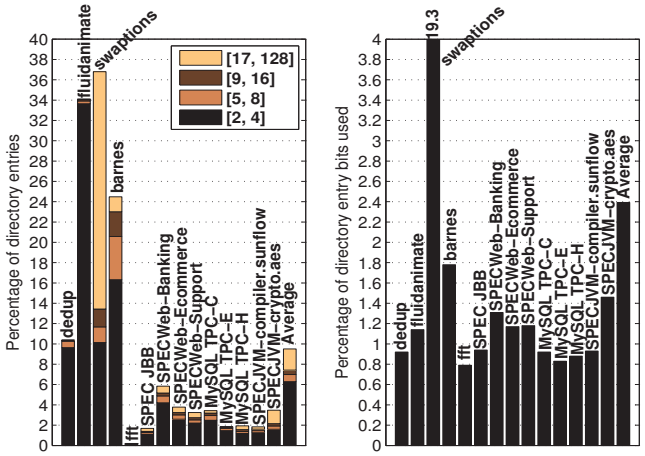

Fig. 1. Left panel: Distribution of maximum sharer count averaged over all allocated directory entries in a 2x sparse directory. Right panel: Percentage of directory entry bits set averaged over all allocated directory entries in a 2x full-map sparse directory.

core count. Due to static partitioning of each set into two types of entries (e.g., six pointer ways and two bitvector ways in an eight-way set-associative sparse directory), such a design cannot react optimally to the changing sharing degree. We will refer to this design as the Hybrid directory organization.

The state-of-the-art hierarchical organization, named scalable coherence directory (SCD) [31], represents the core count as a product of two integers (ideally equal) and the larger integer decides the width of the directory [31]. This organization treats a system with pq cores as p clusters of q cores each, where q is bigger than or equal to p. Each directory entry is q bits wide. Multiple directory entries are used to encode the sharers in a hierarchical fashion. Each bit in the top-level or root entry represents a cluster and in the worst case $p + 1$ directory entries would be needed to encode all the sharers. The non-root entries will be referred to as the leaf entries. In any case, at least two directory entries are required and the best case arises when all the sharers are confined to a single cluster. Additionally, each directory entry can encode a certain number of sharer pointers ($\lfloor q/\log_2(pq) \rfloor$) and the coherence tracking format of a block switches to the hierarchical one only after the number of sharers of the block exceeds this limited pointer count that each directory entry can accommodate. While this proposal is able to achieve a square root factor reduction in the directory width, multiple directory entries have to be invested to encode the sharers of a block. This increases the pressure on the directory as the degree of sharing and the number of shared blocks increase. Since each involved directory entry carries the tag of the shared block, the tracking overhead per shared block increases significantly in the worst case. For example, in a 1024-core system having a 32-bit wide hierarchical directory, 33 directory entries would be needed in the worst case to encode the sharers of a block. If we assume a 24-bit tag, the worst case tracking overhead per shared block is 33×(24+32) bits, which far exceeds the tracking overhead for a full-map bitvector (1024+24 bits). The relative wastage increases in smaller core-count systems. The observed level of inefficiency, however, depends on the actual sharing pattern of the application. An additional inefficiency arises due to a large volume of private blocks. Each of these blocks needs just one limited pointer and wastes the remaining pointers in a directory entry.

Dynamic allocation of tracking information has been proposed in two different forms. The dynamic pointer allocation scheme assigns a new sharer pointer entry to a block as and when a new sharer is added [32], [33]. Searching the list of sharers for a specific sharer or walking the sharer list for sending out invalidations can be costly. A recent proposal carries out dynamic assignment of a full-map bitevector to a sparse directory tag whenever the block corresponding to the tag gets shared by at least two cores [40]. For tracking private blocks, the proposal uses a single owner pointer attached to the tag. While this proposal is able to eliminate the directory bit wastage for tracking private blocks, the space inefficiency is still significant for tracking shared blocks. This design will be referred to as the Select directory conveying the fact that only a selected subset of the sparse directory tags gets dynamically attached to full-map bitvectors as the need arises.

In this paper, we introduce Pool directory, a novel sparse directory organization that aims at optimizing the average number of bits per directory entry. Our proposal attaches a pointer with each sparse directory entry and maintains a separate direct-mapped pool of short vector entries. Each pool entry can store either a few sharer pointers or a sharer vector encoding the sharers in a cluster of cores. To encode all the sharers of a block, multiple pool entries may be needed and these are allocated dynamically as and when the need arises. Our pool management protocol ensures that all these pool entries remain contiguous so that it is sufficient to maintain a pointer to the head entry. The pointer in each sparse directory entry can either encode a sharer (particularly useful for tracking private blocks) or point to a head pool entry. Our proposal enjoys four distinct advantages compared to the state-of-the-art hierarchical representation. First, the private blocks never contend for pool entries and can be encoded in the sparse directory array. Second, exactly one tag is allocated for tracking a block. Each individual pool entry does not need a separate tag. Third, there is no need to maintain a root entry in the pool. At most p pool entries are needed to encode p clusters of q cores each. Fourth, the sharers of a block can be encoded by optimally mixing the limited pointer and the sharer vector representations in the pool entries allocated to the block, thereby offering significant flexibility in encoding the sharers in a space-efficient manner. The directory storage is dynamically allocated to track sharers of a block as and when needed by allocating additional pool entries. Additionally, the decoupled organization of the directory allows us to independently size the number of entries in the sparse directory table and the pool table. We present the detailed design of the Pool directory in Section III. Our simulation results (Sections IV and V) on a 128-core system show that our proposal outperforms the state-of-the-art dynamic hierarchical scalable coherence directory (SCD) [31] by 5% while reducing the interconnect traffic by 20% when using a $\frac{1}{16}\times$ sparse directory. Additionally, our proposal performs within 2.4% of a full-map organization while requiring only one-third of the directory storage of a full-map organization.

