Primary-node Scheme for Cache Coherence in Large Scale Shared-Memory Multiprocessors

Guoying Chen *

Ultracomputer Note #181
January, 1992
(This is revision 1.5, as of Jan. 31, 1992)

Abstract

In this paper we propose a new cache coherence scheme called the *primary-node* method, capitalizing on the observation in [17] that there is usually a primary processor accessing a data block with much higher frequency than other sharing processors (secondary nodes). We associate each memory block with only one pointer that points to the "primary" node responsible for the majority of consistency operations for that memory block, while the remainings are performed collaboratively by both secondary nodes and the primary node. As expected, an overwhelmingly large percentage of consistency operations for the block can be performed locally at the primary node, reducing both the access latency and network traffic related to the coherence actions. The overall memory consumption for this scheme is $O(Clog_2N + Mlog_2N)$, almost the same as for the chained pointer scheme. Unlike the chained pointers, our scheme eliminates the large latency problem caused by sequential invalidations. Performance analysis shows that our scheme is simple, efficient and scalable.

Keywords: Caches, cache coherence, directory schemes, primary-node scheme, shared-memory multiprocessors.

*Supported in part by U.S. Department of Energy under grant number DE-FG02-88ER25052 and in part by US National Science Foundation under grant number MIP-8915488

1 Introduction

Cache coherence is a well-known problem in multiprocessors with multiple caches [4]. Numerous schemes (both hardware-based and software-based) have been proposed in the literature [18]. Directory-based schemes [5] [6], a hardware approach, are among the most promising solutions because they rely only on point-to-point interconnects rather than a network broadcast mechanism. In such schemes, a directory is maintained (either in memory or in caches) to keep track of those caches which are holding the memory blocks (an entry per memory block). However, the memory overhead of the traditional directory scheme (particularly the full-map directory) tends to be prohibitive for building large scale shared-memory multiprocessors if identities of all processors are kept.

Tremendous efforts have been made to reduce the storage overhead while hoping to keep the performance as efficient as the full-map scheme. These efforts include limited directories in [2], limitLESS scheme in [7], chained directory in IEEE Scalable Coherence Interface (SCI) standard and its extensions [12], MIN-based scheme in [10], dynamic pointer allocations in [16], and tree-based and hierarchical full-map directory in [15]. Even so, many of them do not perform as well or as effectively as expected.

As an example, for limited directories, in order to attain a certain level of performance, enough pointers must be included in each entry of the directory to make the probability of pointer overflow small. Typically, the average number of limited pointers are $2-4$. To decrease the probability of pointer overflow and attain higher level of performance, one may use as many pointers as $6-10$. However, using more pointers in each entry results in most of them going unused, since the most frequent cases use only a few $(2-4)$ of pointers (the most important motivation behind proposing the limited directory scheme). As long as the number of pointers is fixed, no measures can be taken to decrease the length of directory. Moreover, situations can still occur in which the limited schemes run short of available pointers so that some mechanism is required to handle the overflow case.

Other approaches besides the limited directories also have their drawbacks one way or another [1]: either low performance is inherited from the conventional limited directory, or their complexity is much increased, or some responsibilities are put on to other components of multiprocessor systems (such as compilers). A simple, efficient and scalable cache coherence scheme is needed.

In this paper, we propose a new cache coherence scheme called the *primary-node* method. This scheme capitalizes on the characteristic of parallel applications observed in [17]: there is usually a primary processor that accesses the data with much higher frequency than other sharing processors. The trace analysis and modeling studies in [17] reveal that the ratio of frequencies with which the primary and secondary processors access the block usually ranges from 10 to 500, and sometimes even higher. This motivates us to devise a new scheme using such useful characteristics.

In our scheme, associated with each memory block, there is only one pointer which points to its "primary" node — the node that accesses the block much more frequently than other nodes (called secondary nodes). The primary node is responsible for most of the work of data coherency with respect to the associated memory block, while secondary nodes collaborate with the primary node for other coherent operations, which are induced on secondary nodes. As expected, an overwhelmingly large percentage of consistency activities of the block can be performed locally at the primary node, reducing both the access latency and the network traffic due to coherence actions

[1] More issues will be addressed in section 4.

and thus boosting system performance. Our technique, allowing the primary node to maintain a linked list of secondary pointers, is to use a portion of local memory as a *free pointer pool* on the primary node. Each secondary pointer of the linked list is allocated (released) dynamically from (to) the free pointer pool. The total size of the linked list in the system is $Clog_2N$, where C is the total number of cache lines, N the number of caches. Let M be the number of total memory blocks. Then the overall memory consumption for this scheme is $O(Clog_2N + Mlog_2N)$, almost the same as the chained pointer scheme, but our scheme eliminates the biggest problem of the chained directories — large latency of sequential invalidation through the chains. Because of the above features, we believe that our scheme is simple, efficient and scalable and has a performance comparable to the full-map directories.

