A Practical Method for Whole Map Directory-Based Cache Coherence Protocol Implementation

Dr. NAGARJUNA *, Mr.SAKTI CHARAN PANDA
Dept. OF Computer Science and Engineering, NIT , BBSR
nagarjuna@thenalanda.com *, sakticharan@thenalanda.com

Abstract– Directories have been utilised in shared memory multiprocessors with private caches to maintain cache coherency. The standard full map directory is made to be effective and straightforward and tracks the precise caching state for each shared memory block. Sadly, it is not appropriate for large-scale multiprocessors due to the inherent directory size inflation. The associative full map directory (ADirpNB), which we suggest in this work, lowers the directory storage requirement. The proposed ADirpNB uses a single directory entry to keep track of the sharing information for a collection of memory blocks that are only cached. As "a full map directory with lower directory memory cost," ADirpNB can be constructed by utilising dynamic cache pointer allocation, reclamation, and replacement hints. Our research shows that, ADirpNB decreases memory overhead of a conventional complete map directory by up to 70-80 percent on a typical architectural paradigm. We demonstrate that the proposed approach may be implemented with the proper protocol modification and hardware addition in addition to having a low memory overhead. Studies using simulations suggest that ADirpNB can perform as well as DirpNB. Due to the removal of directory overflows, ADirpNB exhibits more consistent and reliable performance results on applications across a range of memory sharing and access patterns than limited directory schemes. For moderately large-scale and fine-grained shared memory multiprocessors, we think ADirpNB can be used as a design substitute for full map directories.
Index Terms–Cache coherence, directory protocols, shared memory multiprocessors, computer architecture.

1 Introduction

Shared memory multiprocessors are becoming increasingly popular and attractive platforms for running a variety of applications, ranging from traditional parallel engineering and numeric applications to recent commercial database and Web workloads [4]. Most shared memory multiprocessors use private or local caches to alleviate the impact of interconnection network latency on the memory accesses [30], [49], [29], [35], [28]. Introducing private caches greatly improves system performance, however, cache coherency must be maintained if memory blocks are allowed to reside in different processors simultaneously [14]. Several cache coherence schemes have been proposed in the literature to solve this problem [46], [34], [51], [41]. Snoopy protocols [15], [26], [40], [3] maintain data consistency by monitoring memory access traffic and taking appropriate actions if a memory access violates consistency in data cache. A snoopy protocol is usually implemented on shared bus multiprocessors mounted with a limited number of processors because its performance largely depends on the broadcasting ability of the system interconnection.

Directory protocols, based on point-to-point communication, provide an attractive design alternative to maintain cache coherency in scalable, high performance multiprocessor systems [45]. In this case, a directory entry is maintained for each memory block to keep track of the processors which have cached copies and to decide which action should be taken upon requests to that memory block. Most medium and large scale shared memory multiprocessors of the current generation, such as SGI Origin 2000 [29], Sequent STiNG [35], Stanford FLASH [28], and MIT Alewife [2], employ directory protocols to ensure cache coherence of shared data. The full map directory [6], which employs a presence bit vector to track the identities of processors caching a given block, is designed to be efficient and is the simplest of all directory-based cache coherence protocols [17]. Unfortunately, the storage overhead necessary to maintain a full map directory grows rapidly with the number of processors, making it unscalable.

Many research efforts compromise communication message efficiency for directory memory storage by keeping limited or imprecise sharing information [1], [39], [33], [31] or employ dynamic and complex directory structures (doubly linked list, k-ary tree) with customized protocol operations [23], [8], [38], [48], [51], [9], [20] to resolve this directory scaling problem. Compared with a traditional full map directory, these solutions either inflate coherence message traffic or introduce more latency and complexity in the protocol controller. For instance, in a limited directory [1], when the number of processors sharing a memory block exceeds that of the specified number of pointers, a directory overflow will occur.

For a p processor full map system, the amount of directory memory is $m \cdot p^2$ bits, where m is the total number of blocks in shared memory modules. The memory

• The authors are with the Laboratory for Computer Architecture, Department of Plectrical and Computer Pngineering, University of Texas at Austin, Austin, TK 78712. P-mail: {tliS, ljohn}@ece.utexas.edu.

Manuscript received 10 Mar. 2000; revised 10 Dec. 2000; accepted 11 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IPPPCS Log Number 1116S9.

requirement for a limited directory entry with fixed i pointers is $i \cdot \log_2 p$, where $\log_2 p$ are the minimal bits to uniquely present each processor's identity. Additionally, each pointer is attached with a valid bit to indicate if it contains a valid processor number, making a total of $p \cdot m \cdot (i + i \cdot \log_2 p)$ bits dedicated to directory structure.

The linked list directory [23], [48] scales gracefully to larger numbers of processors with minimal memory overhead. The directory bits dedicated to storing coherence information in the doubly linked list scheme is $p \cdot (m + c + 2 \cdot (m + c) \cdot \log_2 p)$ since each memory and cache pointer requires $\log_2 p$ bits to point a processor, plus an extra bit to point back to memory (c is the number of cache blocks in a cache) [34]. Unfortunately, the inherently complex, sequential, and distributed structure of the linked list directory can cause several disadvantages [17].

A bit-vector directory can be classified as Dir_iX using a nomenclature introduced in [1], where i is the number of pointers in one directory entry and K is either B or NB, depending on whether a broadcast is issued when a cache pointer overflows. Full map directory and limited directory, therefore, can be symbolized with Dir_pNB and $Dir_iNB(i < p)$.

This paper proposes an associative full map directory ($ADir_pNB$) which contributes to reducing the overwhelming memory overhead while maintaining the optimal performance of a full map directory. By examining multiple memory block caching artifacts, we find that directory memory overhead can benefit from associating a shared directory entry with a set of exclusively cached memory blocks, i.e., blocks that are potentially mapped into the same cache line or the same cache set, depending on the cache organization.

The proposed directory has the following features that distinguish this work with previous studies: 1) It uses one directory entry to track multiple memory blocks caching status simultaneously by creating and maintaining multiple centralized linked lists; 2) by exploiting caching exclusiveness of multiple memory blocks, $ADir_pNB$ captures compact yet exact sharing information for each memory block, thus makes an effective use of directory memory; 3) by implementing dynamic cache pointer allocation, reclamation, and replacement hints, $ADir_pNB$ can achieve competitive performance with a traditional full map directory.

This paper examines the quantitative efficiency of the proposed cache coherence protocol on a cache coherent nonuniform memory access (CC-NUMA) machine from the perspective of both memory overhead, coherence traffic, and execution performance. Our performance evaluation is based on SimOS [18], a complete system simulation platform running multiprogramming applications SPLASH-2. The analysis indicates that, on a typical architecture, $ADir_pNB$ reduces the memory overhead of a traditional full map directory by up to 70-80 percent. For some memory and cache configurations, $ADir_pNB$ is even more memory efficient than inexpensive limited directories, such as Dir_4NB and Dir_8NB. By eliminating the cache pointer overflows due to limited directory entries (or pointers) and sequential invalidations by traversing the distributed linked

TABLE 1
Notations Used to Explain the Proposed Scheme

Symbol	Description
p	Number of processors
C_x	Caches in multiprocessors, $x = 1,\ldots, p$
M	A distributed shared memory module
m	Number of memory blocks in M
n	Number of cache lines in C_x
MB_i	A shared memory block, $i = 0,\ldots, m-1$
CL_j	A cache line, $j = 0,\ldots, n-1$
r	Capacity ratio of a shared memory module and a cache, i.e., $r = m/n$
Φ_j	The set of memory blocks which are exclusively mapped into CL_j in M

list through underlying interconnection network, the proposed scheme potentially has lower coherence messagetraffic and protocol controller latency.

Simulation studies indicate, on a 16-processors CC-NUMA system, ADir$_p$NB results in a competitive performance with Dir$_p$NB. Compared with limited directory schemes, ADir$_p$NB shows more stable and robust performance results on applications across a spectrum of memory sharing and access patterns due to the elimination of directory overflows. We believe that ADir$_p$NB can be employed as a design alternative of full map directory for moderately large-scale and fine-grain shared memory multiprocessors.

The remainder of this paper is organized as follows: The detailed rationale of our proposed associative full map directory is described in Section 2. Section 3 examines the memory reduction efficiency of the proposed technique. Section 4 describes our experimental methodology and presents results from the simulation studies. Related workis discussed in Section 5. Finally, concluding remarks appear in Section 6.

2 The Proposed Associative Full MapDirectory

This section introduces a memory-efficient arrangement of directory bits, called the associative full map directory. Our new approach is based on the observation that a traditional full map directory allocates a static, redundant, yet sparsely utilized bit vector for each memory block, ignoring the potential exclusiveness of multiple memory blocks due tothe implication of cache mapping artifacts. By exploiting caching exclusiveness, directory memory requirement can benefit from associating a dynamic, shared directory entry with multiple memory blocks. Our proposed associative full map directory derives its name from the perspective that each directory entry has the ability to keep track of and maintain precise sharing information for multiple memory blocks simultaneously. To clarify the proposed scheme efficiently, we define a set of notations used in our illustration (see Table 1).