II. Related Work

A large body of research on coherence directory store optimization has followed the first proposal that introduced a bitvector as the directory element [6]. The early proposals focused on the distributed shared memory multiprocessor architectures. The proposals for optimizing the width of the directory include limited pointer schemes with broadcast on overflow (Dir_iB), limited pointer schemes with invalidation on overflow (Dir_iNB), limited pointer schemes with software handling on overflow (LimitLESS directory), coarse-vector schemes where each bit in a vector tracks a cluster of sharers requiring a broadcast invalidation within a sharing cluster on a write, and use of gray codes for achieving better compression in the bitvector [3], [7], [17], [26]. The scalable coherent interface standard forms a doubly linked list of sharers with the help of pointers attached to each private cache block, while the directory maintains a pointer to the head of the list [19]. Although such a distributed linked list scheme is more scalable than a bitvector scheme in terms of storage, the protocol operations are significantly more complex than a bitvector protocol. Some of the proposed limited pointer schemes use a distributed linked list or a distributed tree to track the overflown sharers with one of the pointers in the directory entry serving as the head of the list or the root of the tree [8], [9]. A number of proposals organize the directory in a hierarchical tree or multi-level clusters [16], [24], [27], [36]. Although the directory organizations in these proposals achieve low overhead, the hierarchical coherence

protocol makes the overall design complex. Tree-based compression of tracking information and a two-level directory architecture where the second-level directory maintains imprecise compressed information have also been proposed [1], [2]. The segment directory design partitions the system into a few clusters and tracks the sharers in a limited number of clusters [11]. On an overflow, one of the well-known overflow solutions is invoked.

More recent proposals have focused on directory space optimization for chip-multiprocessors. We have already discussed few such schemes in the last section. One of these mixes two types of directory entries, namely, pointers and bitvectors to design a sparse directory set [14] (will be referred to as the Hybrid directory scheme). The other proposal designs a scalable coherence directory (SCD) by representing sharers in two-level hierarchical bitvectors [31]. A directory architecture that eliminates the duplicate tag overhead of the directory by maintaining an array of Bloom filters for answering set membership queries about the sharers has been proposed [42]. This design suffers from scalability issues, since each directory access involves looking up C Bloom filters, C being the number of cores. Proposals that track a small set of sharing patterns and link each active directory entry to a sharing pattern have been proposed [44], [45]. Limited pointer representations that use one pointer for counting the sharers on overflow have been explored [21]. This count is later used to limit the number of acknowledgment messages needed on a broadcast invalidation. A recent proposal has used multi-level memristors to compress the size of the directory entries [43]. In contrast to these, our proposal presents a design for dynamically allocating directory entries while leaving the cache coherence protocol unchanged.

Although we focus on optimizing the average number of bits per directory entry in this study, the number of entries in the sparse directory is an important design parameter and has a significant impact on the life time of a block in the private cache hierarchy. A small number of entries leads to premature invalidations due to directory eviction. Designs exploring smart hash functions and skew-associative organizations for the sparse directory have been proposed [15], [31]. Designs that store the evicted directory entries in a memory-resident hash table and delay invalidations have also been explored [20]. Page-grain classification between private and shared data has been used to exclude private blocks from coherence tracking, thereby effectively increasing the number of available directory entries for tracking shared data [12]. A recently proposed design does not invalidate private blocks on directory eviction, but resorts to broadcast when such a block gets shared after the tracking entry of the block is evicted from the sparse directory [13]. Recent proposals employing coarse-grain coherence tracking for privately cached regions can further reduce the required number of directory entries [4], [14], [41].

Compiler analysis and certain software guarantees have been exploited to reduce the required directory storage. Compiler-generated hints about private data have been used to optimize directory allocation [23]. Data-race-free software, disciplined parallel programming models, and self-invalidation of shared data at synchronization boundaries have been used to significantly reduce the coherence directory size or completely eliminate the coherence directory [10], [29], [34].

III. Pool Directory

We discuss the detailed design of the Pool directory in the following. Section III-A presents the architecture of the Pool directory and Section III-B discusses the implementation of the various operations supported by the Pool directory. In this discussion, we assume a traditional three-hop MESI cache coherence protocol [22].

A. Directory Organization

The Pool directory architecture consists of two structures, namely, the sparse directory and a pool of short sharer vectors, as shown in Figure 2. The sparse directory is a set-associative array with each entry having a tag and tracking/state information of the block corresponding to the tag. The major part of the tracking information is taken up by a pointer (P), which can either encode a sharer or point to an entry in the sharer vector pool. If the number of cores in the system is C and the number of entries in the sharer vector pool is N, each pointer needs to be $\lceil \log_2(\max(C, N)) \rceil$ bits wide. The single sharer (S) bit in a sparse directory entry is set when the corresponding block has a single sharer (i.e. private), which is directly encoded in the pointer P.

