The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors

Håkan Nilsson and Per Stenström

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 Lund, Sweden

Abstract

The problem of cache coherence in large-scale shared memory multiprocessors has been addressed using directory-schemes. Two problems arise when the number of processors increases; the network latency increases and the implementation cost must be kept acceptable.

In this paper we present a tree-based cache coherence protocol called the Scalable Tree Protocol (STP). We show that it can be implemented at a reasonable implementation cost and that the write latency is logarithmic to the size of the sharing set. How to maintain an optimal tree structure and how to handle replacements efficiently are critical issues we address for this type of protocol. Finally we compare the performance of the STP with the SCI (IEEE standard P1596) by considering a classical matrix-oriented algorithm targeted for large-scale parallel processing. We especially show that the STP manages to reduce the execution time considerably by reducing the write latency.

1 Introduction

Shared memory multiprocessors offer a flexible and powerful programming model. However, scaling such computers to a large number of processors has shown to be difficult mainly due to the contention and the latency associated with their memory systems. A unified approach to cope with these problems has been to use caches in conjunction with a directory-based cache coherence protocol implemented in hardware [9].

The objective of directory-based cache coherence protocols is to reduce memory system contention by exclusively sending point-to-point messages to those caches that share a copy of one memory block. This is achieved by associating the same number of cache pointers as the number of caches with each directory entry. In *full-map directory schemes* [3, 8] the cache pointers are represented by a bit vector containing N bits, where N is the number of caches. Since one bit vector is associated with each memory block, the resulting implementation cost for the directory is unacceptable for multiprocessors containing several hundreds of caches. Consequently, researchers have explored other approaches to reduce the implementation cost of the directories.

One approach to reduce the implementation cost is to use a limited number of cache pointers per memory block. Since each pointer requires $\log_2 N$ bits, the memory overhead for each memory block is $i \log_2 N$ bits, assuming i cache pointers. For applications where the number of processors that share data, the *sharing set*, is less or equal to i, the performance of such *limited-directory schemes* is identical to full-map directories. When the sharing set exceeds the available number of cache pointers, three strategies have been proposed. For Dir_iNB schemes [1], cache-pointer replacements take place when the size of the sharing set exceeds i. As a result, the latency is increased because of an increased cache-miss rate. Dir_iB schemes [1] broadcast consistency messages when the directory runs out of pointers for the memory block, resulting in excessive network traffic which increases contention. LimitLESS [4] emulates a full-map directory in software when the directory runs out of pointers which reduces processor utilization. In summary, we note that for applications where the sharing set is large compared to the available number of cache pointers, limited-directory schemes result in poor performance in one way or the other.

To reduce performance degradation for applications with large sharing sets, the challenge is to arrange the directory in such a way that (i) it has a sufficient number of cache pointers, (ii) it has an acceptable implementation cost, and (iii) it handles consistency actions in an efficient way. The IEEE P1596, known as the Scalable Coherent Interface (SCI) [6], is a promising

example of such a protocol. The SCI associates two cache pointers with each cache line. All caches sharing a memory block form a double-linked list. Thus, only $2\log_2 N$ bits are associated with each cache block. Apparently, it never runs out of cache pointers and it offers an acceptable implementation cost for large-scale multiprocessors. Unfortunately, however, the SCI incurs a severe performance penalty in handling write operations. Upon a write request to a memory block cached in n caches, the invalidation or update message has to traverse the list of caches. Thus, the latency incurred by write operations is $O(n)$. Consequently, the write latency can have a detrimental effect on the performance of parallel applications with large sharing sets.

In this paper we present the design and evaluation of a new directory-based cache coherence protocol, called the Scalable Tree Protocol (STP). The STP arranges the caches in the sharing set of a memory block in a tree structure. Like the SCI, the STP never runs out of cache pointers. Although the STP has a slightly higher implementation cost than the SCI, it only associates $(3+K)\log_2 N$ bits with each cache block, where K is the fan-out of the tree. Thanks to the fact that the sharing set is arranged in a tree-structure the write latency is $O(\log_K n)$ instead of $O(n)$. A critical issue in the design of tree-based cache coherence protocols is how to reduce read and write latencies. The design of the STP shows that it is possible to achieve a small read latency by exploiting parallelism in the algorithm that establishes the tree structure. Moreover, our protocol guarantees a logarithmic write latency by always maintaining an optimal tree structure.

