# Performance Evaluation of Link-Based Cache Coherence Schemes

Håkan Nilsson and Per Stenström

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

## Abstract

*Large-scale shared-memory multiprocessors rely on private coherent caches by using directory-based protocols. Directory-based protocols preserve network bandwidth by reducing the number of consistency actions. A critical issue becomes how they maintain state information about the set of caches and how they reduce read and write latencies. These tradeoffs are studied in this paper.*

*We study two link-based approaches, called tree-based and linear-list protocols, and contrast their performance and implementation cost to that of a full-map protocol. Using program-driven simulation and a set of three benchmark programs, we find that tree-based and linear-list protocols (e.g. IEEE P1596 SCI) perform almost as well as full-map protocols but to a considerably lower implementation cost. However, if the sharing set is large, linear-list schemes may suffer because of the large write latency while tree-based protocols still perform well.*

## 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. To cope with these problems, a unified approach has been to use caches in conjunction with a directory-based cache coherence protocol implemented in hardware [14].

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 a memory block – *the sharing set*. In limited directory protocol [5], exact information is maintained only when the sharing set is less or equal to the number of pointers. However, when the sharing set is large, limited directory-based protocols perform poorly due to broadcast, pointer replacement, or software overhead, that is intrinsic to this class of protocols. Instead, in this paper we focus on the performance and implementation tradeoffs of three protocols that maintain exact information about the sharing set but that differ considerably in hardware complexity – full-map, linear-list, and tree-based protocols.

In *full-map directory schemes* [4, 13] the cache pointers are represented by a bit vector containing $N$ bits, where $N$ is the number of caches. Since a bit vector is associated with each memory block, the resulting implementation cost for the directory is unacceptable for multiprocessors containing several hundreds of caches.

For large system configurations, say 1024 nodes, researchers have explored other approaches to reduce the implementation cost of the directories. In *link-based protocols* the hardware cost is reduced by organizing the caches that have a copy of a memory block into a link structure such as a linear list or a tree. IEEE P1596, known as the Scalable Coherent Interface (SCI) [9], associates two cache pointers with each cache line. All the caches that share a memory block form a double-linked list. Unfortunately, the SCI incurs a severe performance penalty in handling write operations because the invalidation or update messages have to traverse the list of caches resulting in a write latency of $O(n)$, given $n$ caches.

In an earlier work [11], we have proposed a new directory-based cache coherence protocol called the Scalable Tree Protocol (STP). The STP arranges the caches sharing the same memory block in a tree structure. It has a slightly higher implementation cost than the SCI. Thanks to the fact that the sharing set is arranged in a tree-structure, the write latency in the STP is $O(\log n)$. A critical issue in the design of tree-based cache coherence protocols is how to reduce read and write latencies.

In this paper we investigate the intrinsic performance tradeoffs of full-map, linear-list, and tree-based protocols. The performance evaluation is based on program-driven simulation and three parallel benchmark applications. One benchmark is an algorithm kernel for the solution of a system of linear equations. The other two benchmarks are taken from the Stanford SPLASH benchmark suite [12].

We show that the read-latency is the same for the three protocols. The write latency for linear-list and tree-based protocols is competitive with a full-map protocol for applications where a small number of caches share the same memory block or when the ratio of communication and computation is small. But if the number of caches that share the same block is large, a linear-list protocol suffers because of the time it takes to traverse the list of caches. Our tree-based protocol is competitive with a full-map protocol even in this case. Our study suggests that since linear-list and tree-based protocols are less expensive to implement than full-map protocols, they are more appropriate for large system configurations.

The organization of the rest of the paper is as follows. In Section 2 we present the organization and the consistency actions of the three protocols investigated. The simulation environment, the architecture models, and the benchmark programs are described in Section 3. In Section 4 we present the simulation results. Finally, we conclude our results in Section 5.

## 2 Directory-based protocols

In this section we present the organization and consistency actions of the three protocols we have studied. To compare the protocols in a consistent manner, we begin with the assumptions we have used to study the intrinsic performance differences between full-map, linear-list, and tree-based protocols in Section 2.1. In Sections 2.2–2.4, we outline the three protocols we experimentally evaluate in Section 4. We especially focus on the read and write latency differences associated with the protocol actions. In Section 2.2 we describe a full-map protocol based on the presence flag protocol originally proposed by Censier and Feautrier [4]. In Section 2.3 we present the linear-list protocol, based on the IEEE P1596 standard (SCI), and in Section 2.4 we present our tree-based protocol, the STP.