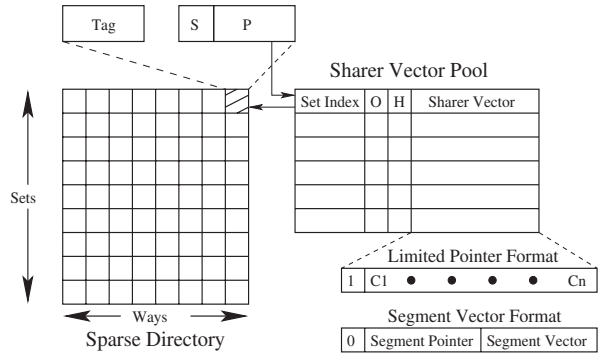

Fig. 2. General structure of the Pool directory.

The sharer vector pool is a tagless direct-mapped array with each entry having a sharer vector. The sparse directory entries for blocks having more than one sharer utilize a collection of consecutive pool entries to encode the sharers. Each such collection of pool entries can be logically seen as a linked list with a head pool entry and a tail pool entry. A sparse directory entry that needs pool entries has its single sharer (S) bit reset and the pointer (P) points to the head entry of the collection of pool entries being used. Each pool entry has a head (H) bit indicating if the entry is a head entry for some sparse directory entry. Each pool entry also contains an occupied (O) bit and the set index of the sparse directory entry it is associated with. The occupied bit is set for a pool entry in use. We will discuss the utility of the occupied bit, the head bit, and the set index field in Section III-B. We note that the private blocks do not need any pool entry.

Each sharer vector can encode sharers, either in *limited pointer* format or in *segment vector* format. As shown in Figure 2, the first bit of each sharer vector identifies the encoding format being used by the vector. When the first bit is set, the sharer vector is encoded using the limited pointer format and when the first bit is reset, the sharer vector is encoded using the segment vector format. The limited pointer format is useful for efficiently encoding a small number of sharers. In the limited pointer format, the sharer vector is organized as an array of pointers and their valid bits. Each pointer can independently point to a sharer of the corresponding sparse directory entry. Each such pointer needs $\lceil \log_2(C) \rceil + 1$ bits, where C is the number of cores. The segment vector format was introduced in [11] for efficiently encoding a cluster of sharers in a segment directory entry. In the segment vector format, a sharer vector consists of a segment vector and a segment pointer. The segment vector is a K-bit wide segment of a full-map vector. The segment pointer is a $\lceil \log_2(C/K) \rceil$-bit field maintaining the position of the segment vector within the full-map vector effectively recording the id of the cluster the segment represents. The limited pointer format and the segment vector format permit each sharer vector to independently encode sharers. This allows a sparse directory entry to simultaneously utilize both the encoding formats to achieve greater storage efficiency. In the worst case, $\lceil C/K \rceil$ pool entries would be required for encoding a full-map vector.

Figure 3 illustrates different sharer vector encoding formats for pool entries with twenty-bit wide sharer vector in a 128-core system. Each sharer vector can encode a maximum of two sharers in the limited pointer format, as a valid bit and a seven-bit wide pointer are required

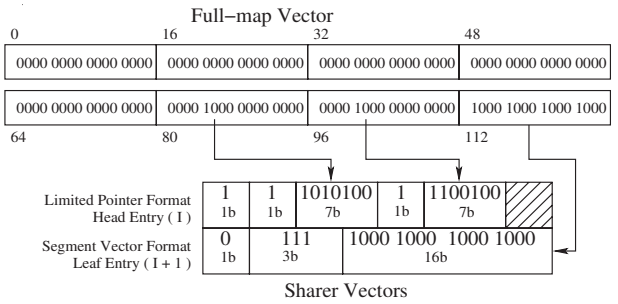

Fig. 3. Pool entries with twenty-bit wide sharer vector in a 128-core system.

for encoding a sharer. When using the segment vector format, sixteen sharers can be encoded in a sixteen-bit wide segment vector. A three-bit wide segment pointer is maintained for encoding eight distinct sixteen-bit wide segments of the full-map vector. In the worst case, eight pool entries would be required to encode a full-map vector. In Figure 3, a sparse directory entry for a block is using two consecutive pool entries I and $I + 1$ for encoding six sharers of the block. The head entry (I) is using the limited pointer format for encoding two sharers from the 5^{th} and the 6^{th} segments of the full-map vector. The tail entry ($I + 1$) is using the segment vector format for encoding all the sharers from the 7^{th} segment of the full-map vector.

B. Directory Operations

The coherence directory is looked up in parallel with every L3 cache access. Depending on the outcome of this lookup, different operations may have to be invoked on the Pool directory. In the following, we discuss four important operations.

1) Adding a Sharer: On a sparse directory miss, the replacement process allocates a sparse directory entry for the requested block and encodes the new sharer directly in the pointer P. At this point, the single sharer (S) bit in the sparse directory entry is also set. When a second sharer needs to be added, the sparse directory entry allocates a pool entry, and encodes the two sharers using the limited pointer format in the allocated pool entry. The single sharer (S) bit is cleared in the sparse directory entry and the pointer P is updated to point to the allocated pool entry. In a general situation, when a new sharer needs to be added to a block B which has already been allocated a number of pool entries, the sharer addition logic explores four possible avenues, as discussed below. Suppose the new sharer falls in segment n of the full-map vector. Let the collection of pool entries already allocated to B be Q. Note that the pool entries in Q are all contiguous. First, the collection Q is looked up to find out if there is an entry already allocated for segment n. If such a pool entry exists, the new sharer is encoded in it. If no such entry exists, the collection Q is looked up to locate a pool entry that is currently using the limited pointer format and has a free pointer. If such a pool entry exists, the new sharer is encoded in the free pointer. When no such free pointer is available, it may be possible to add the new sharer by changing the encoding format of a pool entry. Particularly, when all the sharers encoded in a pool entry using the limited pointer format belong to the same segment of the full-map vector, the encoding of the pool entry is changed to the segment vector format creating new opportunities for finding a pool entry to encode the new sharer. When all these three avenues fail, a new pool entry must be added to the collection Q.