The organization of the rest of the paper is as follows. In Section 2, as a background, we present the organization and the coherence protocol of the Scalable Coherent Interface. We use an example parallel application to build intuition into the performance issues introduced by the linear-list approach taken by the SCI. In Section 3, we present the STP coherence protocol. We especially focus on how the protocol maintains an optimal tree structure and how replacements are handled. We compare the implementation cost and the performance of the SCI and the STP in Section 4. Finally, in Section 5, we conclude the results.

2 Background

The motivation behind our work is to design a cache coherence protocol that achieves a high performance and an acceptable implementation cost for multiprocessors with several hundreds to thousands of processors targeted to applications with large sharing sets.

The IEEE P1596 standard, known as the SCI [6], has similar objectives. The linear-list structure results in an acceptable implementation cost. Unfortunately, the linear-list approach results in performance degradation due to the write latency that is incurred by the list structure. In this section, we begin with reviewing the cache coherence protocol of the SCI in Section 2.1. Then we illustrate how the write latency can impact severely on the performance of applications with large sharing sets by considering a classical matrix-oriented algorithm for parallel processing in Section 2.2.

2.1 The SCI – A linear-list protocol

Linear-list protocols, such as the SCI, maintain the sharing set in a linear list as shown in Figure 1. The basic mechanism of the SCI protocol is two pointers that are associated with each cache line. They point at the predecessor and the successor cache in the list. Moreover, one pointer is associated with each memory block to point at the head of the list. We now review the consistency actions associated with the SCI protocol.

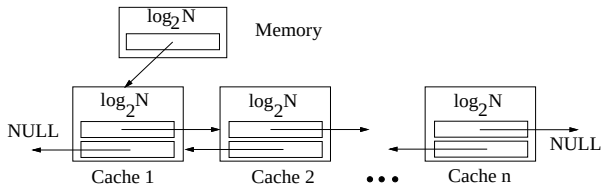

Figure 1: n caches sharing a copy of a memory block according to the SCI protocol.

A cache that reads a block will be linked into the list as the new head of the list in the following way. The reading cache sends a read request to the memory. If the list is empty, the memory establishes a pointer to the reading cache and supplies the cache with data. Otherwise, the memory returns the pointer to the old head of the list and modifies its pointer to point at the reading cache. The reading cache updates its successor pointer to point at the old head and sends a new request to the old head of the list. The old head returns the requested block and updates its predecessor pointer to point at the reading cache, which now is the new head.

The SCI protocol maintains consistency by a write-invalidate protocol, i.e. invalidation messages are sent

to all caches belonging to the sharing set. Consequently, if the cache associated with the writing processor has the only copy in the system, the write operation can proceed locally. Otherwise, the following actions take place. The cache associated with the writing processor is taken out of the list and is put as the head of the list. Then it sends an invalidation message to its successor. When a cache gets an invalidation message, it removes itself from the list and returns the identity of its successor to its predecessor. When all caches are removed from the list, the write operation is completed. Since the invalidation must traverse n caches, given that n caches share the same block, the time for a write operation to be completed is $O(n)$.

Possibly, a cache block needs to be replaced when handling a cache miss. The replacement operation is efficiently supported by the double-linked list. The cache that replaces a shared block sends the identifier of its predecessor to its successor and the identifier of its successor to its predecessor. The cache is now removed from the list.

To what extent the write latency impacts the performance of a parallel algorithm depends on whether the architecture allows multiple outstanding memory requests. The write latency can be overlapped by other write operations and computation provided that the application relies on a relaxed memory consistency model such as the *weakly ordered model* proposed by Dubois et al. in [5]. Under the weakly ordered model shared memory operations may overlap as long as all outstanding memory operations have been completed at the synchronization points. The implication of this is that if the distance between two synchronizations is small, the time for a write operation to be completed may have a significant impact on the execution time. We shall substantiate this issue in the next section.

2.2 An example parallel algorithm

In this section we review a classical generic paradigm to implement parallel algorithms for the solution of a linear system of N equations by iterations. For simplicity, we omit the convergence criterion and focus on the shared read-write memory references generated by the algorithm. This particular generic algorithm is chosen to illustrate the problem of coherence actions to data structures with large sharing sets. Our purpose is to address the problem of how to support such sharing behaviors for large-scale multiprocessors.

