SPACE : Sharing Pattern-based Directory Coherence for Multicore Scalability*

Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas
Deptartment of Computer Science, University of Rochester
{hozhao,ashriram, sandhya}@cs.rochester.edu

ABSTRACT

An important challenge in multicore processors is the maintenance of cache coherence in a scalable manner. Directory-based protocols save bandwidth and achieve scalability by associating information about sharer cores with every cache block. As the number of cores and cache sizes increase, the directory itself adds significant area and energy overheads.

In this paper, we propose *SPACE*, a directory design based on recognizing and representing the subset of sharing patterns present in an application. SPACE takes advantage of the observation that many memory locations in an application are accessed by the same set of processors, resulting in a few sharing patterns that occur frequently. The sharing pattern of a cache block is the bit vector representing the processors that share the block. SPACE decouples the sharing pattern from each cache block and holds them in a separate directory table. Multiple cache lines that have the same sharing pattern point to a common entry in the directory table. In addition, when the table capacity is exceeded, patterns that are similar to each other are dynamically collated into a single entry.

Our results show that overall, SPACE is within 2% of the performance of a conventional directory. When compared to coarse vector directories, dynamically collating similar patterns eliminates more false sharers. Our experimentation also reveals that a small directory table (256-512 entries) can handle the access patterns in many applications, with the *SPACE* directory table size being $O(P)$ and requiring a pointer per cache line whose size is $O(log_2 P)$. Specifically, *SPACE* requires $\simeq$ 44% of the area of a conventional directory at 16 processors and 25% at 32 processors.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—Shared memory; C.1.2 [Processor Architectures]:Multiprocessors;

General Terms: Design, Experimentation, Performance

Keywords: Directory coherence, Cache coherence, Multicore scalability, SPACE.

1. INTRODUCTION

Multicore processors continue to provide a hardware coherent memory space to facilitate effective sharing across cores. As the number of cores on a chip increases with improvements in technology, implementing coherence in a scalable manner remains an increasing challenge. Snoopy and broadcast protocols forward coherence messages to all processors in the system and are bandwidth intensive. They also have inherent limitations in both performance and energy and it is unlikely that they will be able to effectively scale to large core counts.

Directory-based protocols are able to support more scalable coherence by associating information about sharer cores with every cache line. However, as the number of cores and cache sizes increase, the directory itself adds significant area and energy overheads. The conceptually simple approach is to adapt a full map sharer directory [6] and associate a P-bit vector (where P is the number of processors) with every cache line. Unfortunately, this makes the directory size dependent on the number of shared cache lines (M) and the number of processors, resulting in a directory size that is $O(M * P)$.

One possible alternative is shadow tags, which are used in many current processors including Niagara2 [17]. Essentially, each private L1 cache's address tags are replicated at a logically centralized directory structure. On every coherence access, the shadow tags are consulted to construct sharer information dynamically — this is a highly associative and energy intensive lookup operation. Tagless lookup [19] was recently proposed to optimize the shadow tag space by compressing the replicated L1 cache tags. This approach uses a set of bloom filters to concisely summarize tags in each cache set. The energy intensive associative lookup needed by shadow tags is thus replaced with bloom filter tests.

Various other approaches have been proposed to reduce the area overheads of a full bit map directory, including the use of a directory cache [1, 15], a compressed sharer vector [8, 9, 16], and pointers [2, 11]. Directory caches restrict the blocks for which precise sharing information can be maintained simultaneously. Compressed sharer vectors fix the level of imprecision at design time — all cache lines suffer from imprecision. Pointers incur significant penalty (due to the need to revert to either software or broadcast mode) when the number of sharers exceeds the number of pointers. They are inefficient at representing a large number of sharers.

In this paper, we optimize the directory by recognizing and representing the sharing patterns present in an application. The sharing pattern of a cache line is the bit vector representing the processors that share the block. Our approach, SPACE (Sharing PAttern based CoherencE) takes advantage of the observation that many memory locations in an application are accessed by the same set of processors, resulting in a few sharing patterns that occur frequently. A full map directory is highly inefficient and essentially duplicates the same sharing pattern across many cache lines. SPACE decouples the sharing pattern from each cache line and holds them in a separate directory table. Multiple cache lines that have the same sharing pattern point to a common entry in the directory table. The directory area overhead of SPACE is thus $O(M \times log_2 K)$, where K is the logical number of entries in the directory table, assuming that

*This work was supported in part by NSF grants CCF-0702505, CNS-0615139, CNS-0834451, and CNS-0509270.

PACT'10, September 11–15, 2010, Vienna, Austria.

M is significantly larger than P so that this factor dominates the area of the directory table itself (which is $O(K \times P)$). SPACE simplifies the design by using a constant number of entries in the directory table (fixed at design time). The major challenge with this approach is how SPACE accommodates new sharing patterns when the directory table capacity is exceeded. Unlike directory caches that default to broadcast for these cases, SPACE degrades gracefully by dynamically collating patterns that are similar to each other into a single sharer pattern.

We evaluate SPACE using a full system simulator. Our results show that overall, SPACE is within 2% of the performance of a conventional full map directory. Our experiments comparing SPACE against coarse vector [9] approaches demonstrates that dynamically collating similar patterns achieves significant reduction in false sharers (almost completely eliminates it). Our experimentation also reveals that a small directory table (256 entries for a 16 core chip, 512 entries for a 32 core chip) can handle the access patterns in many applications. Empirical results indicate that the number of directory table entries is typically $O(P)$. Hence, the area required for the pointers per cache line is $O(log_2 P)$, which results in increasing savings with SPACE with increasing number of processors — *SPACE* occupies $\simeq 44\%$ the area of a full map directory at 16 processors, and 25% at 32 processors.

2. BACKGROUND

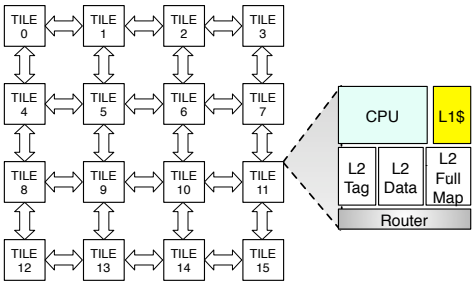


Figure 1: **Baseline tiled multicore architecture. L2 Full Map: sharer vectors associated with cache line.**

In this work, we study the organization of directories in the context of a tiled multicore shown in Figure 1. Each tile in the multicore consists of a processor core, private L1 (both I and D) cache, and a bank of the globally-shared last-level L2 cache. Coherence at the L1 level is maintained using an invalidation-based directory protocol and directory entries are maintained at the home L2 bank of a cache line.

Full bit map directories [6] are an attractive approach that was first proposed for multiprocessors but can be extended to maintain coherence in multicores with an inclusive shared cache. The sharing cores are represented as a bit-vector associated with each cache block, with each bit representing whether the corresponding core has a copy of the block. Sharer information is accessed in parallel with the data. The main disadvantage of this approach is the area overhead. The capacity of the shared L2 cache exceeds the total capacity of the L1 caches and unfortunately the directory size is directly correlated with the number of cache lines in the L2.

To ensure that the coherence mechanism scales to large multicores, a directory design is required that makes effective use of space, maintains accurate information to ensure that coherence traffic does not waste bandwidth, and provides a complexity-effective implementation. We believe these goals can be attained by exploiting the sharing patterns prevalent in applications. The key observation is that applications demonstrate few unique sharing patterns, which are stable across time, and many cache lines essentially display the same sharing pattern. Here, we use the term sharing pattern to refer to the group of processors accessing a single location. In the next section, we describe how we use sharing pattern locality to decouple the directory from the cache sizes and number of cores and provide a scalable coherence framework.