The collection Q can be extended by prepending or appending it with a new pool entry. Let the entry just before the collection Q begins be Q_- and the entry just after the collection Q ends be Q_+. If at least one of Q_- and Q_+ is free, the free entry is added to the collection Q. When both of these entries are occupied, one of them is evicted. The victim is chosen as follows. Suppose each segment vector is K-bit wide. Therefore, in the worst case, a block would need $\lceil C/K \rceil$ entries to encode all its sharers, where C is the number of cores. We divide the pool into equally-sized chunks such that each chunk has $\lceil C/K \rceil$ consecutive entries. If Q_+ and Q_- belong to the same chunk (as Q), we victimize one of them at random. If Q_+ and Q_- belong to two different (adjacent) chunks, we victimize the one that belongs to the chunk which has the larger share of Q. The rationale for this victimization policy is that we let Q grow within the chunk which already has a bigger share of Q so that the interference in the adjacent chunk due to this growth is minimized. When Q_- is used for extending Q, the head (H) bit for the current head entry in Q is cleared and Q_- becomes the new head of the extended collection. The pointer P in the sparse directory entry is also updated to point to Q_-.

In all cases, the sharer addition logic first looks up the sparse directory. On a tag hit, if the single sharer (S) bit in the sparse directory entry is not set, the occupied (O) and the head (H) bits of consecutive $\lceil C/K \rceil$ pool entries are examined starting from the pool entry pointed to by the pointer P. The O and H bit arrays are physically kept separately from the pool array. This examination reveals the number of pool entries that must be read out. The maximum number of pool entries that a block can use is $\lceil C/K \rceil$. We provision the pool with two read ports so that the required number of access rounds is bounded by $\lceil C/K \rceil/2$. For the pool configuration modeled in this study, we verify using CACTI [18] that this latency is comfortably hidden under the last-level cache access latency for 22 nm nodes. This latency can be further improved by internally banking the pool and ensuring that not all pool entries of a block are allocated in the same bank. The accessed pool entries are examined and the new sharer is added according to the aforementioned protocol. The longest critical path, encountered infrequently, involves reading of the sparse directory, examining the O and H bit arrays, reading of pool entries, examining the pool entries, extending the collection of pool entries, and updating the new pool entry. For the workloads considered in this study, on average, 98.6% of the allocated sparse directory entries need at most two pool entries and 99% need at most three pool entries.

2) Removing a Sharer: The coherence protocol modeled by our system generates replacement hints to the directory when a core evicts a shared block. This is done to exclude the already evicted blocks from tracking information so that the information is up-to-date and exact. In addition, dirty evictions always generate writebacks to the L3 cache. In both these cases, the sharer must be removed from the directory. This operation requires looking up the sparse directory followed by pool entry accesses, if needed, and removal of the sharer leading to the possible release of a pool entry. When the number of sharers for a sparse directory entry reduces to one, the sparse directory entry frees its pool entries. The single sharer is encoded directly in the pointer P of the sparse directory entry and the single sharer (S) bit is set. The port requirements are similar to the sharer addition operation.

3) Allocation of the First Pool Entry: A sparse directory entry needs to allocate its first pool entry when the number of sharers of the block that the entry is tracking becomes two. This allocation is treated differently from growing an already allocated collection of pool entries because appropriate positioning of the first allocated pool entry is important so that the future growth of the collection does not lead to pathological conflicts and pool entry evictions. By examining the vector of occupied (O) bits of the pool entries, a free pool entry can be identified. To make the implementation efficient, the occupied bitvector is segmented and one occupied vector is maintained for a segment of $\lceil C/K \rceil$ consecutive pool entries. Each occupied bitvector is also accompanied by a $\lceil \log_2(C/K) \rceil$-bit wide population counter which tracks the number of bits set in the occupied bitvector. The population counter is incremented when an entry in the segment is occupied and decremented when an entry in the segment is freed. Finally, a free entry index is maintained for each segment of pool entries, which points to the next free entry in the segment. By examining the occupied bitvector, the free entry index can be updated. For our configurations, the pool size never exceeds 64 entries per last-level cache bank and therefore,

we need at most eight such occupied bitvectors, population counters, and free indices.

To more or less evenly distribute the density of occupied entries over the entire pool, a round-robin policy is utilized for selecting the segment with a free pool entry. An index of the segment from which the last pool entry was allocated is maintained to assist in the selection of a free pool entry. When a free pool entry cannot be obtained, a tail entry is randomly selected for eviction from the round-robin segment. The protocol for evicting a pool entry is discussed next.

4) Pool Entry Eviction: For evicting a pool entry, the sparse directory entry associated with it needs to be located. Once the sparse directory entry has been located, the sharers encoded in the evicted pool entry are back-invalidated. The occupied (O) and the head (H) bits of the pool entry are cleared. When the sparse directory tag is occupying a single pool entry (the pool entry being evicted), all but one of the sharers encoded in the pool entry are back-invalidated. The single sharer that is not back-invalidated is encoded directly in the pointer P of the sparse directory entry and the single sharer (S) bit in the sparse directory entry is set.