### 2.1 Framework for comparison

To compare the protocols in a consistent manner we have made a set of basic assumptions regarding their implementations that in some cases may differ from the assumptions made in the literature.

First, we assume that they all maintain consistency using a write-invalidate protocol [14]. Second, we consider two different memory models; sequential consistency [10] and weak ordering [6]. Our implementation of sequential consistency is conservative – the processor stalls on every access to shared data until the access is completed. Under weak ordering write operations can be pipelined between synchronization points. Thus, all write latency can be hidden except for the write operations that are pending when a synchronization operation is encountered. Finally, we assume that memory is kept up-to-date for clean blocks. When more than one cache has a copy of the block, a read request can be serviced by the corresponding memory module since the block kept there is clean[1]. In the next three sections we focus on the read and write latency encountered by the three protocols. In this study we assume infinite caches. As a result, we do not focus on how block replacements are handled. We discuss the effect of replacements in Section 5.

### 2.2 Full-map protocols

In directory-based cache coherence protocols, the information of which caches that share the same memory block is distributed across the processing nodes. The sharing information is used to reduce network traffic and memory contention by exclusively sending invalidation messages only to those caches that share the same block.

In full-map protocols [4, 13] the sharing information is stored in a bit vector of length N, where N is the number of caches in the system. The full-map protocol we simulate has one bit vector of length N associated with each memory block, see Figure 1. The bit vectors are located in the memory module. There is also a modified bit, marked $m$ in Figure 1, associated with each block to indicate if the memory block is dirty.

A cache that reads the block sends a read request to the memory. The memory sets the bit associated with the reading cache. If the block is clean, the requested block is returned to the cache in two network traversals. Otherwise, the memory requests a write-back operation from the cache that keeps the dirty block which takes another two network traversals.

When a write is issued to a globally shared block, the cache sends a write request to the memory. The memory sends invalidation messages to all caches whose bits are set in the bit vector by serially issuing invalidation messages to the network. The invalidation messages are then propagated in parallel through

[1]In the SCI, for example, the memory is not kept up-to-date and a longer read-latency is encountered [9].

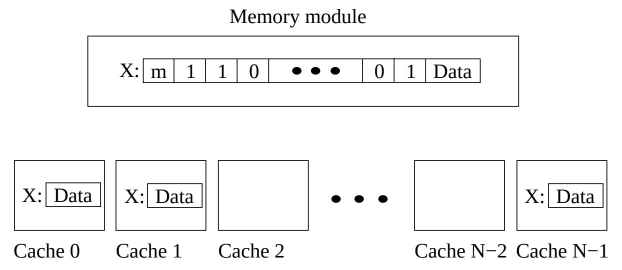

Figure 1: The organization of a full-map protocol where three caches share variable X.

the network. The caches reply with invalidation acknowledgement messages. When the memory has received acknowledgement messages from all the caches in the sharing set, it sends a write acknowledgement message to the writing cache. The memory also sets a modified bit associated with the memory block, not shown in Figure 1. The writing cache has now an exclusive copy and subsequent write operations can be handled locally by the cache.

In summary, we note that a cache miss to a clean block is serviced by two network traversals; to a dirty block in four network traversals; and invalidations propagate in parallel through the network to all caches that share the block and thus incur a write latency of $O(1)$. As for the implementation cost, full-map protocols associate $N$ bits with each memory block.

### 2.3 Linear-list protocols

Linear-list protocols, such as the SCI, maintain the sharing set in a linear list as shown in Figure 2. The basic mechanism of the linear-list protocol is the two pointers ($\log_2 N$ bits each) that are associated with each cache line. They are used to 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 linear-list protocol which is based on the SCI protocol with some modifications according to Section 2.1.

