Cuckoo Directory: A Scalable Directory for Many-Core Systems

Michael Ferdman[‡†] Pejman Lotfi-Kamran[†] Ken Balet[†] Babak Falsafi[†]

[‡]Computer Architecture Lab
Carnegie Mellon University
http://www.ece.cmu.edu/CALCM/

[†]Parallel Systems Architecture Lab
École Polytechnique Fédérale de Lausanne
http://parsa.epfl.ch/

Abstract

Growing core counts have highlighted the need for scalable on-chip coherence mechanisms. The increase in the number of on-chip cores exposes the energy and area costs of scaling the directories. Duplicate-tag-based directories require highly associative structures that grow with core count, precluding scalability due to prohibitive power consumption. Sparse directories overcome the power barrier by reducing directory associativity, but require storage area over-provisioning to avoid high invalidation rates.

We propose the Cuckoo directory, a power- and area-efficient scalable distributed directory. The cuckoo directory scales to high core counts without the energy costs of wide associative lookup and without gross capacity over-provisioning. Simulation of a 16-core CMP with commercial server and scientific workloads shows that the Cuckoo directory eliminates invalidations while being up to four times more power-efficient than the Duplicate-tag directory and 24% more power-efficient and up to seven times more area-efficient than the Sparse directory organization. Analytical projections indicate that the Cuckoo directory retains its energy and area benefits with increasing core count, efficiently scaling to at least 1024 cores.

1. Introduction

Manufacturing technology innovation has led to rapidly growing on-chip core counts in today's processors, highlighting the need for a scalable on-chip cache coherence mechanism. Adapting prior work from multi-chip systems [17], cache coherence between private caches has been achieved on CMPs with a handful of cores. However, quickly growing core counts have exposed the energy and area costs of scaling the existing coherence mechanisms, requiring innovation to achieve power-efficient cache coherence with reasonable area budgets in future CMPs [31,43].

There exist two broad classes of CMP coherence directories. Duplicate-Tag-based schemes in use by several designs [7,16,43] are area-efficient, but require highly associative structures whose power dissipation precludes scaling to large core counts. Conversely, Sparse directory schemes [17] are power-efficient, but incur considerable area cost in over-provisioning the directory capacity to avoid conflicts in low-associativity directory structures.

Sparse directory organizations using compressed representations of sharer bit vectors are myriad [1,3,10,11,13,17,23,36]. Hierarchical directory organizations [44,45] can enable area-efficient uncompressed vector storage through multiple serialized lookups. However, these techniques address only the size of the sharer vectors, not the number of vectors the directory must store. Sparse directories experience set conflicts, forcing evictions of cached blocks that cannot be simultaneously tracked by the directory. To reduce conflict frequency and avoid performance loss, existing Sparse directory implementations over-provision directory capacity [17,35]. Although compressed and hierarchical designs are theoretically scalable (power and area do not grow significantly with core count), the practical area cost of Sparse directories is excessive due to capacity over-provisioning.

In this paper, we present the *Cuckoo directory*, a scalable distributed directory with nearly constant power and area utilization per core, regardless of core count. The Cuckoo organization avoids set conflicts of traditional Sparse directories, eliminating performance loss due to forced invalidations without significantly over-provisioning the directory capacity, achieving scalable power- and area-efficient CMP coherence.

To avoid set conflicts, the Cuckoo directory uses a *N-ary Cuckoo hash table* [15,29], a small associativity (3- or 4-way) structure whose address bits are passed through different hash functions, one for each way. The physical implementation of the Cuckoo directory closely resembles a set-associative structure, having nearly identical energy and latency per lookup. However, unlike the set-associative organization that always picks a replacement victim from a small set of

978-1-4244-9435-4/11/$26.00 ©2011 IEEE

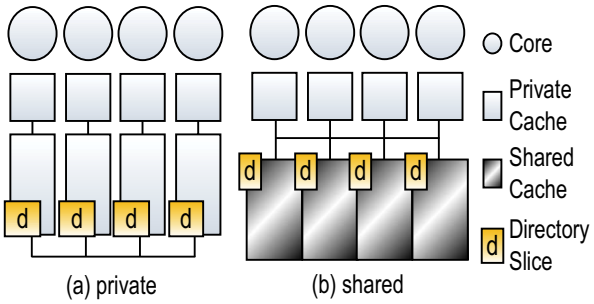

FIGURE 1. CMP organizations. (a) the interconnect connects independent private hierarchies (b) the interconnect is used to access an address-interleaved shared cache

conflicting entries, the Cuckoo directory displaces victims to alternate non-conflicting ways, practically never resorting to eviction.

We perform full-system simulation of CMPs running server and scientific workloads to evaluate coherence directory organizations based on commercial products, industry prototypes, and state-of-the-art research proposals. We use simulation and analytical projections to demonstrate that:

- The Cuckoo directory is a practical power- and area-scalable directory organization, offering up to 80x energy-efficiency over the leading area-efficient Tagless [43] design and more than 7x area-efficiency over the leading power-efficient *Sparse* [17] design at 1024 cores.
- Even at 16 cores, the Cuckoo directory is up to 16x more energy-efficient than the traditional Duplicate-Tag directory and up to 6x more area-efficient than the Sparse organization.

The rest of this paper is organized as follows. Section 2 provides background on CMP coherence and Section 3 explains the scalability of prior directory organizations. Section 4 presents the Cuckoo directory design and hardware. Section 5 provides a detailed evaluation of the Cuckoo directory. We present related work in Section 6 and conclude in Section 7.

2. CMP Coherence Background

Research literature and industry products explore many different CMP cache hierarchies. The *private* organization shown in Figure 1(a) has direct connections between private L1 and private L2 caches. Coherence must be explicitly enforced, invalidating all remotely cached copies of a block on a write and guaranteeing that cache misses are satisfied from a peer L2 if that L2 has a dirty copy of the accessed block. The *shared* organization shown in Figure 1(b) has small private L1 caches and a large shared L2 cache. The address-interleaved shared L2 cache has a unique location for each address, eliminating the need for a

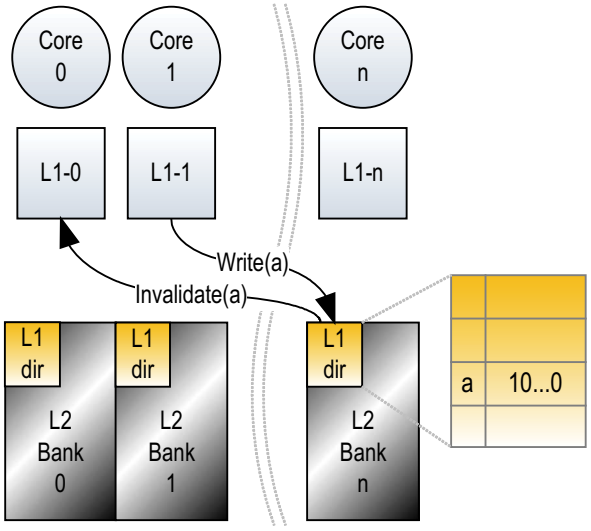

FIGURE 2. Operation of CMP directory. Cache coherence is enforced with a distributed directory located next to the lower-level cache banks.

coherence mechanism. However, even when the L2 is shared, coherence between the private L1 caches must be explicitly maintained. While shared and private organizations form the two extremes, actual designs may mix or extend these organizations (e.g., with software controlled private/shared hierarchies [19] or three-level hierarchies with private L1 and private L2 caches backed by a shared L3 [35]).

