International Journal of Engineering, Management, Humanities and Social Sciences Paradigms (Volume 31, Special Issue of January 2019)
ISSN (Online): 2347-601X and Website: www.ijemhs.com

Honey, I Started to shrink the Coherence Directory (SPATL)

Ms.Pragyan Paramita Panda[1], Ms.Suchitra Mishra[2]*

*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
pragyanparamita@thenalanda.com, suchitramishra@thenalanda.com*

Abstract— The coherence directory, which contains details on the sharing of cache blocks, is one of the main problems with on-chip coherence in a multicore processor in terms of scaling. To reduce area overhead, shadow tags that copy complete private cache tag arrays are frequently utilised. However, getting the sharing information requires an energy-intensive asso- ciative search. A Tagless directory was recently proposed, which uses bloom filters to condense the tags in a cache set. To totally avoid associative lookup and minimise directory cost, the Tagless directory associates the sharing vector with the bloom filter buckets. However. With more cores, Tagless still has space and energy constraints because it still represents the sharing information using a full map sharing vector.

First, we demonstrate in this work how numerous bloom filters basically reproduce the same sharing pattern because of the regular nature of applications. We then take advantage of the shared pattern and suggest SPATL1 (Sharing-pattern based Tagless Directory). By taking use of the sharing pattern similarity, SPATL is able to divorce sharing patterns from bloom filters and get rid of duplicate sharing patterns. In comparison to Tagless, the previous most storage-efficient directory, SPATL offers 34% storage savings at 16 cores and works with both inclusive and non-inclusive shared caches. We investigate a number of methods for periodically eliminating false sharing that results from combining Tagless and sharing pattern compression, and we show that SPATL can achieve the same amount of false sharers as Tagless with 5% more bandwidth. Last but not least, we showKeywords: Directory coherence, Cache coherence, Multicore scalability, Tagless, Bloom Filters

I. INTRODUCTION

In order to utilize the growing on-chip real estate, designers are increasingly turning toward larger numbers of independent compute engines or cores, whether homogeneous or heteroge-neous. To provide fast data access, data is replicated/cached in core-local storage to exploit locality. Further, to ease communication among these compute cores, the potentially multiple copies of data are often kept coherent in hardware. The larger core counts require more bandwidth both for data access and to keep the caches coherent. Cache coherence needs to track information about the various copies of cached blocks in order to keep them consistent with each other. A directory is typically used to provide precise information on the presence of replicas so as to minimize coherence communication.

A typical directory-based coherence protocol [5] maintains a bit vector (the sharing pattern) per coherence unit, represent-ing the processors that currently share the memory locations, resulting in space overhead that is proportional to the number of cores and the size of the shared level of memory. By limiting the communication to a multicast among the actual sharers instead of a broadcast, the bandwidth requirement of directory-based protocol scales better than typical snoop-based protocols.

Several optimizations to reduce the area overhead of the directory have been proposed. For example, a directory cache [1], [15] stores sharing information for a subset of lines in the shared memory. A compressed sharer vector [6], [8], [16] uses fewer bits to represent sharer information, thereby losing some precision in determining the exact sharers. Such techniques also can represent only a limited number of shar-ing patterns and suffer inelegant sharp performance losses for specific types of sharing patterns. Pointers [2], [11] provide precise sharing information for a limited number of sharers of each cache line, resorting to introducing extra hardware and software overhead when the number of sharers exceeds the number of hardware-provided pointers.

Alternatively, shadow tags are used, for example, in Ni-agara2 [17], in which the tags from the lower level caches are replicated at the shared level. An associative search of the shadow tags is used to generate the sharer vector on the fly. Although shadow tags achieve good compression by maintaining only information for lines present in the lower level caches, the associative search used to generate the sharer vector is energy hungry, especially at larger core counts.

Recently, two different approaches have been used to achieve directory compression without loss in precision or extra energy consumption. The Tagless directory [19] starts with the shadow tag design and uses bloom filters per private-level cache set to encode the presence of the tags in each private-level cache. The buckets in the bloom filter repre-sent the sharing pattern. This approach has two advantages, namely, it eliminates the energy-hungry on-the-fly sharing pattern generation, and the shadow tag space is also no longer proportional to the size of the tag.

SPACE [20] was designed for inclusive caches and lever-ages the observation that many memory locations in an application are accessed by the same subset of processors and hence have identical sharing patterns. In addition, the number of such patterns is small, but varies across applications and even across time. SPACE proposes the use of a sharing pattern table together with pointers from individual cache lines to the table. Graceful degradation in precision is achieved when the table's capacity is exceeded.

In this paper, we extend the observation made in [20] that sharing pattern commonality across memory locations can be used to compress the directory without significant loss in precision, to apply to non-inclusive caches. Specifically, we combine the energy and compression benefits of the Tagless and SPACE approaches in a system we call SPATL (Sharing-pattern based Tagless Directory). As in the Tagless approach, tags within individual sets are combined in a bloom filter. However, rather than containing sharer vectors, the individual buckets in the bloom filter contain pointers to a table of sharing patterns. As in SPACE, only the sharing patterns actually present due to current access to shared data are represented in the sharing pattern table. This combination allows directory compression with graceful degradation in precision for both inclusive and non-inclusive cache organizations. Our results show that the use of a sharing pattern table can be used to compress the Tagless directory, resulting in compounded area reductions without significant loss in precision. *SPATL* is 66% and 36% the area of the Tagless directory at 16 and 32 cores, respectively. We study multiple strategies to periodically eliminate the false sharing that comes from combining sharing pattern compression with Tagless, and demonstrate that *SPATL* can achieve the same level of false sharers as Tagless with▪5% extra bandwidth. Finally, we demonstrate that *SPATL* scales even better than an idealized directory and can support 1024-core chips with less than 1% of the private cache space for data parallel applications.

II. Background

In a multicore chip like the one shown in Figure 1, there are private caches associated with each core (or set of cores). In our baseline design, we also have a shared L2 cache that is tiled across the various cores. While conceptually a centralized structure, the directory is distributed across the various tiles. Each cache block is assigned a home tile and the directory associated with the home tile is assigned the task of providing sharer information for cache blocks that map to that tile. For maximum precision, the coherence directory must maintain sharing information for each unique tag in the private caches.

Designs that use an inclusive shared L2 cache piggyback on the L2 tags to implement the tags required by the directory. This requires the addition of a P bit sharing vector (P : # of cores) per L2 tag. Unfortunately, since shared caches are many times larger than private caches, many entries contain no information. For example, if the Niagara2 (8 cores, 8KB L1/core, 4MB shared L2) were to implement an in-cache directory it would consume 64KB of space, which is 100% of the cumulative size of L1 caches across all the 8 cores.

An alternative to piggybacking on the L2 tags is to use a

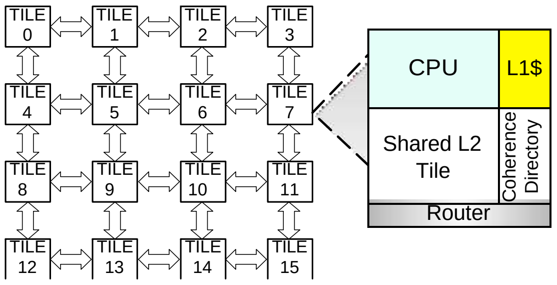


Fig. 1: Tiled 16 processor multicore. Coherence directory distributed to each tile.

directory cache to maintain information only for lines present in the L1. Since each cache line in each core could be unique, to guarantee no loss of information, the directory cache would need to contain as many entries as the sum of the number of cache lines in each L1, along with an associativity that is at least the aggregate associativity of all the L1s (i.e., even on the 8 core Niagara2, we would need a 32 way directory cache). Practical directory cache designs have much lower associativity and pay the penalty of associativity-related eviction of directory information for some blocks. While recently there have been proposals [7] to use sophisticated hash functions to eliminate associativity conflicts, optimizing the directory cache organization is a hard problem.