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. First, if the list is empty, the memory establishes a pointer and supplies data in two network traversals. Second, if the block is dirty, the memory requests the dirty cache to write back the block. Consequently, data is returned after four network traversals. Finally, if the block is shared it supplies the data and in addition a pointer to the cache that is the old head of the

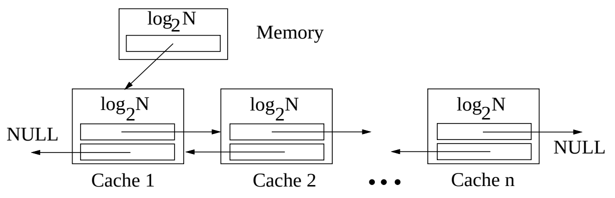

Figure 2: $n$ caches sharing a copy of a memory block according to the linear-list protocol (from Nilsson and Stenström [11]).

list. The memory also 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 updates its predecessor pointer to point at the new head. At this point, the read operation is completed.

On a write operation to a block that is shared by other caches, the cache associated with the writing processor sends a write request to the memory. The memory then sends an invalidation message to the head of the list. The list is invalidated cache by cache in the following way. The head of the list sends an invalidation to its successor. When a cache gets the invalidation, it invalidates its copy, removes itself from the list and returns the identity of its successor to the head. When all caches are removed from the list, the head invalidates its own copy and sends an acknowledgement message to the memory. The memory then sends a write acknowledgement message to the writing cache. The write operation is completed when the writing cache gets the write acknowledgement message. Since the invalidation must traverse $n$ caches, given that the size of the sharing set is $n$, the time for a write operation to be completed is $O(n)$.

In summary, we note that like the full-map protocol a cache miss to a clean block is serviced by two network traversals and to a dirty block in four network traversals. Finally, invalidations propagate sequentially through the linear list. Linear-list protocols thus incur a write latency of $O(n)$, given $n$ copies. As for the implementation cost, linear-list protocols associate $2\log_2 N$ bits with each cache line.

### 2.4 STP – A tree-based protocol

In this section, we outline the consistency actions of the Scalable Tree Protocol (STP) and the support mechanisms associated with it. A complete description of the protocol is found in [11].

In the STP, $K$ subtrees are associated with each cache line, see Figure 3 where $K = 2$. The STP always guarantees an *optimal* tree structure. With an optimal tree we mean that level $i$ is completely filled before caches are inserted at level $i+1$, as shown in Figure 3. A new cache that reads a memory block is inserted in the tree as a leaf at the lowest level. The tree is optimal even if a replacement is done [11].

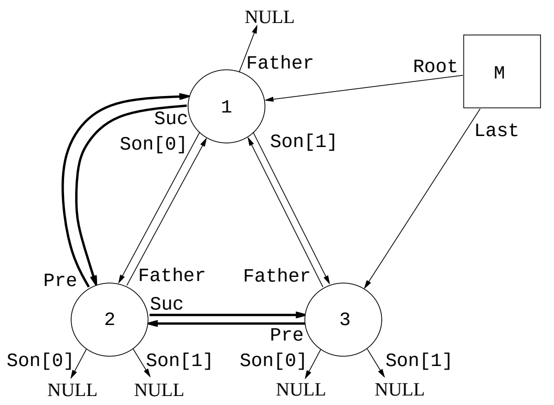

Figure 3: Three caches have read a memory block and all the pointers are set correctly (from Nilsson and Stenström [11]).

In [11] we showed that the overhead is $3\log_2 N$ bits for one memory block and $(3+K)\log_2 N$ bits for one cache line.

Figure 3 shows a tree when three caches have read the block and all pointers are established. The Pre and Suc pointers form a linear list and the Son and Father pointers create a tree. The linear list is used during a block replacement to make sure that the tree is still optimal after the replacement operation. The digits inside the nodes denote the order in which the caches have fetched the block from memory.

We use the notation according to Table 1 to distinguish between different caches and memory modules involved in a coherence operation.

### 2.4.1 Global read operations

We show which actions are taken for read requests to clean blocks shared by other caches. Figure 4 shows which messages are sent when the seventh cache reads the shared memory block. First $C_{rd}$ sends a Read-Request to the memory, M, and then M responds with a Data message 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 has a potential to be overlapped by local computation. The reading cache will

| 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.