Coherence directories track the privately cached addresses in address-interleaved physically distributed directories. To achieve coherence, all accesses from the private caches interrogate the directory, which sends coherence requests to the sharers as needed. Figure 2 shows an example directory operation of a shared-cache CMP. Statically interleaved *home* locations determine which L2 bank and directory slice are responsible for each address. When a *write* request for block *a* arrives at its home location (bank *n* in Figure 2), the L1-directory slice is consulted in parallel with the L2 tags. If address *a* is found in the directory, *invalidation* requests are sent to all sharers. The private-cache organization undergoes a similar procedure, but the local private L2 cache is consulted first and a *write* request is sent to the home directory only in the case of a miss in the L2 cache.

3. CMP Directory Scalability

As the number of cores grows, the aggregate directory must increase commensurately. For each private cache, a *directory slice* is added to track the additional private cache's blocks. Moreover, the core count should not affect the organization of each direc-

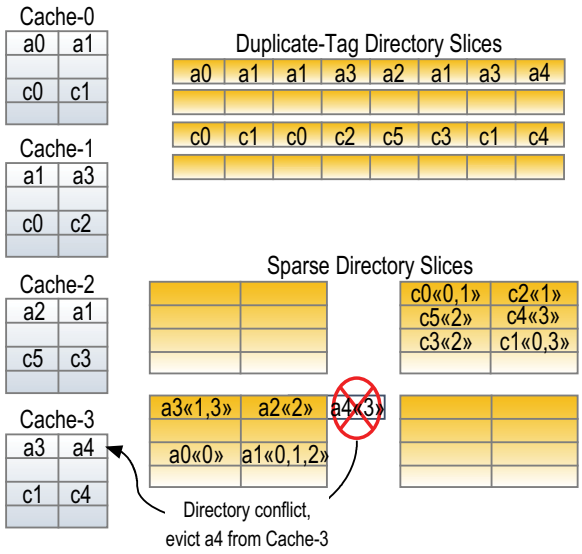

FIGURE 3. CMP directory organizations.

tory slice. If the associativity of each directory slice grows to accommodate more cores, the aggregate directory power dissipation grows quadratically. Similarly, if the storage of each directory slice grows to accommodate more cores, the aggregate directory area grows quadratically.

The operation of two basic CMP directory organizations is presented in Figure 3. Four private 2-way set-associative caches are shown with the four distributed slices of the Duplicate-Tag [7] and Sparse [17] directory organizations. The directory slices are address-interleaved and distributed on chip, each slice tracking blocks in a subset of the private-cache sets.

3.1. Duplicate-Tag Scalability

The Duplicate-Tag organization mirrors the organization of the private-cache tags, ensuring that there is always sufficient space in the directory to track all cached blocks. To construct an invalidation vector, a lookup in the Duplicate-Tag directory compares all stored tags in the directory set against the lookup tag, finding the sharers wherever the tags match.

The Duplicate-Tag associativity must equal the product of the cache associativity and the number of caches [6], resulting in designs with large (e.g., 332-wide [39]) associative directories. The Duplicate-Tag directory power dissipation for designs with 4- and 8-way private L1 caches [35] or 16-way private L2 caches is prohibitive even for today's CMP designs with a few cores.

Figure 4 presents the per-core area and energy scalability of directory designs for a system with 16-way private L2 caches. The aggregate chip energy and area utilization of the directory are the products of

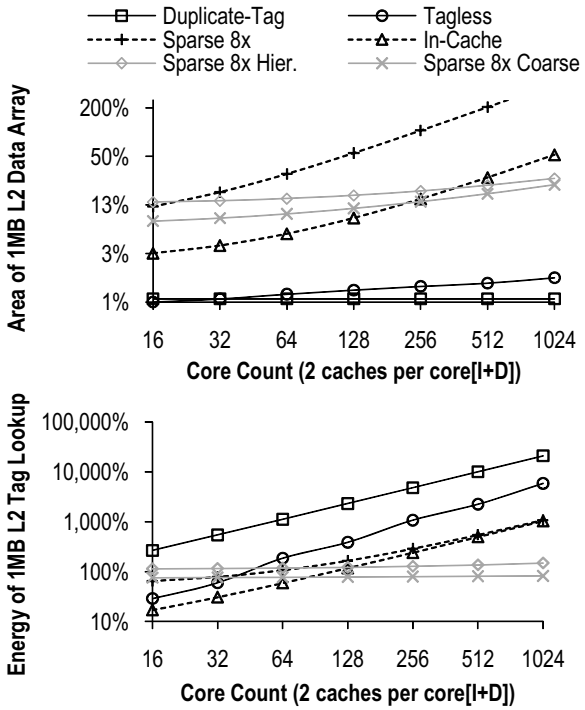

FIGURE 4. Area (top) and energy (bottom) scalability of various coherence directory organizations.

the values shown in Figure 4 and the core count. An increasing core count not only adds new directory slices to the Duplicate-Tag organization, but also linearly increases the associativity of each directory slice, resulting in non-scalable quadratic growth of the aggregate energy consumption of all directory slices.

3.2. Sparse Directory Scalability

The Sparse organization reduces directory associativity by using the low-order tag bits to extend the index of the directory storage, reducing the associativity by increasing the number of directory sets. Because this operation loses the one-to-one correspondence of directory entries to cache frames, each directory entry is extended with explicit sharer information.

Unfortunately, the non-uniform distribution of entries across directory sets in the Sparse organization incurs set conflicts, forcing invalidation of cached blocks tracked by the conflicting directory entries and reducing system performance. An example of a conflict in set *a* is shown in Figure 3. If blocks *a3* and *a4* are tracked by the directory and block *a2* is accessed, one of *a3* or *a4* must be evicted from the private caches because the directory organization cannot simultaneously track these three blocks. Reducing the conflict frequency requires over-provisioning the number of directory sets and associativity [17]. At the limit, the *in-cache directory* organization extends an

inclusive shared cache's tags with the sharer information, implicitly saving directory tag storage, but grossly over-provisioning the sharer storage [35] because the number of tags in the lower-level cache greatly exceeds the number of tracked blocks in the private caches.

The traditional Sparse directory uses bit vectors to track sharers [9]. The bit vectors stored in each directory entry must grow linearly with core count, in turn leading to a quadratic growth in the aggregate directory area as core counts increase. Although energy-efficient, both the traditional Sparse and the in-cache designs are impractical for large core counts. At 256 cores, the aggregate vector-based L1 directory could consume more than 256MB of on-chip storage, exceeding the capacity of the L2 caches [43].