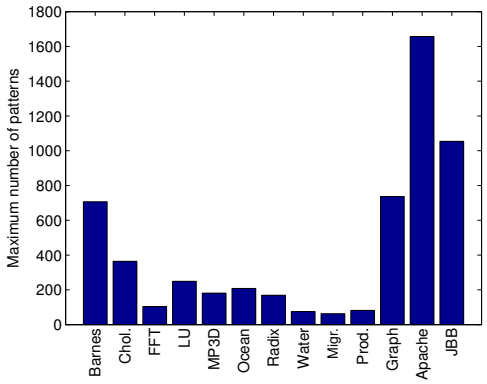


Figure 2: **Maximum number of sharing patterns in any snapshot. (16 processor system. Other system parameters specified in Table 1a.)**

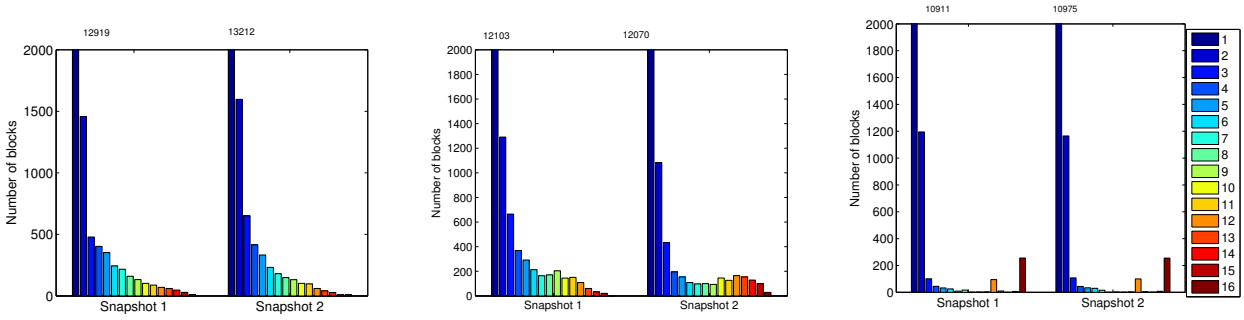
3. SHARING PATTERN-BASED DIRECTORY COHERENCE (SPACE)

3.1 Sharing Patterns

The sharing pattern of a cache line can be represented as a P-bit vector (where P is the number of processors), with each bit specifying if the corresponding processor has a copy of the block. The maximum number of sharing patterns possible is 2^P. A conventional directory will assume that each cache line has a unique pattern and that this pattern could be any one of 2^P. Hence, each cache line has an associated P-bit sharing pattern. We make an observation about sharing pattern locality — many cache lines in the application are accessed by the same set of processors, which essentially leads to the same sharing bit-vector pattern. Because of application semantics and the regular nature of inter-thread sharing, it is also likely for a system to repeatedly encounter the same set of patterns.

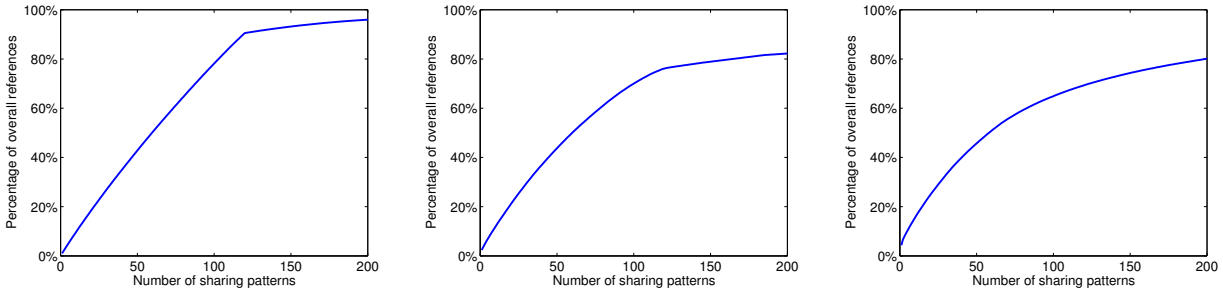
Figure 2 shows the maximum number of patterns encountered in an application during any snapshot of its execution (examined every 100,000 instructions) for a 16 processor system [1]. Although the maximum possible number of patterns is 65536 at 16 processors, we can see that the actual number of patterns observed in our application suite did not exceed 1800. We can also see that the commercial workloads (Apache and SPECjbb) tend to have many different patterns, while scientific workloads (SPLASH2 [18]) have a limited number of sharing patterns with regular data accesses. This 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 the sharing patterns that occur without assuming that each cache line demonstrates a unique pattern.

[1]Except for Apache and SPECjbb, these numbers are only for application data and instructions, with the goal of demonstrating individual application behavior. Since Apache and SPECjbb make a considerable number of system calls, we include operating system references since we cannot separate those made only on behalf of the application.



(a) Apache (b) SPECjbb (c) Barnes

Figure 3: **Number of cache blocks for each of *N* sharers.**



(a) Apache (b) Specjbb (c) Barnes

Figure 4: **Cumulative distribution of references over sharing patterns. X axis represents the patterns, ordered by frequency of access. Y axis is the cumulative distribution of the references accessing the patterns.**

An important metric of interest is the distribution of cache lines with the same sharing patterns. If there exists good sharing pattern locality (many cache lines display the same sharing pattern), it would increase the effectiveness of a directory based on common sharing patterns. A single entry can describe the sharing pattern of many cache lines.

Figure 3 shows the degree of sharing across two different snapshots of three different applications. Each bar in the histogram represents the number of cache lines with patterns with a certain number of processors sharing the cache line. Each cluster in the graph represents a snapshot of the sharing information of cached blocks. We took these snapshots at regular intervals (1000 transactions for Apache and SPECjbb, 1 million instructions for SPLASH2) and observed that snapshots remained steady over time. Here, we only show two snapshots. We observe that private accesses are the dominant sharing pattern, exhibited by over 70% of the blocks in Apache and SPECjbb, and 90% of the blocks in Barnes (other SPLASH2 workloads demonstrate similar behavior).

We also observed that the distribution of the blocks across the patterns remains relatively stable. This indicates that while a given cache line's sharing pattern may change, overall, the application accesses cache lines in a similar manner. Note that the histogram tails off sharply, indicating that the number of sharers per cache line is small on average.

Our next experiment studies the frequency with which the patterns in the directory are referenced. The sharing pattern of a cache line is referred to only on two occasions: downgrades, when a copy is requested by a processor and an L1 has a modified version, and invalidations, when a processor wants to modify the location and the sharer copies need to be invalidated. In addition, the sharing pattern is also updated when a copy of a read-shared line is requested. Figure 4 shows the number of patterns that get frequently referenced. In all these applications, more than 80% of the total references goes to the 200 most frequently accessed patterns. This indicates that imprecisely maintaining the sharers for many shared cache lines (e.g., read-only) will not hurt performance. The linear curve segment in Apache indicates that the number of references are uniformly distributed across all of the frequently accessed patterns.

Overall, we can draw the following important conclusions:

1. Many cache lines have a common sharing pattern and the number of the patterns that are frequently referenced is small. This implies that a directory that supports a few sharing patterns can be sufficiently accurate to save bandwidth.

2. The number of patterns that a directory needs to support for a real application is 40$\times$ (Apache, SPECjbb) - 800 $\times$ (Water) less than the maximum number of sharing patterns.

3. The number of patterns in the directory will remain relatively stable during the execution. This suggests that the total number of sharing patterns that a directory has to support for a given applications can be fixed.

4. Finally, there is significant variation between the applications with regards to the maximum number of sharing patterns : $\simeq$75 (Water) - 1700 (Apache) (see Figure 2. This suggests that a directory sized based on the worst-case (1700) would waste significant space for some applications. On the other

hand a directory with fewer entries would be space efficient but would need an effective way of dynamically managing patterns when there are no free entries. We describe one such scheme in the next section.