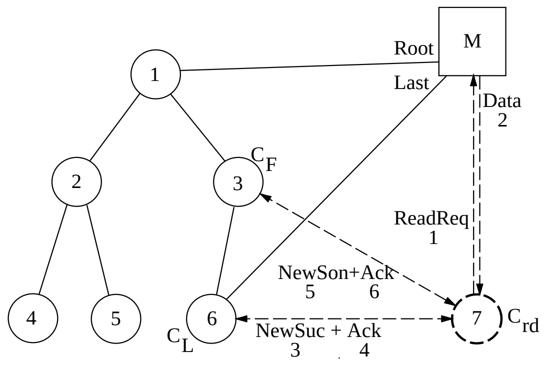

Figure 4: The seventh cache is reading the memory block (from Nilsson and Stenström [11]).

be inserted as son of $C_F$ and successor of $C_L$ in the following way. $C_{rd}$ notifies $C_L$ of its new successor by the message NewSuc. After $C_L$ has acknowledged NewSuc $C_{rd}$ notifies $C_F$ of its new son by the message NewSon. $C_{rd}$ is completely linked into the tree when $C_F$ has acknowledged the NewSon message. The next cache reading the block becomes another subtree to $C_F$, if $C_F$ can have another subtree. Otherwise the next cache becomes a subtree to the successor of $C_F$. $C_{rd}$ is now the new $C_L$.

In summary we note that, like the full-map and the linear-list protocols, a cache miss to a clean block is serviced by two network traversals and to a dirty block in four network traversals. However, some additional network traversals are needed to establish the tree structure. The tree construction can be partly overlapped by computation.

### 2.4.2 Global write operations

The write operation starts with a write request message, WriteReq, from $C_w$ to the memory, see Figure 5. The memory receives the write request, stores the identity of $C_w$ in the WritePending pointer, and sends a CheckLast message to $C_L$. $C_L$ responds with LastOk when it is completely linked into the tree. The memory starts to invalidate all copies when LastOk is received. This is shown in Figure 5 by an invalidation message, Inv, sent from M. Note that the time to construct the tree may show up as an increased write latency. The effect of this is investigated experimentally in Section 4. When a cache receives an invalidation message, it first looks if it has any subtrees. If the cache has subtrees, it sends an invalidation message to each of them. Then the cache waits for an invalidation acknowledgement message, IAck, from each subtree. When the cache has received invalidation acknowledgement from all its subtrees, it invalidates its own copy of the block, and sends an invalidation acknowledgement upwards to its father. If the cache does not have any subtrees, it invalidates its own copy of the block and sends an invalidation acknowledgement message to its father at once.

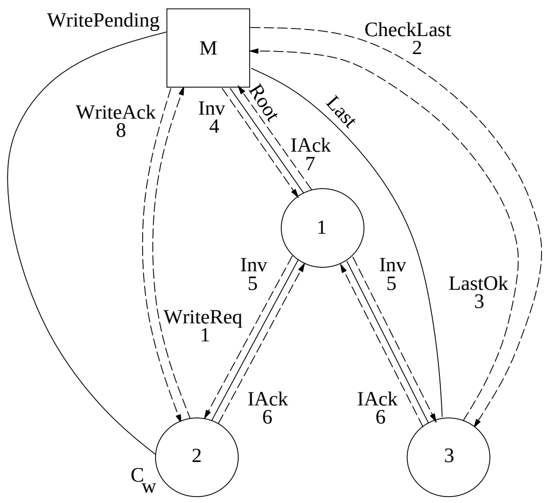

Figure 5: A general view of a global write operation (from Nilsson and Stenström [11]).

When the memory receives an invalidation acknowledgement message, it knows that the whole tree is invalidated and can send a write acknowledgement message, WriteAck, to $C_w$. 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. The memory block is marked modified in the memory. Subsequent write operations by $C_w$ can be performed locally until another cache reads the block.

In summary, we note that invalidations propagate in parallel through the tree structure. Unlike full-map and linear-list protocols, tree-based protocols thus incur a write latency of $O(\log\ n)$, given $n$ copies.