Many current multicore chips (e.g., Niagara2) use a simplified form of directory cache consisting of replicas of the tag arrays of the L1 cache (i.e., maintain shadow tags). An associative search of the shadow tags is used to generate the sharer vector on the fly. Although shadow tags achieve good compression by maintaining only information for lines present in the lower level caches, the associative search used to generate the sharer vector imposes significant energy penalty.

Recently, the Tagless coherence directory [19] was proposed to eliminate the associative lookup. Instead of representing each tag exactly, a bloom filter concisely summarizes the contents of each set in every L1 cache. Overall, we would need only $N_{L1sets} * P$ bloom filters (32–64bits per bloom filter) to represent the information in all the L1 caches. The probing required per L1 in shadow tags is replaced with a simple read of a bloom filter, which eliminates all the complex associative search of shadow tags. Unfortunately, for large multicores the cost of the bloom filters grows proportionately (similar to the sharing pattern vector) and constitutes significant overhead. For example, for an 8 core Niagara2, it would require 3KB (per hash function), but extrapolating to 1024 cores, it would require 3MB, which imposes significant area and energy penalty for sharing pattern information access. We briefly describe the overall architecture of Tagless below and highlight the challenges.

A. Tagless Coherence Directory

Tagless coherence directory uses a set of bloom filters to summarize the contents of the cache. Figure 2 shows the bloom filter associated with each set of the private L1 cache. Essentially, the Tagless directory consists of a $N_{L1sets} * P$ set of bloom filters (N_{L1sets} : number of sets in the L1

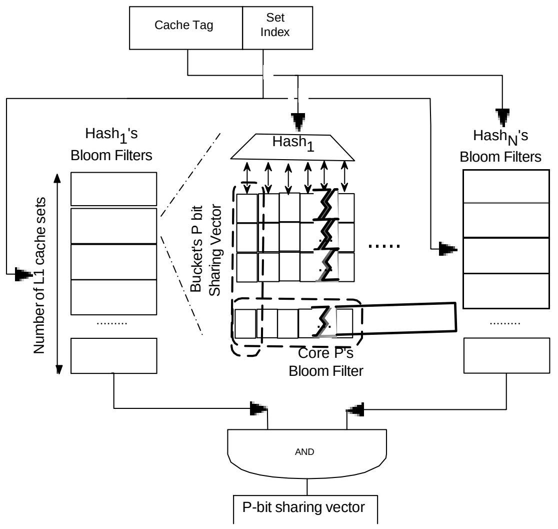


Fig. 2: Tagless Coherence Directory [19].

cache. P : Number of cores). Each bloom filter per set is a partitioned design that consists of $hash_N$ hash functions each of which map to a k bucket (k bitmap) filter. If the size of the bloom filter is comparable to a cache tag, overall this essentially improves the space over shadow tags by a factor of $\frac{N_{L1ways}}{\#of\,hash\,functions}$.

Tagless directory uses this representation to simplify the insertion and removal of cache tags from the bloom filter. Each bloom filter summarizes the cache tags in a single cache set. Inserting a cache block's address requires hashing the address and setting the corresponding bucket (note that each address maps to only one of the buckets). Testing for set membership consists of reading the bucket corresponding to the cache tag in the set-specific bloom filter of each processor and collating them to construct the sharing pattern (in Figure 2, each bucket represents a sharing pattern). Having a bloom filter per set also enables Tagless directory to recalculate the filter directly on cache evictions. While conceptually, the Tagless directory consists of $N_{L1sets} * P$ bloom filters, these filters can be combined since each core uses the same bloom filter organization. A given cache block address maps to a unique set and a unique bucket in the bloom filter. Combining the buckets from all the bloom filters, a P bit sharing pattern is created, which is similar to the sharing pattern in a conventional full-map directory.

Multiple addresses could potentially hash to the same bucket and hence introduce false positives. Using multiple hash functions enables addresses to map to different buckets and possibly eliminate false positives. Simply ANDing the sharing vector from the buckets that an address maps to in each hash function will eliminate many false positives. Consider an implementation with $hash_N$ hash functions, k buckets per hash function, N_{sets} L1 cache sets, and P cores. The Tagless directory requires a P-bit pattern for each of the k buckets, giving rise to an overhead of $hash_N * k * P * N_{sets}$ bits.

Scalability Challenges. For large multicore chips (256+ cores) the storage overhead of the Tagless directory is dominated by P. This introduces challenges to scalability with increasing core counts. Furthermore, reading a large P-bit wide vector from this coherence directory will not be energy efficient. Figure 3a shows the per-core area of the Tagless directory while increasing the number of cores. Since the number of addresses that are mapped to a bloom filter grows with the number of cores, the possibility of false positives increases when using a fixed bloom filter size. We therefore increase the number of buckets per bloom filter so as to maintain the same level of false positives as our baseline design. If we project to a Niagara2 design with a number of cores from 256–2048, the Tagless directory adds significant overhead. At 2048 cores, the total directory overhead is 16MB, which is 100% overhead since the aggregate size of all the L1s in this system is 16MB. We assume that the directory is uniformly distributed amongst all the cores and hence the per-core overhead grows more gradually from 2KB at 256 cores to 8KB at 2048 cores. Figure 3b plots the energy overhead of reading from a directory tile. The size of sharing pattern block read varies linearly with the cores. We see a significant increase in the read energy from 5pJ at 256 cores to 12pJ at 2048 cores.

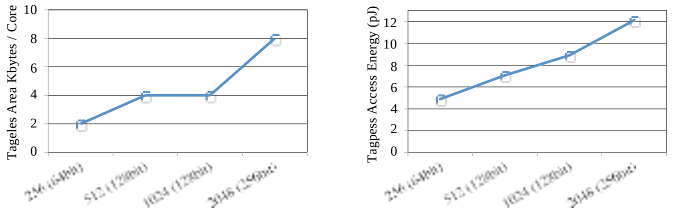


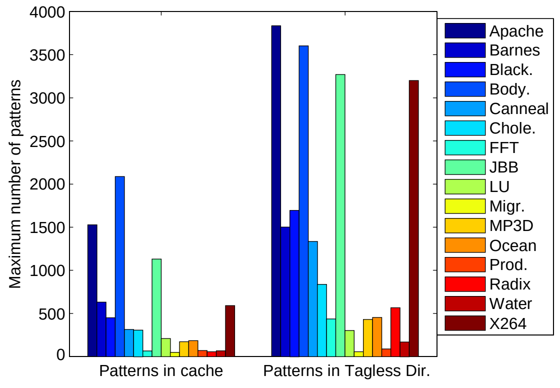
Fig. 3: Left (a): Storage overhead of Tagless directory per core; X axis: # of cores (Bloom Filter size); Y axis: KB of coherence directory per core. Right (b): Access energy of Tagless directory tile per core; X axis: (# of cores); Y axis: pJ.

III. SPATL : Hybrid Coherence Directory

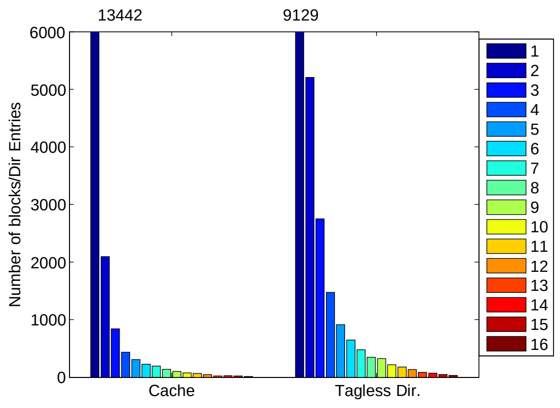
A. Sharing Patterns in the Directory

At PACT 2010, the SPACE [20] design was proposed as a promising technique that compresses directory space for inclusive cache designs. SPACE was based on observations of application semantics that showed the regular nature of inter-thread sharing, resulting in many cache blocks having the same or similar sharing patterns. Thus, the in-cache directory has a lot of redundancy and replicates the same pattern for many cache blocks. SPACE decouples the sharing vectors from the L2 tag and stores the unique sharing patterns in a pattern table; multiple cache lines with the same pattern would point to a common entry in the pattern table. The sharing bit vector per cache tag is replaced with a pointer whose size is proportional to the number of unique sharing patterns. Unfortunately, while this provides better scalability than the base in-cache directory design (reduces the directory overhead to 40KB for the Niagara2), lines not present at the L1s continue to bear the pointer overhead, which limits the overall benefit.

