A Tagless Coherence Directory

Jason Zebchuk
Dept. of Electrical and Computer Engineering
University of Toronto
zebchuk@eecg.toronto.edu

Vijayalakshmi Srinivasan
T.J. Watson Research Center
IBM
viji@us.ibm.com

Moinuddin K. Qureshi
T.J. Watson Research Center
IBM
moinqureshi@us.ibm.com

Andreas Moshovos
Dept. of Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

ABSTRACT

A key challenge in architecting a CMP with many cores is maintaining cache coherence in an efficient manner. Directory-based protocols avoid the bandwidth overhead of snoop-based protocols, and therefore scale to a large number of cores. Unfortunately, conventional directory structures incur significant area overheads in larger CMPs.

The *Tagless Coherence Directory (TL)* is a scalable coherence solution that uses an implicit, conservative representation of sharing information. Conceptually, TL consists of a grid of small Bloom filters. The grid has one column per core and one row per cache set. TL uses 48% less area, 57% less leakage power, and 44% less dynamic energy than a conventional coherence directory for a 16-core CMP with 1MB private L2 caches. Simulations of commercial and scientific workloads indicate that TL has no statistically significant impact on performance, and incurs only a 2.5% increase in bandwidth utilization. Analytical modelling predicts that TL continues to scale well up to at least 1024 cores.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories, shared memory*

General Terms

Design, Experimentation, Performance

Keywords

Cache coherence, Directory Coherence, Bloom Filters

1. INTRODUCTION

Chip-multiprocessors (CMPs) have become the defacto design for high-performance processors. While current CMP models have up to eight cores, shrinking feature sizes may lead to CMPs with tens or maybe even hundreds of cores. The shared memory programming paradigm has become an entrenched component in parallel application development that can exploit the performance potential of these CMPs. To continue supporting this paradigm, CMPs must maintain cache coherence across all of their cores.

CMPs with few cores can use snoop coherence protocols [19,21], but these designs do not scale well as the number of cores increases. Snoop protocols rely on broadcasts that result in overwhelming bandwidth utilization for larger CMPs. Despite recent proposals to reduce the snoop protocol bandwidth requirements [23,29], scaling these protocols to larger systems still remains unlikely.

Directory protocol bandwidth utilization scales more graciously, but directory protocols introduce large structures that consume precious on-chip area and leakage power. Conventional directory implementations either have large area overhead, or suffer poor performance for some sharing patterns, or are simply impractical for large CMPs. Of the two most common directory designs, duplicate tag directories, such those used by Niagara 2 [2], become impractical for larger CMPs due to their highly associative lookups. Sparse directories, the other common directory design, incur large area, bandwidth, or performance overheads [3, 10, 14, 25]. Additionally, sparse directories may restrict which blocks can be cached simultaneously and thus suffer from premature block evictions that can hurt performance, bandwidth, and power. This work attempts to develop a scalable coherence directory that has a minimal impact on performance and bandwidth utilization compared to conventional designs while mitigating the area, power, and complexity overheads.

This work revisits directory design starting from the basic required functionality — in essence, maintaining coherence in a CMP consists of performing set-membership tests on all private caches. This work combines two well known observations: 1) as sparse directory designs have demonstrated [3, 10, 14, 25], the set-membership tests do not have to be precise – a conservative estimate is sufficient for correctness, but an accurate test is desired for performance; 2) Bloom filters are space-efficient structures for performing set-membership tests [6] and have been applied to solve countless problems across many domains [7, 24].

This work proposes the *Tagless Coherence Directory* (TL), a novel, scalable directory structure based on Bloom filters. Conceptually, TL is a grid of Bloom filters, with one column for each CMP core, and one row for each cache set. Each Bloom filter tracks the blocks of one cache set of one core. Given a block, accessing the Bloom filters in the corresponding row in parallel produces a

MICRO'09, December 12–16, 2009, New York, NY, USA.

sharing vector that represents a *superset* of all of the sharers of the block. In practice, relatively small Bloom filters (e.g., 256 bits for 16 cores with 16-way cache sets) are sufficiently precise that the average sharing vector is the *true* set of sharers. TL overcomes the challenge of element removal in Bloom filters by dedicating a Bloom filter to each cache set, allowing it to easily remove elements from the represented set without the area overhead of counting Bloom filters, a commonly used solution to this problem [13, 24].

Experimental results using commercial and scientific workloads running on a 16-core CMP show that TL uses 44% less dynamic energy per lookup, 57% less leakage power and 48% less area than a conventional sparse directory with no statistically significant performance loss, and incurs an average bandwidth increase of only 2.5%. Analytical models predict that TL will scale well for CMPs with increasing numbers of cores.

The rest of this paper is organized as follows: Section 2 describes the motivation and challenges associated with directory protocols for CMPs. Section 3 describes TL. Section 4 reviews three conventional directory designs. Section 5 evaluates TL's effectiveness. Section 6 reviews related work, and Section 7 summarizes this work.

2. MOTIVATION

The cache coherence problem occurs any time multiple locations have copies of the same data block. In a CMP, solving this problem requires the ability to identify all of the cores that have cached copies of a given block. Each cache contains a unique *set* of blocks, and the coherence protocol, in essence, consists of performing set-membership tests to find all copies of a given block.

Snoop coherence protocols make these set-membership tests explicit by directly probing each cache. The resulting bandwidth and energy demands, while reasonable for small CMPs, become overwhelming with tens or hundreds of cores. Recent proposals allow snooping to scale to slightly larger CMPs [4, 8, 12, 23, 24, 29], but their effectiveness has not been demonstrated for the very large CMPs expected to result from increased feature-size shrinking.

Directory coherence protocols rely on a conceptually central structure to track which cores share each block. This reduces bandwidth requirements, but adds an area overhead for the structure, and also increases the latency of requests that must now go through the directory structure.

The first directory protocol was proposed over 30 years ago [33]; however, as CMPs replace traditional multiprocessors, they introduce new challenges and opportunities for directory designs. CMPs have limited area and power budgets. The less area and power a directory uses, the more area and power available to other components. Furthermore, on-chip integration in CMPs provides fast on-chip caches, low latency interconnects, and the possibility of implementing structures that would be less practical with discrete components.

A CMP directory should solve the coherence problem by taking advantage of the opportunities CMPs offer while addressing the new challenges. The directory structure should require little area and energy while providing low-latency lookups, and the overall directory protocol should have low-latency requests and low bandwidth utilization. The next section describes a new directory design that attempts to meet these objectives.

3. TAGLESS COHERENCE DIRECTORY

The Tagless Coherence Directory (TL) provides a space-efficient directory structure that is practical to implement and requires only minor modifications to the coherence protocol. The remainder of this section describes the conceptual directory structure (Section 3.1), its usage (Section 3.2), a practical implementation for it (Section 3.3), and an analytical model of its precision (Section 3.4). Without loss of generality, the description uses an example CMP with 16 cores, 48-bit physical addresses, and 1MB, 16-way set-associative, private, inclusive L2 caches. While many CMPs include large shared caches in addition to private caches, these shared caches have the same problem of needing to maintain coherence amongst the private caches. Using large private caches and no shared cache emphasizes the performance and bandwidth impact of the directory since requests for shared, read-only data cannot be provided by the shared cache.

3.1 Conceptual Structure

Maintaining coherence consists of performing a set-membership test on each cache to determine which ones have copies of a given block. Bloom filters are a proven, space-efficient structure that uses a conservative, implicit representation of set contents to perform set-membership tests [6]. A coherence directory could use a Bloom filter to represent the contents of each cache, as shown in Figure 1(a). While their simple, space-efficient structure makes Bloom filters seem well-suited for a low area, low power CMP coherence directory, they also present two challenges: 1) the conservative representation of set contents, and, 2) the difficulty of removing items from the represented set.