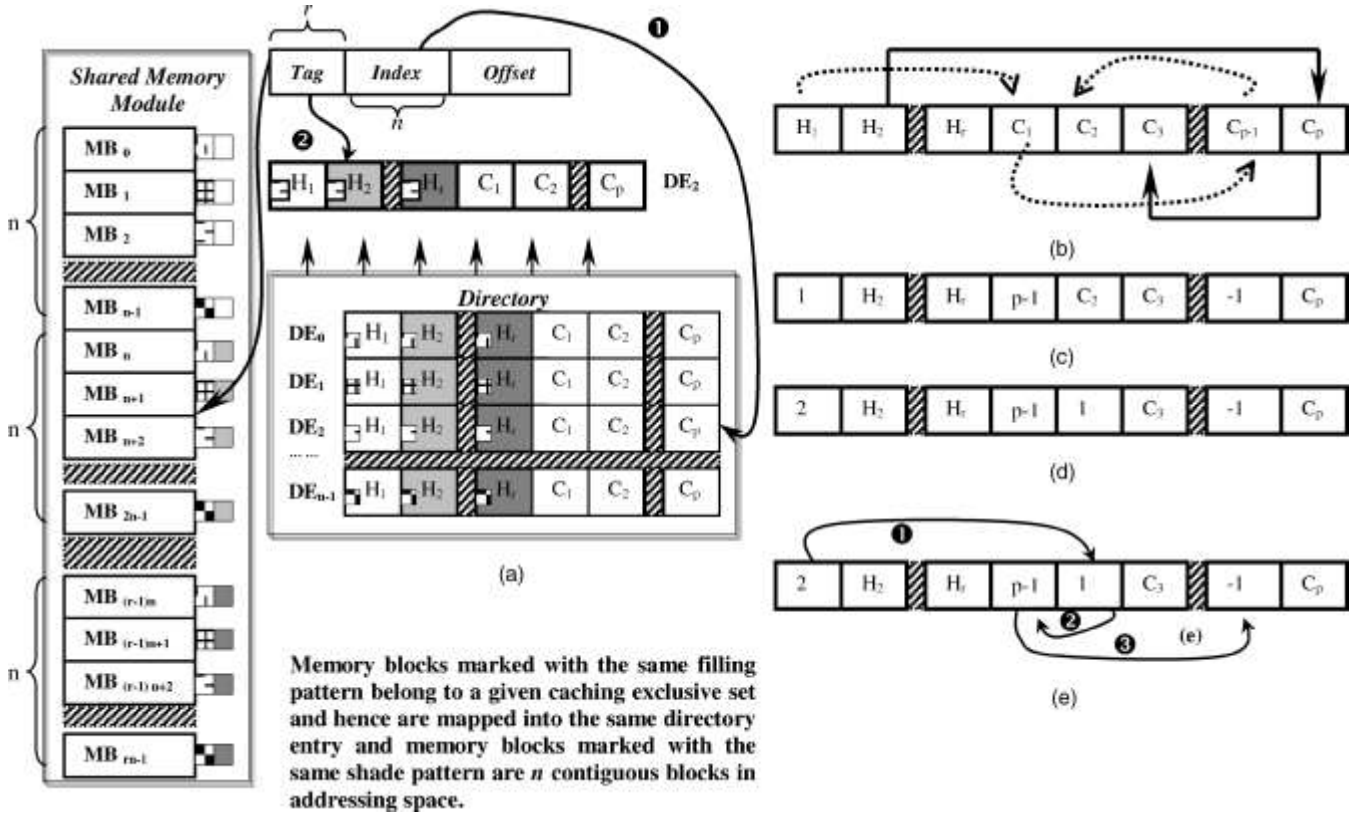

Fig. 1. Structure of associative full map directory. (a) Mapping scheme between caching exclusive shared memory blocks and directory entries. (b) Multiple cache linked lists in a directory entry. (c) Before cache 2 is inserted in the linked list. (d) After cache 2 is inserted in the linked list. (e) Invalidation message transverse linked list one-by-one.

Associative Full Map Directory for Direct Mapped Cache

For simplicity, we first introduce the proposed scheme for direct mapped caches and then extend to more complicated set-associative caches. For a direct mapped cache, the mapping artifact, which translates a memory block to a cache line, is $f : MB_i \rightarrow CL_j$, where $j = i \bmod n$. The capacity ratio r determines how many memory blocks can exclusively be mapped into a specific cache line. If, at any point t in time, a memory block MB_y is cached by C_x, we note it as $MB_y \in C_x(t)$. Otherwise, we note it as $MB_y \notin C_x(t)$. Assuming 16 MB of memory per module, a 16 KB direct mapped cache, and a cache line size of 16 bytes, we have $n = 2^{10}$, $m = 2^{20}$, and $r = 1,024$. Note that $\emptyset_0 = MB_0, MB_{1\times1,024}, MB_{2\times2,024}, \ldots, MB_{1,023\times1,024}$ is the complete set of memory blocks that can potentially be mapped into cache line CL_0.

We mark $\emptyset_0$ as the set of caching exclusive memory blocks with respect to CL_0. More formally, if $MB_y \in C_x(t)$ and $MB_y \xrightarrow{f} CL_i$, i.e., $MB_y \in \emptyset_i$, then $\forall MB_q \in \{\emptyset_i - MB_y\}$, we have $MB_q \notin C_x(t)$, where C_x is the cache in which MB_y is present. Given a case that memory block MB_y is cached by n processors at point t in time, e.g., $MB_y \in C_1(t), \ldots, MB_y \in C_n(t)$, $MB_y \notin C_{n+1}(t), \ldots, MB_y \notin C_p(t)$, and $MB_y \in \emptyset_i$, we have $\forall MB_q \in \{\emptyset_i - MB_y\}$, $MB_q \notin C_1(t), \ldots, MB_q \notin C_n(t)$. That is, memory blocks in set $\{\emptyset_i - MB_y\}$ are at most cached by $C_{n+1}(t), \ldots, C_p(t)$ (p-n) caches.

This rule implies that, in a multiprocessor system with p direct mapped caches, the maximum number of cached copies of memory blocks which belong to a given caching exclusive set $\emptyset_i$ is p, the number of caches (processors) in the system. Moreover, $\forall MB_p, MB_q \in \emptyset_i$ $(p \neq q)$, at any point t in time, if $MB_p \in C_x(t)$ and $MB_q \in C_y(t)$, we have $x \neq y$. These features guarantee that memory blocks falling into the same caching exclusive set actually do not compete for the same cache pointer, even if there are only the fixed p cache pointers for the entire set of memory blocks. Hence, it is economical and safe to allocate total p cache pointers for the r memory blocks in $\emptyset_i$.

Fig. 1 illustrates the structure of associative full map directory, including the mapping rationale from caching exclusive sets to directory entries and coherence operations manipulated on a directory entry. As shown in Fig. 1, each directory entry is comprised of head pointer fields $H_1, H_2, \ldots, H_r$ and cache pointer fields $C_1, C_2, \ldots, C_p$. A head pointer is allocated for each memory block within a shared memory module and is used to indicate the first item in the sharing list of that block. A cache pointer, which serves as a forwarding linker, is used to store the next cache identifier in that list. Either a head or a cache pointer consumes $\log_2 p + 1$ bits since each of them requires $\log_2 p$ bits to point to a processor, plus an extra valid bit to indicate if it contains a valid processor pointer. To clarify our illustration, we mark each memory block with a specific filling pattern and a specific shade pattern in Fig. 1. Memory blocks marked with the same filling pattern fall

into the same caching exclusive set and, hence, are mapped into the same directory entry. Memory blocks marked with the same shade pattern are contiguous in addressing space and are uniformly distributed among different directory entries.

Following this rule, the m memory blocks in a shared memory module can uniquely be mapped into n directory entries, each of which has r head pointers and p cache pointers. Recall m equals n $\times$ r in our definition. In each directory entry, the p cache pointers are dynamically allocated and reclaimed for the clustered r head pointers, which represent a complete caching exclusive memory block set. Indexing of the head pointer for a given memory block, although not as explicit as that in the case of traditional full map directory, is straightforward and needs fairly simple hardware. As shown in Fig. 1a, the n bit index field in physical memory address is used to find the directory entry allocated for the corresponding caching exclusive memory block set (step 1 in Fig. 1a). The r bit tag field is then applied to select a specific head pointer allocated for that memory block (step 2 in Fig. 1a). Fig. 1a shows an example of how to find the head pointer for memory block MB_{n+2}.

Sharing a memory block with different processors causes the directory controller to construct and maintain a directory memory-based linked list with a head pointer dedicated to that memory block. Since a shared directory entry is used to serve a set of memory blocks, the sharing behavior of these memory blocks may produce several linked lists simultaneously. Fig. 1b shows a case in which a memory block (indicated by H_1) has copies in caches 1, p-1, 2, and another memory block (indicated by H_2) has copies in caches p and S. Due to the caching exclusiveness of these memory blocks, the p cache pointers in this directory entry can be shared effectively and safely by the r memory blocks.

As mentioned before, the proposed associative full map directory constructs and maintains the precise sharing information for multiple memory blocks in a linked list style. Fig. 2 highlights the directory memory manipulations that should be taken upon a memory access. Fig. 2a shows a procedure for inserting a new item into linked list when a shared read miss is invoked. Fig. 2b shows a procedure called invalidate(cache) to perform invalidation upon a write, where cache refers to the processor requesting this write operation. The traversal through the list can be performed more effectively within directory memory instead of issuing remote cache accesses through the slower interconnection network.