In this work, we extend the idea of eliminating sharing pattern redundancy to the Tagless buckets. Figure 4a shows the maximum number of patterns displayed in an application during its execution, with and without the Tagless directory (system configuration described in Table I in Section V). The relatively small number of patterns present in the applications compared to the total number of possible patterns suggests an opportunity to design a directory that holds only the sharing patterns present. In the Tagless directory, each bucket combines and holds the union of sharing patterns of cache blocks that map to that bucket. This in some cases causes an overall increase in the total number of patterns since two addresses with different sharing patterns could map to the same bucket (causing false positives). Despite this possibility, as the figure shows, the number of sharing patterns is much smaller than the total number of buckets (65,536 in our experiments), indicating that the same sharing pattern gets replicated across multiple hash table buckets.



(a) Maximum number of sharing patterns.



(b) Number of cache blocks for each of N sharers.

Fig. 4: (a) The maximum number of patterns present for a specific application for Apache. (b) The cache block distribution over different number of sharers for Apache. For example, ▾9,000 cache blocks have a private access pattern (only one processor is accessing the block).

Figure 4b shows the degree of sharing for a snapshot of the application Apache with and without the use of the Tagless directory. Each bar in the histogram represents the number of cache lines with patterns with a certain number of processors sharing the cache line. Private cache lines are the dominant sharing pattern for Apache, exhibited by over 75% of the cache lines. We observe that the percentage of cache lines tagged as private reduces to 40% for the Tagless directory. Some cache lines with private patterns are tagged as 2-sharer or 3-sharer because of conflicts in Tagless's bloom filter buckets.

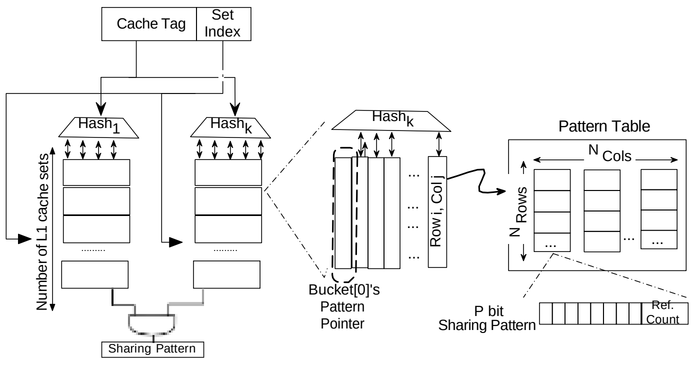


Fig. 5: Hybrid Tagless-Pattern directory approach. Each bucket includes a pointer to the sharing pattern.

Based on our analysis, we observe that despite the possibility of conflicts, the common sharing patterns across many cache blocks continues to be exhibited across many buckets in the Tagless approach. We also observe that the number of patterns that are frequently referenced is small (as corroborated by results in [20]). In our experiments, the number of patterns that applications exhibit is 5.6% (Apache, SPECjbb), 0.8% (SPLASH2), and 3.8% (PARSEC) of the total number of buckets in the Tagless directory. Thus, we propose a solution to enable Tagless directory scalability to a large number of cores by eliminating the redundant copies of sharing patterns. The compression of sharing patterns will complement the compression achieved by the Tagless directory.

B. SPATL Architecture

As shown in the conventional Tagless directory, every bucket in the bloom filter specifies the sharing pattern for blocks mapping to that bucket. We propose to decouple the sharing pattern for each bucket and hold the different unique sharing patterns observed in the Tagless directory in a separate pattern directory table. This eliminates redundancy across the Tagless directory where the same sharing pattern is replicated across different buckets. With the directory table storing the patterns, each bucket now includes a pointer to an entry in the directory, not the actual pattern itself.

We organize the directory table as a two-dimensional structure with $N_{Dir.ways}$ ways and $N_{Dir.sets}$. Each bucket points to exactly one entry in the directory table and multiple buckets pointing to the same entry essentially map to the same sharing pattern. The size of the pattern directory table is fixed (derived from the application characteristics in Section III-F) and is entirely on-chip. Hence, when the table capacity is exceeded, we have a dynamic mechanism to collate patterns that are similar to each other into a single entry.

In this section, we describe our directory implemented on a multicore with 16 processors, with 64KB private, 2-way L1 caches per core, and a Tagless directory with 2 hash functions and 64 buckets per hash function. The conventional Tagless directory design incurs an overhead of

$Hash_N * N_{buckets} * Pcores$ bits = 2 * 64 * 16 bits per set of the L1 cache. Figure 5 illustrates the *SPATL* approach. We have a table with $N_{Dir.entries}$ ($= N_{Dir.ways} * N_{Dir.sets}$) entries, each entry corresponding to a sharing pattern, which is represented by a P-bit vector. The directory table itself is a $N_{Dir.entries} * P$ bit array; at a moderate number of cores, it does not constitute the dominant overhead. For each bucket in the Tagless directory, we replace the sharing vector with $\lceil log_2(N_{Dir.entries})\rceil$ bit pointer to indicate the sharing pattern. Every time the sharer information is needed, the bucket is first hashed into, and the associated pointer is used to index into and get the appropriate bitmap entry in the directory table, which represents the sharer bitmap for the cache tags that map to that bucket. The main area savings in SPACE comes due to the replacement of the P-bit vector per bucket with a $\lceil log_2 N_{Dir.entries}\rceil$-bit pointer.

The next two sections describe how *SPATL* inserts entries into the Tagless buckets and directory table, how patterns are dynamically collated when there aren't any free entries, and how sharing patterns are recalculated on cache block evictions.

C. Cache Block Insertion

When a cache line is brought in and a sharing pattern changes (or appears for the first time), the block needs to modify the sharing pattern associated with its bucket in the Tagless directory. To achieve this, the set index of the cache line is used to index to the specific bloom filter, and the tag is used to map to the specific bucket. When a cache line is inserted into a specific core *i*, logically, it modifies Core *i*'s sharing bit in the bucket mapped to. This operation is carried out in *SPATL* as a sharing pattern change. The current sharing pattern pointed to by the pointer in the bucket is accessed, and Core *i*'s bit is set in the pattern to form the new pattern.

The newly generated pattern needs to be inserted into the directory table. The pattern table is organized as a two-way table with N_{rows} and N_{cols}. Initially, the incoming sharing pattern hashes into a particular row and then compares itself against patterns that already exist in that row (Figure 5). Once a free entry is found in the directory table, the row index and column location are used by the bucket to access the specific entry. Intuitively, the hash function that calculates the row index in the pattern table has to be unbiased so as to not increase pollution in any given row. We also require that similar patterns map to the same row so as to enable useful collation of sharing patterns that do not differ by many bits when the protocol runs out of free directory entries.