3.3. Imprecise and Hierarchical Directories

Inexact and hierarchical representations of sharer vectors enable the reduction of directory storage and energy overheads at the cost of implementation complexity. For example, the *Tagless* directory [43] reduces the number of bits accessed for each directory operation by encoding a super-set of sharers in a Duplicate-Tag-like organization while *Coarse*-grained and inexact conservative encodings [3,11,13,17,23] reduce storage area in a Sparse organization.

Figure 4 shows that the Tagless directory is extremely area-efficient up to 1024 cores. However, this scalability comes at the cost of significant complexity. More importantly, like the traditional Duplicate-Tag organization, the bit-widths of either each read or each update operation of the Tagless directory increase with the number of cores. Therefore, the slope of the energy dissipation line for the Tagless directory in Figure 4 is nearly identical to the Duplicate-Tag organization. Although the energy for each operation in the Tagless directory is lower than in the Duplicate-Tag organization by a constant factor, the Tagless directory's energy dissipation still grows quadratically with core count, limiting its scalability.

Additionally, *hierarchical* organizations reduce directory storage by using coarse bit-vectors at a primary location and exact sub-bit-vectors at secondary locations [44,45]. Hierarchical techniques save storage by breaking up large bit vectors and allocating only the necessary second-level sub-bit-vectors, at the cost of additional storage to replicate the tags multiple times, once for each allocated second-level entry.

Figure 4 demonstrates the storage scalability of the Sparse Coarse [17] design that precisely stores sharers in the available bits (2*log(#caches) bits) and falls back to a coarse vector representation in the case of overflow [24], and Sparse Hierarchical [44,45], a 2-level hierarchical directory organization. Although theoretically scalable, these schemes address only the vector storage inside each entry and not the total number of directory entries. Set conflicts in the directories require over-provisioning the number of directory sets to avoid frequent invalidations, resulting in Sparse directories that can rival the L2 cache size.

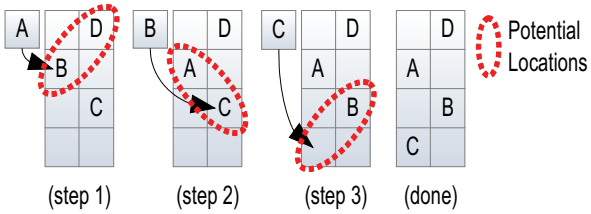

FIGURE 5. Cuckoo hash operation. A conflicts with B and D, B conflicts with A and C. Inserting A displaces B, triggering a cascade of insertions until C is placed into a vacant location.

4. Cuckoo Directory Design

We construct the Cuckoo directory to overcome the power and area scalability limitations of prior techniques. To meet these goals, the Cuckoo directory associativity and total storage per directory slice remain nearly constant, regardless of the core count.

The Cuckoo directory relies on the observation that low-associativity directory tag storage suffers primarily from transitivity of set conflicts. In traditional cache or directory indexing, if block A conflicts with B and block B conflicts with C, then A must also conflict with C. In a 2-way associative structure, there are two locations where A, B, and C can be stored. If B an C are present, inserting A must replace either B or C.

4.1. Cuckoo Hashing

Cuckoo hashing [29] can break transitive conflicts. The Cuckoo hash uses two independent (direct-mapped) tables, indexed through two different hash functions. A new entry is always inserted in one of the two tables, potentially displacing a valid entry. The insertion procedure continues with each displaced entry, alternating probing of the tables until the displaced entry is stored in a vacant position without displacing another entry.

Figure 5 demonstrates the Cuckoo hash operation. Block A conflicts with B and D, and block B conflicts with A and C. In step 1, A is inserted into one of its two possible locations, displacing B. In step 2, because B was previously displaced, B is inserted into its alternate location (where it conflicts with C). In step 3, C is inserted into a vacant alternate location (where it does not conflict with A), ending the insertion process.

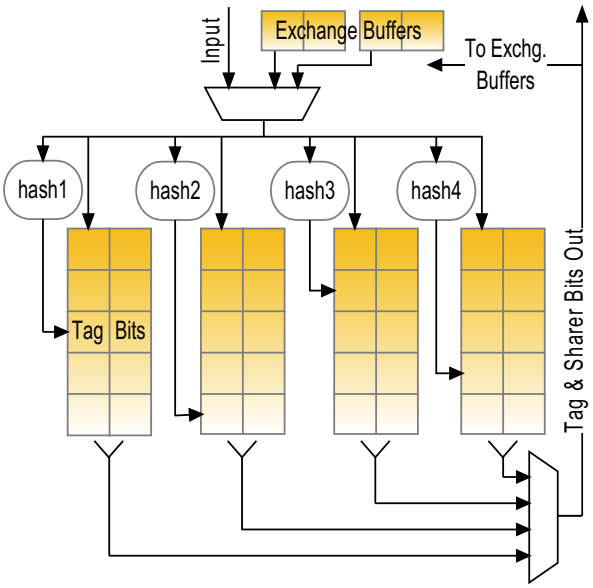

FIGURE 6. 4-way Cuckoo directory hardware. Each direct-mapped way is indexed through a different hash function.

The Cuckoo hash lookup operation is identical to the skewed-associative cache [34]. However, the key difference between the Cuckoo hash and the skewed organization is the insertion procedure. Whereas the skewed-associative cache selects a victim from one of the ways, the Cuckoo organization uses displacement to iteratively move entries until a non-conflicting location is found. Skewed associativity reduces conflict frequency, roughly doubling the perceived associativity of the underlying structure, while the Cuckoo hash provides nearly the equivalent of a fully-associative structure with the same lookup energy and latency, albeit with a more complex insertion procedure.

4.2. Hardware Implementation

The Cuckoo directory is an implementation of the *d-ary Cuckoo hash* [15] that extends the Cuckoo hash to more than two tables. Figure 6 depicts a 4-way Cuckoo directory. To find an element in the Cuckoo directory, all ways are looked up in parallel using hashed values of the searched address. Inserting an entry into the directory requires a lookup followed by a write of an entry in one of the ways. If the write replaces a valid directory entry, the insertion procedure is repeated for the victim entry, iterating until an insertion finds a vacant location.

We limit the maximum number of insertion attempts to avoid infinite loops. A counter tracks how many times an insertion procedure passes way 0. If the counter overflows, the hardware terminates the procedure and discards the most recently displaced entry, maintaining correctness by invalidating the blocks in the private caches that correspond to the evicted entry. To maintain a uniform distribution of entries across the ways, each insertion starts at the way at which the previous insertion stopped.

Upgrade and eviction requests search the directory and update the corresponding entries. For read and write misses from private caches, the directory is searched for a matching tag. If an entry is found, it is updated with a new sharer, and, if necessary, an invalidation vector based on the entry is produced. If the accessed tag is not found, a new entry is inserted.

In the shared-cache configurations, the directory lookup is performed in parallel with the L2 lookup. Because the L2 cache is a larger and slower structure, the latency of the directory lookup is not on the critical path and has absolutely no impact on performance. For shared-configuration directory updates and for all requests in the private-cache configurations, multiple insertion attempts may appear on the critical path. However, in practice, the frequency of long insertions is too low (see Figure 10) to have a measurable impact on performance. Furthermore, long insertions can be immediately prematurely terminated when a new request arrives at the directory, eliminating any potential effect on the lookup latency of the new request.