Bloom filters provide an inherently conservative representation of set contents. The filter's structure is exemplified by the *partitioned* Bloom filter shown in Figure 1(b) which uses a number of bit vectors, each with its own hash function, to represent the members of a set [24, 31].[1] A given item maps to one bit in each bit vector. Adding an item to the set requires setting the corresponding bits in each bit vector. A set-membership test consists of evaluating each hash function for the test item and performing a logical-AND of the selected bits. A zero result indicates the item is definitely *not* a member of the set. A one indicates the item *may* be a member of the set, hence the representation is conservative. A *false positive* occurs when a Bloom filter wrongly indicates the set may contain an item. When using Bloom filters to represent cache contents, false positives do not affect coherence protocol correctness, but they can affect bandwidth utilization and performance.

Removing items from the set represented by a Bloom filter is difficult; as items are removed from the set, a bit should only be cleared when none of the remaining items map to that bit. A naïve solution would check all remaining items to determine which bits can be cleared. For a coherence directory, this amounts to searching an entire cache each time it evicts a block. Counting Bloom filters replace each bit in the filter with a counter [13, 24]. Adding an item to the set increments the counters for each hash function, and removing an item decrements them. This solution significantly increases the size of the filter; even the pathological case of using two-bit counters requires twice as many bits. These bits could be better utilized to increase the size of the Bloom filter bit vectors, thus improving their accuracy.

TL takes advantage of normal cache operation to avoid the overheads of counting Bloom filters. Typical caches are set-associative arrays where each access reads one set of the tag array. If a Bloom filter only represents a single set of a single cache (referred to hereafter as a *cache-set*), then on an eviction, the cache can evaluate all the hash functions for each block in the set in parallel and determine which filter bits can be cleared.

[1] A traditional, non-partitioned Bloom filter allows the hash functions to share a bit vector and thus requires a multi-ported structure.

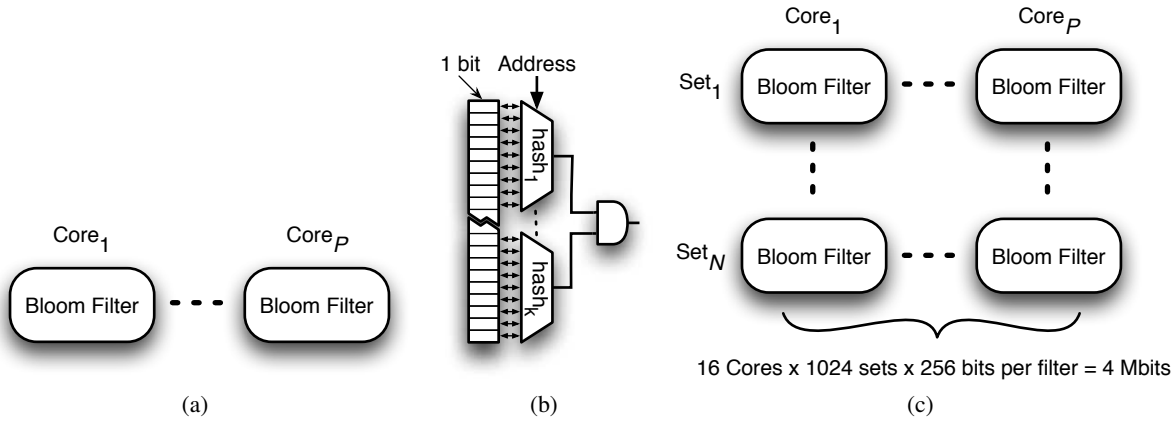


Figure 1: (a) A coherence directory using Bloom filters. (b) Implementation of a single Bloom filter. (c) TL concept.

Conceptually, as shown in Figure 1(c), TL consists of an $N \times P$ grid of Bloom filters. Each of the P columns of filters represents one cache in the CMP, and each column contains N filters, one for each set in the corresponding cache. On a directory lookup, TL probes a row of P Bloom filters to produce a P-bit *sharing vector* that indicates which caches *might* contain copies of the requested block. Our example CMP would have a 1024×16 grid of Bloom filters. The results of Section 5 show that each Bloom filter should use four hash functions and 64 bits for each vector, resulting in a total of $1024 \times 16 \times 4 \times 64 = 4$Mbits. For comparison, the tag arrays for these private caches require a total of 16 caches $\times$ 1024sets$\times$16ways$\times$32-bit tags $=$ 8Mbits. Although conceptually, the 4Mbits form 16K separate Bloom filters, Section 3.3 shows how these filters can be implemented as a few SRAM arrays.

3.2 TL Operation

A TL lookup produces a sharing vector that indicates which cores *might* share the requested block. Attempting to initiate a cache-to-cache transfer might result in requesting data from a node that does not have a valid copy, or there might not be *any* valid copy on-chip. Additionally, a simple MOESI protocol does not track ownership for clean blocks. The imprecise sharing vectors and lack of ownership require some modifications to the coherence protocol. Some existing directory designs, as described in Section 4.2, also maintain potentially imprecise sharing information. The only new scenario that TL must handle is the case when the directory indicates a potential on-chip sharer when there are no valid copies on-chip. As shown in the following sections, this scenario requires little extra complexity, and, thus, TL should not require excessive verification costs. The remainder of this section describes one possible protocol implementation that is evaluated in Section 5.

3.2.1 Read Requests

When the directory receives a read request, there are two cases:

1. The sharing vector indicates no other cores share the block. The directory forwards the request to memory.

2. The sharing vector indicates one or more caches might share the requested block. The directory snoops each potential sharer in turn until one of two things happens:

   (a) A cache containing the block is found, in which case the data is forwarded to the requester.

   (b) All potential sharers have been snooped and none have the block. The directory then forwards the request to memory.

When the requester completes the read request, it sends a final acknowledgement to the directory. At this time a conventional directory would update it's sharing information; in TL the directory does this by adding the block in the set represented by the appropriate Bloom filter. As described, TL uses a *serial snooping* technique [30] to resolve read requests with minimal bandwidth at the expense of potentially extra latency. The evaluation in Section 5 confirms that TL is sufficiently precise that serial snooping does not impose any noticeable performance penalty. Alternatively, a multicast approach could be used that would result in more bandwidth on average, but potentially lower latency. A multicast approach would either require tracking ownership for clean data, or require all sharers to provide data when receiving the multicast, or introduce additional latency to select a single node to supply data.

3.2.2 Write Requests

Handling writes requires an extension of the underlying conventional directory protocol because TL does not know the exact set of sharers or whether there is a sharer in the Owner state. Because clean data does not have an on-chip owner, the sharers cannot decide independently who will provide data to the requester. Similarly, the directory cannot guarantee that the node selected to provide data actually has a copy of the data, or if there is even a copy on-chip. Before examining the modified protocol, consider how the conventional protocol handles writes:

1. A node, the *writer*, initiates a write by sending an appropriate request to the directory.

2. If there are no cached copies, the directory asks memory to provide the data directly to the writer and lets the writer know of the request. Otherwise, the directory takes three actions in parallel: a) it requests that all sharers invalidate their copies; b) it asks one of the sharers, the *provider*,[2] to send the data to the writer prior to invalidating; and c) it notifies the writer how many sharers there are.

[2]In our implementation, the directory chooses the lowest numbered core that is a potential sharer as the provider, but other selection algorithms could be used.

3. The writer waits until it receives acknowledgements from all the sharers. One of these acknowledgements will contain the data. Alternatively, the writer will receive the data from memory.

4. The writer notifies the directory that the transaction is complete.

TL operates identically except for when it selects, in step 2(b), a *provider* that does not have the data. In practice, the probability of the selected sharer not having a copy is very low; thus, the common case for TL proceeds the same as the original protocol. In the less common case, when the *provider* does not have a copy, it sends a NAck to the writer instead of an acknowledgement. This leaves three possible scenarios:

1. **No copy exists:** The writer will wait to receive all of the acknowledgements and the NAck, and since it has not received any data, it will send a request to the directory which then asks memory to provide the data.[3]

2. **A clean copy exists**: For simplicity, this is handled identically to first case. All sharers invalidate their copies and data is retrieved from memory instead. This may result in a performance loss if it happens frequently, but Section 5 shows that no performance loss occurs in practice.

3. **A dirty copy exists:** One of the other sharers, the *owner,* will be in the Modified or Owned state. The owner cannot discard the dirty data since it does not know that the provider has a copy. Accordingly, the owner *always* includes the data in its acknowledgement to the writer. In this case, the writer will receive the provider's NAck and the owner's valid data and can thus complete its request. If the provider has a copy as well, the writer will receive two identical copies of the data. Section 5.5 demonstrates that this is rare enough that it does not result in any significant increase in bandwidth utilization.