Fig. 1c, Fig. 1d, and Fig. 1e illustrate the insertion and invalidation operations in a graphic style. In Fig. 1c, the given memory block (pointed by H_1) is initially cached by processors 1 and p-1. Therefore, the initial values of pointers H_1, C_1, C_{p-1} are 1, p-1, -1. Fig. 1d shows that, when processor 2 accesses directory entry due to a shared read miss, it is inserted as the new head of that linked list. Note that: 1) H_1 points to the most recent requestor, 2) C_2 now serves as forward pointer of H_1, 3) the remainder of the linked list is unchanged. When a write to that block occurs, invalidation messages are sent to those caches present in the linked list by walking through all the items until it meets an

```
Procedure insert (cache)
//Afullmapdir: directory entry associated with a
//given memory block;
//Afullmapdir.head: the head pointer of linked list;
//Afullmapdir.sharelist[i]: one item of the linked list,
//it contains the forward pointer which indicates
//the next sharing item. A special symbol '-1' is used
//to mark the terminal of the linked list;
{ if (Afullmapdir.head=-1) {
  //linked list is empty, create the linked list and
  //insert the current requesting cache as the head pointer;
  Afullmapdir.head=cache;
  Afullmapdir.sharelist[cache]=-1;
  } else {
  //insert the current requesting cache as the new
  //head pointer of the linked list;
  Afullmapdir.sharelist[cache] =Afullmapdir.head;
  Afullmapdir.head=cache; }
}
```

(a)

```
Procedure invalidate (cache)
{ ptr=Afullmapdir.head;
  while (ptr!=-1)
  //invalidate cache copies by walking through the linked list
  { if (ptr!=cache)
  send invalidation message to cache pointer ptr;
  recptr=ptr;
  //point ptr to the next sharing item;
  ptr=Afullmapdir.sharelist[ptr];
  //reclaim the current item in linked list;
  recptr=-1;}
  //linked list now has only one item that owns the
  //exclusive copy of a given memory block;
  Afullmapdir.head=cache;
  }
```

(b)

Fig. 2. Procedures used in the proposed scheme to implement (a) insert and (b) invalidate operation.

invalid pointer, which indicates the end of that list (see Fig. 1e). After invalidation, the linked list has only one item that owns the exclusive copy of that memory block. The invalidated cache pointers are reclaimed to the free list for future use.

In associative full map directory, all cache pointers are dynamically allocated and reclaimed. Exploiting the caching exclusiveness of memory blocks ensures the collision-free characteristic of sharing the fixed p cache pointers among r memory blocks. However, this ability depends on keeping accurate and up-to-date sharing information and does come at a cost. Since there are only p cache pointers, it is possible to run out of pointers if the directory controller does not reclaim a pointer after a memory block has been replaced from the cache.

The proposed scheme, like other dynamic pointer allocation protocols [17], makes use of replacement hints to prevent this pointer exhaustion. Replacement hints complicate design by requiring that the system handle an additional type of message. However, it reduces the number of invalidation and invalidation acknowledgments by only sending coherence messages to the actual sharing processors. Associative full map directory uses replacement hints to locate the replaced item and remove it from the list. Due to the sequential nature of a linked list, time spent on searching a given item can be $\Theta(n)$, where n is length of the

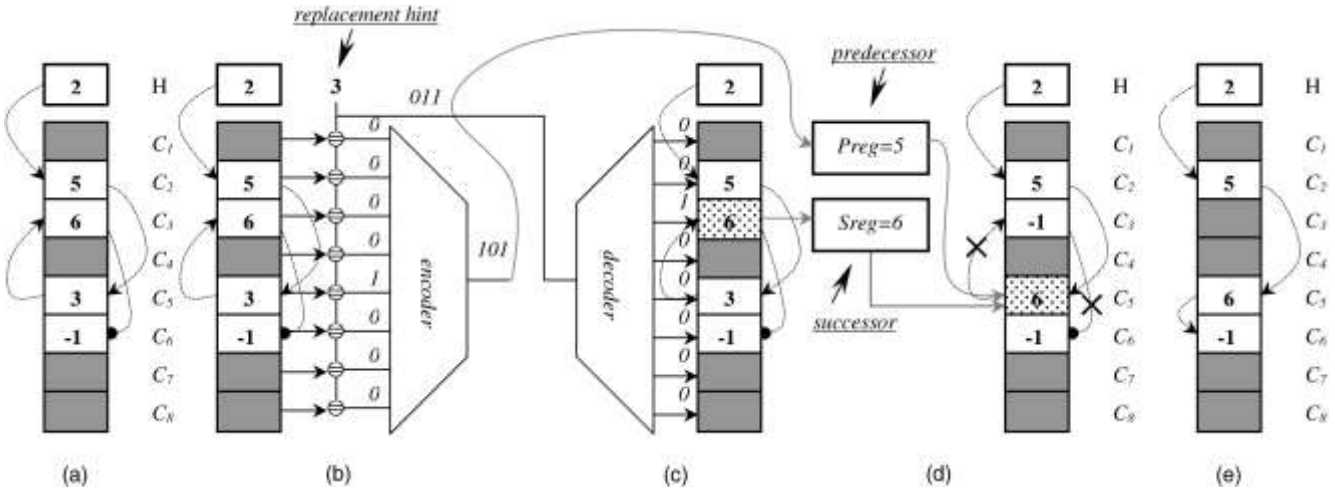

Fig. 3. The proposed efficient replacement algorithm. (a) Initial state of a linked list. (b) Replacement hint is compared with cache pointers in the linked list to find predecessor. (c) Replacement hint is applied to cache pointer array to select the successor of replaced cache. (d) Removing the replaced node from the linked list by updating its predecessor. (d) Final state of linked list after replacement.

inquired linked list. In our proposed scheme, this searching cost can be reduced to Θ(1) since the entire linked list is stored in one directory entry instead of being distributed among different caches. In this paper, we introduce an efficient replacement algorithm for our proposed scheme.

Fig. 3 shows an example of how the efficient replacement algorithm works. Initially, the four cache items (i.e., caches 2, 5, 3, and 6) are scattered throughout the cache pointer array and linked with each other, as shown in Fig. 3a. When a replacement in cache 3 occurs, a replacement hint is sent to the corresponding directory entry. The directory controller uses this information to compare all cache pointers to find the predecessor (cache 5) of the replaced cache item (cache 3) in the linked list, as shown in Fig. 3b. Meanwhile, the replacement hint is applied to index the successor (cache 6) of the replaced cache item (see Fig. 3c). As a result, updating the predecessor with the identity of its successor actually performs removal of the replaced node from the linked list, as shown in Fig. 3d. The unraveled cache pointer should be reclaimed to the free list for future use.[1]

An implication of the above replacement algorithm is that it requires that the replacement hints arrive before the request from the new sharing processor. In a network that allows message reordering, a request could bypass a replacement hint, leaving the directory in a transient state where more pointers are required than are available. In such a case, the directory controller can send NAck and retry signals to the sharing processor until it performs the necessary directory entry reclamation and then allows the new sharing processor to be added to the linked list. Alternatively, the out of order requests could be buffered and serialized through the directory. Our proposed scheme uses NAck/retry solutions for simplicity since messages will frequently arrive in order [44].

Our proposed directory can be symbolized by ADir$_p$NB, where A represents the associative directory because, in this case, a directory entry is associated with and served for multiple memory blocks. Dir$_p$NB is derived from the perspective that this scheme uses dynamic cache pointer allocation, reclamation, and replacement hints to emulate a full map directory with optimal performance.

The directory memory manipulations described above can be either hardwired in a custom coherence controller (HWC) or implemented as a software-based protocol handler executed by a dedicated protocol processor (PP) [36]. The coherence protocol of ADir$_p$NB can be tailored from a traditional full map directory and augmented with the specific directory operations described above. This feature can facilitate protocol verification and thus shorten hardware development time.

Associative Full Map Directory for Set Associative Cache

The proposed scheme can be extended easily to fit set associative caches. In a k-way set associative cache, a memory block is mapped into a given cache set in a modulo fashion, but may be hashed in any one of the k cache lines within a set. Therefore, given p set associative caches in shared memory multiprocessors, at any point t in time, the copies of the memory blocks which can be mapped into the same cache set are no more than kp, where p is the number of caches and k is cache associativity. Similarly, we can associate a shared directory entry with those memory blocks that are mapped into the same cache set. The associative full map directory for the set associative cache has the structure illustrated in Fig. 4.

As shown in Fig. 4, each directory entry is comprised of head pointers (H_1, H_2, … , $H_{\&}$) and a cache pointer matrix $C^{p\times h}$. The symbol & is defined as the number of memory blocks which can be potentially mapped into the same cache set in a memory module. Since those memory blocks are caching exclusive with respect to the same cache set (instead of to the same cache line), the cache pointer array used in the direct-mapped cache is replaced by a cache pointer matrix to handle the situation that several memory blocks may simultaneously be present in one processor's cache.

1. Fig. 3 shows a case where the replaced item is in the middle of a linked list. The replacements of the head pointer and tail pointer, as two special cases, can be handled in a similar way.

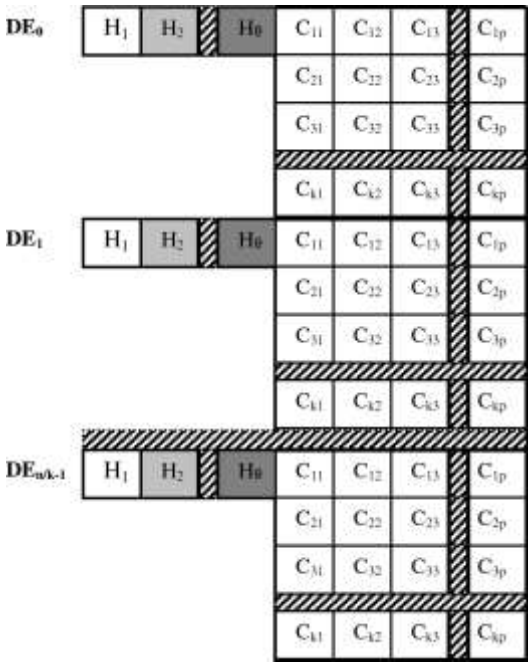

Fig. 4. Directory structure of ADir$_p$NB for set-associative caches.