TABLE 1. System parameters.

CMP Size	16 cores
Processing Cores	UltraSPARC III ISA
L1 Caches	split I/D, 64KB, 2 ways 64-byte blocks, write-back
L2 NUCA Cache	16-core CMP: 1MB per core, 16 ways 64-byte blocks
Main Memory	3 GB memory, 8KB pages 48-bit address space

5. Evaluation

We analyze coherence directory access patterns using full-system simulation executing unmodified applications and operating systems in *FLEXUS* [42]. FLEXUS extends the *Virtutech Simics* functional simulator with models of processor tiles with cores, NUCA cache, on-chip coherence protocol controllers, and on-chip interconnect. We simulate a tiled CMP where the lowest-level cache is the L2 cache. We summarize our tiled architecture parameters in Table 1.

We simulate systems running *Solaris 8* and executing the workloads listed in Table 2. We include two scientific workloads and a range of server workloads from competing vendors, including online transaction processing, decision support system, and web server benchmarks. We start simulation from warm system checkpoints. For server workloads, we measure 100 million instructions after warming the micro-architec-

TABLE 2. Application parameters.

OLTP – Online Transaction Processing (TPC-C v3.0)	
DB2	*IBM DB2 v8 ESE*, 100 warehouses (10 GB), 64 clients, 2 GB buffer pool
Oracle	*Oracle 10g Server*, 100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server (SPECweb99)	
Apache	*Apache HTTP Server v2.0*, 16K connections, fastCGI, worker threading
Zeus	*Zeus Web Server v4.3*, 16K connections, fastCGI
DSS – Decision Support Systems (TPC-H)	
Qry 2,16,17	*IBM DB2 v8 ESE*, 480 MB buffer pool, 1 GB database
Scientific	
em3d	768K nodes, degree 2, span 5, 15% remote
ocean	1026x1026 grid, 9600s relaxations, 20K res., err 1e-7

tural state for 100 million instructions. For scientific workloads, we warm the micro-architectural state for four iterations and measure the 5th iteration.

We present results for two system configurations presented in Section 2. The *Shared-L2* configuration uses a coherence directory that tracks sharers in private L1 caches. The *Private-L2* configuration tracks sharers in larger private L2 caches. The Private-L2 results are also representative of a system with a 3-level cache hierarchy using two private levels and a shared LLC.

5.1. Cuckoo Hash Characteristics

Figure 7 demonstrates the fundamental Cuckoo directory behavior by presenting an analysis of the d-ary Cuckoo hashing technique as a function of occupancy. To avoid bias from hash function selection, we use strong cryptographic functions to index the ways. The left graph shows the average number of insertion attempts until a successful insertion without a victim. The right graph shows the frequency of not finding a vacant location for a victim entry in 32 insertion attempts. Results are presented as a function of occupancy, as the curve is affected only by the occupancy and is completely independent of the total capacity of the structure.

In case of low occupancy, a vacant location is typically found on the initial lookup. Below 50% occupancy, insertions into 3-ary and wider Cuckoo hash tables either succeed immediately or require only a single displacement. Furthermore, for up to 65% occupancy, 3-ary and wider organizations do not experience insertion failures.

Based on the results of Figure 7, we conclude that a Cuckoo directory with occupancy 50% or lower should never invalidate cache blocks due to directory conflicts, successfully inserting all directory entries, on average, after only two attempts. Directory occupancy below 50% is achieved trivially through sizing of the

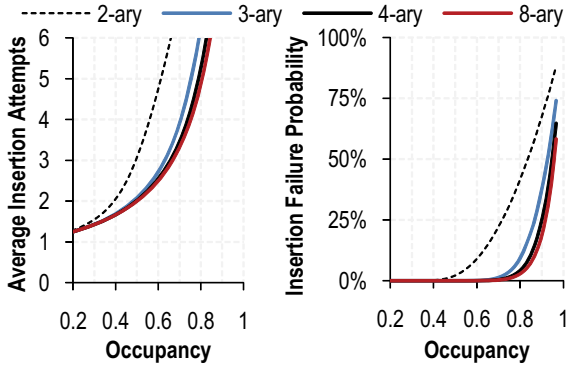

FIGURE 7. Cuckoo hash characteristics. Average insertion attempts and insertion failure rate for 100,000 random values, plotted as a function of occupancy.

Cuckoo directory tables. The maximum number of distinct tags tracked by a directory slice is equal to the number of frames in a private cache. Occupancy below 50% is therefore always guaranteed by a 2x over-provisioning of the Cuckoo directory capacity.

5.2. Cuckoo Directory Under-Provisioning

In practice, presence of shared instruction and shared data blocks limits the number of distinct tags in the aggregate private caches, leading to a natural reduction in the directory occupancy and enabling the Cuckoo directory to function without 2x over-provisioned capacity. We present the average directory occupancy of our workloads in Figure 8. As expected, we observe reduced directory occupancy, indicating that over-provisioning is not needed for the Shared-L2 configuration. For the Private-L2 configuration, decision support queries and the scientific workloads are dominated by large private footprints, resulting in predominantly unique blocks across all private caches, and calling for a slight Cuckoo directory over-provisioning to achieve the desired occupancy. The *ocean* workload demonstrates the extreme case, having nearly 100% unique private blocks in all caches.

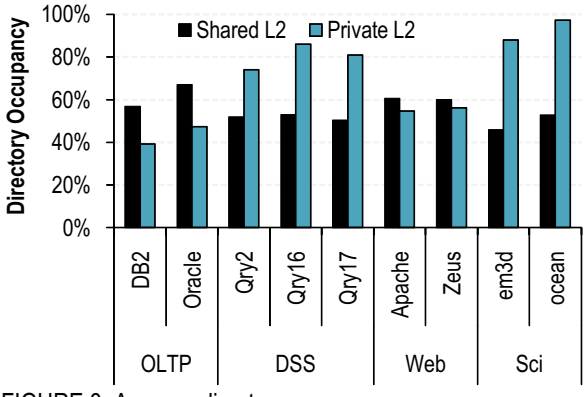

FIGURE 8. Average directory occupancy.

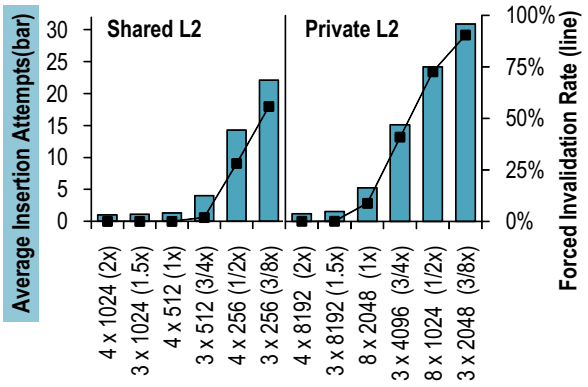

FIGURE 9. Cuckoo directory insertion attempts and failure rates for Shared-L2 and Private-L2 configurations. Cuckoo directory sizes are expressed as (number of ways) x (number of sets). Parenthesized values indicate provisioning factor.