To satisfy these two seemingly contradictory goals, we use a simple hash function to calculate the row index into the pattern table. We use a coarse bit-vector representation of the original sharing pattern as an index. For example, in a pattern directory with 16 rows, we could use a coarse-grain four-bit representation as the encoding to indicate which of the possible four-core clusters is caching the data. It ensures that patterns that map to the same row will differ only in topologically adjacent bits, enabling intelligent collation of patterns, i.e., without excessive extra traffic due to false sharers, by limiting this traffic to neighbors or a specific

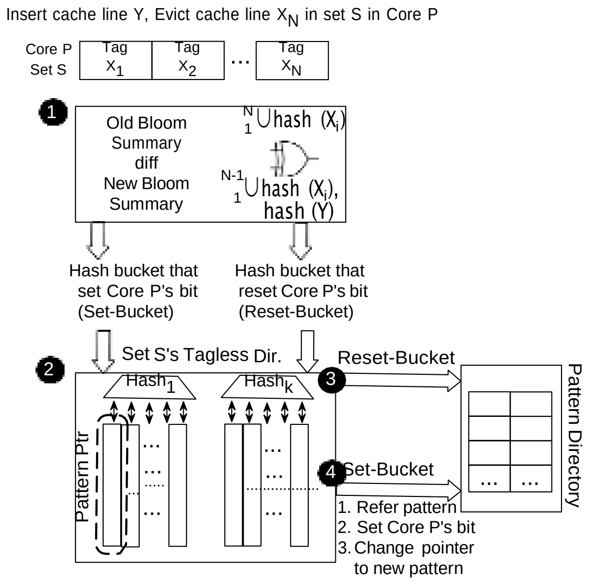

Insert cache line Y, Evict cache line X_N in set S in Core P
Core P
Set S
Tag X_1
Tag X_2
...
Tag X_N
1
Old Bloom Summary diff New Bloom Summary
$\bigcup_1^N hash(X_i)$
$\bigcup_1^{N-1} hash(X_i)$, hash (Y)
Hash bucket that set Core P's bit (Set-Bucket)
Hash bucket that reset Core P's bit (Reset-Bucket)
2
Set S's Tagless Dir.
$Hash_1$
$Hash_k$
Pattern Ptr
3
Reset-Bucket
4
Set-Bucket
1. Refer pattern
2. Set Core P's bit
3. Change pointer to new pattern
Pattern Directory

Fig. 6: Steps involved in inserting a new cache line.

set of sharers (when there are no free patterns available). Since private and globally-shared (all processors cache a copy) patterns appear to be common patterns across all the applications, *SPATL* dedicates explicit directory indices for these $P + 1$ patterns (where P is the number of processors) without the need for actual directory space.

Eviction of cache blocks. When a cache block is evicted from a core *i*, the bloom filter must be modified accordingly. Merely accessing the bucket to which the block hashes and resetting the bit corresponding to core *i* in the sharing pattern specified by the bucket does not suffice, since other blocks in the same core's cache set may map to the same bit. Instead, the Tagless directory will recalculate the *i*th bit (associated with core *i*) of the bloom filter buckets by rehashing all tags in the set to detect collisions.

In *SPATL* we cannot simply recalculate and reset (if necessary) core *i*'s bit in the sharing pattern pointed to by the bucket since other buckets could be pointing to this same sharing pattern. Instead, we treat such recalculations of the bloom filter as essentially sharing pattern changes. When core i's bit needs to be reset in a bucket, we first access the sharing pattern pointed to by the bucket. Following this, we reset core i's bit and try to reinsert into the pattern table as a new sharing pattern.

Illustration : Cache line insertion and eviction. Figure 6 illustrates the steps involved in inserting a cache line into *SPATL*'s directory. Currently, set S holds cache lines X_1,X_2, ...X_N in its N ways and we would like to insert cache line Y and displace cache line X_N. Not all buckets are affected as a result of this change, only the buckets that X_N and Y hash into. Hence, in ; the L1 cache at core P calculates the current Bloom summary, the new bloom summary with Y inserted in place of X_N, and the difference between the two summaries. The difference will include at most two buckets. If Y is the first address in the set to hash into a bucket from the set S, then Core P's bit in that bucket needs to be set (indicated by Set-Bucket in Figure 6). If no other address hashes into the same bucket as X_N then Core P’s bit needs to be reset on the bucket to prune out false positives (indicated by Reset-Bucket). The tuple consisting of (Core id (P), Set id (S), Set-Bucket, Reset-Bucket) is sent to the Tagless directory. In [illegible], the Tagless directory refers to the pattern pointed to by the Reset-Bucket, resets Core P’s bit, inserts the new pattern into the pattern table, and swings Reset-Bucket’s pointer to point to the new pattern. In [illegible], the Tagless directory refers to the pattern pointed to by the Set-Bucket, Sets Core P’s bit, inserts the new pattern into the pattern table, and swings Set-Bucket’s pointer to point to the new pattern. We do not unset Core P’s bit directly in the pattern table because there could be potentially other buckets pointing to the same pattern. Similarly, we do not set Core P’s bit directly in the pattern table because this could induce an extra false sharer for buckets already pointing to the entry.

D. Merging Patterns

A key challenge of fixed size superset representation is the combining of patterns from different cache blocks. In the hybrid approach, which combines Tagless and the pattern directory, sharing patterns need to be merged at two different levels. At the first level, the Tagless directory essentially associates a single sharing pattern vector with each bucket. When cache blocks with different sharing patterns hash into the same bucket, then the Tagless directory will need to store a union of the sharing patterns of each cache block. This arises due to the false positives introduced by bloom filters.

The other form of merging occurs when there are more sharing patterns in the system than the pattern directory can support. Figure 7 illustrates the process of inserting a pattern into the pattern table. When inserting a pattern in the directory, we index into the pattern table and search for a matching entry. If there are no matching or free entries that can be allocated from the set, the incoming pattern is combined with some existing pattern. Note that this merging does introduce extra false positives for the buckets that already point to that entry. The pattern directory tries to minimize pollution by merging the incoming pattern with the sharing pattern that is closest in terms of hamming distance (number of sharers in which they differ). This ensures that the extra sharers/false positives caused by the merging is kept to a minimum. Existing Tagless directory buckets that point to the sharing patterns will experience new false positives, but by ensuring that the merged patterns are similar to each other, we can limit the number of extra sharers and the resulting potential for extra coherence traffic.

Removal of sharing patterns. The last challenge that needs to be addressed is to ensure that entries in the directory are re-usable once no bucket has the sharing pattern in the entry. We use a simple scheme of reference counting to detect when an entry is no longer in use. A counter is associated with each entry in the directory. This counter is incremented when a new bucket starts pointing to the entry and is decremented when a bucket pointing to the entry changes its pointer. The entry can be reclaimed when the counter reaches zero.

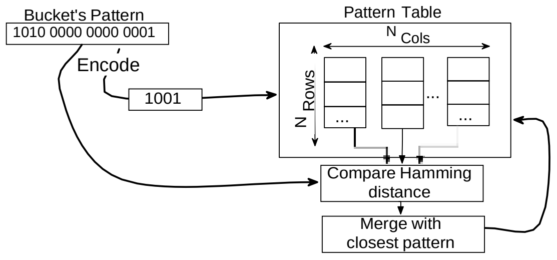


Fig. 7: **Inserting and merging a pattern into the pattern table.**

E. Directory Accesses

An interesting challenge that *SPATL* introduces is that it is possible for the directory to provide an inaccurate list of sharers to the coherence protocols. Coherence protocols use the sharers list in multiple ways. On a write access, the sharing pattern is used to forward invalidations and obtain the latest version of a cache block if any of the processors is holding a modified version. In such cases, we adopt a parallel multicast approach in which the pattern directory pings all possible sharers indicated by the sharing pattern. Cores will respond based on their state; whether they have a modified copy, have simply read it, or do not even cache the block.

Whether the shared cache is inclusive or exclusive determines whether the information in the directory is needed to retrieve data on read misses. Consider an inclusive cache in our baseline system with private caches and shared L2. With an inclusive L2 cache, the shared L2 has a copy of each L1 cache block. In case of a read miss on a clean cache block, the L2 can directly source the data and save the effort of forwarding messages to one of the L1 sharers. We only need to add information in the coherence directory about the new sharer. The directory information is needed for invalidation on write misses and to locate the modified copy when transitioning from modified to shared state. With a non-inclusive (or exclusive) shared L2, on a read miss that doesn’t find the block at the L2 level, we cannot separate the condition when the block does not reside at all on-chip from when the block is cached by one of the L1s without examining the directory. We have no choice but to check the directory and ping each of the sharers to see if they have a copy. Extra sharers/false positives in the directory affect read miss performance and the directory design has to be comparatively more robust than inclusive caches.