Fig. 5 shows a snapshot of a directory entry at the moment that a memory block (represented by H_1) is cached by processor 1, 3, and p and another memory block (represented by H_2) is cached by processor 1, 2, and p. Note that associative hardware search for replacement in set-associative caches needs to be scaled linearly with associativity.

3 Efficiency of Directory Memory Reduction

Direct Mapped Cache

For simplicity, we first consider directory memory overhead of ADir$_p$NB with direct-mapped cache configurations and then extend to a set-associative case. To evaluate the protocol memory efficiency of the proposed scheme and its sensitivity to various memory and cache configurations, we introduce a notation $N_x(.)$, referred to as memory overhead for a given directory scheme in a memory module. Thus, we have: $N_x(\text{Dir}_p\text{NB}) = mp$ and $N_x(\text{ADir}_p\text{NB}) = m \cdot [(\log_2 p + 1) \cdot (1 + p/r)]$. $N_x(\text{ADir}_p\text{NB})$ can be expressed as $(\log_2 p + 1)(m + pn)$, which indicates that one part is linear with the memory size m and another part is linear in size with the total amount of cache (pn) in the system. As described in Section 2, r is the capacity ratio of a shared memory module and a cache, i.e., $r = m/n$.

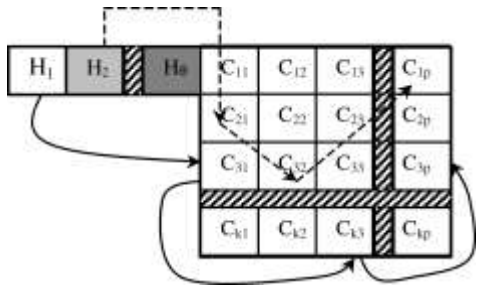

Fig. 5. Cache linked lists in a shared directory entry.

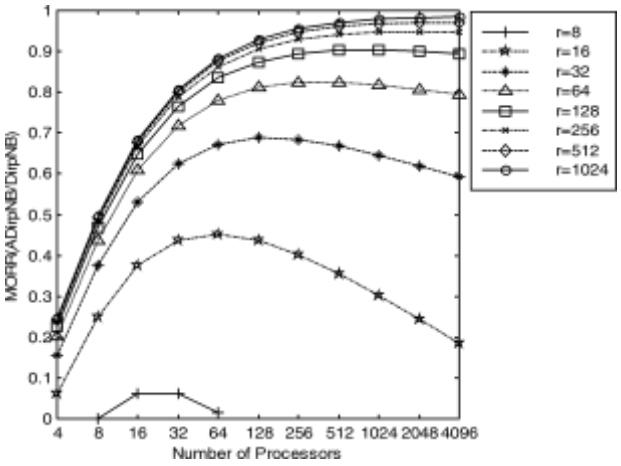

Fig. 6. Impact of memory cache configurations on memory overhead reduction ratio (MORR).

The memory overhead reduction ratio (MORR) of directory scheme DirA to directory scheme DirB can be defined as $\text{MORR}(\text{DirA}/\text{DirB}) = 1 - N_x(A)/N_x(B)$. Thus, we have

$$\text{MORR}(\text{ADir}_p\text{NB}/\text{Dir}_p\text{NB}) = 1 - (\log_2 p + 1) \cdot (m + np)/(mp).$$

Fig. 6 illustrates the impact of different memory cache configurations on MORR. It is seen that, for a given p, an increase in r improves MORR since a directory entry can be shared with more memory blocks as r grows. With a given memory cache mapping configuration, MORR increases as p grows and starts to decrease for high values of p. Note that when $r \geq 32$, the reduction of MORR becomes less sensitive to the growth of system size because the high associativity between multiple memory blocks and a directory entry can efficiently hide memory expansion caused by the increase of number of processors. For example, given $r = 128$, MORR equals 0.84, 0.90, and 0.90 when p is equal to 64, 256, and 4,096. This optimistic result implies that the proposed scheme can be applied to large systems.

Note that, for a given MORR 2, we have $r = \frac{p \cdot (\log_2 p + 1)}{(1-2) \cdot p - (\log_2 p + 1)}$ $(r > 0)$. The value of r can be computed to investigate the minimum required memory module to cache capacity ratio in order to yield any advantage by justifiably employing ADir$_p$NB. From the standpoint of implementation, r is power of 2. Fig. 7 examines the impacts of 2 and p on r, the minimal required memory module to cache capacity ratio given MORR = 2. It is seen that, in general, an increase in 2 naturally increases r for a given p. Fortunately, 2 does not have a significant effect on r provided that $0.1 < 2 < 0.9$ and p goes from 64 to 4,096. If we assume that typical cache sizes are in the range of 64KB words to 256KB words and a typical memory module may contain from 2MB words to 16MB words [34], then the typical values of r will fall into a range of 8 to 256.

Fig. 7 shows that when r falls into this range, 2 can be as high as 0.9. This observation implies that, with optimistic memory module and cache configurations, ADir$_p$NB can

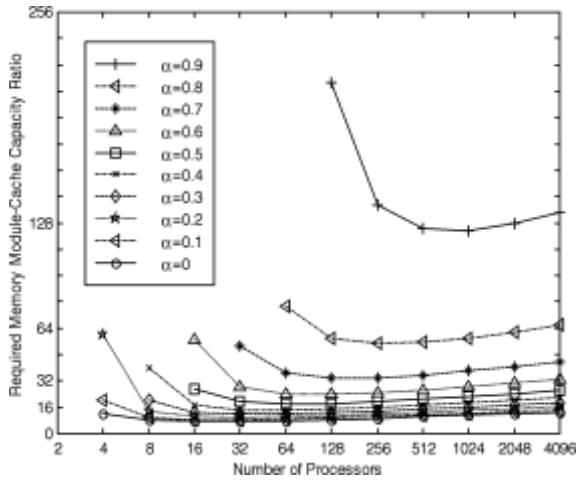

Fig. 7. Impact of p and 2 on r.

help to reduce the directory memory overhead by 90 percent compared with Dir$_p$NB. Fig. 7 shows that, in general, r = 64 is sufficient to reduce 70-80 percent of directory memory overhead.

Fig. 8 provides directory memory overhead comparison of ADir$_p$NB and three inexpensive limited directory schemes, namely Dir$_4$NB, Dir$_8$NB, and Dir$_{16}$NB. To provide a fair comparison, we present only cases where limited directory consumes less directory memory than a full map scheme. ADir$_p$NB is seen to be more economical than even limited directories. The improvement does deteriorate with higher p. For example, compared with Dir$_4$NB, MORR is 0.63, 0.5, and 0.25 when r = 64 and p equals 32, 64, and 128. For a given p, increase of r will improve MORR. For example, given p = 64, MORR increases from 0.25 to 0.73 when r grows from 32 to 1,024. Given p = 128 and r = 64, MORR are 0.25, 0.63, and 0.81 for Dir$_4$NB, Dir$_8$NB, and Dir$_{16}$NB.

Set Associative Cache

The directory memory overhead of ADir$_p$NB with set-associative cache configuration (in a shared memory module) is $\frac{m}{\&}[\&(\log_2 ph + 1) + ph(\log_2 ph + 1)]$ since the head pointers of a directory entry consumes $\&(\log_2 ph + 1)$ bits and $ph(\log_2 ph + 1)$ bits are used for storing the cache pointer matrix. Also, a valid bit is attached with each pointer to indicate whether it is pointing to a valid cache.

The MORR of ADir$_p$NB for set-associative caches can be expressed as $1 - \frac{\&(\log_2 ph+1)+ph(\log_2 ph+1)}{\&p}$. Recall & can be expressed as hr, i.e., & = hr, where k is the set size and r is the capacity ratio of memory module and cache. When h = 1, & = r and the directory structure shown in Fig. 4 equates with the directory structure described in Fig. 1a. For a k-way set associative cache, MORR can be expressed as $1 - \frac{(r+p)\cdot(\log_2 p+1)}{rp} - \frac{(r+p)\log_2 h}{rp}$. Note that the first two terms are identical to MORR in the case of direct-mapped caches. The set associative caches, unfortunately, introduce a factor $\frac{(r+p)\log_2 h}{rp}$, which decreases the memory overhead savings. Typically, k ranges from 2 to 16. Fig. 9 illustrates the impact of set associativity k and p on MORR. As shown in this figure, the increase of k does decrease MORR for a given p and r. Fortunately, given r ≥ 64, the MORR drops less than 0.1 when k goes from 2 to 16.

The results in Fig. 9 indicate that the amount of state required by ADir$_p$NB depends on the amount of associativity in the system and MORR does not scale well on large and highly associated (with small value in r and high value in k) cache configurations. This implies that ADir$_p$NB is not quite suitable for designs in which large fully associative caches (e.g., 4M, 32-way remote access cache) are used to eliminate capacity and conflict misses. In such cases, a COMA-based protocol [47] may provide more design trade-off. Fortunately, even if the cache is only 2 or 4-way set-associative, added structures, such as victim caches [25], prefetch buffers, write-back buffers, and noninclusive L1/L2 caches, can increase the effective associativity of the system drastically. Despite the above limitation, ADir$_p$NB still provides some optimization on implementing a full-map directory based coherence protocol for fine-grain physical shared memory.