We use the average number of insertion attempts and forced invalidation rates to determine the minimum Cuckoo directory size. Figure 9 evaluates a wide range of Cuckoo directory sizes and organizations, ranging from under-provisioned to over-provisioned. In addition to an organization's way and set counts, in parenthesis, we indicate the provisioning factor. Factor "1x" indicates a capacity equal to the worst-case number of blocks that the directory must simultaneously track (equal to the number of cache frames that map to the directory slice). Greater factors indicate capacity over-provisioning, while lower values indicate under-provisioning.

We allow up to 32 insertion attempts to ensure termination in the unlikely event of a loop; in such cases, we count 32 attempts toward the average. A lookup always precedes an insertion to confirm that a new entry should be allocated rather than adding a sharer to an existing entry. The lookup implicitly reveals if an empty position, eligible to hold the searched entry, exists in any of the directory ways. If an empty position is found during the lookup, insertion succeeds on the first attempt, contributing one toward the average. Addition of sharers to already existing directory entries does not affect the reported average number of attempts. Dirty and clean evictions from the private caches are tracked by the directory, with the directory entry becoming empty and eligible for reuse at the time the last sharer evicts the block.

Figure 9 indicates that under-provisioning directory capacity (factor less than 1x) results in an exponential increase in insertion attempts and forced invalidations due to failed insertions. However, as expected from Figure 8, the Shared-L2 configuration does not require over-provisioning the capacity to achieve a small average number of insertion attempts

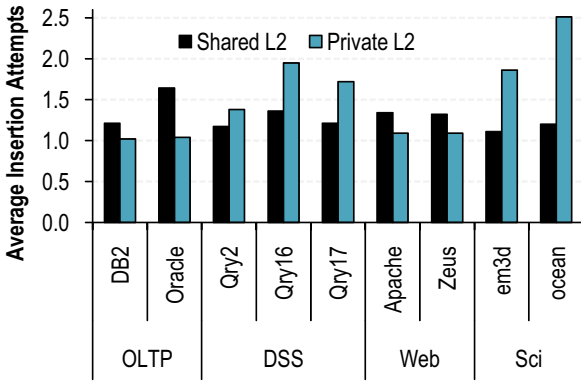

FIGURE 10. Cuckoo directory average insertion attempts.

and near-zero invalidation rates. We find that a small 1.5x capacity over-provisioning is sufficient for the Private-L2 configuration.

5.3. Worst-case Cuckoo Insertion Attempts

The Cuckoo directory capacity in Section 5.2 is selected based on the average behavior across a wide range of workloads. To confirm general applicability, Figure 10 presents the average insertion attempts for all workloads using the selected 4x512 and 3x8192 Cuckoo directory organizations for the Shared-L2 and Private-L2 configurations, respectively. Despite the small directory size, the average number of insertion attempts is typically less than two, indicating that a vacant location is usually found at the time of the initial lookup. Larger average insertion attempts are observed in workloads with more private blocks.

Figure 11 presents the insertion attempts for the benchmarks with the longest-tail distribution. For the Shared-L2 configuration, OLTP Oracle exhibits the worst behavior; for the Private-L2 configuration, *ocean* exhibits the worst behavior. The distribution of

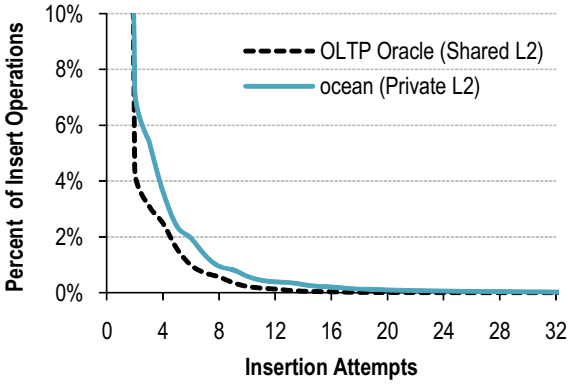

FIGURE 11. Worst-case insertion attempt distributions. Values at 1 insertion attempt (85% for Oracle, 73% for ocean) are not shown to enhance clarity.

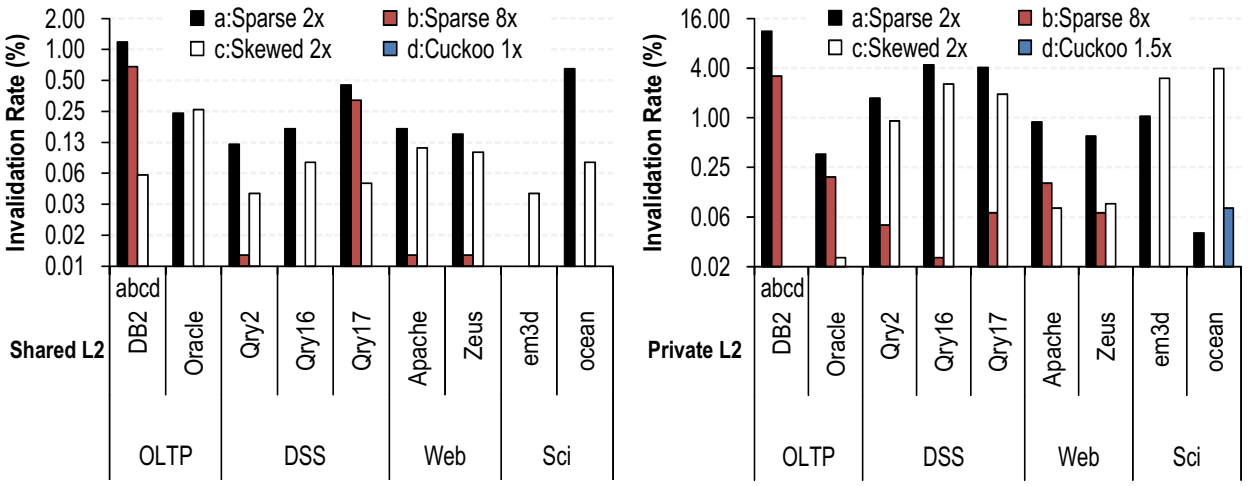

FIGURE 12. Directory invalidation rates with Shared-L2 (left) and Private-L2 (right) configurations.

insertion attempts confirms expectations: each insertion attempt increases the probability of finding a vacant location, exponentially reducing the probability of performing a subsequent attempt. Even for the worst-case benchmarks, the probability of reaching 32 insertion attempts is nearly zero. Additionally, lack of a peak at 32 indicates that longer insertions and loops are practically non-existent.

5.4. Invalidation-Rate Comparison

We compare the forced-invalidation rates of the Cuckoo directory to competing directory organizations in Figure 12. We present the invalidation rate as a fraction of directory entry insertions for (a) an 8-way Sparse directory with two times capacity over-provisioning (Sparse 2x), (b) an 8-way Sparse directory with eight times over-provisioning (Sparse 8x), (c) a 4-way skewed-associative directory (Skewed 2x) adapted from the skewed-associative cache organization [33], and (d) the 3- and 4-way Cuckoo directory organizations selected in Section 5.2. The Sparse 2x directory has the same capacity as the skewed-associative organization, both having two times greater capacity than the Cuckoo 1x directory.