A challenging problem for the primary-node scheme is how to determine the primary node for a given memory block. Our solution is based on the technique of initial random assignment and then sampling the cache accesses for each copy so as to choose the primary node dynamically when the parallel programs are running. Such a method makes it possible to distribute the primary nodes evenly for different memory blocks among all processors, and thus alleviate possible bottleneck problems due to the primary-node becoming a central sharing resource if not appropriately distributed.

The remainder of this paper is organized as follows. Section 2 presents the primary-node cache coherence scheme. Section 3 describes the assignment and distribution problem of primary node and secondary nodes. In section 4, we compare the primary-node scheme with other related work and discuss some possible enhancements. Section 5 covers our preliminary performance analysis of the scheme. We summarize our work and discuss some future research directions in section 6.

2 The Cache Coherence Scheme

2.1 Definitions and Basic Mechanisms

Our objective is to devise a simple, efficient and scalable cache coherence scheme. In our work, we consider general and large scale MIMD shared-memory multiprocessors with N processors and interconnected networks (e.g., multistage or mesh), where N is the number of Processing Elements (PEs) in the system (usually also the number of caches). Each PE has its own cache memory. The only requirement for our scheme is the availability of local memory at each node with an allocated *free pointer pool* for any primary cached copy to link all its secondary node pointers; otherwise, the equivalent amount of space has to be specially arranged by the system. We believe such a requirement is reasonable since the tendency in design of shared-memory multiprocessors is to include a private local memory for each processor.

We assume in our system that there is a path between every cache pair and a path between each cache and memory module. Our scheme is based on the sequential consistency model. Our methodology can in principle be applied to a more relaxed memory consistency model [1] [8] [9].

In our discussion, a *node* often refers to a PE with its cache. A memory block means a logical unit with a number of words as well as an identification. Each memory module consists of a number of memory blocks. Copies of a memory block can reside in more than one cache. For a given memory block, there is a PE, called the *primary node*, which is responsible for the major work of cache coherency for the given memory block; while other PEs are to assist the primary node in keeping the block coherent. We hereby call them *secondary nodes* for the memory block.

Consequently, a cached copy resident on the primary node is called a *primary copy*; otherwise it is called a *secondary copy*.

The primary copy, in our scheme, always has the up-to-date values of its associated block. In other words, it could either be the only owner of that block, or co-own the block with a secondary copy which currently has the write permission. Apparently, the semantics of ownership here is different from the conventional schemes in the situation of co-ownership. By co-ownership, we mean that it happens between the primary copy and a secondary node. The secondary node has the write permission, but the primary node does not, but has the responsibility of keeping the most recent value of the updated secondary copy. Additional care must be taken to define the primary-node scheme in co-ownership situation (see next subsection).

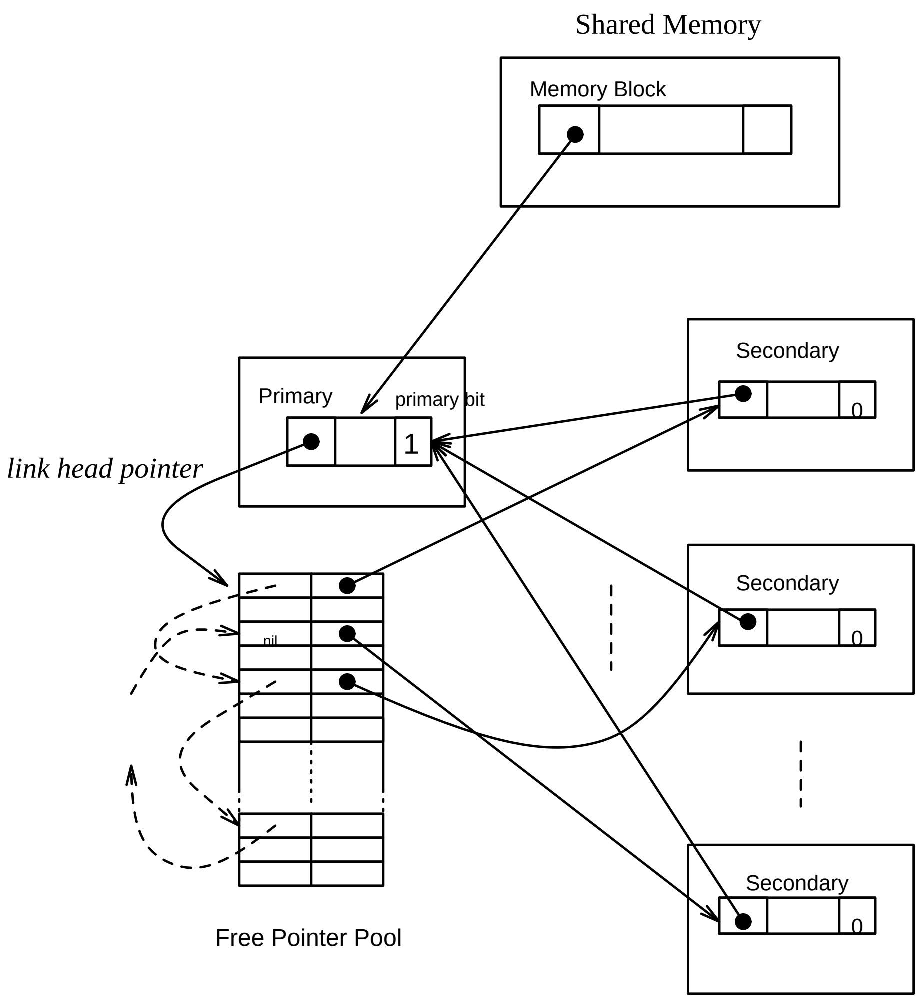
From the observations in [17], we expect that quite a large percentage of accesses to a given memory block can be carried out on the primary node (the percentage ranges from 90% to 99.8%, if the ratio of primary node access to secondary access is between 10 to 500), which will reduce communication cost and make positive contributions to the system performance.