In all cases, all transactions are serialized through the directory, so no additional race considerations are introduced. Similarly, the underlying deadlock and livelock avoidance mechanisms remain applicable.

[3]This request is serialized through the directory to avoid races with evicts.

3.2.3 Evictions and Invalidates

When a block is evicted or invalidated TL tries to clear the appropriate bits in the corresponding Bloom filter to maintain accuracy. Each Bloom filter represents the blocks in a single set, making it simple to determine which bits can be cleared. The cache simply evaluates all the hash functions for all remaining blocks in the set. This requires no extra tag array accesses as all tags are already read as part of the normal cache operation. Any bit mapped to by the evicted or invalidated block can be cleared if none of the remaining blocks in the set map to the same bit. When using an inclusive L2 cache, as in our example, it is sufficient to check only the one L2 cache set. If the L2 cache is not inclusive, then additional L1 accesses may be necessary.

3.3 A Practical Implementation

Conceptually, TL consists of an $N \times P$ grid of Bloom filters, where N is the number of cache sets, and P is the number of caches, as shown in Figure 1(c). Each Bloom filter contains k bit-vectors. For clarity we will refer to the size of these bit vectors as the number of $buckets$. In practice, these filters can be combined into a few SRAM arrays provided that a few conditions are met: 1) all the filters use the same set of k hash functions, and 2) an entire row of P filters is accessed in parallel. These requirements do not significantly affect the effectiveness of the Bloom filters.

Conceptually, each row contains P Bloom filters, each with k hash functions. Figure 2(a) shows one such row. By always accessing these filters in parallel using the same hash functions, the filters can be combined to form a single lookup table, as shown in Figure 2(b), where each row contains a *P-bit sharing vector*. As a result, each lookup directly provides a P-bit sharing vector by AND-ing k sharing vectors. Since every row of Bloom filters uses the same hash functions, we can further combine the lookup tables for all sets for each hash function, as shown in Figure 3(a).

These lookup tables can be thought of as a "sea of sharing vectors" and can be organized into any structure with the following conditions: 1) a unique set, hash function index, and block address map to a single unique sharing vector; and 2) k sharing vectors can be accessed in parallel, where k is the number of hash functions. An example implementation uses a single-ported, two-dimensional SRAM array for each hash function, where the set index selects a row, and the hash function selects a sharing vector, as shown in Figure 3(b).

As a practical example, Section 5 shows that a TL design with four hash tables and 64 buckets in each filter performs well for

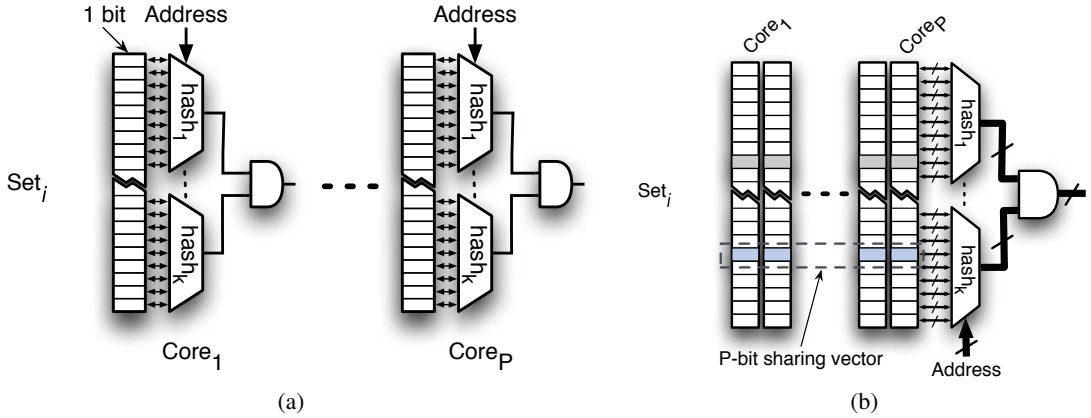


Figure 2: (a) A naïve implementation with one filter per core. (b) Combining all of the filters for one row.

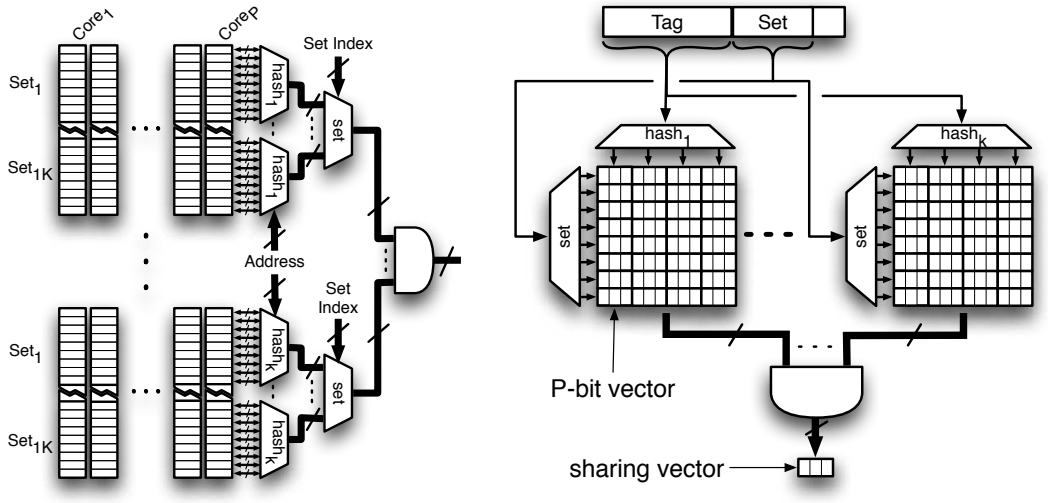


Figure 3: (a) Per set and per hash function Bloom filter combining. (b) A practical implementation.

our 16-core CMP example. Each hash table in this design contains 1K sets $\times$ 64 buckets/set $\times$ 16 bits/bucket $=$ 1M bits. The total number of bits is: 4 hash functions $\times$ 1Mbit per table $=$ 4Mbits. Each of these tables can be banked, for example, by distributing the sets. Assigning one bank per CMP core results in 16 banks of 64 sets $\times$ 64 buckets/set $\times$ 16 bits/bucket $=$ 64K bits. This is equivalent to accessing a 16-bit word from an 8KB SRAM array with 128-byte lines. The entire directory requires four such accesses in parallel.

3.3.1 Network Overhead

In addition to potentially sending unnecessary snoop or invalidate messages as a result of imprecise information in the directory, there are some additional fixed network overheads. Both eviction and invalidate acknowledgement messages contain a small bit vector indicating which hash tables can clear bits in their sharing vectors. For the configurations studied in Section 5, these are small bit vectors (3-7 bits) and can piggyback within the original 64-bit message. Invalidate acknowledgements are sent to the writer, which collects the bit vectors and sends them to the directory with the final write acknowledgement, adding one bit per core per hash function. As a result, write acknowledgements increase from 64-bits to 128-bits in our configurations.

3.3.2 Other Required Hardware

On an eviction or invalidation, the cache needs to search the affected set for blocks that map to any of the sharing vectors used by the evicted block. This requires evaluating all of the directory hash functions for each block in the set, and comparing the results with the hash values for the evicted block. When simple hash functions are used, the necessary circuitry requires only a small amount of additional area and energy compared to the overhead of the directory itself. Additionally, since invalidate acknowledgements are collected by the writer before being sent to the directory, the size of each miss-status handling register (MSHR) has to increase so that a writer can accumulate multiple bit-vectors (up to $P-1$ vectors in the worst case), as mentioned in Section 3.3.1, to indicate which bits to clear in the directory.