Our results indicate that the Sparse 2x directory incurs a significant number of set conflicts with nearly all workloads in both Shared-L2 and Private-L2 systems. Compared to the Sparse 2x directory, the 4-way Skewed 2x organization generally reduces the frequency of invalidations in highly contended directory sets for server workloads, but does not reduce invalidations for scientific workloads that have a more uniform distribution of accesses. The 8-way Sparse 8x directory is large enough to provide reasonable invalidation rates on average, however, the forced-invalidation rates remain significant for many workloads. Finally, we note that the Cuckoo directory — having less capacity and lower associativity compared to the competing designs — experiences near-zero invalidations with all workloads. Robustness of the Cuckoo design is highlighted with the *ocean* workload that has nearly 100% distinct blocks in the Private-L2 system, but experiences invalidations only on 0.08% of directory updates with only 1.5x Cuckoo directory.

5.5. Hash Function Selection

We evaluate the Cuckoo directory implementation using the skewing hash functions from Seznec and Bodin [34]. We repeated the experiments with cryptographic hash functions and observed no measurable benefit for the Cuckoo 2x directory. For more aggressive Cuckoo directory designs with lower provisioning factors, we observed marginally lower average insertion attempts. Additionally, in the case of *ocean*, stronger hash functions eliminate the forced invalidations observed in the Private-L2 configuration at 1.5x over-provisioning. However, fewer insertion attempts and reduced invalidation rates result in minimal energy improvements and are offset by the complex hardware implementation of the hash functions, compared to the trivial implementation of the skewing hash functions that require only several levels of logic.

Strong hash functions offered the most benefit in situations where the directory capacity was severely under-provisioned. In all configurations that exhibit high forced-invalidation rates, the stronger hash functions offer multiple-order-of-magnitude reduction in invalidation rates compared to the skewing functions. However, these needlessly over-constrained directory designs are impractical because they gain marginal area advantages at a huge power penalty due to many unsuccessful insertion attempts.

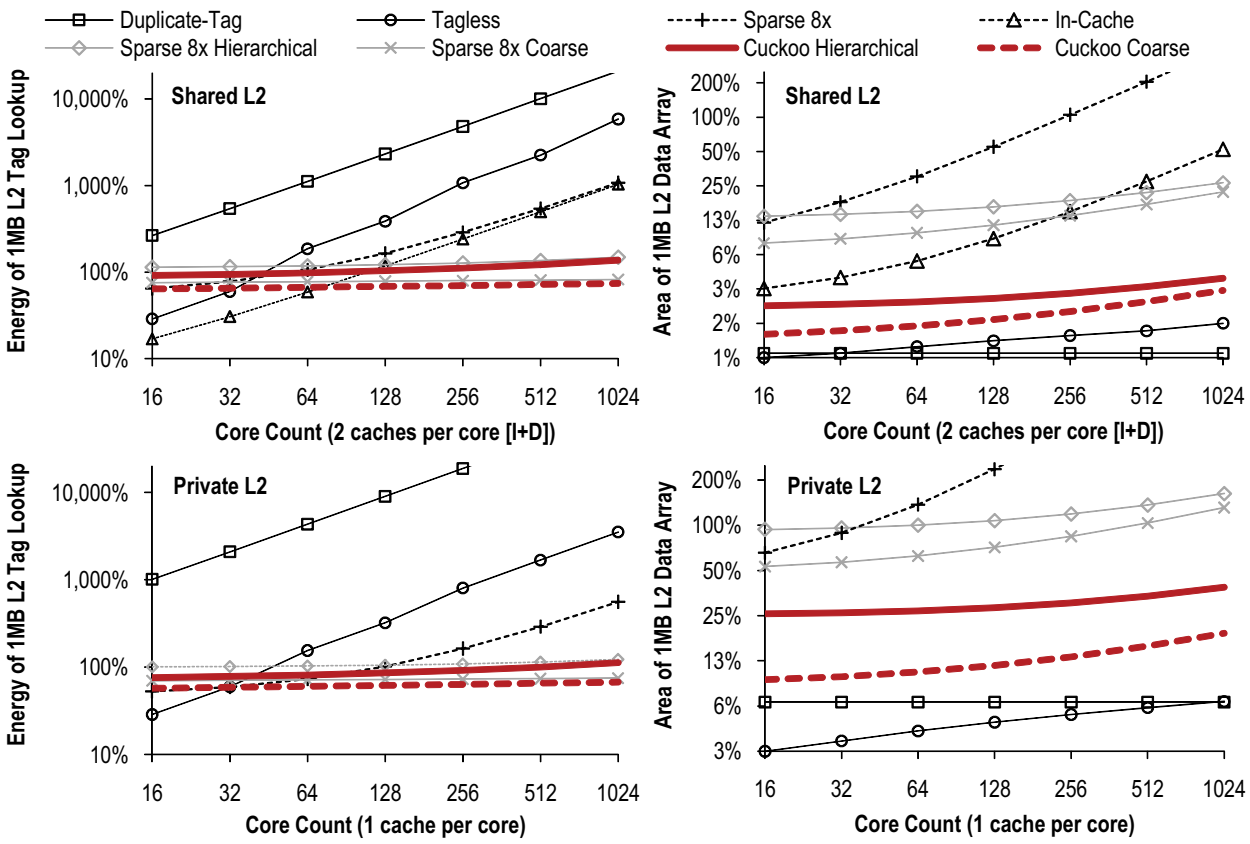

FIGURE 13. Power and area comparison of directory organizations. Directory energy consumption is plotted for varying core counts. Directory for the Shared-L2 cache configuration is depicted on top, Private-L2 configuration is depicted on the bottom.

5.6. Power and Area, Trends and Comparison

Figure 13 presents the per-core energy dissipation and area utilization of the leading directory organizations. We compute the energy dissipation based on the number and size of the read and update operations on the directory storage structures, scaling the energy of each operation by its frequency as determined from our workload suite.[1] We present directory energy dissipation relative to the energy of a 16-way set-associative L2 tag lookup. Directory area is presented relative to the area of the L2 data array (1MB).

The behavior trends of all directory organizations are similar across the Shared-L2 and Private-L2 configurations. Even for 16-core systems, the Duplicate-Tag organization is extremely energy inefficient, despite consuming minimal chip area. At low core counts, the small storage requirements of the Tagless directory lead to energy savings compared to most other directory organizations. The Tagless organization remains extremely area efficient to 1024 cores and beyond. However, like in the Duplicate-Tag organization, the number of bits read and written for each directory access in the Tagless directory scales linearly with core count, resulting in quadratic growth in chip energy dissipation for the aggregate of all Tagless directory slices. Despite being extremely area-efficient, energy dissipation of the Tagless directory becomes prohibitive beyond 128 cores.

Traditional Sparse directories storing full bit vectors for each tag suffer from both energy and area inefficiency. As core counts increase, the vector sizes increase linearly, dominating the energy and area of the directory and making the directory organization irrelevant. Linear per-core power and area increases render these designs unreasonable for large core counts.