3.2 Key Idea

As compared to a conventional full map directory [16] that stores the sharing patterns along with the cache line, SPACE decouples the sharing pattern from each cache line and holds them in a separate directory table. Multiple cache lines that have the same sharing pattern point to a common entry in the directory table. With the directory table storing the patterns, each cache line includes a pointer whose size is proportional to the number of entries in the directory. We organize the directory table as a two-dimensional structure with $N_{Dir.ways}$ ways and $N_{Dir.sets}$. The size of the directory table in our current design is fixed (derived from the application characteristics in Section 3.1) 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.

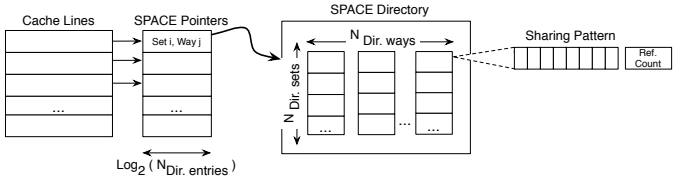


Figure 5: **SPACE directory organization.** $N_{Dir.entries} = N_{Dir.sets} * N_{Dir.ways}$ **represents the number of patterns in the directory. Ref. count indicates reference counter associated with each sharing pattern.**

In this section, we describe our directory implemented on a multicore with 16 processors, with 64KB private L1 caches per core, and a 16MB shared inclusive L2 cache. The conventional full map directory design would include an entry for each cache line for a total of 262144 (16MB/64 byte) entries. Figure 5 illustrates the SPACE 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 16-bit vector. For each cache line in the L2 cache, we replace the sharing vector with a $\lceil log_2(N_{Dir.entries}) \rceil$ bit pointer to indicate the sharing pattern. Every time the sharer information is needed, the data block tag is first accessed, 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 line.

Notice that SPACE essentially decouples the directory organization from the caches and can optimize the size of the directory based on various constraints (e.g., energy, area, and/or latency). If space is not the main constraint, we can choose to have a large $N_{Dir.entries}$, all the way to the extreme case of using a unique entry for each cache line (emulating the baseline full map approach). A SPACE design with $N_{Dir.entries} = 1$ essentially defaults to a broadcast protocol. With on-chip directory area being an important constraint and with applications demonstrating regular sharing patterns, we investigate SPACE designs with varying $N_{Dir.entries}$ (see Section 5).

3.3 SPACE Implementation

The main structure required to incorporate SPACE in a multicore is the directory table that stores the sharing patterns. We organize the sharing pattern table as a set-associative structure with $N_{Dir.ways}$ ways and $N_{Dir.sets}$ sets to support a total of $N_{Dir.ways} * N_{Dir.sets}$ sharing patterns. A pointer in each LLC's (last level cache's) cache line tag identifies the set and way containing the cache line's sharing pattern and is used to index the directory table for a sharing pattern lookup. This section describes how SPACE inserts entries into the directory table, how patterns are dynamically collated when there are no free entries, and how entries are removed from the table.

Inserting a new pattern

When a cache line is accessed and a sharing pattern changes (or appears for the first time), the pattern needs to be inserted in the directory table. Once a free entry is found in the directory table, the set index and way are used by the cache line to access the specific entry. The key challenge is to determine where in the set-associative directory table the new sharing pattern should be stored. Intuitively, the hash function that calculates the set index has to be unbiased so as to not increase pollution in any given set. With SPACE, we also require that similar patterns map to the same set so as to enable better collation of entries when the protocol runs out of free directory entries.

To satisfy these two seemingly contradictory goals we use a simple encoding scheme: we encode the full sharer bit-vector into a compressed bit vector of fewer bits as shown in Figure 6, with each bit standing for sharers existing in the specific cluster. For instance, for a multicore with 16 cores in a 4x4 mesh and with $N_{Dir.sets}$=16, the 16-bit sharing vector will be compressed to a 4-bit vector, each bit corresponding to whether a sharer exists in one of the four 2x2 clusters. Then the compressed $log_2(N_{Dir.Sets})$ bit vector will be used to index into the directory table. This process is illustrated in Figure 6.

The main advantage of this hashing function is that it considers all the bits in the sharing vector and removes bias towards any specific subset of the processors. Since sharing pattern variations within a cluster map to the same set, it also enables us to perform effective collation of patterns (when there are no free patterns available) — extra coherence messages are limited to within the same cluster.

Since private and globally-shared (all processors cache a copy) patterns appear to be common patterns across all the applications, SPACE dedicates explicit directory indices for these $P+1$ patterns (where P is the number of processors). Hence, for lines modified by a specific processor ("M" state), SPACE will represent the processor accurately, which also helps with the implementation of the coherence protocol (see Section 3.4). Note that these entries do not need to be backed by physical storage, since their decoded patterns are fixed and known at design time.

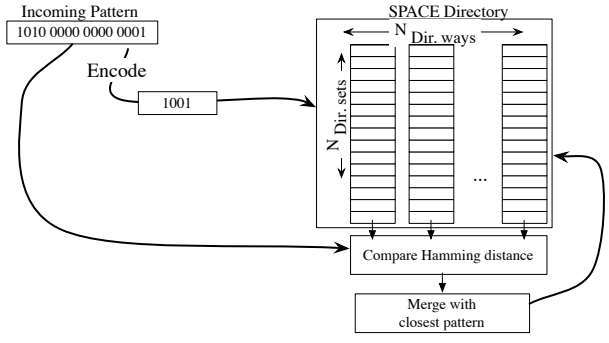


Figure 6: **Inserting a pattern into the directory.**

Merging with existing pattern

A key challenge that a fixed size directory needs to deal with is the appearance of new patterns in the application when the directory is already completely filled. When the pattern is added to the directory, SPACE searches all the ways in the particular set-index for a matching pattern. If there exists a matching entry, SPACE will simply provide the cache line with a pointer to this entry.

In SPACE, when the incoming sharing pattern maps to a set with no free entries, it is merged with one of the existing patterns in the set. Cache lines already pointing to the entry can continue to use the sharing pattern specified although it is no longer precise. SPACE does try to ensure that such merging minimizes the pollution of the existing entry. We achieve this by determining the hamming distance of the incoming pattern to all the existing patterns in the set. This distance is the number of bit positions in which the incoming pattern differs from the existing pattern, and indicates the number of extra false sharers that would be caused by the merging. After determining this distance by an XOR function, the incoming pattern will merge with the existing pattern with least hamming distance (minimal false sharers) using the simple OR operation.

This novel technique of merging sharing patterns ensures that existing cache lines that point to the entry will suffer minimal performance penalty because of the extra sharer bits. This is one of the key contributions of the SPACE design: when the directory runs out of space, SPACE will dynamically collate sharing patterns similar to each other. In comparison, a sparse directory does not consider an application's sharing patterns when compressing entries and uses a static approach based on clusters of processors.

Removal of a stale pattern

Finally, the last challenge that needs to be addressed is to ensure that entries in the directory are re-usable once no cache block has the sharing pattern in the entry. Recycling entries by removing stale patterns is vital for SPACE, because the system would otherwise fall back to broadcast with new patterns continually merging with stale patterns in the limited entries.

SPACE elects to use a simple scheme of reference counting to detect when an entry becomes stale. A counter is associated with each entry in the directory. This counter is incremented when a new cache line starts pointing to the entry and is decremented when a cache line pointing to the entry changes its pointer (either the cache line was evicted or it changed its sharing pattern). The entry is reclaimed when the counter reaches zero. For simplicity, we assume that the counter includes $log_2 M$ bits to deal with the worst case when all the M cached lines have the same pattern. The overhead of these counters itself is a small fraction of the overall area and —- with a 512-entry SPACE directory and a 64MB L2 cache, the counter bits only consumes 0.1% of the overall space consumption. Alternatively, if a smaller reference counter size is used, in case of saturation, additional cache lines could use additional ways in the set or default to broadcast.