F. Challenge: Two-Level Conflicts

The base *SPATL* design without optimization exhibits much higher false positives when compared to the Tagless design. The reason for the increase in false positives is the double conflicts in the Tagless buckets and the pattern table. As we can see from Figure 4b the Tagless table in general introduces new sharing patterns because of conflicts at the Tagless buckets. The pattern table introduces further false positives after merging patterns. The conflict itself is not a problem if the original patterns can be recovered when a cache line is evicted as in the Tagless design. Unfortunately, with the base hybrid design this recovery ability is lost since the

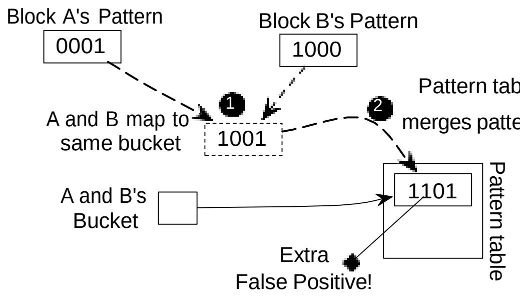


Fig. 8: **Two levels of false positives. Dashed lines indicate operations.**

pattern table introduces new sharing patterns by ORing with other unknown patterns.

To illustrate the problem, consider the example with 4 processors shown in Figure 8. Cache line A has the (private) pattern 0001, while cache line B has the (private) pattern 1000. A and B map to same the bucket in the Tagless directory. This causes the first level of false positives, and the bucket creates the pattern 1001, and inserts it into the pattern table ( ). In the pattern table, the pattern gets merged with pattern 1101. Pattern 1101 is now stored in the pattern table, and the bucket stores the pointer pointing to the pattern 1101 ( ).

Now consider when cache line B is evicted. In the *SPATL* design, on a cache line eviction, we read the pattern table entry (1101) and reset Core 0's bit, which leaves us with 1100. The false positive from Core 2 caused by merging patterns in the pattern table cannot be cleared since we do not know whether Core 2's bit was set due to pollution in the Tagless or the Pattern table. With private patterns being the common patterns, the situation described occurs frequently, leading to pattern table pollution, and soon enough the pattern table does not have free entries, leading to more pollution. In the Tagless design, the signature is recalculated, and the pattern would naturally become 1000, which is the accurate pattern again.

To clean up the polluted entries, we use *pattern recalculation* messages at the time of cache evictions. At the time of cache evictions, we look up the pattern table and multicast a *pattern recalculation* message to other sharing processors (in this case, when Core 0 evicts B, the Tagless directory multicasts messages to Core 2 and Core 3 as indicated by the pattern.) Each individual processor recalculates its signature of the set and sends back the information. Now, we are able to reconstruct the precise pattern for the set and place it in the pattern table. For example, in this case we will be able to precisely recalculate the pattern for the set as 1000. The recalculation results in increased messages in the system as shown in Figure 11. However, the messages are not in the critical path because they are incurred only on cache evictions. We investigated a few simple optimizations to address the increase in traffic in Figure 11. Instead of recalculating the pattern on every eviction, we use simple decision logic to decide when to recalculate based on the importance of the cache line. We evaluate the importance of whether a pattern recalculation is needed based on information such as the number of sharers in the pattern, and the number of entries pointing to the pattern. This information already exists in the base design and we demonstrate that employing such optimizations minimizes the bandwidth cost of *pattern recalculation* messages.

IV. Analytical Model: Bit Budget Trade-offs

We analyze and compare the different coherence directory schemes in order to understand their reasons for efficiency using an analytical model. In the following analysis, we assume that the total number of cores in the system is P, i.e., P private caches need to be kept coherent and each cache has S sets and W ways. The total number of blocks that can be cached across all the L1 caches is $P \times S \times W$. However, typically there are fewer unique blocks due to data sharing. Assuming f is the fraction of blocks that are unique in the total number of blocks $(0 \leq f \leq 1)$, then a coherence directory essentially needs only $f * S * W * P$ entries to support coherence operations. Hence, the total bit budget for an *idealized directory* will be : $f * P * S * W * (T_b + P)$, where T_b is number of tag bits (typically 48 bits on a 64 bit machine). For large multicores if $P >> T_b$, then the Ideal Directory $= f * P * S * W * P$ and $\propto O(P^2)$.

The shadow tags approach that completely replicates the L1 tags has a bit budget as $P * S * W * T_b$, which is suited to the case when most cache lines across the cores are private. Shadow tags has a smaller overhead than the ideal directory when all tags are unique ($f = 1$), because the shadow tags do not store the actual sharing patterns required by coherence. Every coherence access needs a $W * P$-way search on T_b-bit tag fields. Even on small multicores, this is an energy-intensive associative search. Tagless Directory is built on the shadow tags approach. It adopts shadow tags' approach of using the directory to represent the tags in the L1, but unlike shadow tags, it only represents a summary of the tags in each set using a bloom filter. Its overall budget is :

$$\text{Tagless Budget} = S * B * H_n * P$$

B and H_n are related based on false positives

$$E[\text{False positives}] = (P - 1) * (1 - (1 - \frac{1}{B})^W)^{H_n}$$

B: Buckets/Hash function ; H_n : # of Hash functions

With a large number of cores, P will dominate the relation, resulting in significant area overhead. *SPATL* improves over Tagless by decoupling Tagless' relation to P and relates it to the actual sharing patterns in the application. *SPATL* allows the designer to carefully consider the application suite targeted and appropriately size the pattern table. If the pattern table stores 2^{I_p} patterns, then the Tagless table needs I_p bits per entry, which is smaller than P. Therefore, in *SPATL* the pattern table grows linearly with P, but the Tagless table itself grows as log(pattern table size). Overall, *SPATL* performs better than Tagless under the following conditions

$$[S * B * H_n * I_p(\text{Tagless Table}) + 2^{I_p} * P(\text{Pattern Table})] < S * B * H_n * P$$

In many cases for large multicores $P >> I_p$, which implies that $(P - I_p)$ can be approximated as P, so the condition can be reduced to $2^{I_p} < S * B * H_n$.

V. Evaluation

A. Experimental Setup

To evaluate the *SPATL* design, we use a Simics-based [12] full system execution-driven simulator, which models the SPARC architecture. For cache and memory simulation, we use Ruby from the GEMS toolset [13]. Our baseline is a 16-tile multicore with private L1 caches and a 16-way shared inclusive L2 distributed across the tiles. We employ a 4x4 mesh network with virtual cut-through routing. We simulate two forms of packets: 8-byte control packets for coherence messages and 72-byte data payload packets for the data messages. Table I shows the parameters of our simulation framework.

We use a wide range of workloads, which include commercial server workloads [3] (Apache and SPECjbb2005), scientific applications (SPLASH2 [18]), and multimedia applications (PARSEC [4]). We also include two microbenchmarks, migratory and producer-consumer, with known sharing patterns. Table II lists all the benchmarks and the inputs used in this study. The table also includes the maximum number of access patterns for each application, which can be correlated with the performance of a given *SPATL* directory size.

We compare against the following coherence directory designs:

Tagless Directory (TAGLESS). This design studies the original Tagless approach presented at Micro 2009. The number of hash functions is fixed at two, and the number of buckets per set is varied from 16 to 64.

SPATL-N (TAGLESS-SPACE Approach). We also study a range of *SPATL* design points varying the directory table from 512 — 2048 entries. We evaluate two versions, namely, *SPATL*-NOUPDATE (SPATL1024noupdate in chart) and *SPATL*. The *SPATL*-NOUPDATE is a baseline design for the combined approach. *SPATL* includes extra optimizations (discussed in Section III-F) geared to eliminating the transient false-positives that arise due to conflicts in the TAGLESS table. For the *SPATL* design, each tile contains a segment of the directory table. We charge a 2-cycle penalty for each *SPATL* lookup.

B. How accurate is SPATL?

SPATL can achieve false positives similar to the base Tagless design. We do require extra logic in the design to eliminate the pollution arising out of two levels of compression. We eliminate the pollution by designing simple "pattern recalculation" messages to recalculate the sharing pattern. These messages are multicast to possible sharers off the critical path, at eviction time.