To locate the sparse directory entry associated with the evicted pool entry, the set index stored in the pool entry is used to read out the entire sparse directory set. Once the sparse directory set has been read out, the pointers P stored in all the ways of the sparse directory set are compared against the index of the pool entry. When the head (H) bit is not set for the pool entry being evicted, the index of the head of the collection containing the evicted pool entry is used for comparison. We organize the sparse directory's tag and data arrays as two separate direct-mapped arrays with one row of each array containing the tag and data entries for a set. This allows us to efficiently read out one entire set. Such a design is attractive for a set-associative array if the size of the entire set is small, which is true for Pool directories.

IV. Simulation Environment

We use a significantly modified version of the Multi2Sim simulator [35] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. Each core has private L1 and L2 caches with the L2 cache being non-inclusive/non-exclusive with respect to the L1 instruction and data caches. The L1 instruction and data caches are 32 KB in size and eight-way set-associative. The unified L2 cache is 128 KB in size and eight-way set-associative. The L1 and L2 cache lookup latencies are one and two cycles, respectively. The cores along with their private caches are arranged on a mesh interconnect having a single-cycle hop time. The L3 cache is shared among all the cores and non-inclusive/non-exclusive with respect to the L1 and L2 caches. Each mesh switch, in addition to having a core along with its L1 and L2 caches, has a bank of the shared L3 cache and a slice of the sparse directory. Each L3 cache bank is 256 KB in size, has sixteen ways, and requires three cycles for lookup. The sparse directory slice in a switch is responsible for tracking the blocks mapped to the L3 cache bank in that switch. The associativity of the directory slice is same as the per-core L2 cache associativity (eight) and the number of sets in the directory slice is decided relative to the number of L2 cache sets per core. The ratio of the number of sets in one slice of a (R)x sparse directory to the number of L2 cache sets per core is R. All levels of the cache have 64-byte blocks and implement a least-recently-used (LRU) replacement policy. The sparse directory implements a 1-bit not-recently-used (NRU) replacement policy. The simulated system models eight single-channel memory controllers evenly distributed over the mesh. Each memory controller connects to a 2 GB DRAM module modeled using DRAMSim2 [30]. Each DRAM module is eight-way banked single-rank DDR3-2133 with 12-12-12 latency parameters and burst length eight. The memory controllers implement the FR-FCFS scheduling algorithm.

The applications for this study are drawn from various sources and detailed in Table I (ROI refers to the parallel region of interest). The inputs, configurations, and simulation lengths are chosen to keep the simulation time within reasonable limits while maintaining fidelity of the simulation results. The PARSEC and SPLASH-2[1] applications are simulated in execution-driven mode, while the rest of the applications are simulated by replaying an instruction trace collected through the PIN tool. The PIN trace is collected on a 24-core machine by running the multi-threaded applications creating at most 128 threads (including client, server, application, and JVM threads). Before replaying the trace through the simulated 128-core system, it is pre-processed to expose maximum possible concurrency across the threads while preserving the global order at global synchronization boundaries and between load-store pairs touching the same memory block (64 bytes).

We evaluate five directory organizations in this study. These are summarized below. We assume a 48-bit physical address.

Scalable coherence directory (SCD) [31]: Sparse directory with $\frac{1}{16}\times$ sets i.e., each slice has sixteen sets and eight ways. Each directory way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, two limited-pointer fields and their valid bits (total sixteen bits, which can also encode the sharers in a sixteen-core cluster in a hierarchical representation), two bits to encode the type of representation (limited-pointer, root, leaf), and three bits to encode the cluster id in a hierarchical representation. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 55 bits i.e., **110 KB**.

Hybrid directory [14]: Sparse directory with $\frac{1}{16}\times$ sets i.e., each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), and 1-bit NRU state. Out of the eight ways in a set, two ways can encode full-map sharer vectors and each is of size 128 bits. The remaining six ways can encode a single pointer, each of size seven bits. Total size is **142.5 KB**.

Select directory [40]: Sparse directory with $\frac{1}{16}\times$ sets i.e., each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, a 7-bit pointer, and a pointer state bit (single sharer vs. pool pointer). Depending on the pointer state bit, the 7-bit pointer field stores either the private owner of a block or the entry id of a dynamically assigned full-map bitvector from a sixteen-entry fully-associative pool of bitvectors. Each pool entry has a 128-bit sharer vector, a valid bit, and four bits of back-pointer to the associated sparse directory set. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 42 bits + 128 slices $\times$ 16 pool entries $\times$ 133 bits i.e., **117.25 KB**. The bitvector pool exercises FIFO replacement.

Pool directory: Sparse directory with $\frac{1}{16}\times$ sets i.e., each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, a 7-bit pointer, and a pointer state bit (single sharer vs. pool pointer). The pool has 40 entries per slice. Each pool entry has four limited-pointer fields and their valid bits (total 32 bits, which can also encode the sharers in a 32-core cluster), one bit to encode the type of representation (limited-pointer, sharer cluster), one occupied bit, one head bit, two bits to encode the cluster id in a sharer cluster representation, and four bits of back-pointer to the sparse directory set. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 42 bits + 128 slices $\times$ 40 pool entries $\times$ 41 bits i.e., **109.625 KB**. Note that the SCD, Select, and Pool directories are sized to have similar storage overhead.

Full-map directory: Sparse directory with $\frac{1}{16}\times$ sets i.e., each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, and a 128-bit vector. Total size is **324 KB**.

V. Simulation Results

In this section, we quantitatively compare the SCD, Hybrid directory, Select directory, Pool directory, and full-map directory in terms

[1] The SPLASH-2 applications are drawn from the SPLASH2X extension of the PARSEC distribution.