3.4 Protocol Support for SPACE

An interesting challenge that SPACE introduces is that it is possible for the directory to provide an inaccurate list of sharers to the coherence protocols. This occurs when SPACE merges sharing patterns conservatively due to lack of space. Note that SPACE's inaccuracy will at most cause false positives (indicate processors which are not actual sharers) but not false negatives (miss an actual sharer). Similar imprecision also exists in the existing sparse directory designs but SPACE performs an important optimization that resolves the case for the modified state. SPACE keeps the private access sharing pattern locked down and ensures that the single sharer is accurately provided to the coherence protocol. The coherence protocol's action in this case does not have to change. This is a notable improvement over previous approaches [19], which introduced additional protocol complexity to deal with imprecision in this situation. The only inaccuracy the coherence protocol needs to deal with is false-positive sharers for which invalidations will be forwarded. This can be handled with an acknowledgment of invalidation sent by a false sharer even if it does not have the cache block. False-positive sharer invalidation is also only incurred when the number of sharing patterns exceeds the directory capacity (or incurs conflicts).

3.5 Centralized or Tiled Directory Table

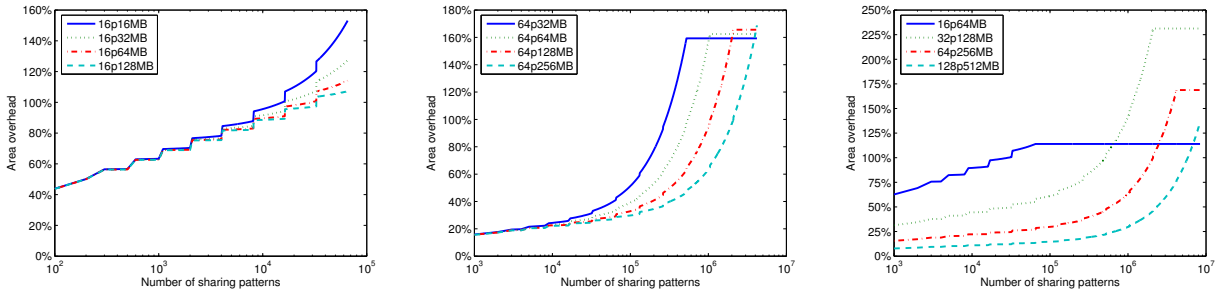
In tiled multicores as the one discussed in Section 2, there is an interesting design choice for SPACE: bank and distribute a fraction of the directory sharing pattern table to each tile or maintain a centralized table for the whole chip. Distribution works well in terms of latency without an increase in area overhead if the sharing patterns across tiles are disjoint. However, with a line-interleaved LLC (last level cache), overlap of sharing patterns is highly likely. A centralized table would thus save area, since to support $N_{patterns}$ sharing patterns, a centralized directory would require $N_{patterns}$ directory entries, while a tiled directory would in the worst case require $N_{patterns}$ per tile.

Interestingly, comparing the overheads of tiled versus centralized SPACE directories is dominated by the pointers in the cache line. For both the tiled case and the centralized case, to support $N_{patterns}$ requires $log_2(N_{patterns})$ per cache line. In a 16-processor multicore, the area overhead of the centralized-SPACE is only 1% smaller than the tiled-SPACE with a 512-entry pattern directory table. The centralized-SPACE does impose significant latency since each cache bank has to communicate over the network to initiate a directory lookup. The overheads of this directory access penalty appeared to have a significant impact on performance and in this paper, we focus on investigating the tiled-SPACE design.

3.6 Area Overheads

The area overhead for SPACE includes two main components (1) the pointers to the directory entry associated with each cache line and (2) the sharing bit-vectors in the directory itself. Consider a system that has M cache lines and P cores. Let $N_{patterns}$ be the number of sharing patterns in the application. A conventional full map directory associates a P bit pointer with each cache line and consumes $M*P$ bits. With SPACE, the pointers associated with each cache line requires $\lceil log_2 N_{patterns} \rceil$ and for M cache lines consumes $M * \lceil log_2 N_{patterns} \rceil$ bits. The directory table itself is a $N_{patterns} * P$ bit array. Overall, to achieve better space efficiency than the conventional directory, the following condition has to be satisfied: $M*P > M*\lceil log_2 N_{patterns} \rceil + N_{patterns}*P$. The directory can only have as many patterns as the number of cached blocks and this value is also bounded by the number of patterns possible with a P-bit vector. Hence, the maximum value for $N_{patterns}$ is $MIN(2^P, M)$.

In our implementation of SPACE (see Figure 5), there is an additional $log_2 M$ reference count bits per table entry. We include this overhead in Figure 7(a) and (b) to show SPACE's storage requirements compared to the traditional full bit vector directory design with varying L2 sizes for two different processor configurations. In the case of 16 cores and a 64MB L2 cache (see Figure 7(a)), SPACEs' directory will have less storage overhead comparing to the full map directory if it supports less than 42000 patterns. The maximum possible patterns in this configuration is limited by the number of processors to 65536 (Min (65536,1024*1024)) — SPACE needs one 16-bit vector per cache line for the pointer to



(a) 16 processors (b) 64 processors (c) 4MB L2 Cache

Figure 7: **SPACE's area overhead compared to a full map directory. X axis is the number of patterns in the directory. (a) and (b) present the overhead varying the L2 size at a fixed number of cores (16 and 64 respectively). Each line represents a specific configuration of L2. (c) presents the overhead varying the number of cores while keeping the L2 size/core constant. The curves flat-line (or terminate) when they reach the maximum possible number of sharing patterns possible in the system, which is $MIN(2^P, M)$.**

represent all the patterns, which is exactly the same as the per cache line vector size in full map. Even in this worst case, the extra overhead SPACE is paying is simply the size of the directory table, which is a small fraction of the overall L2, 0.2% of a 128MB L2 and 0.4% of a 64MB L2. In the 64 processor multicore, the maximum number of patterns that would need to be supported is limited by the number of cache lines, M ($MIN(2^{64}, M)$). As long as the maximum number of patterns required by an application is less than $2.5 * 10^5$, SPACE will have a smaller area overhead than the conventional directory.

As discussed in Section 3.1, most applications require the directory to maintain a small number of patterns varying between 75 – 1700. Interestingly, in this range, for multicores in the near future (64-256 cores), the number of cache lines in the shared cache $M >> N_{patterns}$ and empirically, $N_{patterns} << 2^P$. Overall, in SPACE, the overhead of the pointers associated with each L2 cache line is the dominant overhead since $M * \lceil log_2 N_{patterns} \rceil >> N_{patterns} * (P + log_2 M)$. To study the properties of such a system, we study the overheads of SPACE varying the number of cores while keeping the cache size / core constant. Figure 7(c) demonstrates that at 1000 entries (X axis start), SPACE requires $\simeq$60% of the area when compared to a full map directory for 16 cores, and $\simeq$20% for 64 cores. Figure 7(c) also shows that at a large number of entries (not required by our applications) the directory table itself becomes a large fraction of the overall area overhead. The curves are all terminated when they reach the maximum possible number of sharing patterns possible in the system, which is $MIN(2^P, M)$.

Note that the size of the sharing vector will also grow linearly with respect to the number of processors in the system (P). As a result, in the conventional full map directory the percentage of the whole cache used as the directory will grow linearly, and does not scale well with the number of cores. For the SPACE design, as we will show in Section 5, with the per cache line pointer being the dominant overhead, the size of the pointer will grow sub-linearly with respect to P since $\lceil log_2 N_{patterns} \rceil$ bits are needed per cache line, and $N_{patterns}$ grows linearly with P according to our empirical data. SPACE is therefore a more scalable approach to directory design for multicore platforms.