Like many other directory schemes, our primary-node scheme relies on the caches containing data that is reasonablely distributed across the processing nodes in main memory. However, the reference patterns and behavior for a given memory block are primary-node dominated as observed in [17]. Our scheme, unlike the traditional limited directories, provides flexibility for expanding the size of pointers as needed by programs, as we will see later on.

In our scheme, the cache coherency of a given memory block is maintained mostly by the primary node, and the primary copy either owns or co-owns the block whenever it is cached. Remaining coherence operations (i.e., induced from secondary nodes) are cooperatively performed by all associated nodes with the coordination of the primary node. With each memory block, there is a single pointer used to point to its *primary* copy. Unlike the traditional directory scheme, the dirty bit associated with the memory block is not needed because the equivalent job is done by the primary node. In other words, the primary node represents the memory block in keeping the values and states consistent among the memory block itself and all its cached copies.

For each local cache, several bits are used: (1) one primary bit; (2) a number of bits for a counter; (3) two bits for cache states. The primary bit indicates whether or not its node is primary. The counter bits are used to keep track of the access frequencies on the copy (used for determining the primary node, as described later). The state bits maintain the states in a processor's cache. In our primary-node scheme, four states are possible: *private, invalid, shared-clean, dirty.* The *private* state means that no other node is caching the copy. It behaves like a cache copy in a uniprocessor. The *invalid* state means this copy is stale and cannot be used, and the primary node must be consulted. The *shared-clean* state implies that there may be other nodes caching that block. Finally, the *dirty* state implies that this cache has an exclusive copy of the memory block and the block has been modified.

The difference between our scheme and other directory schemes is that we introduce the notion of the primary-node, as described above. The primary node records the locations of the secondary copies, each with a Log_2N bit pointer. Obviously, these pointers can not be stored in the cache itself. Otherwise it would make the cache tag bit fields too large and require the tag to be dynamically expandable, which is unlikely. Storing the secondary pointers in caches is in any case expensive. However, it is possible to store the secondary pointers in shared-memories as the dynamic pointer allocation scheme does. But for MIMD shared-memory architectures, looking up these secondary pointers would go through a network, increasing both the latency and introducing more network



(Note: Not shown are state bits and count bits in caches)

Figure 1: The Directory Organization for Primary-node Scheme

traffic.

As shown in Figure 1, our approach is to allocate these secondary pointers dynamically from a free pointer pool resident on each node (either in its local memory if available or in a special area provided by the system). Thus a special register for *free pool pointer* (not shown) must be allocated at each node to contain the address of the first available free pointer in the pool. The pointer from the primary copy is the head of a linked list of secondary pointers, each with an auxiliary link (in dashed lines). We refer to the pointer from the primary copy as the *link head pointer*. If such a *link head pointer* is empty, we know that no secondary nodes exist and the memory block is not shared. Otherwise, it implies that at least two caches (including the primary node) share the block. In secondary nodes, the corresponding pointer bits are used to point back to the primary node. Thus, the paths between the primary node and the secondary nodes are established. The communication between secondary nodes and the memory module has to go through the primary node. This makes our scheme distinct from other directory schemes. By the Dir notion introduced in [2], our scheme can be denoted as a $Dir_{1^*_p}NB$ directory scheme, since from each memory block only one pointer connects to the primary node, while the primary node stores all secondary node identities, where p symbolizes the primary-node method and NB means no broadcasting is used.

2.2 Operations and Behavior

In a private state, only one copy is cached in the system, and no cache coherency enforcement is required. In a shared state, however, at least two copies are cached in two different nodes. One node is *initially* assigned to be the primary and the others are secondary node(s). The initial assignment can be done on a pseudo-random basis, as it will be discussed in the next section about the primary node determination problem.