In our first set of experiments, we estimate the accuracy of sharing patterns maintained in *SPATL*. In a directory-based coherence protocol, coherence operations refer to the sharer information to forward coherence messages and the accuracy of tracking sharers has an impact on overall network utilization and hence energy spent in communication. For cases in which the sharing pattern is represented inaccurately, we evaluate the average number of extra false sharers experienced on each directory probe.

Our baseline shown in Figure 9a evaluates the Tagless directory approach with different hash functions and buckets. 64 buckets and 2 hash functions appears to be the optimal design with negligible false positives. Figure 9b shows the *SPATL* approach. As we can see the *SPATL*-noupdate (naively combining TAGLESS with a Pattern table) introduces many false positives. Once we introduce the optimization to recalculate the sharing pattern on evictions, we reduce the false positives and are able to approach Tagless' level of accuracy. In applications including MP3D, FFT, and Water, the *SPATL* design does not add any inaccuracy on top of the Tagless design. This is due to the over provisioning of entries in the pattern table, which needs to support other applications as well. In the baseline *SPATL*-noupdate design, Apache, Barnes, Bodytrack and SPECjbb experience the lowest accuracy with the relatively large number of sharing patterns that merge in complex ways to introduce many false sharers.

There are two possible scenarios where the directory needs to be referenced. A cache miss in the L1 to look up the directory to decide which cache could possibly provide the data. If *SPATL* were integrated with an inclusive shared L2 cache then we can decide to source data for all misses from the L2, except in the case when one of the caches holds a modified copy. If *SPATL* was integrated with a non-inclusive shared cache then cache misses need to possibly source the data from one of the L1s and need the directory for determining the possible sharers. Write misses (get exclusive access and update messages) probe the directory for sharer information to forward invalidations.

Figure 10 demonstrates an interesting trend that the average false positive sharers is much smaller for invalidation probes. Most of the *SPATL* false sharers are introduced as a result of probes on read misses. If *SPATL* were integrated with a non-inclusive cache it would need to satisfy both read misses and forwarded invalidations; we would need 1024 entries in the pattern table. If *SPATL* were integrated with an inclusive shared L2 cache, we can eliminate all the false positives due to the cache misses and reduce the pattern table size by 4×.

C. Interconnect Traffic

In this section, we study the interconnect traffic for applications in *SPATL* and show that the *SPATL* directory introduces minimal increase in on-chip traffic.

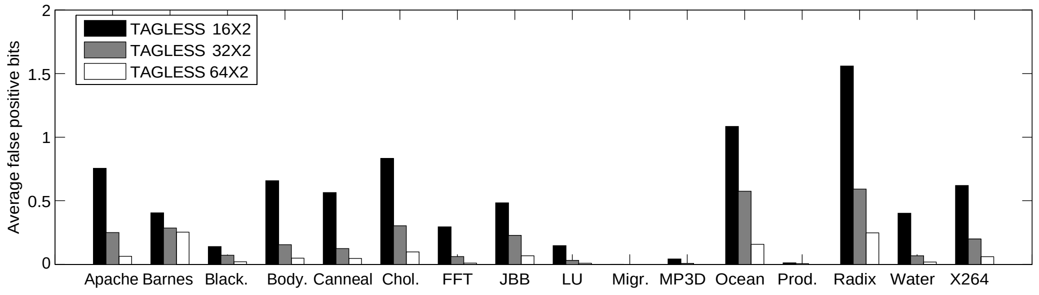
SPATL increases traffic compared to fully accurate directory in two ways. The false positives per reference generates additional messages, which are on the critical path of invalidations and lookups. In addition, "pattern recalculation" (presented in Section III-F) at evictions also multicast messages to sharers. Figure 11 plots the increase in traffic due to the false positives and the recalculation. In applications with few sharing patterns, both the traffic caused by false positives and recalculations are minimal. This is the case for Blackscholes, Canneal, and all the scientific benchmarks except Barnes. The additional traffic is less than 2%. Due

TABLE I: Target System parameters

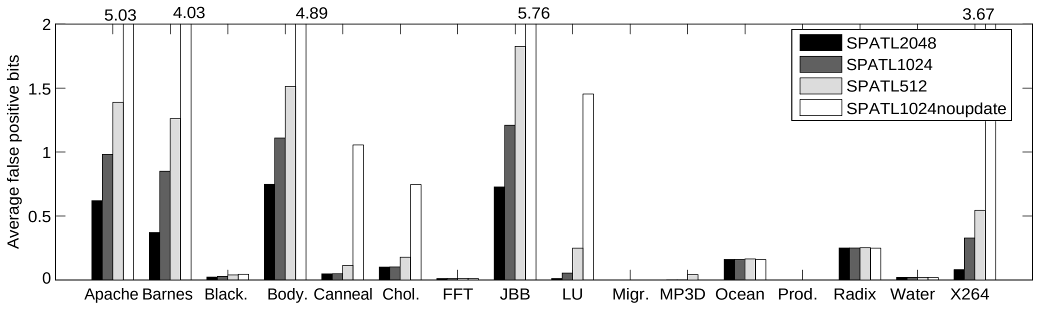
Cores: 16-way, 3.0 GHz, In order L1D/I : each 64KB, 2way, 64byte block
Shared Tiled L2 Cache
16 banks, 4MB/Tile, 16way, 14 cycles
Interconnect: 4x4 mesh 128bit wide 2cycle links
Main Memory : 500 cycles

TABLE II: Application Characteristics

Benchmark	Setup	# of sharing patterns	Network Utilization
Apache	80000 requests fastforward, 2000 warmup, and 3000 for data collection	1657	11.6%
JBB2005	350K Tx fastforward, 3000 warmup, and 3000 for data collection	1054	8.5%
Barnes	8K particles; run-to-completion	707	3.3%
Cholesky	lshp.0; run-to-completion	364	2.6%
FFT	64K points; run-to-completion	104	3.7%
LU	512x512 matrix,16x16 block; run-to-completion	249	1.9%
MP3D	40K molecules; 15 parallel steps; warmup 3 steps	181	6.1%
Ocean	258x258 ocean	208	5.7%
Radix	550K 20-bit integers, radix 1024	169	5.0%
Water	512 molecules; run-to-completion	75	2.7%
Migratory	512 exclusive access cache lines	63	0.6%
ProdCon	2K shared cache lines and 8K private cache lines	82	1.5%
Blackscholes	4096 options	450	3.5%
Bodytrack	4 cams, 100 particles, 5 layers, 1 frame	2087	2.2%
Canneal	100K elements, 10K swaps per step, 32 steps	313	4.3%
X264	640 x 360 pixels, 8 frames	590	2.2%



(a) Tagless false positives



(b) *SPATL* false positives

Fig. 9: (a) Average number of false positives per reference with Tagless approach. (b) Average number of false positives per reference with *SPATL* approach.

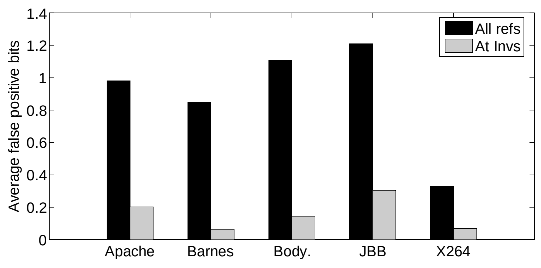


Fig. 10: False sharers on coherence write invalidations.

to recalculation, multicast only happens when there is a hint of pollution (i.e., pattern table indicates that more than one sharing pattern has been ORed at the entry). Therefore both types of traffic is minimal. In applications with many sharing patterns (i.e., Apache, JBB, Bodytrack), traffic overhead due to false positives is limited to 5%. On the other hand, traffic due to multicast is increased by up to 15%. This traffic is off the critical directory lookup path, so its impact on the performance could be minimal compared to the traffic due to false positives. Note that our overall network utilization for most applications is moderate, which allows the network to support the increase in traffic.