TABLE I. SIMULATED APPLICATIONS

Suite	Applications	Input/Configuration	Simulation length
PARSEC	dedup, fluidanimate, swaptions	sim-medium, sim-medium, sim-small	Complete ROI
SPLASH-2	barnes	32K particles	Complete ROI
	fft	4M complex doubles	Complete ROI
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions
TPC	MySQL TPC-C	10 GB database, 2 GB buffer pool, 100 warehouses, 100 clients	500 transactions
	MySQL TPC-E	10 GB database, 2 GB buffer pool, 100 clients	Five billion instructions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 100 clients, zero think time, even distribution of Q6, Q8, Q11, Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.2 running Banking, Ecommerce, Support	Worker thread model, 128 simultaneous sessions, mod_php module	Five billion instructions
SPEC JVM	compiler.sunflow, crypto.aes	Five operations	Five billion instructions in ROI

of interconnect traffic, volume of private cache misses, the number of sparse directory fills, and overall application performance.

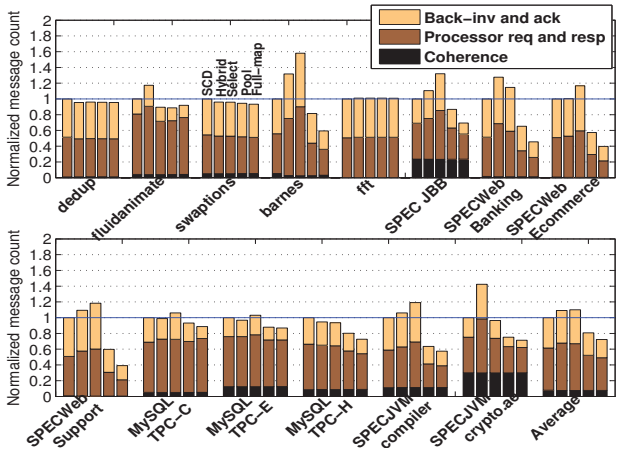

Fig. 4. Interconnect message count normalized to SCD.

Figure 4 shows a comparison of interconnect message count. Each group of bars corresponds to an application and the rightmost group in the bottom panel shows the average. The bars in a group correspond to SCD, Hybrid, Select, Pool, and full-map organizations from left to right. Each bar is divided into three segments representing three different types of messages. The forwarded requests, their responses, invalidations due to writes, and their acknowledgments constitute the coherence messages. The private cache misses, their responses, writebacks, and writeback acknowledgments constitute the processor requests and responses. Back-invalidations induced by sharer evictions from the directory and their acknowledgments constitute the third category of messages. We exclude the messages between the L3 cache banks and the memory controllers from these results because, as expected, the volume of these messages is not affected by changes in the directory organization. For each application, the message counts are normalized with respect to SCD. Across the board, we observe that the Pool directory organization is able to save a significant volume of interconnect messages. While the coherence message count remains largely constant across the different directory organizations, the primary savings achieved by the Pool directory arise from reduction in private cache misses and back-invalidations. Since the Pool directory manages the directory space more efficiently, the pressure on the directory goes down significantly leading to a less number of back-invalidations. The harmful subset of the back-invalidations causes an increase in the volume of the private cache misses. The savings achieved by the Pool directory are particularly impressive for fluidanimate (11% reduction), barnes (18% reduction), SPEC JBB (13% reduction), SPECWeb (35% to 43% reduction), TPC-E (12% reduction), TPC-H (20% reduction), and the SPEC JVM applications (25% to 36% reduction). In these applications, SCD suffers from high directory pressure because it requires multiple directory entries to encode more than two sharers. This leads to premature invalidation of directory entries tracking active blocks causing an increased volume of private cache misses. On average, the Pool directory organization achieves a 19% reduction in interconnect message count and is able to bridge a big portion of the wide gap between SCD and full-map organizations (see the Average group of bars). The Hybrid and Select organizations suffer from respectively 9% and 10% average increase in interconnect message count compared to SCD. The Hybrid organization's static partitioning of the directory space among the two types of directory entries fails to match the dynamic demand of directory entries over time and across applications. The Select organization, on the other hand, fails to track all the active shared blocks with the few wide sharer vectors. Figure 5 further shows the interconnect traffic (total message size) normalized to SCD. The trends are similar to those shown in Figure 4. On average, compared to SCD, the Hybrid and Select organizations experience respectively 6% and 7% more interconnect traffic and the Pool directory organization enjoys a 20% reduction in interconnect traffic.

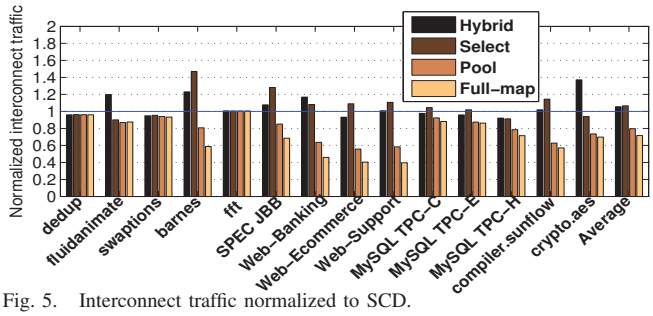

Fig. 5. Interconnect traffic normalized to SCD.