Consider the following set of linear equations:

$$\overline{x}_{i+1} = \tilde{A}\overline{x}_i + \overline{b}$$

where $\overline{x}_{i+1}, \overline{x}_i$, and $\overline{b}$ are vectors of size N and $\tilde{A}$ is a matrix of size $N \times N$. Suppose that each iteration (the calculation of vector $\overline{x}_{i+1}$) is performed by N processors, where each processor calculates one vector element. The code for one iteration of the algorithm is shown in Figure 2.

```
par_for J := 1 to N do
begin
    XTEMP[ J ] := B[ J ];
    for K := 1 to N do
        XTEMP[ J ] := XTEMP[ J ] +
            A[ J,K ] * X[ K ];  — read(X[ K ])
end;
barrier_sync;
par_for J := 1 to N do
    X[ J ] := XTEMP[ J ];      — write(X[ J ])
barrier_sync;
```

Figure 2: An example parallel algorithm for an iterative solution of a linear system of equations.

N processes are initiated by the **par_for** statement. Each process calculates a new value which is stored in XTEMP. The last parallel loop in the iteration copies back the elements of XTEMP to vector X. This requires a barrier synchronization.

In order to understand the impact of cache coherence actions on the execution time for the algorithm, we specifically consider the read and write operations to vector X. These are marked in Figure 2. For simplicity reasons, we assume that the block size is exactly one vector element which thus eliminates the issue of false sharing.

Now recall the consistency actions taken by the SCI protocol reviewed in the previous section. The first parallel for-loop creates a list of size N for each vector element. The second parallel for-loop purges (invalidates) all lists. Under the weakly ordered model, which is one of the most relaxed memory consistency models, the correctness of the above program requires that all write operations from a processor have been completed before the processor can access a synchronization variable. The implications of this is that the time to execute the above program depends on the write-latency time. The write-latency time is $O(N)$ in the SCI protocol. As a result, the execution time will grow with the size of the sharing set. In the next section, we consider the cache coherence protocol of the Scalable Tree Protocol. By arranging the caches belonging to the sharing set in a tree, instead of a lin-

ear list, we will show that the write latencies can be logarithmic to the size of the sharing set.

3 The consistency actions of the Scalable Tree Protocol

In this section, we present the coherence actions of the Scalable Tree Protocol, the STP, and the support mechanisms associated with it. We begin with describing the necessary pointers needed to maintain an optimal tree structure in Section 3.1. In Section 3.2–3.4, we describe the coherence actions taken on processor read and write operations, and block replacements. We especially show that (i) the read latency is better in the STP than in the SCI by exploiting parallelism in the coherence algorithm, (ii) the write latency is logarithmic by always maintaining an optimal tree, and (iii) replacements of blocks associated with caches in the middle of the tree are handled in a constant time.

3.1 Main structure and state memory

The Scalable Tree Protocol maintains an optimal tree structure for all caches that share a memory block. The fan-out of the tree is K, i.e. each node has K pointers to its subtrees, see Figure 3. A write operation can be implemented by an invalidation-based or update-based protocol. We describe the coherence actions required for an invalidation-based protocol.

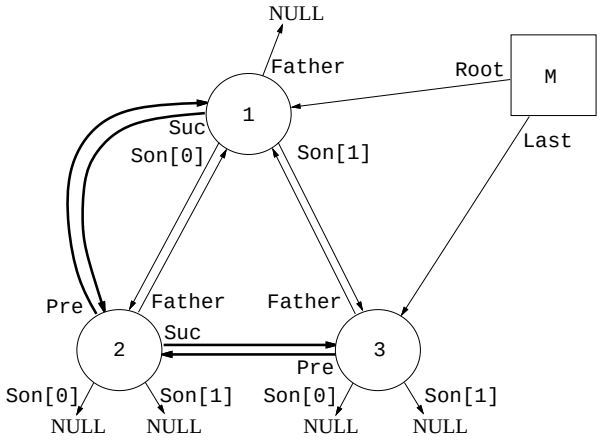

Figure 3: Three caches have read a memory block and all the pointers are set correctly.

We show in the subsequent sections that the protocol always guarantees an *optimal* tree. Level i in an optimal tree is completely filled before caches are inserted at level $i+1$. A new cache that reads a shared memory block is inserted in the tree as a leaf at the lowest level. We also show that the optimal tree structure is consistent even if a cache in the middle of the tree performs a block replacement.