Let us define the scheme and its operations, two parts need to be discussed: coherency operations on the primary node, and coherent activities on any secondary node.

On the primary node. The primary node does most of the work for the data consistency of the associated memory block because a large percentage of block accesses take place on it. So, to boost the performance, it is good to make consistency operations on the primary node as simple as possible.

On a read hit, the operation is trivial, the primary node simply supplies it. Since the primary node always keeps its copy consistent with the memory block and also coordinates the consistency with secondary copies, the read can be carried out locally without having to inform secondary nodes.

Similarly, a write hit can also be done locally by updating the cached block and writing it through the memory module, if the current state of primary copy is *dirty* (i.e., with the write permission). If the state is *not dirty* on the primary node, it means there must be a secondary node which has a dirty copy and holds the write permission. As the values of the block are changed on the primary node, the secondary copies must be notified and their corresponding values must be updated or invalidated, depending on a protocol's policy[2]. At this point, the primary node is the only owner of the block, the secondary node with the dirty copy must transfer the write permission to the primary node. Then, copies on secondary nodes are invalidated, and the current state of the primary copy is changed to *dirty*.

[2]As in other directory schemes, whether to adopt the policy of update or invalidation is an interesting question. Studies are needed on this aspect. We assume that the invalidation policy is used in our scheme.

It should be considered as an unusual case if a read miss occurs, for by the primary copy we usually mean it owns the block by itself only or co-owns it with secondary nodes. However, a read miss can occur if it is the first time accessing the cache (in cold start) or was previously replaced out. As the node is primary, for both cases, the read request is satisfied by the memory module, and some initialization is made for the primary copy: secondary copies are invalidated if any; and then the cache state is set to *shared-clean*.

The procedure for a write miss is as follows: the memory module supplies the data; secondary copies must be sent invalidation signals so that the cache with the miss may load an copy from the primary node. The primary node now holds the write permission. The update to the memory module is carried out immediately, and the cache state on primary node is changed to *dirty*.

On any secondary node. The work done by a secondary node a little bit complicated, because there are some interactions between the primary node and the secondary node in question. It depends on the operation types and the current state of the cache.

Consider a read request. It can be a hit or a miss. Upon a read hit, if the cache state is *shared-clean*, or dirty (having write permission), the local cache can supply it. The cache state remains unchanged. Upon a miss, the primary-node must be sent a request. The up-to-date value is then supplied by the primary node. Special care must be taken here: ont one hand, if the state of primary copy is *dirty*, the memory locations must be updated, and the state of primary copy must be changed to *shared-clean*; but if the dirty copy resides not on primary copy, it must be owned by another secondary copy, that secondary copy must change it state to *shared-clean*. On the other hand, if the primary node does not have copy in its cache at the time of request, a read miss occurs and the memory module supplies the data, and then the primary node forwards it to the requesting node. In both cases, the state of requesting cache is changed to *shared-clean* upon completion of the request. Additional work should be done if the miss is not due to an invalid state (i.e., cold miss): the primary node has to maintain the linked list: allocate a free pointer for the new secondary copy; and include it in the linked list.

A write request can also be a hit or a miss. Upon a write hit, if the current cache state is *dirty*, the write operation can be performed locally and its state is unchanged. In the meantime, the primary copy has to be notified of the new value since the primary must maintain consistency among secondary nodes and the primary node. If the current state on the secondary node is *shared-clean*, the ownership request must be sent to the primary node along with the new value. The primary copy invalidates other secondary copies and waits for their acknowledgements. Until the acknowledgements are received, the primary node will not guarantee write permission to the requesting node. Once it receives the permission from the primary node, the secondary node in question can write the cache immediately. The cache state is then changed to *dirty*. Upon a write miss, on the other hand, a write permission is resuested to the primary node. Then, the primary node informs all other secondary copies about the new assignment of the ownership and invalidates them. Similarly, as in the previous case, only after having received the write permission can this secondary node write cache and set the cache state to *dirty*.

As we see, the operations on the primary node are very simple, while those on the secondary nodes are somewhat complicated. As expected, this fits very well with our objective, i.e. a large percentage of data accesses performed on the primary node, and a small portion of accesses made on the secondary nodes. Our coherence protocol will enable the operations on the primary node to be carried out efficiently. Hopefully, it will contribute to reducing both the access latency and the network traffic. The memory consumption for this scheme is comparatively small. Let M be the number of memory blocks, N the number of Processors (caches), C the total number of cache block copies, T the counter bits, and B the bits for states. Then the total memory overhead is $Clog_2N + C(T + B + Log_2N) + MLog_2N$. Compared C and M, T and B are relatively small and thus can be neglected. Thus, the total overhead becomes $O(Clog_2N + MLog_2N)$, almost the same as the chained pointer scheme, However, because the primary node and its associated linked list are centralized (in contrast to the distribution of the chained linked list in chained directories), the primary scheme does not suffer large penalties due to sequential invalidations as for the chained directories.