In this section we have described the protocol actions and the implementation cost for full-map, tree-based, and linear-list protocols. As far as the implementation cost is concerned, full-map protocols have the largest state memory overhead followed by the STP and the linear-list protocols. As for the latency penalty, we note that the read latency as seen by the issuing processor is the same for all protocols. However for tree-based protocols, it takes additional time to link the cache into the tree. The question is whether these actions can be overlapped by computation. As far as the write latency is concerned, we note that full-map protocols have a potential to propagate invalidations in parallel through the network and thus incur a write latency of $O(1)$ as opposed to tree-based and linear-list protocols that incur a write latency of $O(\log n)$ and $O(n)$ respectively. In the following two sections, we experimentally study these performance tradeoffs.

## 3 Experimental methodology

This section presents the simulation environment, architectural models, and the benchmark applications we use to make a quantitative comparison between full-map, linear-list, and tree-based protocols.

### 3.1 Simulation environment

We use a simulated multiprocessor environment to study the behavior of parallel applications under the three different cache coherence protocols. The simulation environment consists of two parts: (i) a functional simulator that executes the parallel applications and (ii) the three memory system simulators for full-map, linear-list, and tree-based protocols.

We have used the CacheMire testbench [3], which is a program-driven 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 [1]. The application programs are compiled using gcc 2.0 with the -O2 switch turned on. The generated binary code is executed by the functional simulator. The functional simulator generates a sequence of memory references which is passed to the memory system simulator.

The sequence of references are accurately interleaved since the processor simulator executes the instructions one by one. A global clock is incremented after each processor cycle and takes into account the contention and latency in the memory system.

The memory system simulator takes care of references to shared data; instruction fetches and private data references are assumed to hit in the processor cache. We also assume that synchronization accesses are handled in a single processor cycle. The reason is twofold. First, we are not interested in having a particular synchronization scheme to influence the performance results, and second, the synchronization accesses in the applications we study are rare (as shown in Table 3).

## 3.2 Architecture models

The architecture consists of a number of processing nodes connected by an infinite-bandwidth network with constant latency time. A processing node consists of a processor, a cache, a local bus arbitrated using round-robin, and a part of the shared (local) memory, see Figure 6. Contention occurs if several functional units want to access the local bus at the same time. This contention is modeled correctly by the infinite buffers connected to the local bus. We use infinite caches and write buffers in our simulator. Note that the linear-list protocol has a greater advantage of the infinite write buffers than the other two protocols. We simulate a small configuration of 16 processing nodes. The block size is 16 bytes and the branching-factor is $K = 2$ for the STP.

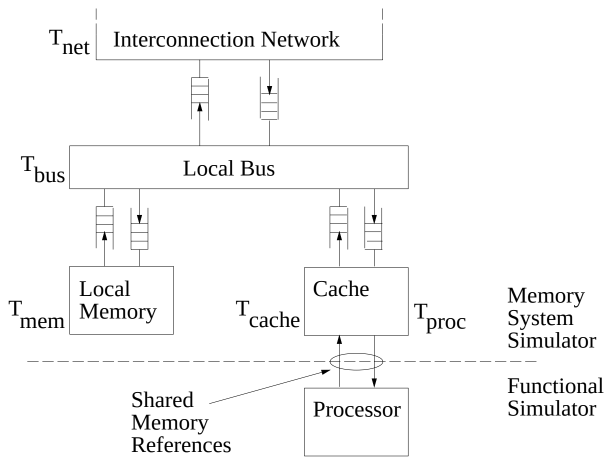

Figure 6: The organization of a processing node and the architecture parameters for each functional unit.

The shared memory is distributed across the memory modules in a fine-grain interleaved fashion to avoid influence of the memory allocation strategy according to the scheme in RP3 [2].

We now focus on the latency times of memory requests in our architecture. To do this in a consistent manner we have defined a set of architecture parameters that are used to construct latency times for all the three cache coherence protocols. Table 2 summarizes these architecture parameters and lists the latency numbers, assuming a processor clock cycle time of 10 ns.

| Architecture Parameter | Notation | Latency (pclocks) |
|---|---|---|
| Cache Access Time | $T_{cache}$ | 1 |
| Cache Fill and Restart | $T_{proc}$ | 6 |
| Local Bus Time | $T_{bus}$ | 4 |
| Network Latency | $T_{net}$ | 100 |
| Memory Access Time | $T_{mem}$ | 15 |

Table 2: The architecture parameters for each functional unit counted in processor clock cycles (1 pclock = 10 ns).