3.4 Analytical Model

In TL, false positives in the sharing vectors can result in increased bandwidth and reduced performance. An analytical model can be useful in predicting TL's behavior especially for larger CMPs where simulation becomes prohibitively expensive. For simplicity, we assume randomly distributed memory accesses and pair-wise independent uniform hash functions. Following the analysis of previous work [7], the probability of a Bloom filter having a false positive is given by

$$P(\text{false positive}) = \left(1 - \left(1 - \frac{1}{b}\right)^{a}\right)^{k} \quad (1)$$

where b is the number of buckets for each hash function in the filter, k is the number of different hash functions used, and a is the associativity of the cache. Ignoring sharing, we can extend Broder and Mitzenmacher's analysis to an N-core CMP using a series of $N-1$ Bernoulli trials, one for each of the additional cores in the CMP. As a result, the expected number of false positive bits (FPB) in the sharing vectors becomes:

$$E[\text{FPB}] = (N-1)\,P(\text{false positive}) \quad (2)$$

$$= (N-1)\left(1 - \left(1 - \frac{1}{b}\right)^{a}\right)^{k} \quad (3)$$

Section 5.2 validates this analytical model against simulation results. Section 5.6 then uses it to demonstrate how TL is expected to scale with increasing numbers of cores.

4. CONVENTIONAL DIRECTORIES

Conventional directories offer three potential solutions for CMP directories: *Duplicate tag directories* require a small amount of area and achieve good performance and bandwidth, but they are impractical for CMPs with many cores. *Sparse directories* are more practical, but because they restrict which blocks can be cached simultaneously they either impose large area overheads, or sacrifice performance and bandwidth. *In-cache* directories offer a practical design with no block placement restrictions, but they have area,

bandwidth, and performance overheads similar to sparse directories and can only be use with inclusive, shared caches.

4.1 Duplicate Tag Directories

A conceptually simple directory implementation replicates the tag arrays of each private cache at a "central" location [33]. This design uses the smallest *explicit* representation of the cache contents without constraining which blocks can be cached simultaneously. The Piranha prototype [5] and the Niagara 2 CMP [2] both use such a *duplicate tag directory* (referred to hereafter as DUP). However, for CMPs with many cores, DUP requires an impractical, highly associative lookup. For our example CMP, DUP has to compare 256 tags simultaneously.

4.2 Sparse Directories

Sparse directories with full-map vectors (referred to hereafter as SPARSE$_{FULL}$) use a set-associative directory structure where each entry contains a block tag and a vector with one bit per core identifying the cores caching the block [14, 25]. The associativity of SPARSE$_{FULL}$ is much lower than the aggregate associativity of the caches it tracks. As a result, allocating a new entry can require evicting an existing entry and invalidating all cached copies of that block. This restricts which blocks can be cached simultaneously, potentially hurting performance, bandwidth and power. To minimize the effect of these placement restrictions, SPARSE$_{FULL}$ usually contains at least one entry for each cached block. The area of SPARSE$_{FULL}$ grows as $O(P^2)$ making it impractical for large CMPs.

Coarse vectors [14], segment directories [10], the SGI Origin directory [18], and limited pointer directories [3] use more compact directory entry formats. Such designs (referred to hereafter collectively as SPARSE$_{COMP}$) can only accurately represent a limited number of sharing patterns. When other sharing patterns occur, the directory either restricts sharing and suffers a performance loss, or maintains a conservative sharing representation and relies on a broadcast or multicast protocol to attempt to maintain performance at the cost of higher bandwidth utilization. SPARSE$_{COMP}$ designs use less area, but still have placement restrictions and may hurt performance and bandwidth.

TL provides imprecise sharing information similar to some SPARSE$_{COMP}$ designs. However, TL embeds the sharing information in Bloom filters and does not explicitly associate sharing information with specific block addresses. Additionally, the SPARSE$_{COMP}$ and TL differ significantly in the conditions that result in increased bandwidth or performance degradations. Existing SPARSE$_{COMP}$ designs perform poorly for specific sharing patterns. Regardless of the block address, any block being shared by specific combinations of nodes results in increased bandwidth. Typically, this behavior is application dependant, making certain applications always perform poorly with certain SPARSE$_{COMP}$ designs. TL, on the other hand, only performs poorly for specific combinations of block addresses in different nodes. More specifically, because there are seperate filters for each cache set, only certain combinations of physical page addresses cause TL to perform poorly. This makes it very difficult to purposefully construct any application that will degrade the performance of TL, and typical applications will only have degraded performance for a small fraction of physical address combinations. Ultimately, the likelihood of truly pathological behavior with TL is vanishingly small, while any SPARSE$_{COMP}$ design will always exhibit poor behavior for applications with sharing patters that it cannot accurately represent.

4.3 In-Cache Directories

Censier and Feautrier proposed a coherence directory for traditional multiprocessors that associated each memory block with a bit-vector representing which nodes had cached copies of that block [9]. CMPs with inclusive shared caches can include such directory entries with each cache block to track which cores share each block. This avoids the cost of accessing a seperate directory structure, and avoids adding new restrictions on which blocks can be cached simultaneously. However, this design does not apply to CMPs without shared caches, or with non-inclusive shared caches.

Also, since shared caches are typically many times larger than the total capacity of the private caches, most of the directory entries in an in-cache directory contain no information. Usually a SPARSE$_{FULL}$ or SPARSE$_{COMP}$ design would be more space-efficient. An in-cache directory is only more space-efficient when

$$\textit{entry bits} > \frac{\textit{tag bits}}{\left(\frac{\textit{shared cache blocks}}{\textit{private cache blocks} \times \textit{number of cores}} - 1\right)} \quad (4)$$

For modern CMPs, such as the Intel Nehalem [1], this only holds true when the directory entry is less than one seventh the size of the tag. For a reasonably sized tag, SPARSE$_{FULL}$ is more space-efficient than an in-cache directory with full bit-vectors for CMPs with more than four or five cores. Some in-cache directories with very small entries may be more space-efficient than comparable SPARSE$_{COMP}$ designs, but such designs result in increased bandwidth and potentially lowered performance. Since SPARSE$_{FULL}$ and SPARSE$_{COMP}$ are generally more space-efficient, the remainder of the paper focuses on these designs.

4.4 TL vs. Conventional Directories

TL is an alternative to the three conventional coherence directory designs: DUP, SPARSE$_{FULL}$, and SPARSE$_{COMP}$. As mentioned, DUP scales poorly due to its highly associative lookups for large CMPs. SPARSE$_{FULL}$ scales poorly due to its large area overhead. SPARSE$_{COMP}$ represents an entire family of possible designs, but the performance and bandwidth of such designs varies depending on specific application, and, secondary effects aside, these designs offer, at best, the same performance as DUP and SPARSE$_{FULL}$ with more bandwidth for some workloads. Also, both SPARSE$_{FULL}$ and SPARSE$_{COMP}$ place restrictions on which blocks can be cached simultaneously.

TL avoids the complexity of DUP by requiring only simple direct-mapped SRAM lookups, and the implicit representation of each cache set avoids the placement restrictions of SPARSE$_{FULL}$ and SPARSE$_{COMP}$. The next section shows that TL requires less area than all three conventional designs, while achieving the same performance as DUP and SPARSE$_{FULL}$ with only slightly more bandwidth than SPARSE$_{FULL}$.

5. EVALUATION

This section demonstrates the benefits of TL. Section 5.1 describes the evaluation methodology. Section 5.2 performs a design-space exploration to determine hash table configurations that result in few false positives. Based on these results, Section 5.3 selects promising configurations and demonstrates that properly configured TL designs offer performance comparable to conventional directories. Section 5.4 shows that these designs provide significant energy and area savings. Section 5.5 shows that these benefits come at the cost of only a small increase in bandwidth utilization. Finally, Section 5.6 demonstrates that TL scales well with increasing numbers of cores.

5.1 Methodology

Simulations modeled a tiled, 16-core CMP with private, inclusive L2 caches. The simulator is based on the full-system simulator *Flexus* [16] which uses Virtutech Simics [20]. Each tile contains a processor core with private L1 and L2 caches as described in Table 1, as well as some portion of a distributed directory. The directory banks are interleaved at 64-byte granularity, and directory lookups have a 3-cycle latency for all designs as estimated by CACTI assuming 4GHz processors (see Section 5.4). The L2 caches use a MOESI coherence protocol. The tiles are connected using a mesh network with 64-bit links and using XY routing. Off-chip memory is connected to four memory controllers which are distributed around the edge of the mesh network. The network latency is five cycles per hop [17, 26]. A number of commercial and scientific workloads were simulated, as described in Table 2. All simulated systems ran the Solaris 8 operating system.