The key to reducing this traffic is the frequency of the pattern recalculation. Recalculating on every eviction might be unnecessary because multiple hashing functions could filter out some of the false positives, meaning the recalculation traffic is unnecessary in such cases. Recalculating lazily and

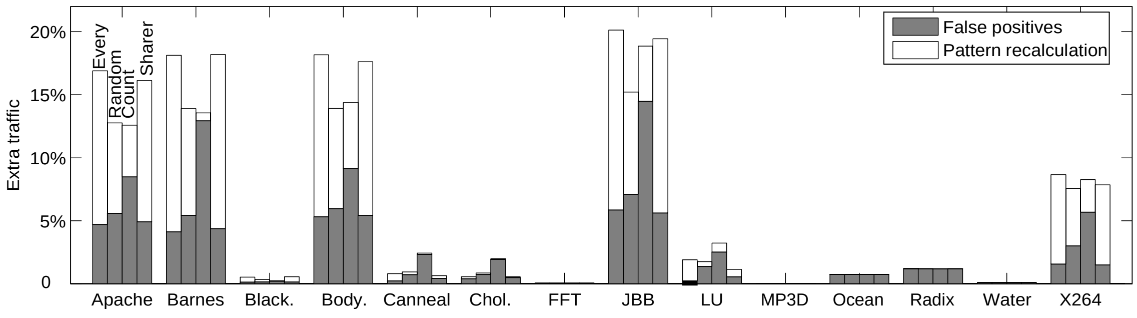


Fig. 11: Extra interconnect traffic. The four columns from left to right indicate traffic using the Every, Random, Count, and Sharer policies.

infrequently on the other hand leads to a heavily polluted pattern table, and introduces further conflicts. We explore three simple techniques to reduce the recalculation traffic here. **Random** chooses to send the recalculation message every third eviction. **Count** will only send the recalculation message if the entries pointing to the pattern reach a certain threshold (48 in the experiment). **Sharer** will send the message when the pattern indicates more than 4 sharers. Figure 11 evaluates the effects of the three methods on both traffic caused by recalculation and false positives. In general, less frequent recalculation leads to more false positives, therefore causes slight increase in traffic due to false positives. The simple random method is very effective, reducing the recalculation traffic to less than 7% for all the applications, while adding less than 1% traffic from false positives. Count method does not perform better than random because the number of entries pointing to a pattern does not translate to the frequency with which the pattern is referenced. The sharer method has the highest accuracy. However, the traffic reduction is limited.

D. Area, Energy, Delay

This section reports the area, energy, and access time of the *SPATL* directory. CACTI 6.0 [14] is used to estimate the delay and energy for a 32nm process technology. The estimated numbers at 16 cores are shown in Table III. The additional cost of accessing the small pattern directory table adds little to the access time and energy. The accessing can be finished within two CPU cycles, and both the accessing time and power consumption is significantly better than alternative directory designs including a FULL directory cache and the shadow tags directory.

The last column in Table III shows the relative area of the *SPATL* directory. The area for *SPATL* includes both the buckets of pointers and the pattern table. On top of the Tagless directory, *SPATL* further compresses the directory by 25% to 42% at 16 cores. This translates to 28% to 37% of the area of a FULL directory cache. The leakage power is proportional to the size of the memory structures. We estimate a 74% reduction in leakage power for *SPATL* with 512 entries compared to a FULL directory cache.

E. Scalability

The performance of the *SPATL* directory directly depends on the number of sharing patterns present in the cache. This is mainly influenced by the application's characteristics, the parallelization strategy, and programming patterns. However, in most architectures the cache block size and cache size have a key influence on the sharing patterns observed since they affect properties like false sharing and working set size in the cache. Figure 12 shows the influence on false positives by varying the L1 cache parameters. As the size of the L1 caches increase, the average false positives increases with more sharing patterns. However, the increase is minor after the size of the working set is reached. The influence of larger cache lines is mixed, because false sharing could lead to either increasing or decreasing sharing patterns. The false positive increases when line size increases from 32B to 64B, then decreases when line size further increases to 128B. Characterizing the influence of false sharing on sharing patterns is beyond the scope of this work.

TABLE III: **CACTI estimates for various directory settings. (The access time and read energy for *SPATL* include access of the pointer in SPACE buckets and the pattern table entry.)**

Configuration	Access Time(ns)	Read Energy(fJ)	Storage Relative to Tagless
FULL dir cache	0.55	16812	2.03×
Shadow tags	0.92	67548	1.53×
Tagless-lookup	0.27	4104	1×
SPATL-512	0.40	4299	0.58×
SPATL-1024	0.41	4394	0.66×
SPATL-2048	0.43	4486	0.75×

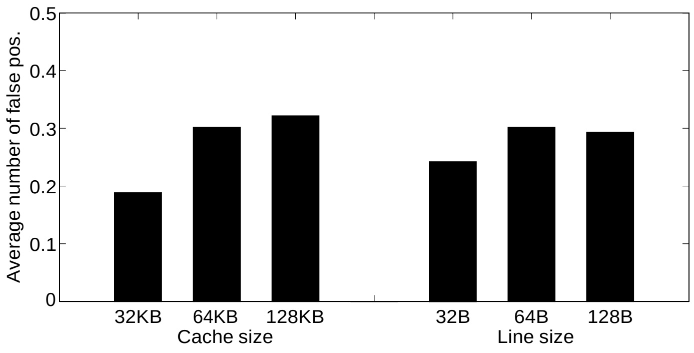


Fig. 12: Average false positives under varying L1 cache settings. The group on the left keeps the cache line size constant (64B) and varies the number of sets. The group on the right keeps the number of sets constant and varies the cache line size.

To study the scalability of the *SPATL* directory, we simulate three multicore systems (8-core CMP, 16-core CMP, and 32-core CMP). For each system, we experimented with three *SPATL* directory setups by varying the size of the pattern table. Figure 13 shows that *SPATL* with a limited number of pattern entries consistently performs similar to FULL. The network traffic is within 5% of FULL for *SPATL*-128 using 8 cores, *SPATL*-1024 using 16 cores, and *SPATL*-4096 using 32 cores. Interestingly, to achieve an effective directory, *SPATL* appears to need a pointer size of $K \ast logP$ ($K = 2.4$ in our experiments). On top of the tag compression by TAGLESS, the directory of size $M \ast P$ is further compressed to $M \ast K \ast logP$.

Figure 14 projects the size of the directory to systems up to 512 cores. Compared to the tagless directory, *SPATL* further compresses the directory by 34% at 16 cores, and by 78% at 64 cores. We also show the size of the ideal directory for 8, 16, and 32 cores. The ideal directory is a directory cache that magically holds only the tags present in the L1 caches. It represents the minimum space for an accurate directory cache. The size varies across applications and in execution, and we show the captured maximum size.*SPATL* has less overheads compared to the ideal directory cache.

Accelerator-based Manycore Architectures. An important design decision in *SPATL* is the size of the pattern table (fixed at design-time), which determines how many unique sharing patterns can be simultaneously supported. In our experience, we observed large variations between the different workload suites and in some cases outliers even within a workload suite (e.g., Barnes in SPLASH2). In our current set of experiments, we assume general-purpose multicores that can target any of these workloads. Hence, the pattern table is sized to support commercial applications like Apache and SpecJBB, which have myriad read-sharing patterns. Unfortunately, this severely over-provisions the pattern table for workloads such as SPLASH2. We now consider accelerator-like manycore architectures which target only data parallel algorithms like SPLASH2. We found that a 32 entry pattern table is sufficient for many SPLASH2 applications (other than Barnes) to perform optimally at 16 cores. Assuming linear growth in patterns with increase in cores (a reasonable assumption for data parallel workloads), we only require a 2048 entry pattern table for 1024 cores. We believe providing a cache coherence directory for a hypothetical 1024-core accelerator (64KB L1 per core) would only require 0.6MB, less than 1% of the total L1 capacity.