An exceptional case (although it is very rare) must be taken care of, i.e. when the free pointer pool is full and no more pointers can be dynamically allocated from the pool. The correct action in this situation is to use some means to select a pointer or pointers from the linked list and then release it by sending an invalidation to the cache identified by the pointer. Similar techniques can be used as in [16]. Of course, there is some trade-off to set the pool size: the larger the pool, the smaller the chances of running short of free pointers; but if the pool is set too large, many free pointers are potentially wasteful.

In summary, the primary-node scheme utilizes the characteristics of access distribution. Coherent operations on the primary node are simple and efficient, while the chances of accesses are large; Coherent operations from the secondary nodes are allowed a little bit complicated, but it is reasonable because their access frequencies are much less than those on the primary node. The small memory consumption of the directory makes the primary-node scheme scalable to the large scale systems.

3 Primary vs. Secondary

3.1 Primary-node Determination

The challenge for our scheme is to determine the primary node for each shared memory block. A very simple approach is to initially randomize it among the available processors. That is to say, for any given shared memory block, the initial primary node is designated randomly. However, such a method is a static solution. When parallel codes are being executed, the data accesses are dynamic and the primary nodes cannot be determined statically. In other words, it is very likely that the initial assignment of the primary node may not be correct. Thus, we must resort to a dynamic approach.

A most beautiful case is to do nothing when there is imbalance in accesses (towards heavy accesses in primary node). The simplest scheme, picking a random accesser, works well when there is gross imbalance; picking the first random accesser which has two successive accesses is even better, and so on. Unfortunately, this doesn't work if the accesses to the various cached copies are just partially counted. However, a similar idea could be made to work by requiring each cache to send some information to the primary node every 1 — 10 microseconds (depending on the system parameters) about the access. The idea is similar to a sampling technique. Recall that a counter is used in the cache. The purpose of doing so is to maintain the statistical information about the past accesses to the associated cached copy. We are not counting all accesses, but a fraction of them can tell us which node is access dominant for that block. This would make the space for the counter (in T bits) much less than the tag, for example. 12 to 16 bits for T may be enough (if the $log(i)$ approximation technique is used as described below, the T would be much smaller). Hence, the storage overhead is not so significant.

One might be concerned by frequent message sending between the primary node and the secondary nodes. Actually, the overhead is not big. The number of messages can be saved if the access information reporting goes together with the normal messages (in both directions). By doing so, the secondary nodes and primary node know each other's current access count. One of the following actions can be taken:

- Every cache notification from the secondary nodes that goes to the primary node would contain the statistics for this block; presumably this includes all writes and some reads (such as the reads without ownership in dirty state), so at least the statistics for writers and some reads would be up-to-date.
- The primary node should be notified whenever a cache copy on the secondary node has an access count apparently larger than the access count on the primary node.

This gives the primary node much information about the use of the cache copies on the secondary nodes. In the first case, control could be yielded to a dominant writer, and, in the second, by any combination of dominant accesses.

A more economical solution would be to approximate $log(i)$, where i is the current count. The algorithm is simply:

Whenever an event occurs, increment i with probability 2^{-i}.

A variant would be to keep a clock count K in each processor, and increment a count i if bit i of K has just changed its value. The logic for this is trivial.

3.2 Primary-node Switching

When the primary node is changed, switching must take place. The method for switching is three simple steps: First, a packet containing pointers is sent from the current primary node to the new primary node, and then the used pointers are released to the free pointer pool. Second, when the packet is received by the new primary node, the same size of pointers are allocated from the free pointer pool on the new primary node and the received pointers are copied over, linked together one by one in the newly allocated space. In the meantime, the pointers of the old and new primary node are exchanged, and the old primary node is reset as the secondary node with its primary bit being 0. The new primary node is set with its primary bit being 1. Third, the primary pointer from the memory block is directed to the new primary node, and all secondary copies are pointing to the new primary node.

When the primary-node switching takes place, the access to the corresponding copy may have to be stalled until the swicthing is completed.

In summary, it is possible to solve the primary-node determination problem dynamically by a very simple method. Switching is required if there is unfavorable imbalance.

4 Related Work and Possible Enhancements

4.1 Related Work and Comparisons

A conventional directory protocol such as the one used in DASH project [13] is obviously not a good solution for large scale parallel machines simply because the memory overhead prohibits the scalability. The limited directory [2] performs very poorly when pointers are exhausted, especially if a broadcast mechanism is used. Although the LimitLESS scheme [7] provides a flexible way to handle the pointer overflow, it is likely to perform poorly for blocks with moderately high read/write ratios, since a substantial fraction of read misses as well as writes will require software traps [17].