## 3.3 Benchmark programs

To understand the relative performance benefits of the different cache coherence protocols we use a variety of scientific and engineering parallel applications. First, we use a parallel algorithm for the solution of a linear system of N equations by iteration, called Solve. Second, we use two of the SPLASH benchmarks [12] to get insight into the performance of the protocols when executing larger applications.

Statistics acquisition starts when the parallel section of the application is entered, because the initialization part is expected to be negligible for full-scale runs.

### 3.3.1 Solve

In this section we review a classical generic parallel algorithm for the solution of a linear system of $N$ equations by iteration. 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 $P$ processors, where each processor calculates $N/P$ vector elements. The code for one iteration of the algorithm is shown in Figure 7.

```
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 7: Solve, an example of a parallel algorithm for iterative solution of a linear system of equations.

$P$ processes are initiated by the **par_for** statement. Each process calculates $N/P = Delta$ new values which are stored in XTEMP. In the last parallel loop in the iteration each processor copies *Delta* elements of XTEMP back to the vector X.

In order to understand the impact of cache coherence actions on the execution time for the algorithm we focus on the read and write operations to vector X. These are marked in Figure 7. We use Solve in two different versions. In the first version, referred to as *Solve1*, we only consider the read and write accesses to the shared vector X. First, the vector X is distributed to all the caches. Then, each processor updates a part of the vector. Note that in Solve1 there is no computation at all. In the second version, referred to as *Solve2*, we execute all the additions and the multiplications just as Figure 7 shows. In our simulations we use a vector size of N = 256 elements, i.e *Delta* = 16 elements for 16 processors.

### 3.3.2 SPLASH benchmarks

We use two of the applications in the SPLASH [12] benchmark suite. The first application is MP3D which is a particle-based wind tunnel simulator. There are two types of data structures in MP3D. First, there is a space array containing the information of how the simulated 3-dimensional space look like. The space array is also used to determine collision partners for the molecules. The second data structure is the particle array which holds the molecules and records their positions and their velocities. The particles are statically allocated to the processors. The data set we use is 10000 particles simulated for 10 time steps.

The second application we use is Water. Water simulates the evolution of a system of water molecules. The main data structure is a large array of records which stores the state of each molecule. During each time step the interactions between the molecules is calculated. The molecules is statically assigned to the processors. We run Water with 288 molecules for 4 time steps.

## 4 Experimental results

In this section we present the experimental results we obtained executing our applications on the three different memory system simulators. We begin with some general characteristics of the applications in Section 4.1. Then we discuss the performance results for applications with large sharing sets and applications with small sharing sets in Section 4.2 and 4.3 respectively.

### 4.1 Application behavior

The behavior of the parallel applications is summarized in Table 3. The fraction of instruction references is relative to all references. The fraction of private data references and shared data references is relative to all data references.

MP3D has a large portion of shared writes and this influences severely on the execution time as we show in Section 4.3. Although both Solve and Water has a small fraction of shared writes they have very different behavior. A write operation in Solve invalidates as many copies as there are caches in the system. In Water, almost all write operations only need to invalidate one copy. This difference in invalidation pattern influences significantly on the behavior of the applications as we will see.

### 4.2 Applications with large sharing sets

Figure 8 shows the relative performance between full-map, the STP, and the linear-list protocol under

| Application | Instructions | Private data | | Shared data | | Synchronizations |
|---|---|---|---|---|---|---|
| | | read | write | read | write | (Test&Set) |
| | (x1.000.000) | (x1.000.000) | (x1.000.000) | (x1.000.000) | (x1.000) | |
| Solve1 | 0.038(74.4%) | 0.004(34.3%) | 0.004(31.9%) | 0.004(31.8%) | 0.256(2.0%) | 32 |
| Solve2 | 1.8(84.9%) | 0.20(60.0%) | 0.07(20.0%) | 0.066(19.9%) | 0.256(0.08%) | 32 |