To get a feeling for the overhead in implementation complexity of the STP, we briefly describe the different pointers associated with a memory block and a cache line, see Figure 3. The pointers are more extensively explained when the different operations are described in subsequent sections. Each memory block uses three pointers called Root, Last and WritePending. Root points at the root of the tree, Last points at the cache that fetched the memory block most recently, and WritePending (not shown in Figure 3) points at the cache with a pending write request. Since each pointer is $\log_2 N$ bits, the overhead for a memory block is:

$$3 \log_2 N \quad bits \tag{1}$$

Each cache line uses $3 + K$ pointers, called Father, Son[0..K-1], Pre, and Suc, to maintain the tree structure. Father points at the father to the cache. Son[0..K-1] point at the K subtrees of the cache. Pre and Suc point at the caches that fetched the shared memory block before and after the cache, respectively. Thus, the state memory overhead for a cache line is:

$$(3 + K) \log_2 N \quad bits \tag{2}$$

In addition there is a small constant number of state bits associated with each memory block and each cache line[1].

We use the notation according to Table 1 to distinguish between different caches and memory modules involved in a consistency operation. This notation is used throughout the paper.

In Figures 3–6 we assume a tree with fan-out K = 2, but the algorithms work correctly for any fan-out. The digits inside the nodes denote the order in which the caches have fetched the block from the memory.

The digits on the message arrows denote in which order the messages are sent. For an example, look at Figure 5. Three caches have read the shared memory block from the memory and cache number two issues a write request. The messages WriteReq, CheckLast and LastOk are sent in sequence. When invalidations start to propagate in the tree, they can be sent in parallel as shown by the invalidation messages sent by the root node.

A cache line containing the shared memory block is called a node. Figure 3 shows a tree when three caches have read the block and all pointers are set correctly.

[1] Such state bits are needed in all cache coherence protocols.

Notation	Description
C_{rd}	The cache performing a global read.
C_w	The cache performing a global write.
C_{rm}	The cache performing a block replacement.
C_L	The cache that fetched the memory block most recently.
C_F	The cache which becomes father to the next cache issuing a global read.
C_R	The cache in the top of the tree, the root.
M	The memory containing the shared block.

Table 1: The notations used for caches involved in a coherence operation.

The Pre and Suc pointers form a linear list, and the Father and Son pointers create a tree. The linear list is used during block replacement to make sure that the tree is still optimal after the replacement operation.

3.2 Consistency actions for global read operations

When a cache reads a shared memory block, two situations are possible. First, the memory has a valid copy of the block and can satisfy the read request at once. Second, another cache has an exclusive copy of the block and the memory has to be updated before it can satisfy the read request. In the first situation, there is two different cases when serving a read request: (1) no cache has read the memory block and (2) at least one cache has already read the block.

We have followed two guide lines when we designed the protocol for read operations. First, we want to keep the tree optimal and second, we want the memory to send data to C_{rd} (see Table 1), to achieve that the processor gets the data and continues as soon as possible.

Case 1: C_{rd} sends a ReadReq message to the memory, M. M responds with the message Data which contains the requested block. C_{rd} is C_R, C_F, and C_L after the operation.

Case 2: Figure 4 shows which messages are sent when the seventh cache reads the shared memory block. When C_{rd} sends a ReadReq message (1) to M, M responds with a Data message (2) to C_{rd}. The Data message contains a copy of the memory block and the identity of C_L. Note that the processor can continue after the data transaction. In other words, all actions taken to link the cache into the tree can be

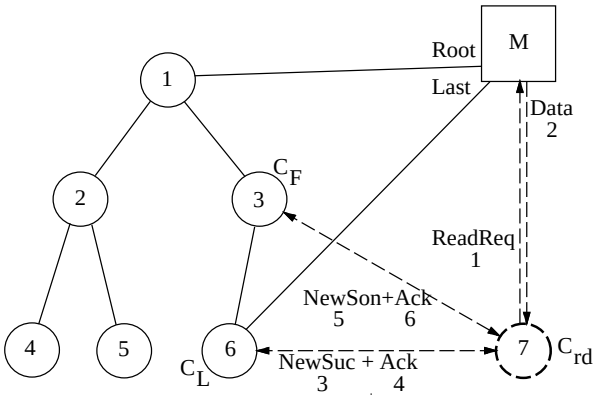

Figure 4: The seventh cache reads the memory block.

overlapped by local computation.