4. RELATED WORK

There have been primarily three different approaches for implementing the directory, (1) Shadow tag, which maintain a copy of the address tags of each L1 cache and probes these tags to construct the sharing vector, (2) Full map, which associates the sharer bit vector with the cache line in the shared level, and (3) Directory-cache, which stores the sharing information for a subset of the cache lines. Shadow tags are conceptually simple and replicate the private L1's address tags for each processor at a logically centralized directory structure. While shadow tags work well for current designs (e.g., Niagara2 [17]), which have a small number of cores and small L1s, they are challenging to implement. There are area and energy penalties for organizing and accessing a tag table with $N_{L1-sets} * N_{L1-ways} * N_{Cores}$ entries and associatively searching $N_{L1-ways} * N_{Cores}$ on each directory reference. Most recently, Tagless-Lookup [19] optimized shadow tags by using a bloom filter to conservatively represent the address tags of all the ways in an L1 cache set. Tagless-lookup uses multiple independent bloom filters to eliminate false positives. Each bucket in the bloom filter holds a sharing vector, which represents a conservative sharing pattern (false positives only) for addresses that map to that bucket. It optimizes the size of the shadow table down to $N_{L1-sets} * N_{Cores}$ entries, and has an empirical saving of 2$\times$ compared to the base shadow tag approach. The tradeoff between SPACE and Tagless-lookup is one of area versus dynamic energy. SPACE has higher area overheads due to the pointer needed per L2 cache line, while tagless's area is proportional to the L1 cache size (much smaller than the L2). Similar to shadow tags, Tagless-lookup requires higher dynamic energy to lookup the bloom filters and dynamically construct the sharing pattern on each directory reference. With the tagless lookup, each directory reference requires a lookup of multiple independent bloom filters. Also, bloom filters are known to be a challenge to design [20] since false positive rates scale sub-linearly with the filter size.

Full map directories [6] can be used to primarily target multicore chips with inclusive shared caches. Essentially, since a line at the L1 cache level is included at the shared level (L2 in our baseline system), a bit vector is directly associated with the shared level cache line. This provides an efficient representation for shared lines since it only uses a single address tag to represent the cache line and represents the sharer core using one bit per core — an overhead of T+N_{Cores} bits. Furthermore, the directory information is accessed along with the data access and this simplifies the mechanism needed to update and read the directory information. The major challenge with this approach is that typically a shared cache is much larger than the total capacity of the L1s and hence the capacity requirements are greater than the shadow tag approach. For example, for an Intel core 2 duo [10] (with 32KB private D-cache and 2MB shared L2) the shadow tags approach would consume 48*512*2 bits while the full map approach would consume 32 *

1024 * 2 bits. The full map approach consumes 1.33× more area than the shadow tag approach.

Coarse sharer vectors [9, 16], sharer pointers [2, 7, 11], and segmented sharer vectors [8] all change the encoding of the sharer map to represent (a subset of the) sharers more compactly. Such designs, however, can introduce significant false-positives in the sharer maps and/or represent only a limited number of sharers precisely and efficiently, incurring significant penalties when there is a mismatch between the sharing pattern and the hardware encoding scheme.

Directory caches [1, 15] seek to optimize the space by holding sharing information for only a subset of the total number of cached blocks. This introduces a high latency penalty for blocks that cannot be represented in the directory. The coherence protocol either has to invalidate all the cached copies or default to broadcast in this situation. Since each directory cache entry is used by a single cache line, the cache may be filled with identical sharing patterns, making space utilization inefficient. Previous research [1] has proposed the coupling of a small on-chip directory cache with a much larger in-memory directory cache. This requires coherence protocol extensions to support an off-chip directory structure and pathological cases can cause some cache lines to suffer long latencies.

4.1 Our Approach: SPACE

In many ways, SPACE's directory table is similar to a directory cache and tries to optimize for the common case. SPACE, however, focuses on frequently needed sharing patterns and exploits the overall sharing trends in the application to minimize space while directory caches track commonly accessed cache lines. Compared to directory caches in which each entry represents the pattern of a unique cache block, SPACE's pattern directory improves the utilization of every entry. Each entry represents the information for multiple cache blocks (possibly all the cache blocks) in the shared cache. SPACE achieves this by eliminating the address tags and using pointers in the shared cache to explicitly map a location to a given pattern entry. This permits effective use of each pattern entry since the entry is no longer associated with one specific cache block. Furthermore, when the pattern table has no free space, SPACE dynamically collates the blocks' sharing pattern with a similar pattern leading to a conservative expression that gradually increases the possibility of false-positive sharers without defaulting to the extreme measures (evicting or broadcasting) of the directory cache. SPACE has area overhead similar to the coarse vector approach, while keeping the network utilization comparable to the full map approach. Our analytical evaluation and empirical experiments reveal that we can attain performance comparable to the baseline full map approach while requiring much less area.

5. PERFORMANCE EVALUATION

5.1 Experiment Setup

To evaluate our proposed SPACE design, we conduct our experiments on 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 cache. The multiple tiles are connected with a 4x4 mesh interconnect (for the 16-tile case). For the baseline coherence protocol, we use a non-uniform-shared L2 (L2S) MSI directory coherence protocol. For the SPACE design, each tile also contains a segment of the pattern directory table. We charge a 2 cycle penalty (averaged across lookups and updates) for each SPACE directory access. The cache lines in a particular tile can only use entries from the SPACE directory associated with the tile. Table 1a shows the parameters of our simulation framework.

For the network modeling, we use GEMS' interconnect model. We employ a 4x4 mesh network with virtual cut-through routing. The link width is 16 bytes and can transmit 16 bytes/cycle. We simulate two forms of packets: 8-byte control packets for coherence messages and 72-byte payload packets for the data messages.

We use a wide range of workloads, which include commercial server workloads [4] (Apache and SPECjbb2005), scientific applications (SPLASH2 [18]), and graph mining (Graphmine [5]). We also include two microbenchmarks, migratory and producer-consumer, with specific sharing patterns. Table 1b 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 SPACE directory size.

We compare against the following coherence designs:

Full Map Directory (FULL)

In this system, a 16 bit sharer vector is associated with each cache line and accurately specifies if a particular processor holds a copy of the cache line. In our base configuration, there are a total of 1M cache lines in the L2, leading to an overall directory overhead of 2MB.

Coarse Bit Map Directory (COARSE-8)

A potential solution to the directory space is to represent sharing at a coarser granularity. Each cache line is associated with an 8-bit sharer vector, with each bit representing if a block is cached in any core of a 2-core cluster. The area overhead of the 8-bit vector directory is exactly 50% of FULL. We demonstrate that COARSE's simple encoding mechanism leads to many false positives.

Broadcast (BCAST)

In this scheme, we simulate a form of ordered broadcast similar to the AMD Hammer [3]. A request is sent from the L1 cache to the home L2 slice before being forwarded to all tiles. The broadcast design is representative of an extreme design point in which the coherence protocol does not keep any sharer information. Note that in our baseline the inclusive shared L2 will supply the data for read accesses when the L2 copy is up to date.

BroadcastM (BCAST-M)

BCAST-M performs a simple optimization over BCAST: it keeps extra information at each L2 cache line when an L1 has a modified copy of the line. Maintaining a pointer to any one of 16 processors requires associating a 4-bit pointer with the cache line. BCAST-M also represents an extreme design point for SPACE, since in our design, the smallest SPACE directory requires owners to be represented accurately.

SPACE-N

We also study a range of SPACE design points varying the per-tile directory table from 32 — 512 entries. We fix the number of sets in the table at 16 and vary the associativity.