To prune the design space, the initial results in Section 5.2 rely on functional simulations. These results depend mostly on the set of memory addresses accessed by each core, and are mostly independent of the order or timing of these accesses. Functional simulations executed 500 million instructions per core. Sections 5.3 and 5.5 use detailed timing simulations to evaluate the most practical TL designs. Timing simulations use the SMARTS sampling methodology [36]. Each sample measurement involves 100k cycles of detailed warming followed by 50k cycles of measurement collection. Results are reported with errors with a 95% confidence interval. Matched-pair sampling is used to measure change in performance [11]. Performance is measured as the aggregate number of user instructions committed each cycle [35]. Average results are shown per workload class (i.e., OLTP, DSS, Web, and Scientific), and overall averages are taken across all workload classes.

5.1.1 Baseline Designs

Two baselines represent conventional directory configurations:

1. The **Duplicate Tag Directory** (Section 4.1) baseline, or **DUP**, requires a minimal amount of storage capacity without any cache placement restrictions and is similar to the directory used in the Niagara 2 processor [2]. However, it requires a 256-way tag comparison that makes it impractical.

2. The **Sparse Directory** (Section 4.2) baseline, or **SPARSE$_{\text{FULL}}$**, is a more practical directory implementation. The baseline configuration has 16K sets, with 16 ways each, and each entry is a 16-bit vector. Using a full bit vector results in the best performance and lowest bandwidth, but designs with smaller entries are also possible. Any sparse directory design that uses a compressed directory entry format (e.g., coarse vector, or limited pointer) is referred to as **SPARSE$_{\text{COMP}}$** or **SPARSE$_{\text{COMP-}n}$** where n is the number of bits in each directory entry.

Simulations use the same three cycle latency for all directories. As Section 5.4 demonstrates, the DUP design will likely have a higher latency than the other directory designs.

5.2 Hash Function Effectiveness

The key to TL's good performance is eliminating collisions so that the directory can identify the *true* set of sharers and not simply a *superset* of sharers. The collision rate can be controlled by varying the type and number of hash functions used, and by varying the size of each hash table. The *number of false positive bits* (FPB) per directory lookup, that is the number of cores that the directory mistakenly reports as having a cached copy, was measured to compare different designs. A lower FPB results in better performance and bandwidth utilization.

Table 1: Processor core configuration

Branch Predictor	Fetch Unit
8K GShare, 16K bi-modal, and 16K selector 2K entry, 16-way BTB 2 branches per cycle	Up to 8 instr. per cycle 32-entry Fetch Buffer
	Scheduler
	256/64-entry ROB/LSQ
ISA & Pipeline	**Issue/Decode/Commit**
UltraSPARC III ISA 4 GHz, 8-stage pipeline out-of-order execution	any 4 instr./cycle
	Main Memory
	3GB, 240 cycles
L1D/L1I	**Private UL2**
64KB, 64B blocks 4-way set-associative LRU replacement 1 cycle latency	1MB, 64B blocks 16-way set-associative LRU replacement 2 cycle tag latency 14 cycle data latency

Table 2: Workload Descriptions

Online Transaction Processing (OLTP) — TPC-C	
DB-A	100 warehouses (10GB), 16 clients
DB-B	100 warehouses (10GB), 64 clients
Decision Support (DSS) **TPC-H on a commercial database system**	
Qry 2	Join-dominated, 450 MB buffer pool
Qry 6	Scan-dominated, 450 MB buffer pool
Qry 17	Balanced scan-join, 450 MB buffer pool
Web Server (Web) — SPECweb99	
Apache 2.0	16K connections, FastCGI, worker threading
Zeus 4.3	16K connections, FastCGI
Scientific —- SPLASH-2	
EM3D	600K nodes, degree 2, span 5, 15% remote
Ocean	1026×1026 grid, 9600 seconds
Sparse	4096×4096 matrix

The bars in Figure 4 show the average FPB for several TL configurations, and the dashed lines show the area of each design relative to SPARSE$_{\text{FULL}}$. Configurations are labeled as $\texttt{B}-\texttt{h}_1+\ldots+\texttt{h}_\texttt{N}$, where B is the number of buckets allocated to each hash table, and $\texttt{h}_1+\ldots+\texttt{h}_\texttt{N}$ is a list of hash functions used (one per table). The goal is not to determine the best possible hash function combinations, but merely to see the trends for different numbers of hash tables. For further clarity, the number of buckets per set in each hash table is also indicated by the shading of the bars. The solid line indicates the predictions of the analytical model of Section 3.4.

There are three types of hash functions, each producing a bucket index of $m = \log(\text{bucket size})$ bits:

sN: The bucket index uses m bits from the tag starting at bit position N.

xor: The tag is split in half and the two portions are xor'd and the lower m bits form the bucket index.

prime: The number of buckets is reduced to the nearest, smaller prime number. The bucket index is the tag modulo this prime number.

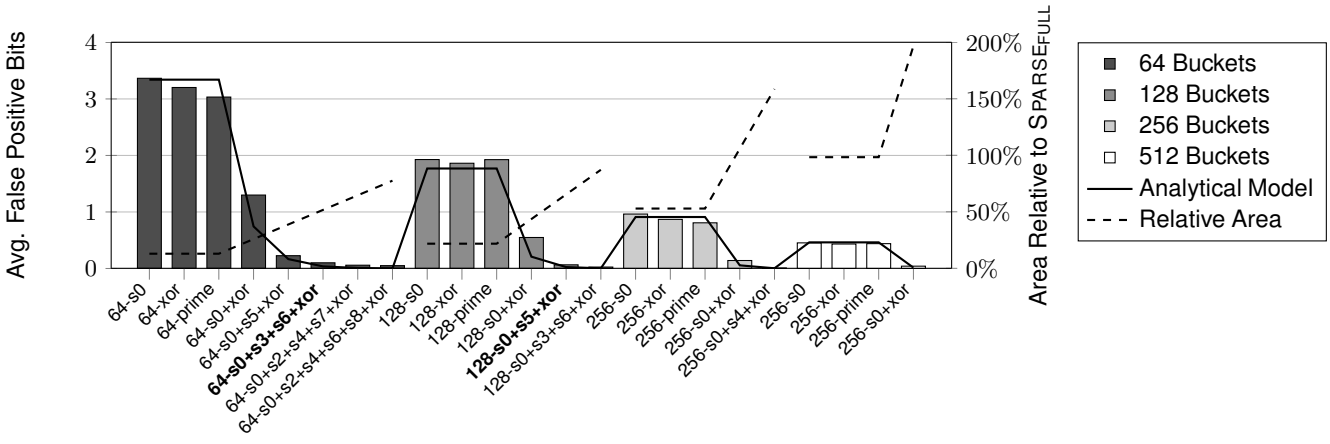


Figure 4: The average number of false positive bits in the sharing vector for different TL configurations as measured by simulation and as predicted by the analytical model. The dashed line shows area relative to SPARSEFULL.

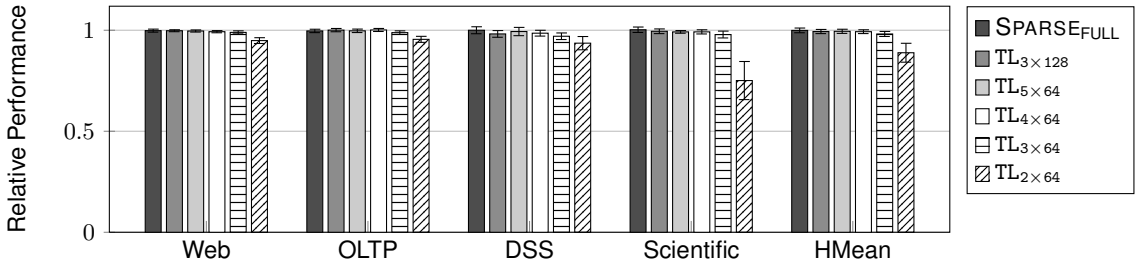


Figure 5: Performance of various configurations relative to DUP.