The tree-based scheme and the hierarchical full-map directory (HFMD) may be among the potentially good candidates for large scale multiprocessors. In particular, the HFMD directory can be embedded in different interconnection topologies. For example , the hierarchical directories can reside in the switches of a multistage omega network, which resembles very much the MIN-based scheme [10]. However, the viability of such schemes relies on several factors: First, the hierarchical locality is the key point in making HFMD efficient. Compiling optimization techniques are required to achieve such locality. Second, false sharing seems to have a large impact on the HFMD so that techniques to eliminate it are vital. Third, the performance of HFMD is likely to be sensitive to the cache configuration, thus special care must be taken to choose the cache size and the block size.

On the other hand, the tree-based directory is a reasonable extension from the linked-list chained directory. In fact, it is, to a certain degree, a mixture of the limited directory and a tree structure. Therefore, it is not surprising that it inherits the lower performance. It apparently suffers penalties when the limited pointers run out and the tree must grow. Initial performance study in [15] shows that although in some cases the tree-based scheme outperforms the chained directory, the tree-based directory does not perform so well as the full-map directory. The advantage of the tree-based scheme is that the storage requirement by the tree-based directory is very small comparing to the full-map directory. So, in some sense, the tree-based directory scheme offers a trade-off between the performance and the memory overhead (i.e., allowing for very efficient storage at a reasonable performance level).

The SCI extension is a good step towards more efficient directory-based protocols. However, the extension is restrictively based on the SCI base protocol. Compatibility is strongly enforced. Innovative ideas are not easy to adapt into the base protocol. The extension becomes more and more complicated in order to obtain the logarithmic steps of invalidations from the SCI base protocol. Apparently, many more refinements need to be made; in particular, the operations on dynamic trees remain to be optimized and simplified.

We are aware of the notions of home cluster and owner cluster in the DASH prototype. Our notion of primary-node is totally different from them since, first, we introduce a new cache coherence scheme with the characteristics of primary accesses on one processor. In contrast, DASH prototype uses a full-map directory scheme. Second, the home cluster there is not equal to our primary node, not because the home cluster is a collection of a number of processors, but because the home cluster refers to the cluster which contains the main memory and directory for a given physical memory address, while the primary node means the node on which most accesses to a given memory block are performed and by which the coherent activities with respect to the memory block are coordinated; the latter may not contain the main memory.

The scheme closest to our scheme is dynamic pointer allocation. Actually we borrow the idea of *dynamic allocation* from them. However, our allocation strategy differs from theirs in being from a local node (private memory) rather than from only one commonly accessible pool. The allocated secondary pointers in our scheme are mainly for the new protocol, although it has the same effect in extending limited pointers. Moreover, our proposed scheme is distinct from all other existing directory schemes in many ways:

First, we utilize the characteristic of major accesses on the primary node and have a designated node accomplish the job. The coherent operations induced from the primary node can be done locally. The fraction of accesses from other nodes is believed very small so that the coherent operations of that part are allowed to be a little more complicated, compared to those on the primary node. Even so, their costs are comparable to the traditional full-map directory (as discussed in the next section). Therefore, the total system overhead related to the coherent activities in our scheme is reduced dramatically.

Second, from the storage point of view, our scheme consumes almost the same directory space as the chained directory. However, since our primary node maintains the linked list of its secondary pointers on its local node, we do not have to traverse the linked list from one cache to another, thus, we eliminate the weakest point in chained directory schemes.

Third, our scheme can still enjoy the benefit of the conventional limited directory, i.e. only a few pointers are usually needed at a given time. So managing our secondary pointers is not difficult.

4.2 Possible Enhancements

Some enhancements can be made to improve the performance of the scheme further by optimizing the coherence activities among primary nodes and secondary nodes. For instance, rather than using a sequential consistency model, we can extend our scheme with some more relaxed memory consistency models (such as the release consistency model [8]) to improve the performance.

Having introduced the notion of the primary node may influence cache replacement policy. Primary copies are hopefully being kept in caches as long as possible, but they may block other secondary or primary copies from being cached. New replacement policies should consider the overall cost of replacements and somehow predict the future accesses. In addition to the factors such as the LRU (Least Recently Used) policy, the cost should take into account other factors such as whether the copy is the primary, and if so, how many pointers will be invalidated.

Software (compiler) may be used to assist the determination of the primary node. For example, at compile time, analysis can be made from parallel programs about the possible data access patterns and behavior so that information can be provided to estimate which node can be the potential primary node. It may be very hard to estimate, but possibly heuristic means can be used to help the determination so that it can reduce chances of primary-node switches. This should be included in future research.

In summary, the primary-node scheme provides a simple, efficient and scalable means to solve the cache coherence problem, compared to other existing directory schemes. Possible enhancements can be furtherly made to improve its performance.

5 Performance Analysis

A good and economical evaluation would be probabilistic analysis with a more realistic model. However, it is very difficult to develop such a model within a short time. A better approach is to use trace-driven (or execution-driven) simulations, which require a lot computing power for very large size systems — we are currently undertaking the methodology of hybrid simulations (trace-driven and/or execution-driven) to speed up simulations.