In the proposed scheme, each memory block is mapped to a home node which keeps a directory entry for memory blocks exclusively mapped to a cache line or a cache set. One implication of the proposed scheme is that directory hardware is dependent on the cache size

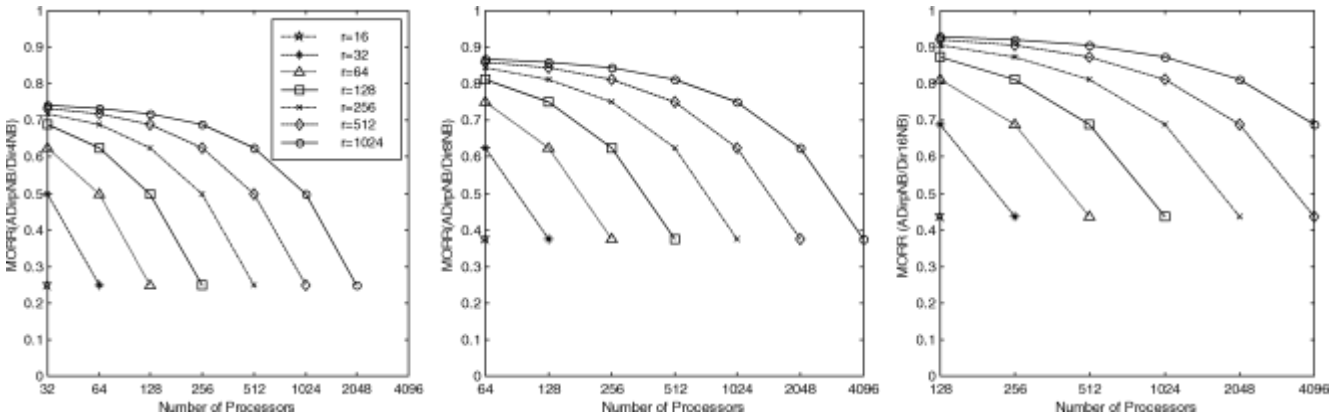

Fig. 8. Directory memory overhead comparison of associative full map directory and limited directories Dir$_4$NB, Dir$_8$NB, and Dir$_{16}$NB.

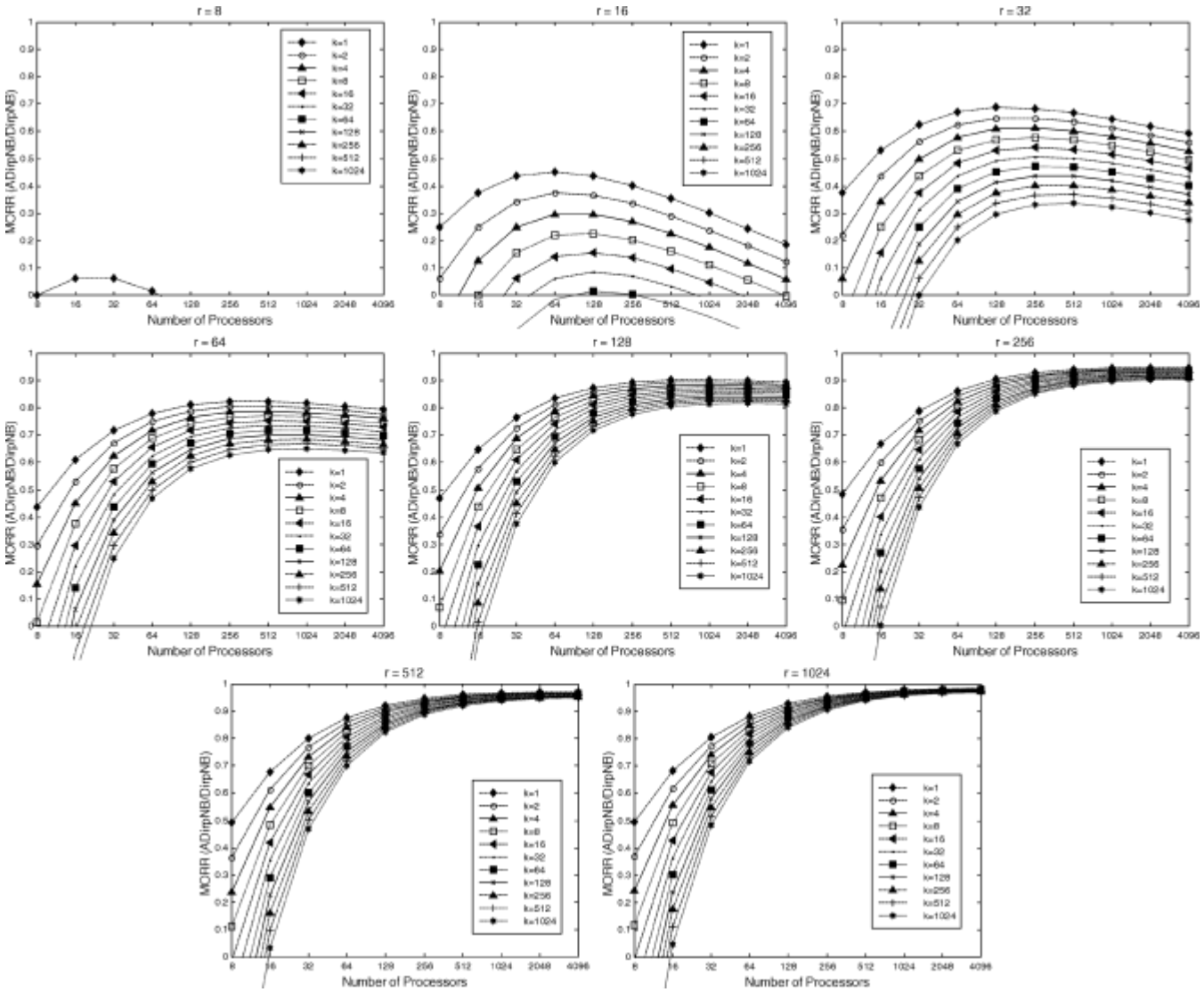

Fig. 9. Impact of cache set associativity on MORR.

and its associativity, eliminating some of the flexibility in allowing users to choose and upgrade cache sizes in their systems. Nevertheless, the relatively low memory overhead still makes it an attractive design alternative. Additionally, since the number of entries in this directory is small, it can be implemented in fast SRAM instead of slower DRAM, which may help to reduce directory information access time. This access time is in the critical path that determines the latency seen by the processor for many types of memory references [11].

4 Performance Evaluation

This section evaluates the performance of the proposed directory scheme quantitatively. We compare ADir$_p$NB with a limited nonbroadcast, a limited broadcast, a coarse vector, and a dynamic pointer directory running on a CC-NUMA system with applications from the SPLASH-2suite.

Experimental Methodology and Architectural Assumptions

The experimental platform used to evaluate the above directory protocols is SimOS [42], [19], a complete simulation environment that models hardware components with enough detail to boot and run a Silicon Graphics IRIX5.3 OS. SimOS includes multiple processor simulators (Embra, Mipsy, and MXS) that model the CPU at different levels of detail and supports simulation for both uniprocessor and multiprocessor architectures [18]. The performance results of this study are generated by Mipsy, which models a single-issue pipelined processor with a one-cycle result latency and a one-cycle repeat rate [19].

We modify SimOS numa memory model by porting the proposed scheme and other examined directory protocols. The default cache coherence protocol for numa model is a traditional full map directory. All simulated directory schemes use invalidation-based protocol and replacement hints. Our simulator can accurately model memory controller and DRAM, directory controller and directory

TABLE 2
Simulation Parameters and Architectural Assumptions

Processor Parameters	
Processors	16 @ 200MHz
Cache Parameters	
L1 cache	2-way associative, 32K
L1 cache line size	32 bytes
L1 hit time	1 cycle
L2 cache	2-way associative, 1M
L2 cache line size	128 bytes
L2 hit time	6 cycles
Memory Parameters	
Shared memory	256MB, 10 cycles access time

memory, network interface, and contention for these resources. The memory system is sequentially consistent.

We configure SimOS to simulate a CC-NUMA multiprocessor composed of 16 nodes connected by a network with fixed delay. Each node includes a 200 MHz compute processor with 32 KB split L1 and 1MB L2 caches, a portion of globally shared memory and directory, a directory controller implemented with simulated directory protocol, and a network interface. All caches are 2-way associative LRU caches with write miss allocation. Main memory consists of a total 256 MB DRAM with a 10-cycles access time. A memory access is local to a node if the accessed memory is allocated within the referring node. References that are not local to a node are classified as remote.[2] In our simulation study, headers for data packets and all other overhead packets (e.g., remote data request message, invalidations, acknowledgments, replacement hint, and NAck) are assumed to be 8 bytes long. The simulation parameters, architectural assumptions, and no-contention latencies of memory accesses and directory operations are summarized in Table 2 and Table 3. These latencies are set to be consistent with the relative processor, memory, and network speeds of the simulated machine.

Cycle-by-cycle simulation of the described architecture is performed. The instruction and data accesses of both applications and OS are modeled [32]. Because directory protocols vary the execution behaviors of applications by influencing their communication characteristics, an implication in comparing the performances of different directory protocols is to ensure that each simulation does the same amount of work. For this reason, the entire execution of each application is simulated to provide a fair comparison.