To further understand the directory pressure in SCD, Figure 6 quantifies the number of sparse directory entry allocations (not to be confused with pool entry allocations in the Pool and Select directory organizations) normalized to SCD. Across the board, we see that the SCD organization experiences a high volume of directory allocations indicating a significant amount of directory conflicts. The rest of the organizations all have almost equal number of sparse directory entry allocations, as expected. Only SCD requires multiple directory entries for encoding the sharers of a single block in a hierarchical manner. On average, SCD suffers from almost double the number of directory entry allocations compared to the other four organizations. The applications that enjoy significant savings in message count and traffic with Pool directory are also the ones that experience relatively higher volume of sparse directory allocations in SCD (e.g., fluidanimate, barnes, SPEC JBB, SPECWeb, TPC-E, TPC-H, and the SPEC JVM applications). Even though the Hybrid, Select, Pool, and full-map organizations have nearly the same number of tag allocations in the sparse directory, their differences in the message traffic arise due to the eviction of sharer tracking information from the tags and not due to eviction of tags from the sparse directory (e.g., eviction of a pool entry in the Pool directory or a swap between bitvector and pointer ways in the Hybrid directory).

Figure 7 presents the volume of private cache misses normalized to SCD. We also show the breakdown of private cache misses into code and data misses. These results closely correlate with the processor request and response message count data shown in Figure 4. The Pool directory organization, on average, enjoys 19% less private cache misses compared to SCD, while the Hybrid and Select organizations suffer from respectively 9% and 10% increase in the volume of private

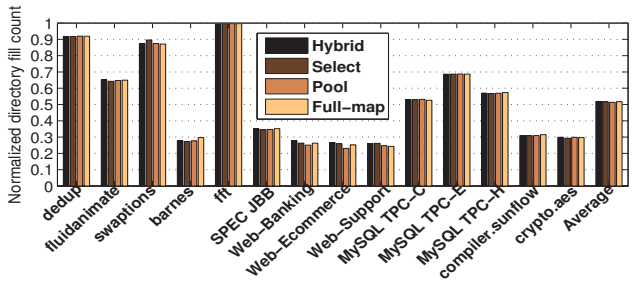

Fig. 6. Sparse directory fill count normalized to SCD.

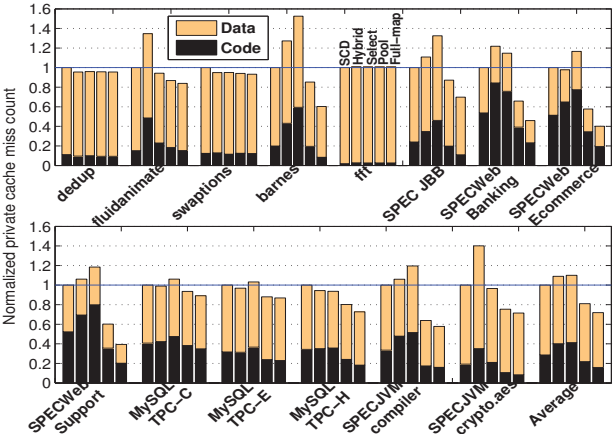

Fig. 7. Private cache miss count normalized to SCD.

cache misses. The Pool directory is able to save both code and data misses, while the Hybrid and Select organizations suffer primarily due to increased volumes of code misses compared to SCD. Since the code blocks experience good amount of sharing, these results indicate that the Hybrid and Select organizations are unable to track all the actively shared code blocks with their resources for tracking shared blocks.

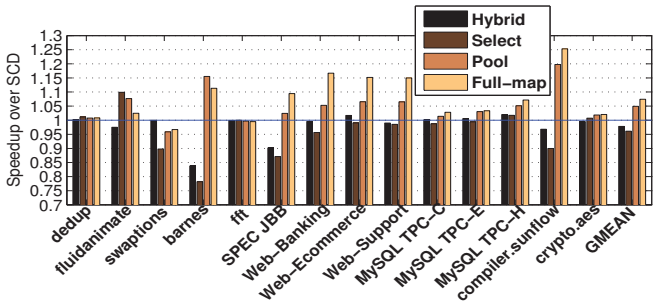

Fig. 8. Speedup over SCD.

Figure 8 summarizes the performance speedup achieved by the Hybrid, Select, Pool, and full-map directory organizations over SCD. The performance improvement achieved by the Pool directory organization is 5%, on average; significant gainers are fluidanimate (8%), barnes (16%), SPECWeb (5% to 7%), TPC-H (5%), and SPEC JVM compiler.sunflow (20%). Most importantly, while using only one-third of the directory space of a full-map organization, the average performance of the Pool directory organization comes within 2.4% of the full-map directory organization. The Hybrid and Select directory organizations, on average, perform respectively 2% and 4% worse than the SCD organization.

For a budget-constrained sparse directory such as $\frac{1}{16}\times$, the directory organization and the directory replacement policy may play an important role in determining the end-performance. In the following, we evaluate a sparse directory design that uses a four-way skew-associative organization with the timestamp-based three-level least-recently-used Z-cache replacement protocol (52 replacement candidates) [31]. Table II summarizes the interconnect traffic, speedup, and dynamic energy expended by the directory structures (using 22 nm nodes) for the Pool directory design. The results are averaged over all the applications and normalized to SCD. The results are shown for two different organizations of the $\frac{1}{16}\times$ sparse directory, namely, eight-way set-associative exercising NRU replacement (this is the design we have been discussing so far) and four-way Z-cache. While the Z-cache organization reduces the performance gap between SCD and Pool directory, the latter continues to save 16% interconnect traffic, on average. The Pool directory saves 85% dynamic energy in the coherence directory reads and writes compared to SCD in the set-associative organization. There are two primary reasons for this large saving. First, the Pool directory-based design enjoys 19% less private cache misses leading to a significantly reduced volume of directory accesses. Second, due to the hierarchical encoding, the SCD design may require multiple set-associative lookups into the sparse directory array to complete one private cache miss request. Although the Pool directory may also need multiple lookups into the pool, the expended energy is significantly less due to the tagless direct-mapped design of the pool. For the Z-cache organization, the Pool directory continues to save 82% energy compared to SCD, on average.