Of the three types of hash functions, sN has the lowest cost since it directly uses part of the address. The *xor* hash function is slightly slower, but requires relatively little extra logic. The *prime* hash function should produce a distribution closer to random uniform values, but it is slower and requires more logic.

The results of Figure 4 show that several three- and four-table TL designs using sN or *xor* result in very low average FPB while offering significant area savings compared to SPARSEFULL. Specifically, `64-s0+s3+s6+xor` and `128-s0+s5+xor` result in area savings of 48% and 35% respectively (as indicated by the dashed line) while only having 0.1 and 0.025 false positive bits on average (as indicated by the bars). Single-table designs perform similarly irrespective of the hash function, and are inferior to two-table designs with the same number of total entries (e.g., `128-prime` vs. `64-s0+xor`). Comparing different designs with the same number of total buckets divided into different numbers of hash tables (e.g., `64-s0+s3+s6+xor` and `128-s0+xor`), the design with more smaller tables performs better than the design with larger tables.

Also in Figure 4, the solid line shows the expected FPB predicted by the analytical model of Section 3.4. While the model assumes randomly distributed accesses and independent hash functions, its predictions are accurate. Thus, Section 5.6 uses it to argue that TL scales well with increasing numbers of cores.

The remainder of the paper focuses on the five most promising designs, namely `64-s0+xor`, `64-s0+s5+xor`, `64-s0+s3+s6+xor`, `64-s0+s2+s4+s7+xor`, and `128-s0+s5+xor`. Also, the simpler naming convention of $TL_{n\times B}$ is used, where `n` is the number of hash functions, and `B` is the number of buckets allocated to each cache set in each hash table.

5.3 Performance Comparison

This section demonstrates that sufficiently precise TL designs offer performance comparable to conventional directory designs. Figure 5 shows the performance of SPARSEFULL (leftmost bar) and of a number of TL configurations normalized to DUP. The figure does not show performance for any SPARSECOMP design as, secondary effects aside, any such design would perform at most as well as SPARSEFULL.

SPARSEFULL suffers from no significant performance loss even though it restricts which blocks can reside simultaneously in the L2 caches and can introduce additional L2 cache misses compared to the other designs. The first three TL configurations suffer no statistically significant performance loss either. For $TL_{4\times64}$, the worst individual workload is *EM3D* which sees a negligible slowdown of $2.9\% \pm 1.3\%$. Based on these results, the rest of the paper focuses on $TL_{3\times128}$ and $TL_{4\times64}$.

5.4 Delay/Area/Power/Energy Comparison

Having shown that TL offers similar performance to DUP and SPARSEFULL, this section demonstrates that TL requires less energy and area. CACTI 5.3 [34] was used to model a single bank of DUP, SPARSEFULL, SPARSECOMP-8, $TL_{4\times64}$ and $TL_{3\times128}$, in a 32nm technology. SPARSECOMP-8 uses just eight bits per vector which severely restricts what sharing patterns it can represent. This design illustrates that the TL designs are superior even to aggressive SPARSECOMP designs. Table 3 shows in order from left

Table 3: CACTI estimates for various conventional and TL directories.

Configuration	Delay (ns)	E_{read} (pJ)	$P_{leakage}$ (mW)	Area (mm^2)	KBits of Storage
$TL_{4\times64}$	0.24	9.64	26.63	0.0845	256
$TL_{3\times128}$	0.26	9.11	37.53	0.1068	384
SPARSE$_{FULL}$	0.55	17.76	72.04	0.1634	704
w/o comparators	–	17.21	61.45	–	–
SPARSE$_{COMP-8}$	0.55	13.60	61.01	0.1364	464
w/o comparators	–	13.05	50.42	–	–
DUP	0.85	127.90	2,785.75	0.3967	576
w/o comparators	–	90.25	74.64	–	–

to right: the configuration name, the access time, dynamic read energy, leakage power, area, and number of Kbits of storage for each design. The TL designs require 16 3- or 4-input NAND gates for AND'ing sharing vectors and six or seven 2-input XOR gates to calculate the xor hash function, but the area and energy of the directory should be dominated by the sharing vector SRAM array. By default CACTI uses comparators that have high leakage power, but other, more energy efficient comparator implementations exist. This is of particular concern for the highly-associative DUP. Accordingly, Table 3 reports energy and power with and without the comparators. For clarity, the discussion focuses on the measurements without the comparators.

The two TL designs are first compared against each other, and then the best is compared with the conventional directories. $TL_{4\times64}$ requires 20% less area and 29% less leakage power than $TL_{3\times128}$, but uses 6% more dynamic energy per lookup. The higher dynamic energy may result in higher average power consumption, but as feature sizes continue to shrink, leakage power is becoming a more dominant constraint, making the $TL_{4\times64}$ the better design overall.

Compared to the conventional designs, TL is significantly better. While DUP has been promoted as a good option because it requires fewer bits, the results from CACTI demonstrate that DUP is worse than all other designs for all metrics. Although the CACTI implementation of DUP may not be optimal, the differences with TL are large and further optimizations are challenging due to its high associativity. Relative to SPARSE$_{FULL}$, $TL_{4\times64}$ uses 44% less dynamic energy for reads, and consumes 48% less area and 57% less leakage power. Compared to SPARSE$_{COMP-8}$, $TL_{4\times64}$ uses 26% less dynamic energy, 47% less leakage power, and 38% less area. Although timing simulations were not performed for SPARSE$_{COMP-8}$, at best it will offer the same performance and bandwidth as SPARSE$_{FULL}$, and in practice it will likely be worse than SPARSE$_{FULL}$.

In terms of overall area, using $TL_{4\times64}$ reduces the area of each CMP tile, including the core, L2 cache and directory bank, by 0.33%, 0.17%, and 1.66% compared to SPARSE$_{FULL}$, SPARSE$_{COMP-8}$, and DUP, respectively. These values are based on CACTI's estimate of a 1MB cache and the assumption that the processor core (including L1 caches) is about 3.3 times larger than the 1MB cache [28].

Finally, the delay of the TL designs is half that of SPARSE$_{FULL}$ and SPARSE$_{COMP-8}$, and less than a third of the delay of DUP. This allows time for the extra logic necessary to compute the hash functions and to perform the AND operation to form the final sharing vector. Depending on the delay of this logic, the latency of TL might be slightly lower than SPARSE$_{FULL}$ or SPARSE$_{COMP-8}$, resulting in a slight performance benefit.

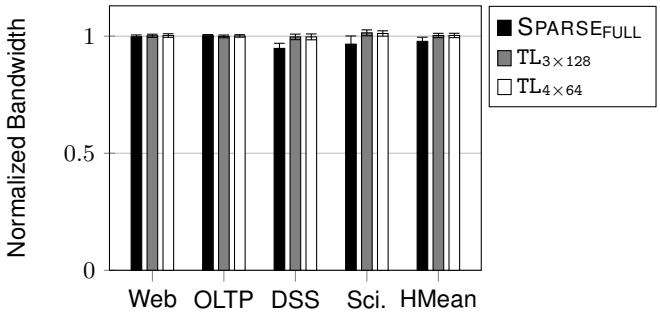
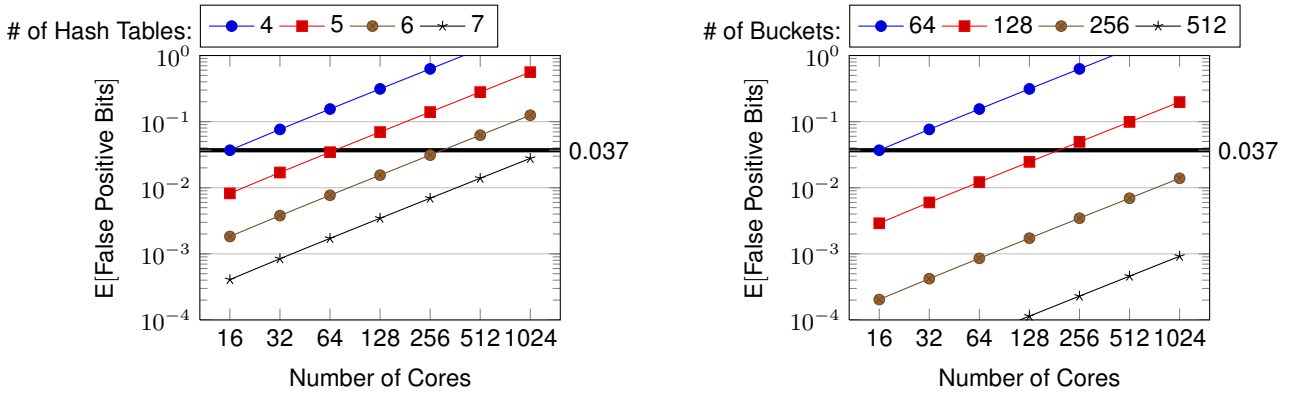


Figure 6: On-chip network utilization normalized to DUP.