Next we consider the actions taken to link the cache into the tree. C_{rd} will later become the new C_L. C_{rd} is inserted as son of C_F and successor of C_L in the following way. C_{rd} first sends a message, NewSuc (2), to C_L and sets its Pre pointer to point at C_L. C_L receives NewSuc, sets its Suc pointer to point at C_{rd} and sends Ack (4) to C_{rd}. The identity of C_F is stored in C_L and Ack contains the identity of C_F. Then C_{rd} sends a new message, NewSon (5), to C_F and sets its Father pointer to point at C_F. C_F inserts C_{rd} as a subtree and sends Ack (6) back to C_{rd}. The next cache reading the block becomes another subtree to C_F. If C_F can not have another subtree, the next cache is inserted as a subtree to the successor of C_F. In Figure 4 is the next reading cache inserted under cache number 4.

We keep the tree optimal by placing a maximal number of subtrees under a node before we place subtrees under another node. The next node to store subtrees is the successor of the node we just filled.

A list of caches waiting to link themselves into the tree can occur, since the linking is done one cache at the time. To speed up the tree creation time, a LinkIn message is sent from C_F to the cache waiting after C_{rd}. The LinkIn message contains the identity of the new C_F. Without the LinkIn messages the tree creation takes $O(2n)$ time, given n caches waiting to link themselves into the tree. The tree creation takes only $O(n)$ time when using the LinkIn messages.

3.3 Consistency actions for global write operations

The instantiation of the tree-based protocol we describe maintains consistency using an invalidation-based protocol under the weakly ordered model [5].

A write request invalidates all copies of the shared memory block. Write requests by the same processor can be pipelined. However, at a synchronization point the processor stalls until all outstanding write requests have been acknowledged. Figure 5 shows how cache number 2 issues a global write request and how the write operation is performed.

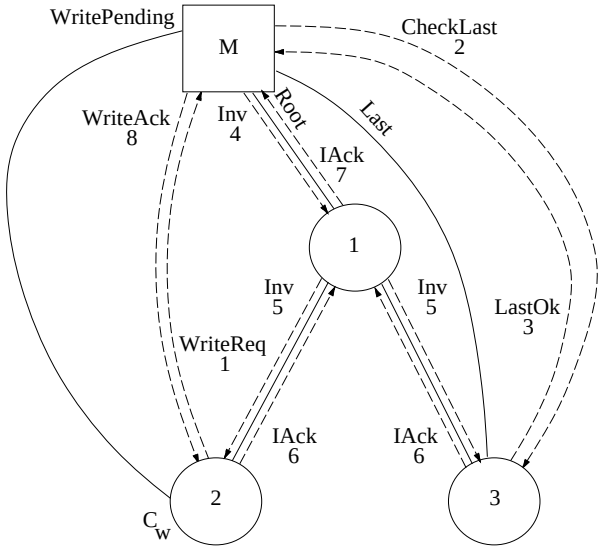

Figure 5: A general view of a global write operation.

The write operation starts with a WriteReq message (1) from C_w to M, see Figure 5. M receives the WriteReq, stores the identity of C_w in the WritePending pointer in M and sends a CheckLast message (2) to C_L. Since C_L read the memory block most recently, it must be linked into the tree before the invalidation messages start to propagate in the tree. C_L responds with a LastOk message (3) to M. If C_L is linked into the tree, it sends LastOk at once. Otherwise, C_L waits until it is linked into the tree before it sends the LastOk message.

When M receives LastOk, it starts to invalidate all copies of the block. This is shown in Figure 5 by an Inv message (4) sent from M. When a cache receives an Inv message, it first looks whether it has any subtrees. If the cache has subtrees, it sends an Inv message (5) to each of them. Then the cache waits for an IAck message (6) from each subtree. When the cache has received IAck from all its subtrees, it invalidates its own copy of the block, and sends an IAck message (7) upwards to its father. If the cache does not have any subtrees, it invalidates its copy of the block and sends an IAck message to its father at once.

When M receives an IAck message, it knows that the whole tree is invalidated and can then send a WriteAck message (8) to C_w. C_w is the only cache in the tree and is both C_L and C_R. When C_w receives the WriteAck message it knows that the write is globally performed, and C_w has the only valid copy of the memory block. Subsequent writes by C_w can be performed locally until another cache reads the block.