In this study, we use nine benchmarks, which cover a spectrum of memory sharing and access patterns from the SPLASH-2 suite [50], to evaluate the performance of different directory protocols. The applications and the input data/problem size are listed in Table 4. We use m4 macro preprocessor and Argonne National Laboratories (ANL) parmacs macros to automatically generate parallel code of each studied SPLASH-2 benchmark. All the benchmarks are compiled with MIPSpro CC compiler with optimization level —O2.

2. The minimum local miss time is $2 \times$ BUS_TIME + PILOCAL_DC_TIME + MEM_TIME and the minimum remote miss time is $2 \times$ BUS_TIME + PIREMOTE_DC_TIME + NILOCAL_DC_TIME-+NIREMOTE-DC-TIME + MEM-TIME + $2 \times$ NET-TIME.

Simulation Results

The effectiveness of the ADir$_{16}$NB directory scheme and its impact on the system performance are compared with those of a fully mapped Dir$_{16}$NB, a limited nonbroadcast Dir$_4$NB, a limited broadcast Dir$_4$B, a coarse vector Dir$_2$CV$_2$,[3] and a dynamic pointer allocation directory DynP. The DynP scheme is assumed to contain 1K pointer/link store entries and is simulated based on Simoni's model [44].

Fig. 10 shows directory overflow characteristics (measured by memory system traffic) that the studied directory organizations produce for each of the applications. Traffic is calculated as described in Section 4.1 and is normalized to the traffic produced by the Dir$_{16}$NB. Fig. 10 illustrates that Dir$_4$NB yields the largest number of memory traffic compared with other directory schemes. In the Dir$_4$NB scheme, the directory makes room for an additional requestor by invalidating one of the caches already sharing the block. This results in an increased number of misses and an increase in the data and coherence traffic. For applications that are well-suited to limited-pointer schemes (such as Water), the traffic is uniformly low for all directory entry organizations. On applications with a large fraction of mostly read data (such as Barnes and FMM), the explosion in memory system traffic caused by the nonbroadcast Dir$_4$NB can be as high as 960 percent and 700 percent, respectively.

The broadcast scheme Dir$_4$B outperforms Dir$_4$NB on all of the studied applications. Nevertheless, visible increases (2.15 times in Barnes, 2.06 times in FMM, and 1.8 times in Raytrace) of memory traffic are observed on applications (e.g., Barnes and FMM) where broadcasts are relatively frequent. In the Dir$_4$B, when a pointer overflow occurs, the broadcast bit is set. A subsequent write to this block causes invalidations to be broadcast to all caches. Some of these invalidation messages go to processors that do not have a copy of the block and, thus, the overall memory traffic is increased. Coarse vector directory Dir$_2$CV$_2$ further reduces memory traffic by only sending invalidations to a subset of processors in the system. Like Dir$_4$B, Dir$_2$CV$_2$ can also inflate memory traffic when broadcast becomes frequent. For example, in comparison with Dir$_{16}$NB, a Dir$_2$CV$_2$ can still yield 1.8 and 1.7 times traffic on benchmarks Barnes and FMM while showing competitive performance on most of the remaining applications. Additionally, we expect the performance gap between broadcast schemes (limited directory, coarse vector) and a full map directory to widen with increased number of processors because broadcast invalidations become increasingly more expensive on large systems.

Compared with the optimal Dir$_{16}$NB, the DynP scheme produces competitive performance on benchmark Choleksy, FFT, LU, and Water, in which a few cache blocks are widely shared. In these cases, the use of on-the-fly directory pointer allocation efficiently reduces directory

3. Dir$_2$CV$_2$ has two 4-bit coarse vectors and each coarse vector bit points to a region of two processors.

TABLE 3
Latencies of Different Memory and Directory Operations

Latency	Description	Cycle
BUS_TIME	Bus operation latency from a CPU to its local dc	15
PILOCAL_DC_TIME	Occupancy of dc on local miss	20
PIREMOTE_DC_TIME	Occupancy of local dc on outgoing remote miss	5
NILOCAL_DC_TIME	Occupancy of remote dc on remote miss	70
NIREMOTE_DC_TIME	Occupancy of local dc on incoming remote miss	5
MEM_TIME	Latency for the dc to fetch a block from local memory	10
NET_TIME	Fixed latency for going between dc across the network	30

TABLE 4
SPLASH-2 Benchmarks and Input Data/Problem Size

Programs	Description	Data Size
Cholesky	Blocked sparse matrix Cholesky factorization	tk15.O
Radix	Parallel Radix sort	1M integer keys, radix 1K
Ocean	Simulation of large-scale ocean movements	258×258 ocean grid
Raytrace	3-D scene rendering using ray tracing	teapot.env
Barnes-Hut	Hierarchical N-body system simulation	16K particles
FFT	Complex 1-D radix - $\sqrt{n}$ six step FFT	64k complex doubles
Water	$O(n^2)$ study of forces and potentials in water molecules	512 molecules
FMM	2-D hierarchical N-body simulation using Fast Multipole Method	16k particles
LU	Blocked dense linear algebra	512×512 matrix, 16×16 blocks

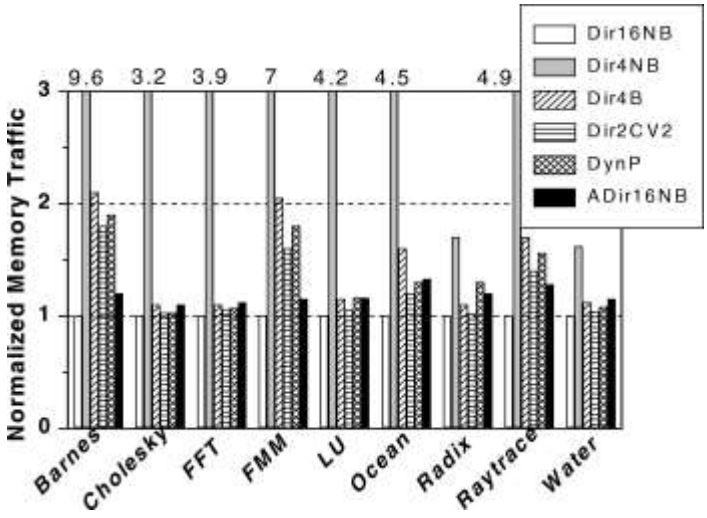

Fig. 10. Normalized memory traffic.

pointer overflows due to the small set of heavily shared cache lines. For example, less than 5 percent of memory traffic increases are observed in Choleksy, FFT, and Water. DynP suffers a performance penalty when it runs out of directory pointers, as it does on benchmark Barnes and FMM. On the two benchmarks, the memory traffic caused by extraneous directory overflows in DynP increases 0.9 times and 0.7 times compared with those on a full map directory. The competitions on fixed resource, such as pointer/link store entries, due to the different sharing patterns of various applications make the performance of DynP less robust.

By exploiting caching exclusiveness, $Dir_{16}NB$ yields attractive performance (in terms of memory traffic) across a spectrum of SPLASH-2 benchmarks. The traffic produced by $Dir_{16}NB$ is close to the ideal traffic of the $Dir_{16}NB$ for most applications. The extraneous memory traffic caused by $Dir_{16}NB$ is due to the NAck and retry messages used to maintain the exact sharing information in a centralized linked list style.

Fig. 11 further shows the execution time of the studied directory schemes normalized to that of the $Dir_{16}NB$. The performance results are found to be tightly correlated with memory traffic patterns shown in Fig. 10. The poor performance of the Dir_4NB, which stems from the largest directory overflows, is shown on most of the studied benchmarks. For example, the Barnes and FMM with the Dir_4NB run 6.4 and 5.8 times slower than those with the $Dir_{16}NB$. By only increasing invalidation traffic but not the miss ratio over that of the $Dir_{16}NB$, Dir_4B and Dir_2CV_2 run and 1.18 times slower than $Dir_{16}NB$ on Barnes.

On benchmarks LU, Radix, Cholesky, and Water, more than 95 percent of the invalidating writes produce only one invalidation [44]. In such cases, there are a few invalidating

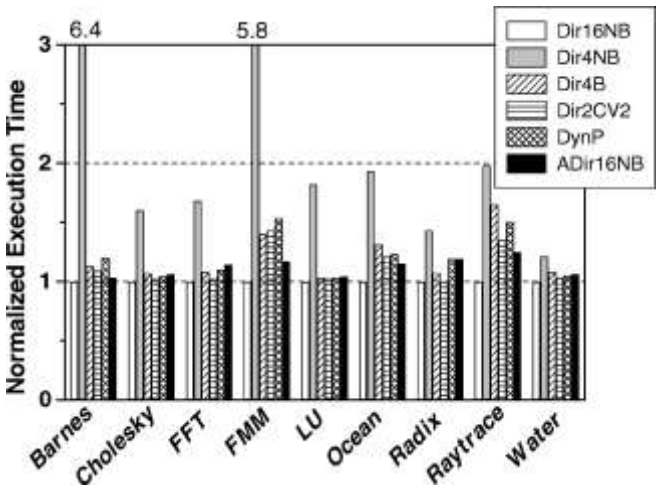

Fig. 11. Normalized execution time.

writes to the blocks which experience directory overflows. Thus, Dir_4B and Dir_2CV_2 exhibit approximate performance of the $Dir_{16}NB$. For benchmarks characterized by many mostly read and small migratory memory blocks, like FMM and Raytrace, the performance degradation of the limited directory schemes is not small due to the fact that the invalidating writes to the memory blocks of mostly read nature result in large invalidations. Not surprisingly, the directory overflow-free $ADir_{16}NB$ and $Dir_{16}NB$ are most robust and outperform other schemes across a spectrum of applications with various memory access patterns.