The inclusive in-cache directory is not applicable to the Private-L2 configuration, as inclusion of private L2s in other private L2s is not possible. However, the inclusive in-cache directory for the Shared-L2 configurations can store full sharer bit vectors and leverage the L2 cache to avoid tag storage and tag lookup energy. In-cache directory designs remain practical as long as the bit-vector size (number of cores) remains moder-

1. The following event frequencies were used: Insert new tag into the directory: 23.5%, add sharer to existing entry: 26.9%, remove sharer from existing entry: 24.9%, remove tag from the directory: 23.5%, invalidate all sharers: 1.2%.

ate. However, beyond 128 cores, in-cache directories lose their advantages and become dominated by bit-vector storage, limiting applicability to larger systems.

Sparse directories storing a limited number of bits per entry (Sparse Coarse and Sparse Hierarchical) are subject only to a logarithmic increase in energy with an increasing core count. These organizations scale well with respect to energy and area, resulting in nearly flat horizontal lines in Figure 13. However, although these designs address the bit-vector storage scalability, the Sparse organization must be over-provisioned to avoid performance loss due to set conflicts. Over-provisioning results in a significant area increase, rendering these designs unattractive and showing the need for more area-efficient organizations.

The Cuckoo directory organization eliminates Sparse directory over-provisioning by resolving set conflicts while still leveraging the benefits of the Coarse and Hierarchical bit-vector storage mechanisms. The Cuckoo directory area utilization rivals the area-efficient Tagless and Duplicate-Tag directory designs, but also has a low nearly-constant per-core energy dissipation regardless of core count. The Cuckoo directory organization achieves up to 7x area reduction compared to the Sparse Coarse and Sparse Hierarchical organizations, maintaining reasonable energy dissipation while bringing the area of the directory storage under 3% of the L2 area for the Shared-L2 configuration with 1024 cores (2048 sharing caches) and under 30% of the L2 area for the Private-L2 configuration with 1024 cores. Constant scalability within the directory slice results in expected linear growth in the aggregate directory energy and area consumption as the number of cores increases, yielding a practical and scalable design to at least 1024 cores.

6. Related Work

Like the Cuckoo directory, a number of prior hardware structure proposals borrow ideas from software hash tables. Caches using multiple hash functions were proposed in hash-rehash caches [2] by Agarwal et al. and later in column-associative caches [4] by Agarwal and Pudar. Broder and Kalin proposed parallel hash functions to access independent memory banks [8]. Seznec used parallel hashing memories to reduce conflicts in skewed-associative caches [33], the basic organization used in the design of the Cuckoo directory. Other mechanisms to reduce conflicts through hashing [40] and software-controlled functions [41] were evaluated by Topham and González and Vandierendonck et al. The skewed-associativity mechanism was adapted as a replacement for CAMs [25] by Mahoney et al., who later provided a mathematical model for the parallel hashing structure [26] similar to the skewed-associativity model by Michaud [28]. Cho et al. relied on a hardware implementation of a hash table to create content-addressable memories from random access memories [12].

Like the skewed-associative cache, the Cuckoo directory relies on the family of hash functions proposed by Seznec and Bodin [34]. However, unlike skewed-associative caches and parallel hashing memories, the Cuckoo directory uses an insertion algorithm based on moving entries within the structure, as proposed for Cuckoo hash tables by Pagh and Rodler [29]. Hagersten and Hill proposed displacement to improve storage efficiency of skewed structures [18]. Spjuth et al. evaluated a displacement-based insertion algorithm in Elbow caches [37,38], crediting the idea behind displacements to Mark Hill. Similarly to Cuckoo directories, Elbow caches displace conflicting elements. However, the Elbow cache is limited to one displacement per insertion and requires multiple lookups to select a displacement victim, resulting in a complex and power-hungry design that experiences more forced invalidations than the Cuckoo directory.

Fotakis et al. generalized Pagh and Rodler's Cuckoo hash to a d-ary Cuckoo hash [15], improving the space utilization of the structure. The Cuckoo directories we evaluate use the d-ary organization. Panigrahy proposed an alternative mechanism to improve Cuckoo-hash space utilization by storing multiple elements per bucket [30]; we do not investigate this approach, but note that it may offer additional improvement in the behavior of the Cuckoo directory at high directory occupancy, potentially allowing a smaller and more power-efficient 3-ary design instead of a 4-ary organization for some systems.

Significant benefit of relocation in hardware hash tables [21] was noted by Kirsch and Mitzenmacher. Another hardware proposal by Kirsch et al. augmented a Cuckoo hash implementation with a CAM based *stash* to maintain overflow entries [22]. Arbitman et al. mathematically formalized the stash approach [5]. This technique is effective if all elements must be stored. However, the Cuckoo directory can invalidate blocks in the rare cases of overflow and does not benefit from a stash.

Many proposals exist to reduce directory overheads by reducing the directory entry size. Agarwal et al. evaluated schemes of storing a small number of pointers with various overflow handling policies [3]. Krafka and Newton evaluated a sectored organization to reduce tag storage overhead [23]. Gupta et al. suggested switching to a coarse representation when limited pointers overflow [17]. Chaiken et al. proposed

software fallback [10] and Chen proposed chained pointers to handle directory overflow [11]. Choi and Park proposed a hybrid of bit-vector and limited pointer organizations [13]. The Cuckoo organization dictates only the organization of the directory itself, not the contents of each entry or its home node. In this work, we constructed the Cuckoo directory with the coarse [17] and hierarchical [44,45] approaches, although the Cuckoo organization can be used in conjunction with any of these space-reduction techniques.

Finally, there exist proposals for coherence mechanisms that avoid directories and cannot directly benefit from the Cuckoo organization. Among these are: Tagless directories [43] from Zebchuk et al., demonstrated to be highly area-efficient but not energy-scalable; SCI [20] from James et al., using sharer pointers in the private caches rather than a directory structure; token coherence [27] from Martin and Hill, also avoiding a coherence directory; software-controlled address indirection by Fensch and Cintra [14] and Hardavellas et al. [19] handle coherence in software on shared cache substrates; and DiCo [32] from Ros et al., eliminating directories in favor of storing coherence information within the cache tags.

7. Conclusions

The exponential growth in the number of on-chip cores has highlighted the need for a scalable on-chip cache coherence mechanism, exposing the energy and area costs of scaling the coherence directories. Duplicate-Tag directories require highly associative structures that grow rapidly with core count and approach prohibitive power consumption. Sparse directories overcome power barrier by reducing associativity while over-provisioning the number of directory sets, but have a larger area cost and affect performance when directory overflows must invalidate active cache blocks.

In this work, we proposed the Cuckoo directory, a directory organization that eliminates set conflicts and enables energy- and area-scalable coherence for large core counts. Based on the Cuckoo hash table, a dense constant-time lookup structure, the Cuckoo directory avoids set conflicts without significant capacity over-provisioning. Through simulation and analytical projections, we showed that the Cuckoo directory provides energy and area benefits to existing 16-core designs and scales to hundreds of cores. At 1024 cores, the Cuckoo directory is up to 80 times more power-efficient than the area-efficient Tagless directory and 11% more power-efficient and seven times more area-efficient than the power-efficient Sparse directory.