3.4 Consistency actions for block replacements

The design of the consistency actions taken on a block replacement has followed two guide lines. First, we want to keep the tree optimal even if a cache in the middle of the tree performs the replacement. This is done by moving C_L to the place in the tree where C_{rm} (the cache that replaces its copy) is. Second, we want a constant replacement time. New global read and write operations are not allowed to be served during the block replacement. The motivation is that the pointers in the tree may be inconsistent during the operation. We have three cases:

1. There is only one cache in the tree.
2. There are at least two caches in the tree and C_L does the replacement.
3. There are at least two caches in the tree and a random cache, not C_L, does the replacement.

Case 1: There are two situations when only one cache has a copy of the shared memory block. First, the only copy of the block is consistent with the block in memory. Second, the cache has an exclusive copy which is inconsistent with the memory. In the first situation C_{rm} only needs to notify M that C_{rm} has invalidated its copy and the only copy is in M. In the second situation C_{rm} must copy the block back to M before it is purged from the cache. When M confirms the update, C_{rm} invalidates its copy of the block.

Case 2: There are at least two caches in the tree and C_L does a block replacement, see Figure 6. The operation starts with a ReplaceReq message from C_{rm} to M. M responds with a ReplacePermission message to C_{rm} and disallows new global read and write operations until the block replacement is finished.

When ReplacePermission arrives to C_{rm}, C_{rm} sends two messages in parallel. The first, SetLast, is sent to the cache pointed at by C_{rm}'s Pre pointer. This cache responds with an Ack message and becomes the new C_L. The second, RemoveSon, is sent to the father to C_{rm}. The father responds with an Ack message and becomes the new C_F.

When C_{rm} has received both the Ack messages, C_{rm} is allowed to use the cache line for a new memory

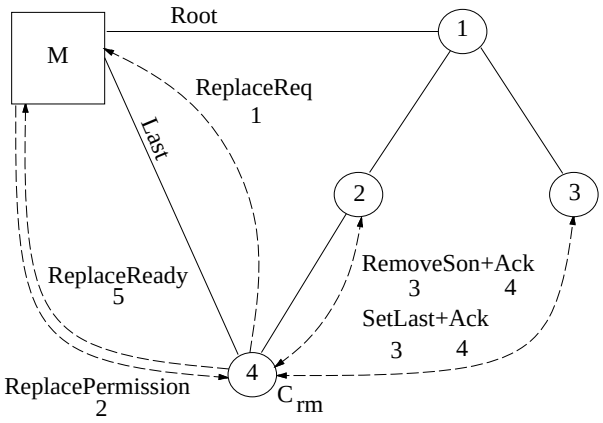

Figure 6: The Last cache, C_L, in the tree performs a block replacement.

block. At the same time, C_{rm} sends a ReplaceReady message to M. The ReplaceReady message contains the identity of the new C_L. New global read and write operations to the block are permitted when M receives the ReplaceReady message.

Case 3: There are at least two caches in the tree and a random cache, not C_L, does the replacement, see Figure 7. C_{rm} sends a ReplaceReq message to M. ReplaceReq contains the state memory (the pointers) of the cache line in C_{rm}. M responds with a ReplacePermission message to C_{rm} and disallows new global read and write operations until the block replacement is finished. C_{rm} is allowed to use the cache line for a new memory block when it receives ReplacePermission. When M sends ReplacePermission, it sends a Move message to C_L. The Move message means that C_L moves itself to the place in the tree where C_{rm} is.

On a move, C_L first removes itself from its place in the tree the same way as in case 2. C_L then informs all the caches that surround C_{rm} in the tree so they know they shall point at C_L instead of C_{rm}. The caches informed are all the sons of C_{rm}, C_{rm}'s Father, and the caches pointed at by the Pre and Suc pointers. Each of these caches sends an acknowledgement to inform C_L when all the pointers are established. C_L then sends a ReplaceReady message to M. M knows by this message that the replacement is done and new global read and write operations to the memory block are allowed.