5.2 How accurate is SPACE?

In our first set of experiments, we estimate the accuracy of sharing patterns maintained in SPACE. In a directory protocol, coherence operations refer to the sharer information to forward coherence messages and the accuracy of tracking sharers can have a significant performance, network utilization, and energy impact. Overall, we find that SPACE-256 has close to perfect accuracy, while COARSE-8 has a significant number of false-sharer specifications, in all cases specifying at least 1 false sharer (see Figure 8). In terms of area overhead, SPACE and COARSE-8 are comparable

Table 1: **Evaluation Framework**

(a) **Target System parameters**

Processors	
Processor	16 cores, 3.0 GHz, In-order
Cache parameters	
Private L1	64KB, 2-way, 64-byte blocks
L1 Latency	2 cycles
Shared L2	64MB, 16-way, 64-byte blocks, 4MB per tile
L2 latency	2 cycles tag, 14 cycles data
Interconnect	
Topology	4x4 mesh
Links	2 cycles, 1.5GHz, 128-bit width
Main Memory	
4GB, 300 cycles	

(b) **Benchmark Suite**

Benchmark	Simulated problem size	Max. # sharing patterns	Network Utilization	Invalidation messages rate
Apache	80000 Tx ffwd, 2000 Tx warmup, and 3000 Tx execution	1657	11.6%	2.3%
JBB2005	350K Tx ffwd, 3000 Tx warmup, and 3000 Tx execution	1054	8.5%	2.5%
Barnes	8K particles; run-to-completion	707	3.3%	2.3%
Cholesky	lshp.0; run-to-completion	364	2.6%	2.0%
FFT	64K points; run-to-completion	104	3.7%	1.4%
LU	512x512 matrix,16x16 block; run-to-completion	249	1.9%	1.6%
MP3D	40K molecules; 15 parallel steps	181	6.1%	20.3%
Ocean	258x258 ocean	208	5.7%	1.8%
Radix	550K 20-bit integers, radix 1024	169	5.0%	1.7%
Water	512 molecules; run-to-completion	75	2.7%	0.2%
GraphMine	340 chemical compounds, 24 atoms, 66 atom types, and 4 bonds; 300 nodes exploration	736	2.0%	2.3%
Migratory	512 exclusive access cache lines	63	0.6%	24.5%
ProdCon	2K shared cache lines and 8K private cache lines	82	1.5%	24.9%

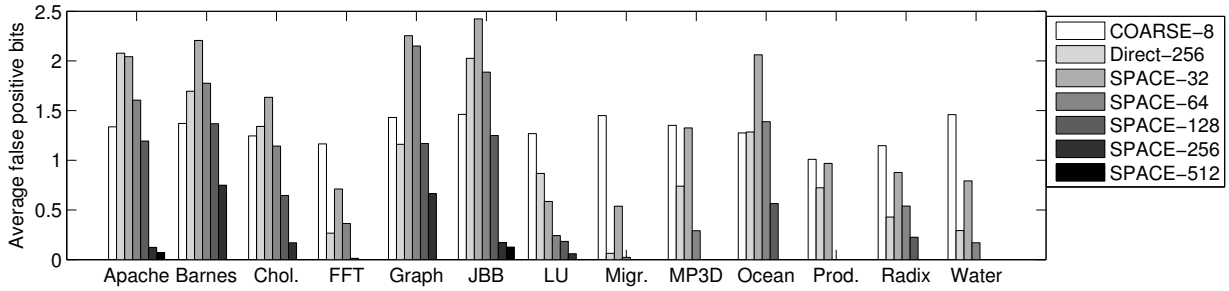


Figure 8: **Average number of false positive bits per reference to the directory.**

at 16 processors — ≃50% the size of FULL. Even SPACE-128 will have fewer false positive bits than the 8-bit vector directory in all the applications — SPACE-128 is 44% the size of FULL. The results confirm that SPACE's sharing pattern collation mechanism is an effective technique that adds false sharers incrementally, compared to COARSE-8, which coarsely represents all sharing patterns and introduces many false sharers.

SPACE also makes effective use of increased directory space. We show that SPACE uses the increase in directory entries to accurately represent the sharing patterns in the system. Interestingly, with 32 entries, SPACE has a significant number of false positives varying between 1-2 false positives per directory reference for most applications. SPACE-256 corresponds to 8× more entries than SPACE-32, and translates to a 62% increase in area compared to SPACE-32. This indicates that SPACE also has an effective sweet spot : it requires a certain number of directory entries to effectively eliminate false positives and reducing the entries any further sees diminishing returns (less area reduction and noticeable increase in false positives).

An important operation in SPACE is merging two patterns when the directory is out of free entries (see Section 3.3). We use a hash function to index the incoming pattern into the directory and identify a set of entries (those with the same hash index) to compare with. This indexing function hence directly influences the pollution of entries in a given set, and this pollution manifests itself as false-positive bits in the sharer pattern. Here, we compare the encoding we described in Section 3.3 with a more simple direct hash function. In the *Direct index* (DIRECT) we use the lower-order processor bits in the sharing pattern as the index. We compare it against the baseline SPACE approach, which uses an encoding technique to cluster sharing bits corresponding to all processors when generating the encoding function. To illustrate DIRECT and SPACE; consider the sharing patterns **[0000 0010 1100 0011]** and **[1111 1010 1010 0011]** that need to be mapped to a directory with 16 sets (4-bit index). DIRECT maps both sharing patterns to the same set **[0011]** while SPACE maps **[0000 0010 1100 0011]** ⟶ **[0111]** and **[1111 0010 1100 0011]** ⟶ **[1111]**. Clearly, while the level of pollution in each case is a function of the access pattern distribution, the penalty for messages caused by false positives when the cores are in the same cluster as in SPACE may be smaller.

Figure 8 shows the effect of this pollution: with the same number of entries (256), DIRECT-256 introduces significantly more false sharers and performs similar to SPACE-64. Since nearest neighbor sharing can improve application performance, it also tends to be more common. If adjacency in the bit pattern represents distance, merging patterns/cores that are likely to share data anyway results in fewer false positives. DIRECT-256 essentially wastes ≃ 192 entries. All the SPACE designs make effective use of

the space by distributing the entries and effectively merging similar patterns in a set in the absence of free entries.

The consequences of false positives in the directory are a function of the number of accesses to the entry. Figure 9 shows that for all applications evaluated, SPACE accurately indicates sharers for over 70% of directory references with just 128 entries at 16 processors. Some applications like MP3D, FFT, and Water experience 100% accuracy since they have a very small number of sharing patterns. Apache, Barnes, and SPECjbb have the lowest accuracy because of their relative large number of irregular sharing patterns.

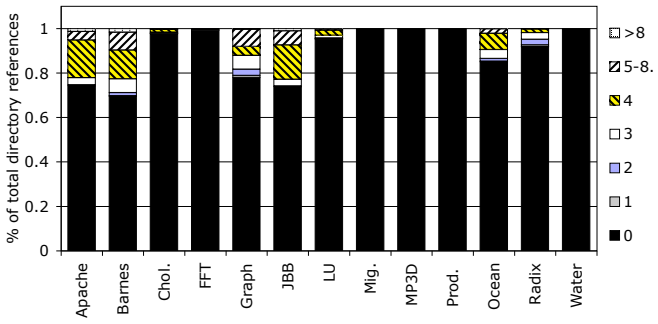


Figure 9: **False sharer distribution in directory references at 16 processors. SPACE directory has 128 entries.**

5.3 Interconnect Traffic

Here, we demonstrate that the false sharers in SPACE do not result in a significant increase in on-chip network traffic. The bandwidth overheads compared to FULL directly correlate with the small false positive bits per reference discussed in the previous section. Bandwidth utilization would mainly be influenced by the invalidation requests that are forwarded to the false sharers caused by processor write requests and L2 evictions. In Table 1b, we list the number of invalidation messages measured in each application.

Figure 10(a) shows the bandwidth overhead for the different coherence systems. The bandwidth utilized is measured as the total number of flits sent over the physical network. Results show that BCAST (which does not use any sharer information) incurs more than 10% overhead for most applications and in the worst case not including microbenchmarks (MP3D) sends up to 2.7× the number of flits required by FULL. SPACE with 128 entries will increase the overhead to an average of 0.8% over FULL. Interestingly, SPACE can exploit even fewer entries (32-64) for many applications (OCEAN, Radix, Water). For Apache, SPECjbb, and Barnes, which have a large number of sharing patterns (see Table 1b), SPACE-128 limits the bandwidth overhead to 1% and 6% compared to FULL. SPECjbb, which experienced many traffic hot spots in the mesh network, is a challenging workload for SPACE and it needs 256-512 entries to match the performance of FULL. SPACE demonstrated significant benefits over BCAST-M, which demonstrates that tracking sharer information for shared cache lines has benefits over merely identifying private data. Note that while SPACE-32 is comparable in area overheads to BCAST-M[2], it makes use of the extra indices to track information for more patterns than just private accesses.