5.5 Bandwidth Utilization

The on-chip interconnect consumes a significant amount of power, thus it is important to ensure that the benefits of TL do not come at the expense of any significant increase in bandwidth utilization. Additional bandwidth results from transmitting extra data for some write transactions (Section 3.2.2), from false-positives in the sharing vectors, and from increasing the size of write acknowledgements from one to two flits (Section 3.3.1). Figure 6 compares the on-chip network utilization for conventional directory designs and TL designs normalized to the DUP baseline. Network utilization is measured by counting the number of flits that traverse each link every cycle. The sum of these traversals is then normalized to the sum for DUP. In principle, ignoring secondary effects (e.g., changes to cache replacement decisions due to directory restrictions), DUP should require less on-chip bandwidth than all other designs. In practice, SPARSE$_{FULL}$ uses up to $5.6\%\pm2.0\%$ more bandwidth for *Sparse* , but up to $12\%\pm6.6\%$ less bandwidth than DUP for *Ocean*. SPARSE$_{FULL}$ may invalidate cached blocks to allocate new directory entries. When these prematurely evicted blocks change subsequent cache replacement decisions, the overall cache miss rate can be reduced, resulting in the observed bandwidth reduction. In addition, prematurely evicted blocks may require fewer link traversals to fetch from memory in SPARSE$_{FULL}$ instead of another cache in DUP.

Both TL designs incur statistically insignificant amounts of overhead compared to DUP, and both result in only about 2.5% more bandwidth utilization than SPARSE$_{FULL}$ on average. The only workload that results in any significant bandwidth increase compared to either baseline is *Sparse*, which has an increase of $4.2\%\pm2.5\%$ for $TL_{4\times64}$ compared to DUP. However, as mentioned previously, SPARSE$_{FULL}$ incurs an even higher overhead for *Sparse*. Overall, none of these minor bandwidth variations result in any significant change in performance.



(a) Scaling Number of Hash Tables

(b) Scaling Number of Buckets

Figure 7: Expected number of false positive bits with increasing core counts.

5.6 Scalability

Using the analytical model of Section 3.4 this section demonstrates that TL is expected to scale well with an increasing number of cores. This section and Section 5.6.1 examine how the expected FPB and area of TL scale, respectively for CMPs of up to 1024 cores.

The expected FPB in the sharing vector for TL grows as shown earlier in Eq. 2. While this model ignores sharing, the results of Section 5.2 validated the model for 16-core CMPs. Maintaining a similar expected FPB should result in similarly negligible performance and bandwidth overheads as demonstrated in Sections 5.3 and 5.5. As the number of cores increases, the expected FPB can be controlled by increasing either the number of hash tables, the size of each hash table, or both.

Figure 7(a) shows how increasing the number of hash tables affects the expected FPB for different sized CMPs. Each curve represents a TL design with a different number of 64-bucket hash tables. The solid line shows the average FPB with the $\text{TL}_{4\times64}$ for a 16-core CMP. To achieve a similar or lower FPB, CMPs with up to 64, 256 or 1024 cores require five, six, or seven hash functions respectively.

Figure 7(b) performs a similar analysis this time keeping the number of hash tables constant at four while the size of each hash table increases from 64 to 512. The expected FPB is evaluated for various CMPs, with the solid line indicating the average FPB of $\text{TL}_{4\times64}$. The results show that 256 bucket hash tables should be sufficient for CMP designs with up to 1024 cores.

5.6.1 Area

To illustrate how the area of TL scales better than that of SPARSE$_{\text{FULL}}$ and SPARSE$_{\text{COMP}}$, Figure 8 shows the storage requirements of each design as a percentage of the number of bits used in the tag and data arrays of all L2 caches. The results demonstrate that TL offers the most practical design for these CMPs. Figure 8 does not include DUP because the high associativity requirement makes it an impractical design for large CMPs.

The solid black line in Figure 8 represents a SPARSE$_{\text{FULL}}$ that always contains one entry per cache block. The poor scalability shown makes SPARSE$_{\text{FULL}}$ impractical for large CMPs. Many alternative SPARSE$_{\text{COMP}}$ designs have been proposed to improve upon the poor scalability of SPARSE$_{\text{FULL}}$. Since there are an unlimited number of possible SPARSE$_{\text{COMP}}$ designs, Figure 8 shows three designs as dashed lines: SPARSE$_{\text{COMP-}n/4}$ contains one bit

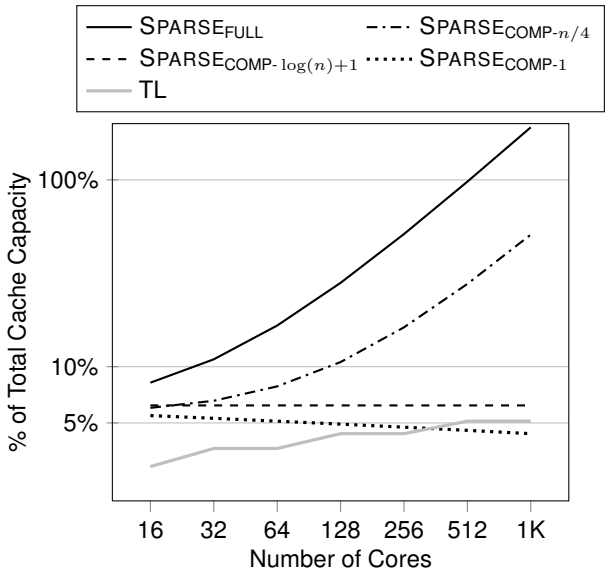


Figure 8: Storage requirements of different SPARSE$_{\text{FULL}}$, SPARSE$_{\text{COMP}}$, and TL designs relative to the total storage requirements of all L2 caches.

for every four cores, representing a coarse-vector, or SGI Origin style design, SPARSE$_{\text{COMP-}\log(n)+1}$ contains a single core pointer per directory entry, and SPARSE$_{\text{COMP-1}}$ contains only a single bit per entry. The first design represents a realistic SPARSE$_{\text{COMP}}$ design that scales similarly to SPARSE$_{\text{FULL}}$ and results in moderate bandwidth increases. The other two designs are extremely restrictive and would result in large increases in bandwidth utilization and likely a significant performance loss. SPARSE$_{\text{COMP-1}}$ is an unrealistic, impractical design, and SPARSE$_{\text{COMP-}\log(n)+1}$ would only be realistic for workloads that have no sharing. More realistic SPARSE$_{\text{COMP}}$ designs with better bandwidth or performance would necessarily have larger area.

Despite the very aggressive, low area nature of the last two SPARSE$_{\text{COMP}}$ designs, the solid grey line in Figure 8 shows that TL offers better scalability. Based on the observation in Section 5.2 that more, smaller tables are generally better than fewer, larger tables, the TL designs shown have been chosen based on the results

in Figure 7(a). Each design uses 64-bucket hash tables, and the number of tables increases from four to seven as the number of cores increases from 16 to 1024. These designs always require less area than SPARSE$_{\text{COMP-}n/4}$ and SPARSE$_{\text{COMP-}\log(n)+1}$, and up to 256 cores, they require less area than SPARSE$_{\text{COMP-1}}$. Based on the predictions of the analytical model, combined with the simulation results for the 16-core CMP, the TL designs should have negligible impact on performance and bandwidth. On the other hand, SPARSE$_{\text{COMP-}\log(n)+1}$ and SPARSE$_{\text{COMP-1}}$ will result in significant bandwidth increases for almost any workload.