As described before, the $ADir_{16}NB$ requires replacement hints, NAck, and retry messages to maintain the exact memory block sharing information while exploiting caching exclusiveness. The introduction of replacement hints, NAck, and retry messages, however, could potentially increase coherency traffic. To understand this implication, we simulate a $Dir_{16}NB$ without the replacement hint, a $Dir_{16}NB$ with the replacement hint and $ADir_{16}NB$. The performance results of the above three directory schemes normalized to the traffic produced by a $Dir_{16}NB$ without replacement hints are shown in Fig. 12. We break traffic down into five major categories:

1. local data, which is the amount of data transmitted between processor and local memory,
2. remote data, which is the traffic transferred between nodes,
3. invalidations and acknowledgments, which are traffic associated with cache coherence maintenance,
4. replacement hints, which are the amount of messages used by the bookkeeping of accurate sharing status, and
5. NAck and retry, which are overhead to avoid running out of pointers during the transient state in an associative full map directory.

With the aid of replacement hints in the finite cache run, the directory knows about all cache replacements and is able to send fewer invalidations for some invalidating writes. Such benefit can be found in benchmarks Barnes and FMM. For other benchmarks, the use of replacement hints contributes to less than 10 percent of memory traffic increase. The visible increases are found on benchmarks Radix, Cholesky, and FFT, which show higher conflict miss rates [50]. In all studied benchmarks, NAck and retry messages slightly increase memory traffic. These indicate that the impact of replacement hints, NAck, and retry messages is not very detrimental on the total traffic.

5 Related Work

The Stanford DASH [30] and HAL-S1 [49] both implement a bit-vector protocol. Many hybrid directory schemes have been proposed as design alternatives of a full map directory [34]. One example is a pointer cache tagged directory [33] that organizes cache pointers as a cache, each entry of which is indexed by an address tag. The tag cache directory [39] is a variation of the pointer cache idea that uses two levels of caches in the directory. In both cases, when the directory cache runs out of space, a free entry has to be created by randomly choosing an active entry and invalidating the selected block in the indicated processor. In [21], Ho et al.

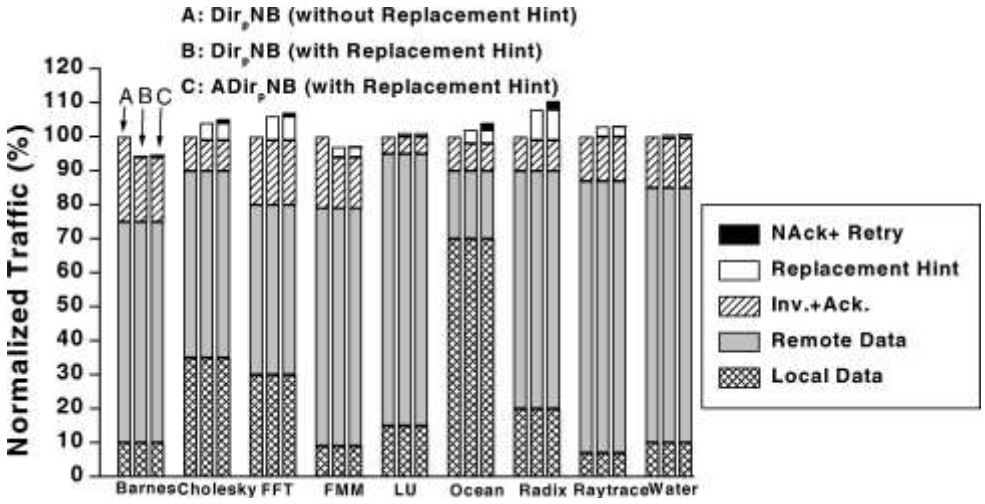

Fig. 12. Impact of replacement hint, NAck, and retry messages on traffic.

proposed a scheme called in-memory directories to eliminate the cost of directories by storing directory entries in the same memory used for the data that they keep coherent. ADir$_p$NB is a hybrid between directory cache and linked list directories and introduces a new efficient directory configuration between these two. By exploiting cache exclusiveness, ADir$_p$NB eliminates the need to store a tagged address for each directory entry and avoids directory overflows in an elegant manner.

The coarse vector directory [16] incorporates a versatile directory structure which can be dynamically interpreted, depending on the data sharing degree of a given memory block. Due to the introduction of coarseness in sharing information, invalidation messages may have to be sent to all processors identified by a unique group, regardless of whether they have actually accessed or are caching the block. The SGI Origin 2000 [29] implements a bit-vector/ coarse vector directory where the coarseness transitions immediately from 1 to 8 above 128 processors.

The LimitLPSS directory, which was implemented in the MIT Alewife machine [7], combines both hardware and software to implement a directory protocol. Overflow pointers are handled by software and the major overhead is the cost of the interrupts and software processing. For example, on a 16-processor system, the latency of five invalidations handled in hardware is 84 cycles, but a request requiring six invalidations handled by software intervention needs 707 cycles.

The dynamic pointer allocation scheme [43] is the default directory organization for the Stanford FLASH multiprocessor. It uses a directory header and a static pool of data structures, called the pointer/link store, to maintain precise sharing information in a linked list style. In FLASH implementation, all linked list manipulations are done in hardware by a special purpose protocol processor, MAGIC.

Our proposed associative full map directory differs from this scheme in that: 1) In ADir$_p$NB, a linked list created for a memory block is stored in one directory entry to facilitate indexing and replacement; 2) in our proposed scheme, the number of bits need to be stored for each pointer is $\log_2 p$, which is smaller than that for a pointer/link store, which can potentially point to a random portion of memory; 3) for the purpose of good performance, their pointer/link store should have a number of entries equal to 8 to 16 times the number of cache lines [44]. In ADir$_p$NB, however, the number of entries for a directory is only as many as those for a cache.

The Scalable Coherent Interface (SCI) [22], also known as IEEE Standard 1596-1992, is a typical linked list-based directory protocol. The basic SCI uses doubly linked lists that are distributed across the nodes. Various derivatives of the SCI protocol are used in several machines, including the Sequent NUMA-Q [35], HP Exemplar [5], and Data General Aviion [13]. The key trade-off is storage requirement, controller occupancy, number of network transactions, and serialization latency. Several SCI extensions [24], [27] have been proposed to help parallelized directory operations and reduce invalidation latency. The proposed ADir$_p$NB is generally simpler than doubly linked list based schemes.

The Scalable Tree Protocol (STP) [38] proposed by Nilsson and Stenström constructs and maintains the caches in the sharing set of a memory block in a tree structure. The STP guarantees logarithmic write latency by always maintaining an optimal tree structure and exploiting parallelism in the algorithms. Unfortunately, this approach sacrifices message efficiency and low read latency in order to construct and maintain a balanced tree, making it unsuitable for an application with a smaller degree of data sharing. The SCI tree extensions [24] is another example of tree-based protocols.

Agarwal et al. [1] first evaluated the performance of directory schemes (Dir$_1$NB, Dir$_0$B, and Dir$_4$NB) using traces generated by the ATUM address tracing tool on a four processor VAX 8350 running parallel applications, i.e., POPS, THOE, and PERO, on MACH. It is hard to compare our results to theirs because of the differences in both simulation methodology and benchmarks.

Chapin et al. [10] studied the memory system performance of IRIX 5.3 on CC-NUMA multiprocessors and concluded that OS data accesses do not follow the patterns discovered in application reference streams that motivated the design of limited directory schemes. However, they did not show the impact of different directories on CC-NUMA architecture quantitatively as we do.

More recently, Michael et al. [36] studied the performance of a full map directory cache protocol with alternative coherence controller architectures on a 4×16 CC-NUMA system. They found that the occupancy of coherence controllers can be a bottleneck for applications with high communication requirements (i.e., ocean, radix, and FFT). Dual protocol engines improve performance by up to 18 percent (with HWC implementation) and 30 percent (with PP implementation) relative to the single protocol engine. Our proposed scheme has simplified and atomic directory operations and can be implemented with either an HWC or a PP.

Dahlgren et al. [12] evaluate the combined performance gains of several extensions to a directory-based invalidation protocol, namely, adaptive sequential prefetching (P), migratory sharing optimization (M), and competitive-update (CW) mechanism. They found that the performance of a directory protocol augmented by appropriate extensions (e.g., P+CW, P+M) can eliminate a substantial part of the memory access penalty without significantly increasing the complexity of either the hardware design or the software system. These optimizations can be used in ADir$_p$NB because they are orthogonal to our technique.

Heinrich et al. [17] evaluate the performance of four scalable cache coherence protocols, including coarse vector, dynamic pointer allocation, SCI, and COMA protocol, using SimOS Mipsy and FlashLite simulators. They found that the optimal protocol changes for different applications and can change with processor count, even within the same application. Wood et al. [51] explored the complexity of implementing directory protocols by examining their mechanisms ranging from directory primitive operations to network interfaces. It is found that, with increasing network latencies, the performance effect of directory operation overhead decreases, which provides the opportunity to sequence

directory operations in a processor rather than a dedicated directory controller.

Nanda et al. [37] studied the impact of applying parallel mechanisms, such as multiple protocol engines, pipelined protocol engines, and split request-response streams, on the occupancy of the coherence controllers. Their experimental results showed that each mechanism is highly effective at reducing controller occupancy by as much as 66 percent and improving execution time by as much as 51 percent on both commercial and scientific benchmarks.