VI. Related Work

This section discusses different directory designs for CMP. Shadow tags duplicate all the tags present in the private cache and construct the sharing vector by looking up the tags when accessed. The design is simple in concept and works well in current multicores including SUN's Niagara2 [17]. The bigger challenge is that it requires an energy-intensive associative search to construct the sharing pattern. We have shown that using the techniques described in this paper we can improve space consumption by a factor of 3× at 64 cores without the need for associative lookup.

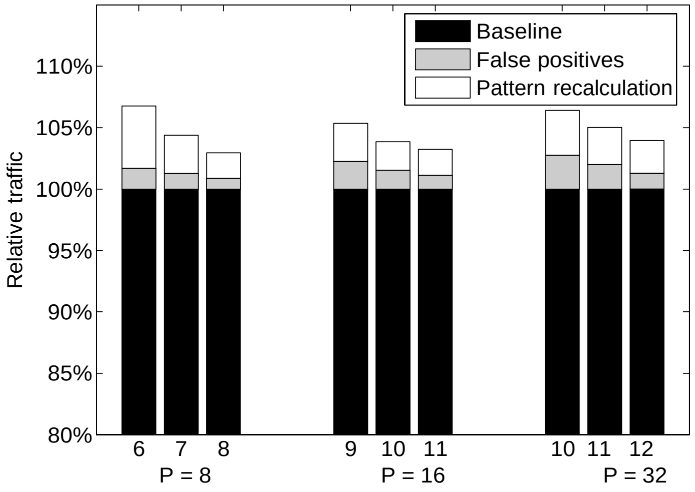


Fig. 13: Interconnect traffic for *SPATL* normalized to a full map in-cache directory. The stacked bars show the extra traffic caused by false positives and extra traffic caused by pattern recalculation. X axis represents three multicore systems (8-core, 16-core, and 32-core). We experiment with three different *SPATL* pattern table sizes (2nd X axis: # of bits of the pattern pointer. Pattern table size ($2^{\#of\ patternbits}$)).

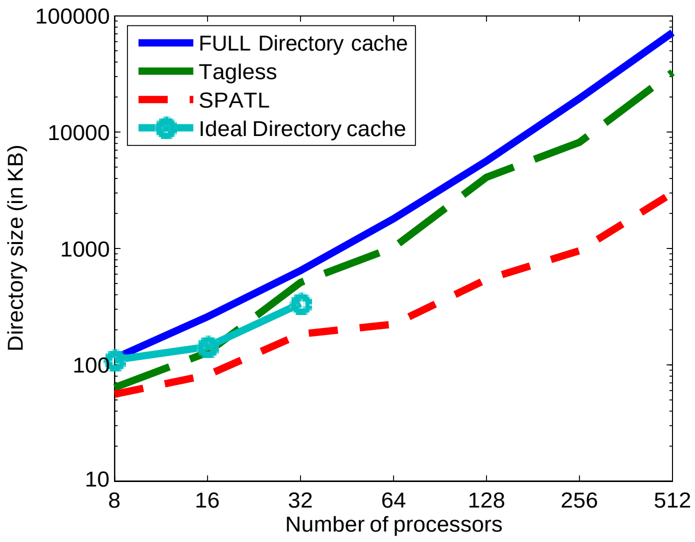


Fig. 14: Storage requirements for a FULL directory cache, Tagless, *SPATL* and ideal (all unique tags) directory cache. Each core has a 64KB private L1 cache.

Tagless directory [19] uses bloom filters to map the tags in each cache set. The bloom filters concisely summarize the tags for each set in every core and completely eliminates the associative search on lookups. Overall, it reduces storage compared to shadow tags by a factor of the number of ways in the L1 cache. The benefits of the bloom filters are limited for multicores with a large number of cores since the per-bucket sharing vector becomes a significant area overhead.

Directory cache [1], [15] limits the size of the directory by restricting the number of blocks that the directory holds the sharing information for. With this limitation, if one block is not present in the directory cache, either all the shared copies have to be invalidated, or the cache block must be defaulted to shared by all the processors. Cuckoo directory [7] uses an improved hashing algorithm to eliminate associativity-related tag evictions in the directory cache. Other proposals try to combine a small directory cache with a larger in-memory directory [10], [15]. Such designs essentially emulate a big directory cache, but they require complex protocol extensions that touch off-chip metadata, and some directory accesses will suffer long latencies.

Full map directory [5] is a simple solution for CMPs with an inclusive shared last-level cache. The bit vector indicating the sharers is associated with the cache line at the shared cache. Full map directory imposes significant storage penalty because the shared cache is usually much larger (24MB on the latest Itanium [9]) and includes lines that are not cached at lower levels. SPACE [20] sought to optimize full map by making the observation that many entries in the shared cache store redundant patterns. It decouples the sharing pattern from the directory entries, and only represent patterns present in the application. Each cache block in the inclusive cache includes a pointer to the pattern table. Unfortunately, even uncached blocks include the pointer and this leads to significant space overhead compared shadow tag-based approaches.

Coarse vectors [8], [16], sharer pointers [2], [11], and segmented vectors [6] all try to compress the sharing vector using more compact encodings. Based on the encoding type, these compressed directories can represent only a limited number of sharing patterns, and introduce imprecision (hard-coded at design time) or extra latency for other patterns.

Overall, *SPATL* is agnostic to the type of shared cache (inclusive or exclusive), affords significant compression over the previously known best approach, Tagless, and loses precision more gracefully based on an application's coherence requirements.

VII. Conclusions

We presented *SPATL*, a coherence directory that requires minimal storage (83KB at 16 cores) and can scale at least up to 512 cores (3MB storage required). *SPATL* achieves this by combining two complementary techniques that compress both the tags and the sharing patterns in the directory. *SPATL* adopts Tagless directory's approach [19] of compressing the tags using bloom filters to summarize the information in each set. *SPATL* further compresses the sharer bit vectors in the bloom filters based on the observation that due to the regular nature of programs, many cache blocks exhibit the same sharing pattern, i.e., there are only a few sharing patterns and they are replicated in many bloom filters. *SPATL* maintains a separate table to hold only the unique patterns that appear in the application. Multiple bloom filters with the same pattern point to a common entry. *SPATL* provides significant benefit over the Tagless's tag compression and achieves 34% savings in storage at 16 cores, and 78% at 64 cores. *SPATL*'s storage overhead is the minimum amongst all previous coherence directory proposals and scales better than even an idealized directory cache from 16—512 cores. Finally, the directory storage can be tuned based on the sharing patterns in the application. Many parallel workloads in SPLASH2 have few sharing patterns and we find that for a 1024-core (64KB L1) accelerator architecture that targets only these workloads, *SPATL* would need only 600KB of space (less than 1% of total aggregate L1 space).

References

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(1):67–79, 2005.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 280–298, 1988.

[3] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2m commercial server on a $2k pc. *Computer*, 36(2):50–57, 2003.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27:1112–1118, 1978.

[6] J. H. Choi and K. H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *Proc. 13th International Parallel and Distributed Processing Symp.*, pages 258–267, 1999.

[7] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: Efficient and scalable CMP coherence. In *HPCA '11: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[8] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *International Conference on Parallel Processing*, pages 312–321, 1990.

[9] Intel Corporation. Intel Itanium Processor 9300 Series Datasheet. http://download.intel.com/design/itanium/downloads/322821.pdf, Feb 2010.

[10] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel. WAYPOINT: scaling coherence to thousand-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 99–110, 2010.

[11] J. Laudon and D. Lenoski. The SGI origin: a ccNUMA highly scalable server. *SIGARCH Comput. Archit. News*, 25(2):241–251, 1997.

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

[13] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[14] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 3–14, 2007.

[15] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 138–147, 1990.

[16] R. T. Simoni, Jr. *Cache coherence directories for scalable multiprocessors*. PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[17] Sun Microsystems, Inc. Opensparc T2 system-on-chip (SoC) microarchitecture specification. http://www.opensparc.net/opensparc-t2/index.html, May 2008.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[19] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. Tagless coherence directory. In *the 42nd Annual International Symposium on Microarchitecture*, Dec. 2009.

[20] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE : Sharing pattern based directory coherence for multicore scalability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2010.