TABLE II. POOL DIRECTORY RESULTS RELATIVE TO SCD FOR $\frac{1}{16}\times$ DIRECTORY

Organization	Traffic	Speedup	Dynamic energy
Set-assoc., NRU	0.80	1.05	0.15
Z-cache	0.84	1.04	0.18

VI. SUMMARY

We have presented a novel coherence directory organization that has a set-associative sparse directory and a direct-mapped pool, each entry of which can act as a limited-pointer entry as well as a short sharer vector entry encoding the sharers in a cluster of cores. A dynamically allocated collection of such pool entries can efficiently track all the sharers of a block. Each sparse directory entry has a pointer, which can either encode a sharer (useful for tracking private blocks) or point to a pool entry. The pool entries allocated to a shared block are contiguously placed in the pool so that maintaining a pointer to the head entry is enough. Simulation results on a 128-core chip-multiprocessor show that our proposal performs 5% better than the state-of-the-art dynamic hierarchical directory organization while reducing the interconnect traffic by 20%. Our proposal delivers performance within 2.4% of a full-map organization while consuming only one-third of the directory space of a full-map organization.

REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A New Scalable Directory Architecture for Large-scale Multiprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 97–106, January 2001.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, **16**(1): 67–79, January 2005.

[3] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, May/June 1988.

[4] M. Alisafaee. Spatiotemporal Coherence Tracking. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture*, pages 341–350, December 2012.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, September 2008.

[6] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. In *IEEE Transactions on Computers*, **C-27**(12):1112–1118, December 1978.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[8] Y. Chang and L. Bhuyan. An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 172–179, August 1996.

[9] G. Chen. SLiD – A Cost-effective and Scalable Limited-directory Scheme for Cache Coherence. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, pages 341–352, June 1993.

[10] B. Choi et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, October 2011.

[11] J. H. Choi and K. H. Park. Segment Directory Enhancing the Limited Directory Cache Coherence Schemes. In *Proceedings of the 13th International Parallel and Distributed Processing Symposium*, pages 258–267, April 1999.

[12] B. A. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 93–104, June 2011.

[13] S. Demetriades and S. Cho. Stash Directory: A Scalable Directory for Many-core Coherence. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, pages 177–188, February 2014.

[14] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang. Building Expressive, Area-efficient Coherence Directories. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 299–308, September 2013.

[15] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-core Systems. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture*, pages 169–180, February 2011.

[16] S. Guo, H. Wang, Y. Xue, D. Li, and D. Wang. Hierarchical Cache Directory for CMP. In *Journal of Computer Science and Technology*, **25**(2): 246–256, March 2010.

[17] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *Proceedings of the International Conference on Parallel Processing*, pages 312–321, August 1990.

[18] HP Labs. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Available at http://www.hpl.hp.com/research/cacti/.

[19] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Distributed Directory Scheme: Scalable Coherent Interface. In *IEEE Computer*, **23**(6): 74–77, June 1990.

[20] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel. WAYPOINT: Scaling Coherence to Thousand-core Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 99–110, September 2010.

[21] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-core Cache-coherent Processor with On-chip Optical Network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 477–488, September 2010.

[22] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[23] Y. Li, R. G. Melhem, and A. K. Jones. Practically Private: Enabling High Performance CMPs through Compiler-assisted Data Classification. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 231–240, September 2012.

[24] Y. Maa, D. Pradhan, and D. Thiebaut. Two Economical Directory Schemes for Large-scale Cache Coherent Multiprocessors. In *ACM SIGARCH Computer Architecture News*, **19**(5): 10–18, September 1991.

[25] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. In *Communications of the ACM*, **55**(7):78–89, July 2012.

[26] S. S. Mukherjee and M. D. Hill. An Evaluation of Directory Protocols for Medium-scale Shared Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 64–74, July 1994.

[27] H. Nilsson and P. Stenstrom. The Scalable Tree Protocol – A Cache Coherence Approach for Large-scale Multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 498–506, April 1992.

[28] B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, May 1990.

[29] A. Ros and S. Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 241–252, September 2012.

[30] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, **10**(1): 16–19, January-June 2011.

[31] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2012.

[32] R. Simoni and M. Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proc. of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[33] R. T. Simoni, Jr. Cache Coherence Directories for Scalable Multiprocessors. *PhD dissertation*, Stanford University, 1992.

[34] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–26, March 2013.

[35] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 335–344, September 2012.

[36] D. Wallach. PHD: A Hierarchical Cache Coherent Protocol. *Ph.D. dissertation*, MIT, September 1992.

[37] W.-D. Weber. Scalable Directories for Cache-coherent Shared-memory Multiprocessors. *Ph.D. dissertations*, Stanford University, January 1993.

[38] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[39] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[40] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. SelectDirectory: A Selective Directory for Cache Coherence in Many-core Architectures. In *Design, Automation, and Test in Europe*, March 2015.

[41] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain Coherence Directories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, December 2013.

[42] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 423–434, December 2009.

[43] L. Zhang, D. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin. SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, August 2014.

[44] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 33–44, October 2011.

[45] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, September 2010.