6. RELATED WORK

The most common techniques for implementing coherence directories were discussed in Section 4. This section discusses some additional possible CMP directory designs, as well as works that use concepts similar to TL.

Two-level directory structures have been proposed that use a small first level directory for most accesses and rely on a larger second level directory when necessary [25]. Placing the first level on-chip and leaving the second level of the directory off-chip in main memory may provide a viable solution for a CMP directory. Similarly, it has been suggested that an on-chip directory *cache* can be used in conjunction with an in-memory directory [22]. While such a solution eliminates the cache placement restrictions of SPARSE$_{\text{FULL}}$, it requires significant off-chip memory and the long latency of requests that miss in the on-chip directory can hurt performance.

The Scalable Coherent Interface (SCI) stores directory information as a series of doubly-linked lists within the caches [15]. The head of each sharer list is stored in main memory, making this impractical for a CMP directory. While the head pointers could be stored on-chip, this would require area comparable to the SPARSE$_{\text{COMP-}\log(n)+1}$ design, and Section 5.6 demonstrated that the size of TL scales better than this design. SCI may require less bandwidth than SPARSE$_{\text{COMP-}\log(n)+1)}$, but the latency of traversing the sharer list might result in worse performance.

Simoni proposed the idea of Directory Entry Combining that used a large direct-mapped directory with no tags [32]. Blocks that mapped to the same directory entry shared the entry the same as TL shares sharing vectors. TL differs significantly from Simoni's design. TL provides a mechanism for removing sharing information for evicted blocks, and it incorporates multiple tables with different hash functions. TL provides improved accuracy at a lower area, resulting in a more area-efficient design that achieves higher performance.

Raghavan et al. recently proposed the PATCH protocol with combines a directory coherence protocol with token counting to improve performance [27]. Because TL does not explicitly track individual cache blocks, it might be difficult to apply such a performance optimization to TL. For larger CMPs, PATCH can be used to reduce the bandwidth overhead of SPARSE$_{\text{COMP}}$ designs without significantly reducing performance. However, as shown in Section 5.6, TL should require less area with little expected performance or bandwidth penalty.

JETTY reduces the need to probe caches in snoop coherence protocols using counting Bloom filters to represent the set of *all* blocks cached in a local cache [24]. RegionScout grouped addresses into coarse-grain regions tracked by counting Bloom filters [23]. TL uses similar filters, but by partitioning these filters per cache set, it avoids the need for counters and utilizes area and power more efficiently to achieve higher precision.

7. CONCLUSION

Directory protocols can maintain cache coherence without the bandwidth overhead of snoop protocols, especially as the number of cores increases in a CMP. While directory protocols offer more scalable bandwidth utilization than snoop protocols, conventional directory designs either have large area overhead, suffer poor performance for some sharing patterns, or are simply impractical for large CMPs.

This paper presents a new Tagless Coherence Directory structure that uses an implicit, conservative representation of the blocks stored by each cache. The new directory uses a grid of small Bloom filters, one for each set in each private cache in the CMP. This organization preserves the effectiveness and space-efficiency of Bloom filters, and it avoids the well understood problem of removing elements from the set represented by conventional Bloom filters. By efficiently removing the sharing information when a block is evicted from the cache, on average, the *superset* of sharers maintained per block includes very few, or no, non-sharers, resulting in a negligible impact on performance and bandwidth utilization. The resulting implementation requires only minor extensions to an underlying conventional coherence protocol. In addition, the bulk of the implementation uses SRAM arrays while imposing very few restrictions on how these are partitioned and organized.

As a result, the new directory structure uses 48% less area, 57% less leakage power, and 44% less dynamic energy than a conventional directory for a 16-core CMP with 1MB private L2 caches. Simulations of commercial and scientific workloads indicate that this design results in no statistically significant change in performance, and only a 2.5% increase in bandwidth utilization. In addition, an analytical model predicts that this design continues to scale well for CMPs even up to 1024 cores. Moving forward, future work can leverage TL to improve performance by reducing the need for indirections through the directory for cache-to-cache transfers.

TL finally provides a practical, scalable directory implementation for future CMPs. In the debate between snoop-based and directory-based coherence protocols, snoop-based protocols have benefited from the lack of a truly scalable CMP directory option. TL provides a new champion for the side of directory-coherence protocols and allows for a proper and fair debate.

8. ACKNOWLEDGEMENTS

This work was supported in part by an NSERC Discovery grant and an IBM Faculty Award. Jason Zebchuk is partially supported by an NSERC Postgraduate Scholarship. We thank Matthias Blumrich, Ioana Burcea, Natalie Enright Jerger, Babak Falsafi, Valentina Salapura and the anonymous reviewers for their helpful comments and suggestions. We also thank the Impetus group at Carnegie Melon University and at Ecole Polytechnique Fédérale de Lausanne for their assistance with the Flexus simulator and with the commercial workloads. Finally, we thank Virtutech for the Simics simulator.

9. REFERENCES

[1] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White Paper, 2008.

[2] OpenSPARC^{TM} T2 system-on-chip (SoC) microarchitecture specification, May 2008.

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In

Proc. of the Int'l Symposium on Computer Architecture, June 1988.

[4] C. S. Ballapuram, A. Sharif, and H.-H. S. Lee. Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[5] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture base on single-chip multiprocessing. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[8] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

[9] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, C-27(12):1112–1118, Dec. 1978.

[10] J. H. Choi and K. H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *Proc. of the Int'l Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 258–267, Apr 1999.

[11] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proc. of the Int'l Symposium on the Performance Analysis of Systems and Software*, Mar. 2005.

[12] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network mulitcast for scalable cache coherence. In *Proc. of the Int'l Symposium on Micorarchitecture*, Dec. 2008.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[14] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proc. of the Int'l Conference on Parallel Processing*, Aug. 1990.

[15] D. Gustavson and Q. Li. The scalable coherent interface (sci). *Communications Magazine, IEEE*, 34(8):52–63, Aug 1996.

[16] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):31–35, Mar. 2004.

[17] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, Sept.-Oct. 2007.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. of the Int'l Symposium on Computer Architecture*, June 1997.

[19] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER 6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, Nov. 2007.

[20] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[21] P. Mak, C. R. Walters, and G. E. Strait. IBM System z10 processor cache subsystem. *IBM Journal of Research and Development*, 53(1), 2009.

[22] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2007.

[23] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. of the Int'l Symposium on Computer Architecture*, pages 234–245, June 2005.

[24] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proc. of the Int'l Symposium on High-Performance Computer Architecture*, Jan. 2001.

[25] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. of the Int'l Symposium on Computer Architecture*, June 1990.

[26] L.-S. Peh and W. Dally. A delay model and speculative architecture for pipelined routers. pages 255–266, 2001.

[27] A. Raghaven, C. Blundell, and M. M. K. Martin. Token tenure: PATCHing token counting using directory-based cache coherence. In *Proc. of the Int'l Symposium on Microarchitecture*, Dec. 2008.

[28] S. Rusu, S. Tam, H. Mulijono, D. Ayers, and J. Chang. A dual-core multi-threaded Xeon processor with 16MB L3 cache. In *Proc of the Int'l Solid-State Circuits Conference*, Feb. 2006.

[29] V. Salapura, M. Blumrich, and A. Gara. Design and implementation of the Blue Gene/P snoop filter. In *Proc. of the Int'l Symposium on High Performance Computer Architecture*, Feb. 2008.

[30] C. Saldanha and M. H. Lipasti. *Power Efficient Cache Coherence*. Springer-Verlag, 2003.

[31] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proc. of the Int'l Symposium on Microarchitecture*, Dec. 2007.

[32] R. Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, Oct. 1992.

[33] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *AFIPS '76: Proc. of the June 7-10, 1976, National Computer Conference and Exposition*, pages 749–753, 1976.

[34] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi. CACTI 5.0: An integrated cache timing, power, and area model. Technical report, HP Laboratories Palo Alto, 2007.

[35] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

[36] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2003.