[2]Both require $log_2P + 1$ pointers per cache-line. BCAST-M requires $log_2P + 1$ because it needs log_2P bits to track the processor that modified the cache line and 1 bit to indicate if the line defaults to broadcast.

5.4 Performance Comparison

In this section, we study the execution time of applications in SPACE. We show that the latency overheads on coherence messages due to false sharers are minimal when SPACE has a moderate number of entries (128 — 256) and its performance is comparable to FULL (see Figure 10(b)). Here, we also study the effect of a decoupled SPACE directory on performance. Compared to FULL in which the sharer information is accessed in parallel with the data access, in SPACE only the directory pointer can be accessed in parallel with the data. Further overhead is required to index into the SPACE directory (Section 5.5 talks about access time for various SPACE directory sizes). We model this as a 2 cycle access penalty (averaged over lookups and updates).

Comparing FULL to BCAST and BCAST-M, FULL has the best performance since each cache line has the exact sharing information. BCAST performs (up to 12.7%) worse than FULL, but BCAST-M's optimization for private cache lines (the dominant access pattern, see Figure 10(b)) improves performance significantly. The traffic overheads in our regular mesh network has little impact on performance for the microbenchmarks (Prod. and Migr.) since the overall network utilization is low — BCAST and BCAST-M perform similar to FULL. With SPACE, the dominant overhead for such applications is the directory access penalty that SPACE adds to all coherence requests that reference the directory. SPACE performs marginally worse (1% and 0.5%) than BCAST and BCAST-M in such applications.

In applications with higher network utilization, which include Apache and SPECjbb, the network traffic saved by SPACE plays a dominant role. SPACE-128 performs better than both BCAST and BCAST-M. SPACE-256 and SPACE-512 have performance comparable to FULL. Overall, the performance of SPACE with 128 entries only suffers minimal performance penalty compared to FULL — slowdown less than 2% for all the applications except SPECjbb (7.5% for SPECjbb). SPACE-128 is ≃44% the area of FULL.

5.5 Area and Energy Metrics

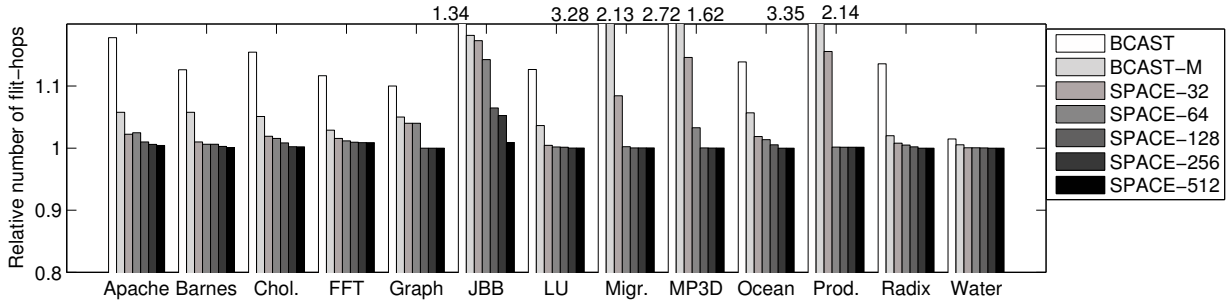
Table 2: **CACTI estimates for various SPACE directory sizes (The read energy includes access of the directory pointer and the directory table entry.)**

Configuration	Access Time(ns)	Read Energy(fJ)
SPACE-32	0.11	654
SPACE-64	0.13	706
SPACE-128	0.14	753
SPACE-256	0.17	838
SPACE-512	0.18	1482
Tagless-lookup [19]	0.34	6381
Shadow tags	0.48	43016

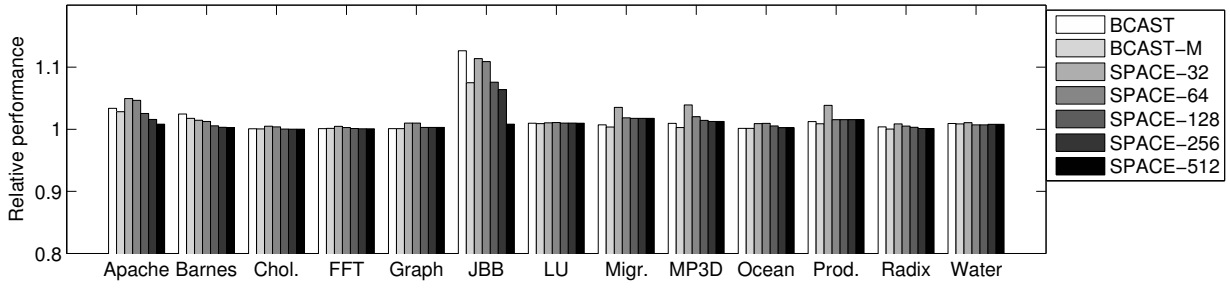
Table 3: **Area overhead for various SPACE directory sizes**

SPACE dir. size	32	64	128	256	512
Area overhead (relative to FULL)	31.35%	37.70%	44.14%	50.78%	57.81%

We use CACTI 6.0 [14] to model energy, delay, and area at a 45nm process technology. Table 2 shows the estimated parameters of the pattern directory table at each tile. With the small size of the pattern directory table in all the configurations, table lookup can be accomplished in one CPU cycle and the area overhead is less than $0.01mm^2$. The dynamic energy numbers include the per-cache-line directory pointer. We also compare the SPACE parameters against the tagless-lookup and shadow tags designs. Compared to shadow



(a) Interconnect traffic(measured in flits)



(b) Performance

Figure 10: **Interconnect traffic and performance of various directory designs normalized to FULL.**

tags, which construct the sharing pattern dynamically on each access with associative lookups, SPACE achieves significant dynamic energy savings. The SPACE directory also consumes less dynamic energy compared to the tagless design.

The area overhead incurred by SPACE relative to FULL is shown in Table 3. The overhead includes both the directory table and the pointer per cache line, which is the dominant overhead in SPACE.

Dynamic energy consumption (in fJ) per access with 64KB private L1 and 16MB shared L2

L1	L2 (BCAST)	L2 (SPACE-256)	L2 (FULL)
19252	134921	135455	135989
Router/Interconnect			
BufRd	BufWr	Xbar	Arbiter
762	1690	24177	402

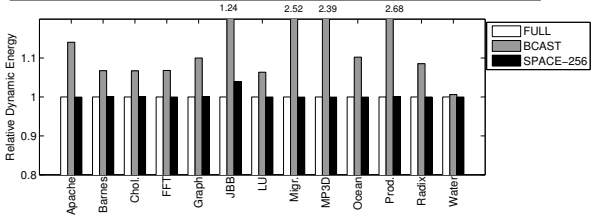


Figure 11: **Dynamic energy of the memory subsystem normalized to FULL. Table shows the per-access energy used for the various components used to drive the model.**

The total dynamic energy consumption of the memory subsystem is calculated based on L1 and L2 access statistics, flit traffic in the on-chip network, and the SPACE directory accesses. As Figure 11 shows, SPACE consumes energy similar to FULL while BCAST consumes 1.05 — $\simeq$2.7$\times$ more energy than either. BCAST consumes energy in the network and for probing L1 caches on many coherence accesses.