REFERENCES

[1] M.E. Acacio, J. González, J.M. García, J. Duato, "A New Scalable Directory Architecture for Large-Scale Multiprocessors," HPCA '01: 7th International Symposium on High-Performance Computer Architecture, Washington, DC, 2001.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads," ACM Transactions on Computer Systems, vol. 6, 1988.

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," ISCA '88: 15th Annual International Symposium on Computer Architecture, Los Alamitos, CA, 1988.

[4] A. Agarwal and S.D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," Massachusetts Institute of Technology TR, 1992.

[5] Y. Arbitman, M. Naor, and G. Segev, "De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results," ICALP '09: 36th International Colloquium on Automata, Languages and Programming, Berlin, Germany, 2009.

[6] J. Baer and W. Wang, "On the inclusion properties for multi-level cache hierarchies," ISCA '88: 15th Annual International Symposium on Computer Architecture, Los Alamitos, CA, 1988.

[7] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," ISCA '00: 27th Annual International Symposium on Computer Architecture, New York, NY, 2000.

[8] A.Z. Broder and A.R. Karlin, "Multilevel Adaptive Hashing," SODA '90: First Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, 1990.

[9] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Mulicache Systems," IEEE Transactions on Computers, vol. 27, 1978.

[10] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," ASPLOS-IV: 4th International Conference on Architectural Support for Programming Languages and OS, New York, NY, 1991.

[11] G. Chen, "SLiD — A Cost-Effective and Scalable Limited-Directory Scheme for Cache Coherence," PARLE '93: Parallel Architectures and Languages Europe, Heidelberg, Germany, 1993.

[12] S. Cho, J.R. Martin, R. Xu, M.H. Hammoud, and R. Melhem, "CA-RAM: A High-Performance Memory Substrate for Search-Intensive Applications," International Symposium on Performance Analysis of Systems and Software, Los Alamitos, CA, 2007.

[13] J.H. Choi and K.H. Park, "Segment Directory Enhancing the Limited Directory Cache Coherence Schemes," IPPS '99/SPDP '99: 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, Washington, DC, 1999.

[14] C. Fensch and M. Cintra, "An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs," HPCA '08: 14th International Symposium on High Performance Computer Architecture, Salt Lake City, UT, 2008.

[15] D. Fotakis, R. Pagh, P. Sanders, and P.G. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," STACS '03: 20th Annual Symposium on Theoretical Aspects of Computer Science, London, UK, 2003.

[16] R. Golla, "Niagara2: A Highly Threaded Server-on-a-Chip," Fall Microprocessor Forum 2006, San Jose, CA, 2006.

[17] A. Gupta, W. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," ICPP '90: 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, 1990.

[18] E. E. Hagersten, M. D. Hill, "Skewed finite hashing function" U.S. Patent 6308246, filed September 4, 1998.

[19] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," ISCA '09: 36th Annual International Symposium on Computer Architecture, New York, NY, 2009.

[20] D.V. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi, "Scalable Coherent Interface," Computer, vol. 23, 1990.

[21] A. Kirsch and M. Mitzenmacher, "The Power of One Move: Hashing Schemes for Hardware," INFOCOM '08: 27th International Conference on Computer Communications, Cambridge, MA, 2008.

[22] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More Robust Hashing: Cuckoo Hashing with a Stash," ESA '08: 16th Annual European symposium on Algorithms, Berlin, Germany, 2008.

[23] B.W. O'Krafka and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," ISCA '90: 17th Annual International Symposium on Computer Architecture, New York, NY, 1990.

[24] J. Laudon, D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA '97: 24th Annual International Symposium on Computer Architecture, New York, NY, 1997.

[25] P. Mahoney, Y. Savaria, G. Bois, and P. Plante, "Parallel Hashing Memories: an Alternative to Content Addressable Memories," NEWCAS '05: The 3rd International IEEE-NEWCAS Conference, 2005.

[26] P. Mahoney, Y. Savaria, G. Bois, and P. Plante, "Performance Characterization for the Implementation of Content Addressable Memories Based on Parallel Hashing Memories," Transactions on High-Performance Embedded Architectures and Compilers II, 2009.

[27] M.M. Martin and M.D. Hill, "Token Coherence: Decoupling Performance and Correctness," ISCA '03: 30th Annual International Symposium on Computer Architecture, New York, NY, 2003.

[28] P. Michaud, "A Statistical Model of Skewed-Associativity," ISPASS '03: 2003 International Symposium on Performance Analysis of Systems and Software, Washington, DC, 2003.

[29] R. Pagh and F.F. Rodler, "Cuckoo Hashing," Algorithms, vol. 51, 2004.

[30] R. Panigrahy, "Efficient Hashing with Lookups in two Memory Accesses," SODA '05: 16th Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, 2004.

[31] S. Patel, S. Phillips, and A. Strong, "Sun's Next-Generation Multi-threaded Processor - Rainbow Falls," Hot Chips 21, Stanford, CA, 2009.

[32] A. Ros, M.E. Acacio, and J.M. Garcia, "DiCo-CMP: Efficient Cache Coherency in Tiled CMP Architectures," IPDPS '08: 22nd International Parallel & Distributed Processing Symposium, Miami, FL, 2008.

[33] A. Seznec, "A Case for Two-Way Skewed-Associative Caches," ISCA '93: 20th Annual International Symposium on Computer Architecture, New York, NY, 1993.

[34] A. Seznec and F. Bodin, "Skewed-associative Caches," PARLE '93: 5th International Conference on Parallel Architectures and Languages Europe, London, UK, 1993.

[35] R. Singhal, "Inside Intel® Next Generation Nehalem Microarchitecture," Hot Chips 20, Stanford, CA, 2008.

[36] R. Simoni, "Cache Coherence Directories for Scalable Multiprocessors," Stanford University TR, Stanford, CA, 1992.

[37] M. Spjuth, M. Karlsson, and E. Hagersten, "The Elbow Cache: A Power-Efficient Alternative to Highly Associative Caches," Technical Report, 2003.

[38] M. Spjuth, M. Karlsson, and E. Hagersten, "Skewed Caches from a Low-Power Perspective," CF '05: 2nd Conference on Computing Frontiers, New York, NY, 2005.

[39] SUN Microsystems, "OpenSPARC T2 Processor Megacell Specification," 2007.

[40] N. Topham and A. González, "Randomized Cache Placement for Eliminating Conflicts," IEEE Transactions on Computers, vol. 48, 1999.

[41] H. Vandierendonck, "Application-Specific Reconfigurable XOR-Indexing to Eliminate Cache Conflict Misses," DATE '06: Conference on Design, Automation and Test in Europe, Belgium, 2006.

[42] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," IEEE Micro, vol. 26, 2006.

[43] J. Zebchuk, V. Srinivasan, M.K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," MICRO '09: 2009 42st International Symposium on Microarchitecture, New York, NY, 2009.

[44] D.A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol," Massachusetts Institute of Technology Master's Thesis, Cambridge, MA, 1992.

[45] S.L. Guo, H.X. Wang, Y.B. Xue, C.M. Li, and D.S. Wang, "Hierarchical Cache Directory for CMP," Journal of Computer Science and Technology, 25(2), 2010.