6 Conclusion

This paper proposes a new coherence scheme called associative full map directory (ADir$_p$NB), which behaves like a traditional full map directory and gracefully decreases the directory memory requirement. The associative full map directory is unique and distinguishes itself from previous schemes by dynamically examining and exploiting caching exclusiveness of multiple memory blocks. Directory bits are dynamically allocated and reclaimed for a set of caching exclusive memory blocks. By implementing replacement hints, the proposed technique can emulate a traditional full map directory with lower memory overhead, fairly simple protocol modification, and appropriate hardware addition. Our analysis shows that the directory memory efficiency of the proposed scheme is promising: On a typical architectural paradigm, ADir$_p$NB reduces the memory overhead of a traditional full map directory by up to 70-80 percent. For some optimal memory and cache configurations, ADir$_p$NB is more memory-efficient than even inexpensive limited directories such as Dir$_4$NB and Dir$_8$NB.

We evaluate the performance of the proposed technique by using a SimOS simulation platform that runs the IRIX5.3 OS and SPLASH-2 applications. Our simulation results show that, due to the elimination of directory overflows, the speed up of ADir$_p$NB can be competitive with that of a Dir$_p$NB on the studied workloads. Thus, we believe that ADir$_p$NB can be employed as a design alternative of full map directory for moderately large-scale and fine-grain shared memory multiprocessors.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," Proc. 15th Ann. Int'l Symp. Computer Architecture, pp. 280-289, 1988.
[2] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," Proc. 22nd Ann. Int'l Symp. Computer Architecture, pp. 2-13, 1995.
[3] J.K. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," ACM Trans. Computer Systems, vol. 4, no. 4, pp. 273-298, Nov. 1986.
[4] L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," Proc. 25th Ann. Int'l Symp. Computer Architecture, pp. 3-14, 1998.
[5] T. Brewer and G. Astfalk, "The Evolution of the HP/Convex Exemplar," Proc. COMPCON Spring '97: 42nd IPPP CS Int'l Conf., pp. 81-86, 1997.
[6] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problem in Multicache Systems," IPPP Trans. Computers, vol. 27, no. 12, pp. 1112-1118, Dec. 1978.
[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), pp. 224-234, 1991.
[8] D. Chaiken and A. Agarwal, "Software Extended Coherent Shared Memory: Performance and Cost," Proc. 21st Ann. Int'l Symp. Computer Architecture, pp. 314-324, 1994.
[9] Y. Chang and L.N. Bhuyan, "An Efficient Tree Cache Coherence Protocol for Distributed Shared Memory Multiprocessors," IPPP Trans. Computers, vol. 48, no. 3, pp. 352-360, Mar. 1999.
[10] J. Chapin, S.A. Herrod, M. Rosenblum, and A. Gupta, "Memory System Performance of UNIX on CC-NUMA Multiprocessors," Proc. 1995 ACM SIGMPTRICS Conf. Measurement and Modeling of Computer Systems, pp. 1-13, 1995.
[11] D.E. Culler, J.P. Singh, and A. Gupta, Parallel Computer Architecture: A Hardware/ Software Approach. Morgan Kaufmann, 1999.
[12] F. Dahlgren, M. Dubois, and P. Stenström, "Combined Performance Gains of Simple Cache Protocol Extensions," Proc. 21st Ann. Int'l Symp. Computer Architecture, pp. 187-197, 1994.
[13] Data General Corp., "Aviion AV 20000 Server Technical Overview,"Data General White Paper, 1997.
[14] M. Dubois, C. Scheurich, and F.A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," Computer, vol. 21, no. 2, pp. 9-21, Feb. 1998.
[15] J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," Proc. 10th Ann. Int'l Symp. Computer Architecture, pp. 124-131, 1983.
[16] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Scheme," Proc. Int'l Conf. Parallel Processing, pp. 312-321, 1990.
[17] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta, "A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," IPPP Trans. Computers, vol. 48, no. 2, pp. 205-217, Feb. 1999.
[18] S. Herrod, M. Rosenblum, E. Bugnion, S. Devine, R. Bosch, J. Chapin, K. Govil, D. Teodosiu, E. Witchel, and B. Verghese, "The SimOS User Guide," http://simos.stanford.edu/userguide/ 1998.
[19] S.A. Herrod, "Using Complete Machine Simulation to Understand Computer System Behavior," PhD thesis, Stanford Univ., Feb. 1998.
[20] M.D. Hill, J.R. Larus, S.K. Reinhardt, and D.A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), pp. 262-273, 1992.
[21] C. Ho, H. Ziegler, and M. Dubois, "In Memory Directories: Eliminating the Cost of Directories in CC-NUMAs," Proc. 10th Ann. ACM Symp. Parallel Algorithms and Architectures, pp. 222-230, 1998.
[22] IPPP Std 1596-1992: IPPP Standard for Scalable Coherent Interface, New York: IEEE, Aug. 1993.
[23] D.V. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi, "Distributed-Directory Scheme: Scalable Coherent Interface," Computer, vol. 23, no. 6, pp. 74-77, June 1990.
[24] R.E. Johnson, "Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors," PhD thesis, Univ. of Wisconsin-Madison, 1993.
[25] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," Proc. 17th Ann. Int'l Symp. Computer Architecture, pp. 364-373, 1990.
[26] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon, "Implementing a Cache Consistency Protocol," Proc. 12th Ann. Int'l Symp. Computer Architecture, pp. 276-283, 1985.
[27] S. Kaxiras, "Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors," PhD thesis, Univ. of Wisconsin-Madison, 1998.
[28] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," Proc. 21st Ann. Int'l Symp. Computer Architecture, pp. 302-313, 1994.
[29] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," Proc. 24th Ann. Int'l Symp. Computer Architecture, pp. 241-251, 1997.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford DASH Multiprocessor," Computer, vol. 25, no. 3, pp. 63-79, Mar. 1992.
[31] T. Li and B.W. Rong, "A Versatile Directory Scheme (Dir$_2$NB L) and Its Implementation on BY91-1 Multiprocessors System," Proc. IPPP Advances on Parallel and Distributed Computing, pp. 180-185, 1997.
[32] T. Li, L.K. John, N. Vijaykrishnan, A. Sivasubramaniam, A. Murthy, and J. Sabarinathan, "Using Complete System Simulation to Characterize SPECjvm98 Benchmarks," Proc. Int'l Conf. Supercomputing, pp. 22-33, 2000.
[33] D.J. Lilja and P.-C. Yew, "Combining Hardware and Software Cache Coherence Strategies," Proc. 1991 Int'l Conf. Supercomputing, pp. 274-283, 1991.
[34] D.J. Lilja, "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," ACM Computing Surveys, vol. 25, no. 3, pp. 303-338, Sept. 1993.
[35] T.D. Lovett and R.M. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," Proc. 2Srd Ann. Int'l Symp. Computer Architecture, pp. 308-317, 1996.
[36] M.M. Michael, A.K. Nanda, B.-H. Lim, and M.L. Scott, "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," Proc. 24th Ann. Int'l Symp. Computer Architecture, pp. 219-228, 1997.
[37] A.K. Nanda, A.T. Nguyen, M.M. Michael, and D.J. Joseph, "High-Throughput Coherence Controllers," Proc. Sixth Int'l Symp. High Performance Computer Architecture, pp. 145-155, 2000.
[38] H. Nilsson and P. Stenström, "The Scalable Tree Protocol—A Cache Coherence Approach for Large-Scale Multiprocessors," Proc. IPPP Symp. Parallel and Distributed Processing, pp. 498-506, 1992.
[39] B.W. O'Krafka and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," Proc. 17th Ann. Int'l Symp. Computer Architecture, pp. 138-147, 1990.
[40] M.S. Papamarcos and J.H. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," Proc. 12th Ann. Int'l Symp. Computer Architecture, pp. 348-354, 1985.
[41] S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Tempest and Typhoon: User-Level Shared Memory," Proc. 21st Ann. Int'l Symp. Computer Architecture, pp. 325-336, 1994.
[42] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," IPPP Parallel and Distributed Technology: Systems and Applications, vol. 3, no. 4, pp. 34-43, Winter 1995.
[43] R. Simoni and M. Horowitz, "Dynamic Pointer Allocation for Salable Cache Coherence Directories," Proc. Int'l Symp. Shared Memory Multiprocessing, pp. 72-81, 1991.
[44] R. Simoni, "Cache Coherence Directories for Scalable Multiprocessors," PhD dissertation, Stanford Univ., Oct. 1992.
[45] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy, "Flexible Use of Memory for Replication/ Migration in Cache-Coherent DSM Multiprocessors," Proc. 25th Ann. Int'l Symp. Computer Architecture, pp. 342-355, 1998.
[46] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," Computer, vol. 23, no. 6, pp. 12-24, June 1990.
[47] P. Stenström, T. Joe, and A. Gupta, "Performance Evaluation of Cache-Coherent NUMA and COMA Architecture," Proc. 19th Int'l Symp. Computer Architecture, pp. 80-91, 1992.
[48] M. Thapar, B. Delagi, and M.J. Flynn, "Linked List Cache Coherence for Scalable Shared Memory Multiprocessors," Proc. Int'l Symp. Parallel Processing, pp. 34-43, 1993.
[49] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke, "The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers," Proc. 24th Ann. Int'l Symp. Computer Architecture, pp. 98-107, 1997.
[50] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Consideration," Proc. 22nd Ann. Int'l Symp. Computer Architecture, pp. 24-36, 1995.
[51] D.A. Wood, S. Chandra, B. Falsafi, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, S.S. Mukherjee, S. Palacharla, and S.K. Reinhardt, "Mechanisms for Cooperative Shared Memory," Proc. 20th Ann. Int'l Symp. Computer Architecture, pp. 156-167, 1993.