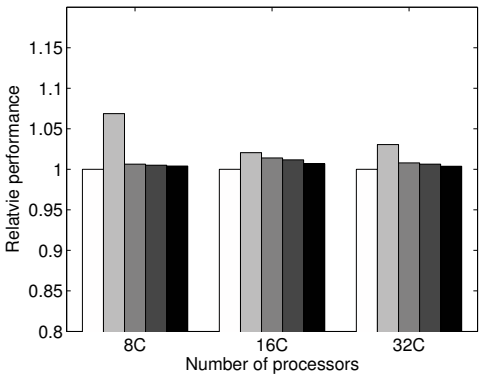
5.6 Scalability

The efficiency of SPACE implicitly depends on the number of sharing patterns frequently used and the number of cache lines mapping to a sharing pattern. Both of these properties vary with the machine parameters. To study the general applicability of SPACE, we study it under three different multicores: 8C (8 core CMP, 8 MB L2), 16C (16 core CMP, 16 MB L2) and 32C (32 core CMP, 32 MB L2). We also vary the size of the SPACE directory between $4*P$ — $32*P$. We evaluate its scalability using application execution time and interconnect traffic as the metrics.

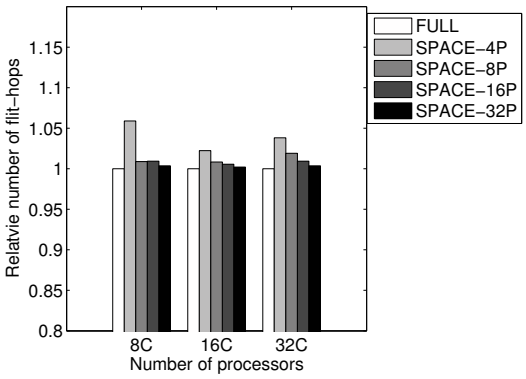
Figure 12 shows that SPACE with a limited number of entries (SPACE-64 for 8C, SPACE-128 for 16C, and SPACE-256 for 32C) consistently performs similar to FULL and has performance penalties within 2% for all multicore configurations we tested. The network traffic demonstrates a similar trend. SPACE's performance tends towards FULL when it is able to accurately maintain the patterns present in an application when it runs on a particular configuration. Interestingly, the minimum number of entries required by an effective SPACE system to deliver performance comparable to FULL appears to be $8*P$ (for the applications we tested). This suggests that the number of frequently referenced patterns tend to be linearly correlated with P. With this trend, SPACE needs $K*P$ entries to efficiently support a system with P cores, where K is a small constant (our experiments seem to suggest K=8). With $K*P$ entries in the table, each cache line needs a pointer of size $log_2(P)+log_2(K)$ (If K = 8, then $log_2(k)$=3). Now consider the case with the total number of shared cache lines as M. When FULL is implemented in such a system, the total overhead of the directory is $O(M*P)$. With SPACE, the overhead of the pointers is $O(M(log_2P+log_2K)) \longrightarrow O(Mlog_2P)$.

Comparison with Tagless-Lookup [19] and Shadow tags

Figure 13 shows a comparison of the area overheads. Full map directory increases linearly with the number of cores. SPACE's overheads are dominated by the pointers associated with each L2



(a) Execution Time



(b) Interconnect Traffic

Figure 12: **Execution time and interconnect traffic for SPACE normalized to FULL. Each bar represents a specific SPACE directory size. SPACE-KP represents SPACE with K * P entries, where P is the number of processors. X axis represents 3 different multicore systems: 8C (8 core CMP, 8 MB L2), 16C (16 core CMP, 16 MB L2) and 32C (32 core CMP, 32 MB L2). Y axis represents the geometric mean of relative execution time and interconnect traffic of all applications.**

cache line. SPACE does impose a constant factor overhead over tagless-lookup (and shadow tags); tagless-lookup improves over shadow tags by $N_{L1-ways}$. Both tagless-lookup and shadow tags seek to represent the information only for lines cached in the L1; this provides notable area benefits over SPACE. The main advantage of SPACE is that it streamlines the access to the directory entry when a cache block is accessed, thereby requiring less energy to access the sharing pattern. The pointers to the sharing pattern are accessed along with the L2 tag lookup and the directory table can be organized to require fewer comparators. Tagless-lookup (and shadow tags) require more energy-intensive table lookups on each cache access due to the highly associative comparators necessary to construct the sharing pattern dynamically. We believe SPACE's design is better suited to support large multicores. Overall, SPACE is 8× and 52× more energy efficient than Tagless-lookup and shadow tags respectively (see Section 5.5 for a detailed quantitative comparison).

6. CONCLUSION

Future generation multicores with a great many processors bring with it the challenge of effectively maintaining cache coherence. Directory-based protocols achieve performance scalability by accurately maintaining information about sharers. Unfortunately, the conventional directory structure does not scale in size and consumes a significant fraction of precious on-chip area.

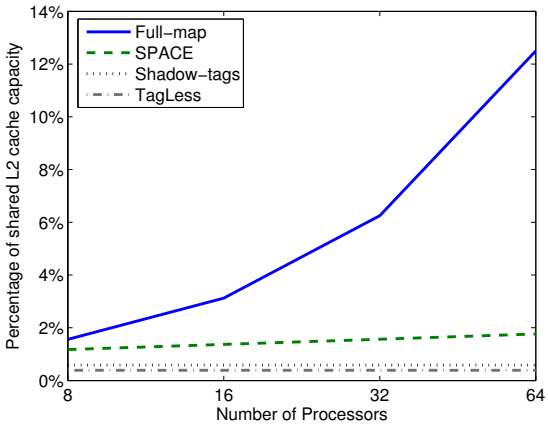


Figure 13: **Storage requirements of SPACE, Shadow tags, Tagless and Full map directory for a 64KB private L1 cache and a shared L2 cache with 1MB / core. Note that the X axis is using a log scale. Y axis : Percentage of the shared L2 cache capacity.**

In this paper, we first investigate the sharing patterns in applications. The sharing pattern of a location refers to the set of processors accessing it. We demonstrate that many applications possess sharing pattern locality, i.e., there are a few unique patterns that are referenced frequently and many cache lines have a common sharing pattern. A conventional directory hence essentially stores duplicate copies of the same sharing patterns. We exploit this observation and propose the SPACE approach, which leverages the sharing pattern commonality by completely decoupling the sharing patterns from the shared cache and storing only a pointer to the specific pattern with each cache line. This permits all cache lines that have the same sharing pattern to point to a common pattern entry. We find that with a small number of entries in the pattern table, we can effectively support a large fraction of the directory references from cache lines: 128 — 256 entries for a 16 processor multicore and 256 — 512 entries for a 32 processor multicore. We show that SPACE can perform within 2% of a conventional full map directory in spite of occupying only 44% at 16 processors and 25% at 32 processors of the full map's area. SPACE's directory deals with space constraints by using a dynamic collation technique to merge patterns that are similar to each other. We demonstrate that this technique helps reduce network traffic due to false sharer bits with area overheads comparable to coarse-vector directories.

A key challenge with the SPACE approach is the pointers required per cache line in the shared cache. This leads to extra overhead compared to approaches such as tagless lookup [19], which store information only for lines in private caches. We believe that SPACE's key insight of exploiting sharing pattern locality and introducing an extra level of indirection to store only the unique entries, is generally applicable. We plan to investigate hybrid approaches between SPACE and tagless lookup [19]: combining techniques from SPACE to help eliminate redundant copies of sharing patterns with tagless lookup to eliminate the per LLC cache line overhead.

7. 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. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD opteron shared memory mp systems. In *Proceedings of the 14th HotChips Symposium*, 2002.

[4] 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.

[5] G. Buehrer, S. Parthasarathy, and Y. Chen. Adaptive parallel graph mining for CMP architectures. In *Proceedings of the Sixth International Conference on Data Mining*, pages 97–106, 2006.

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

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Apr. 1991.

[8] 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.

[9] 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.

[10] Intel Corporation. Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process. http://download.intel.com/design/mobile/datashts/30922106.pdf, Jan 2007.

[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. A tagless coherence directory. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–434, 2009.

[20] C. Zilles. Brief announcement: Transactional memory and the birthday paradox. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.