In the following we attempt to make a preliminary analysis estimating the cost of the coherent actions of our scheme as compared to the full-map directory scheme, which is considered the most efficient existing directory scheme.

As we said in section 2, we considered a general MIMD shared-memory architecture. Table 1 gives an example with the default latencies of primitive operations in processor clock cycles (due to [20]). On the basis of such a model, coherent operation costs can be estimated by following and tracing the protocols case by case and step by step. For example, in the case of *write hit with a shared-clean state but without write permission*, the full-map protocol first looks at the local cache (with the cost of T_{cache}), and then issues a write request to the memory module containing the block (with a cost of $T_{net.cmd} + T_{mem}$). The memory module issues invalidation requests to the sharing caches (the cost is $T_{net.cmd}$) and waits for the acknowledgements (ACKs) from them. Those sharing caches receive the invalidation requests, set appropriate bits and prepare for the ACKs (with a cost of T_{ack}), and send the ACKs back to the memory module (another cost of $T_{net.cmd}$). The memory module receives the ACKs and sets the dirty bit, clears the presence bits for previous sharing caches (the cost is T_{mem}) and sends the write permission to the requesting cache (another cost of $T_{net.cmd}$). The requesting cache receives the write permission message, updates the state in the cache (at the cost of $_{cache}$) and the operation is done. Thus the total cost for this operation is $T_{cache} + T_{net.cmd} + T_{mem} + T_{net.cmd} + T_{ack} + T_{net.cmd} + T_{mem} + T_{net.cmd} + T_{cache}$, which is $1 + 10 + 8 + 10 + 2 + 10 + 8 + 10 + 1 = 60$ pclocks.

Similarly, other operations of the full-map directory and all operations of the primary-node scheme can be estimated. Table 2 shows our results of various read and write costs of the full-map directory scheme and the primary-node scheme.

Table 1: **Latencies for primitive operations in processor clock cycles**

Primitive Operation	**Prameter**	**Default Value**
Cache Access Time	T_{cache}	1 pclock
Miss Detection, Cache Fill, Proc. Restart	T_{proc}	2 pclocks
Network Latency for a Req. and Ack.	$T_{net.cmd}$	10 pclocks
Network Latency for Data	$T_{net.data}$	14 pclocks
Local Bus Request Time	$T_{bus.cmd}$	2 pclocks
Local Bus Reply Time	$T_{bus.data}$	1 pclock
Memory	T_{mem}	8 pclocks
Time to Process an Ack.	T_{ack}	2 pclocks

Table 2: **Results of latencies of operations with two directory schemes**

Operation	**Cases**	**Full-map**	**Primary-node**	
			From the primary	From a secondary
Read	with a cache hit	1 pclock	1 pclock	1 pclock
	with a miss (uncached)	46 pclocks	35 pclocks	53 pclocks
	with an invalid state	45 pclocks	— pclocks	26 pclocks
Write	with a hit & write permit	1 pclocks	1 pclocks	16 pclocks
	with a hit (shared $\rightarrow$ dirty)	60 pclocks	3 pclocks	62 pclocks
	with a miss (uncached)	61 pclocks	30 pclocks	63 pclocks
	with an invalid state	60 pclocks	— pclocks	62 pclocks

From Table 2, we can see that most coherent operations of the primary scheme have lower latencies than the full-map scheme. In particular, the operation costs from the primary node are significantly less expensive [3]. The cases with an invalid state do not happen at all on the primary node because the primary copy has the write permission all the time, no matter where the write is induced. So, such cases in the full-map directory scheme turn out to be a hit on the primary node in the primary-node scheme, definitely a performance gain.

There are two cases that the cost on a secondary node has a larger value than in the full-map scheme. One is *read with an uncached miss*. The reason is that it involves the interactions between the secondary node and the primary node; the primary node has to allocate a new pointer from the free pointer pool and connect it to the linked list maintained by the primary copy. The other operation is a *write hit with the write permission* on the secondary node. Usually, in the full-map scheme, it is just an 1 pclock operation, while in the primary-node scheme, the primary node wants to co-own the write permission with the requesting secondary node, so one more interaction with the primary node is added to the cost.

Other operations on the secondary nodes such as *write with a hit and write permission*, *write with a hit from shared-clean to dirty* and *write with an invalid state* have almost the same cost as the corresponding cases in the full-map protocol (with a little bit of overhead). This can be explained by the fact that the write operation performs similarly to the full-map scheme except that the primary node may have to be consulted instead of the memory module. The primary node has to look up its linked list of secondary nodes, which increases the cost of accessing the memory module, making the total cost of the operation almost the same but a little bit higher.