4 Comparison of the STP and the SCI

In this section we analyze the implementation cost and the performance of a tree-based protocol, the STP,

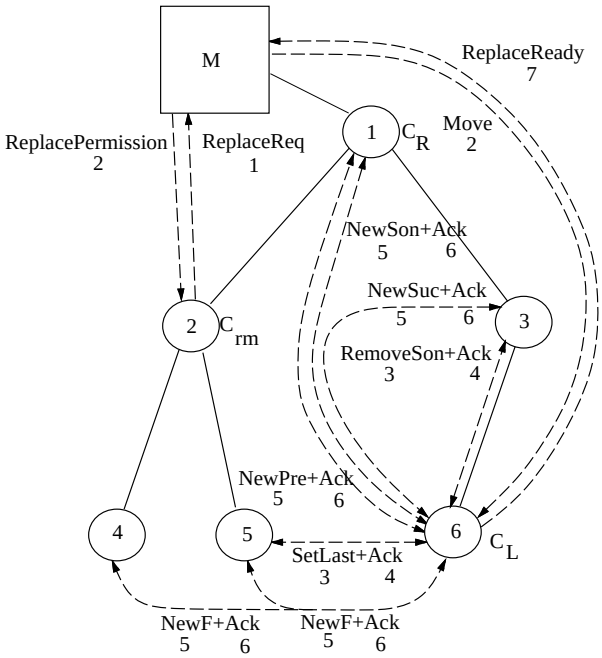

Figure 7: Node 2, in the middle of the tree, performs a block replacement.

and a linear-list protocol, the SCI. First, we investigate the implementation cost for the protocols. And second, we give performance results for the protocols in terms of execution time for the example algorithm described in Section 2. We also discuss the network traffic the protocols generate.

4.1 Implementation cost

In this section we discuss the state memory overhead in the STP for each cache line and for each memory block. We also compare the state memory overhead for a cache line in the STP, with the state memory overhead for a cache line in the SCI.

The state memory overhead for a cache line, relative to the size of the cache line, is found in Table 2. In the table we assume N = 1024 caches. The size of the state memory for a *memory block* is found in Equation 1 and the size of the state memory for a *cache line* is found in Equation 2, see Section 3.1.

The SCI has two pointers associated with each cache line. The pointers are 16 bits long in the IEEE standard, but here we assume that they are $\log_2 N$ bits long. The overhead for a cache line, relative to the size of the cache line, is found in Table 2. We notice that the SCI has a lower implementation cost than the STP, but the implementation cost for the

Block size (bytes)	SCI	STP		
		K = 2	K = 3	K = 4
8	0.31	0.78	0.94	1.09
16	0.16	0.39	0.47	0.55
32	0.08	0.20	0.23	0.27
64	0.04	0.10	0.12	0.14

Table 2: The state memory overhead for a cache line, assuming N = 1024.

STP seems also quite acceptable. For example, assuming 32 bytes memory blocks the SCI results in 8% state memory overhead and the STP results in 20% overhead if $K = 2$. The IEEE standard specifies the block size to 64 bytes.

4.2 Performance comparison

In this section we evaluate the performance of the STP and a linear-list protocol through program-driven simulation. The linear-list protocol we simulate is very similar to the SCI. The only difference is that in our linear-list protocol the memory is kept up-to-date if more than one copy exists. In the SCI the memory is only updated when no cache has a copy of the memory block. The read latency is reduced with 50% if the memory is kept up-to-date (further explained in [7]).

We use the algorithm from Section 2 as a workload, see Figure 2. The motivation for this choice is that the algorithm stresses the efficiency of the protocol implementations in several respects. In the first phase, the *read phase*, there is no copy of the vector X. In the second phase, the *write phase*, all copies of the vector X are purged. Therefore, the implementation of global read and write operations will be particularly stressed. The memory block size is 16 bytes and each processor calculates 4 elements (1 block) of the vector X.

First, we look at the execution time for the algorithm. Second, we look at the network traffic generated by the two protocols. The simulation environment is based on the CacheMire test bench [2] which is a multiprocessor simulator based on a SPARC processor simulator. The parallel applications running on top of the simulator are written using the *parmacs* macros from Argonne National Laboratory. We do not account for the implementation and performance aspects of the barrier synchronizations. The methodology we use is extensively explained in [7], where we perform an extensive evaluation of tree-based and linear-list protocols and contrast their performance with a full-map protocol.

We simulate a multiprocessor architecture with 16 processing nodes connected by an infinite bandwidth network. Each processing node consists of a processor, an infinite cache, a part of the shared memory, and a local bus. The cache line size is 16 bytes. The simulations are done assuming the weakly ordered model [5]. For the STP, the fan-out of the tree, K, is 2. We make the following assumptions about the architecture:

- The network latency between two nodes is 100 processor clock cycles (pclocks, 1 pclock = 10 ns).
- The cache access time is 1 pclock.
- The memory access time is 15 pclocks.
- The transition time on the local bus is 4 pclocks.

The results from the simulation is shown in Figure 8. The execution time is split into three parts. The lower part is the busy time, i.e. the time the processors perform useful work. The middle part is the time added due to read latency, i.e. processor stall time due to cache misses. The upper part is the time added due to write latency, i.e. the time the processors wait at synchronization points for pending writes to be completed. Although the weakly ordered model

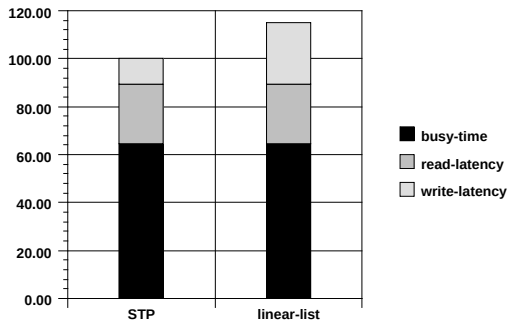

Figure 8: Normalized execution time for the example algorithm.

can hide most of the write latency, we still see a 15% longer execution time for the linear-list protocol than for the STP. This difference will become larger when the number of processors increases.

The STP generates 35% more network traffic than the linear-list protocol does (see Table 3). This larger amount of traffic is acceptable since our belief is that increasing the network bandwidth is easier than decreasing the network latency. Therefore, it is more appropriate to focus on decreasing the latencies in the memory system.

Protocol	Generated traffic
STP	2140 messages
Linear-list	1568 messages

Table 3: The network traffic generated for the protocols when we execute the example algorithm.

5 Conclusions

In this paper we present the design and evaluation of a new cache coherence protocol called the Scalable Tree Protocol (STP). The STP is especially targeted to large-scale multiprocessors containing hundreds to thousands of processors and applications with large sharing sets.

The objective of our study has been to find a cost-effective approach to design a directory-based protocol. The SCI [6], which has a similar objective, maintains the information of which caches sharing the same memory block by organizing them into a linear list. As a result, the memory overhead per cache line is $2 \log_2 N$, assuming N caches. Unfortunately, the linear-list approach results in a write latency which grows as $O(n)$, where n is the number of copies.

Our approach to obtain an acceptable implementation cost and to reduce the write latency is to arrange the caches sharing a memory block in a tree structure. This approach, the STP, results in a write latency of $O(\log_K n)$ and a memory overhead per cache line of $(3 + K) \log_2 N$ bits, where K is the fan-out of the tree. We find the slightly higher implementation cost acceptable compared to the improved write latency.

Important challenges in the design of a tree-based protocol are (i) to maintain an optimal tree structure and (ii) to reduce read and write latencies. We show that this is possible by exploiting parallelism between the data-fetch actions and the actions taken to maintain the tree structure. Results from our program-driven simulations demonstrate that the SCI has a 15% longer execution time than the STP for the example algorithm. Even though our simulations use the weakly ordered memory model [5], which effectively hides write latency, the SCI performs worse than the STP. We have conducted a more detailed performance evaluation in [7]. There are also drawbacks with the STP. First, replacements are more complicated than in the SCI. Second, the STP generates 35% more network traffic than the SCI for the example algorithm. However, this is acceptable since high network bandwidth seems easier to achieve than low latency in memory systems targeted for large configurations.

This study suggests that the STP is more useful than the SCI for machines targeted for parallel processing using applications with large sharing sets.

Acknowledgements

This work was supported by the Swedish National Board for Industrial and Technical Development (Nutek) under the contract number 9001797.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of 15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[2] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström. The CacheMire Test Bench – A Flexible and Effective Approach for Simulation of Multiprocessors. Technical report, Dept. of Computer Engineering, Lund University, Sweden, 1992.

[3] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, 1978.

[4] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Protocol. In *Proc. of the ACM conference ASPLOS-IV*, pages 224–234, 1991.

[5] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[6] IEEE. IEEE – P1596 Draft Document, Scalable Coherent Interface Draft 2.0, March 1992.

[7] H. Nilsson and P. Stenström. Performance Evaluation of Link-Based Cache Coherence Schemes. In *Proc. of the 26st Hawaii International Conference on System Sciences*, January 1993. To appear.

[8] P. Stenström. A Cache Consistency Protocol for Multiprocessors with Multistage Networks. In *Proc. of 16th International Symposium on Computer Architecture*, pages 407–415, May 1989.

[9] P. Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.