In summary, the primary-node scheme achieves its performance gains from data imbalance accesses favorable towards the primary node by making the coherent operations on the primary nodes simple and efficient, while coherent operations induced from the secondary nodes are comparable with those of the full-map directory schemes. The more accesses on the primary nodes, the more advantageous the primary-node scheme. Detailed cost estimation shows that, compared to the full-map directory scheme, the primary-node protocol is simple, efficient and scalable.

6 Conclusion and Future Work

In this paper, we presented the primary-node scheme for multiprocessor cache coherency and addressed the primary node assignment problem. We discussed some possible enhancements to our initial primary-node scheme proposal. We also gave a preliminary performance analysis of the proposed scheme.

As has been seen, our scheme is simple, efficient, and scalable for building large scale shared-memory multiprocessors. In addition, it requires as small memory overhead as for chained directory schemes. In contrast to the high latencies of chained directories which enforce sequential invalidations through the linked-list structure connected from one node to another, the primary-node scheme records locations of secondary nodes in its local store, a central source easy to manage.

There are many issues that are needed to be explored in the future pertaining to the primary-node scheme. One straightforward extension to our scheme is to employ a more relaxed memory consistency model instead of the sequential consistency model. Performance gain can be achieved by

[3] Notice that the 3 pclocks on the primary node with a write hit (from shared-clean to dirty) is an approximate average number. Recall that the primary node holds write permission not only of its own, but also for any writes from secondary nodes. We estimate 90% of write accesses take place on the primary node — for 80% estimation, the cost will be approximately 5 pclocks.

using a relaxed consistency model. Another improvement could be done by designing a new primary-node determination algorithm so that the chances of primary-node switching can be minimized.

Performance evaluation of our scheme compared to other existing directory schemes is underway using a hybrid simulation methodology. A more ambitious simulation plan is to include those software-based schemes. We intend to simulate them by integrating our currently-being-built cache simulator with a network simulator [14] and/or a TLB simulator [19] and a low-level architectural simulator [3]. Much more detailed and more realistic simulations are planned to be done. We expect the simulation results will match our estimations made in the last section.

Other future work with respect to the primary scheme includes furtherly optimizing the scheme and implementing it in a real multiprocessor system, such as the NYU Ultracomputers[11].

Acknowledgement: The randomization technique and the primary-node dynamic determination was suggested by Malcolm Harrison. Thanks for his email discussion with me. Thanks also go to Allan Gottlieb and Susan Dickey for their reading the original manuscript and providing many useful comments.

References

[1] S.V. Adve and M.D. Hill. Weak ordering–a new definition. In *Proceedings of the 17th Annual International Symposium of Computer Architecture*, pages pp2–14, May 1990.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium of Computer Architecture*, pages 280–289, 1988.

[3] AMD. *Am29050 Microprocessor User's Manual.* Advance Micro Devices, 1991.

[4] James K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, University of Washington, Seattle WA USA, November 1986.

[5] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27(12):pp1112–1118, Dec. 1978.

[6] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):pp49–58, June 1990.

[7] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Systems*, pages P224 – 234, April 1991.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium of Computer Architecture*, pages 15–26. IEEE Computer Society Press, 1990.

[9] Kourosh Gharachorloo, Anoop Gupta, and John Hennesy. Performance evaluation of memory consistency models for shared memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Systems*, pages P245 – 257, April 1991.

[10] Kanad Ghose and Sreenivas Simhadri. A cache coherency mechanism with limited combining capacities for MIN-based multiprocessors. In *1991 International Conference on Parallel Processing*, pages I–296 – I–300, August 1991.

[11] Allan Gottlieb, Ralph Grishman, Clyde Kruskal, Kevin McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February, 1983.

[12] SCI Extension Working Group. Scalable coherence extensions to SCI. Draft 0.20 IEEE Standard P1596.2/Draft, August 1991.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium of Computer Architecture*, pages 148–159, May 1990.

[14] Yue-Sheng Liu. *Architecture and Performance of Processor-Memory Interconnection Network for MIMD Shared Memory Parallel Processing Systems.* PhD thesis, Courant Inst., New York University, September 1990.

[15] Yeong-Chang Maa, Dhiraj K. Pradhan, and Domminique Thiebaut. Two economical directory schemes for large-scale cache coherent multiprocessors. *Computer Architecture News, SIGARCH*, 19(4), September 1991.

[16] Richard Simoni and Mark Horowitz. Dynamic pointer allocation for scalable cache directories. In *Inter. Symp. on Shared Memory Multiprocessing*, April 1991.

[17] Richard Simoni and Mark Horowitz. Modeling the performance of limited pointers directories for cache coherence. In *Proceedings of the 18th Annual International Symposium of Computer Architecture*, pages P309 – 318, May 1991.

[18] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):pp12–24, June 1990.

[19] Patricia J. Teller and Allan Gottlieb. TLB performance in multiprocessors. UCN 173, Ultracomputer Lab, New York University, September 1991.

[20] xxx, yyy, and zzz. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. 1992. To appear.