| MP3D | 14.7(72.8%) | 1.18(21.3%) | 0.55(10.0%) | 2.33(42.2%) | 1460(26.4%) | 2478 |
| Water | 618.4(64.1%) | 202.1(58.4%) | 88.0(25.4%) | 51.9(15.0%) | 4019(1.2%) | 424176 |

Table 3: The memory accesses divided into different groups according to their type.

sequential consistency (sc) [10]. Note that the processor must stall for the full write latency in our implementation. As expected, we see that the full-map

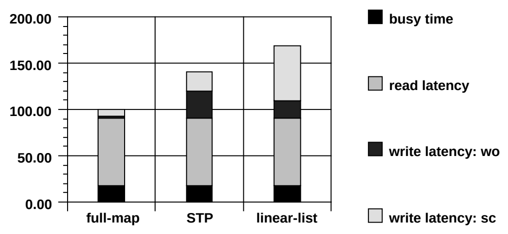

Figure 8: The busy time, read latency, and write latency for Solve1.

protocol outperforms both the STP and the linear-list protocol because it is more successful in reducing the write latency. For the linear-list protocol the write latency is substantial.

The stick diagrams of Figure 8 are broken down into four sections. The bottom section is the busy time which is determined by measuring the execution time on an ideal memory system where each memory reference takes a single cycle. The middle section is the processor stall time due to cache misses or read penalty, and the two topmost sections are the penalty due to outstanding write operations. From this, we see that the difference in performance stems from how well the protocols manage to reduce write latency–the read latency is handled equally well by all the protocols as expected.

Relaxed consistency models are known to be capable of hiding most of the write latency [7]. Therefore, we investigated whether it makes sense to consider more aggressive directory-based protocols than linear-list protocols under weak ordering (wo) [6]. The relative performance between the three protocols under weak ordering is shown in Figure 8 by removing the topmost section. The STP now performs worse than the other protocols. The reason for this is that the penalty due to tree construction cannot be hidden completely. The reason why this effect shows up at all is that the communication to computation ratio for Solve1 is very high. In Solve1, we have removed all computation by only considering reads and writes. This can be seen from the bottom section (processor busy time) which is very small.

To study a more realistic ratio of communication to computation, we introduced the multiplications and additions associated with the algorithm in Figure 7. In Table 4, we see the relative performance between the protocols for this case. As can be seen, the linear-

| | full-map | STP | linear-list |
|---|---|---|---|
| Busy time | 90.1% | 90.1% | 90.1% |
| Read latency | 8.7% | 8.7% | 8.7% |
| Write latency(wo) | 0.3% | 1.0% | 2.4% |
| Write latency(sc) | 0.9% | 2.9% | 7.1% |
| Total time | 100.0% | 102.7% | 108.3% |

Table 4: The busy time, read latency, and write latency for Solve2.

list protocol now performs slightly worse than the STP under weak ordering. This is because of two factors: (i) the tree construction can now be hidden by local computation and (ii) the write latency is not completely hidden. To understand the latter, we consider the possibilities for hiding write latency for the linear-list protocol. The write latency is introduced in the second for loop. The question whether the write latency can be hidden or not will be determined by the distance between the two barriers. When the processors reach the second barrier, there will be at least one outstanding write operation which cannot be hidden. In general we expect the difference between the

STP and the linear-list protocol to be high if the distance between synchronizations is small or when the distance between write operations and the succeeding synchronization is small. Note also that we consider an infinitely sized write buffer. If the write buffer is not sufficiently large the processor may have to stall even though the consistency model allows it to continue. For the linear-list protocol this might be a problem. We therefore believe that the write-buffer size is a critical design parameter for linear-list protocols. In these experiments we have used a small configuration. For large configurations, we expect the write latencies to be higher for the linear-list protocol while the time lost for tree construction is constant for the STP.

In summary, we have shown that for applications with large sharing sets, the write latency for linear-list protocols can be detrimental for performance as compared to full-map and tree-based protocols. Full-map protocols perform better than the STP, but the difference is fairly small even for small branching factors for the STP. We have also seen that the write latency can impact the performance of linear-list protocols, also under relaxed consistency models. A critical design parameter for such protocols is the size of the write buffer. Finally, the tree construction for the STP can be hidden given a reasonable ratio of communication to computation.

### 4.3 Applications with small sharing sets

We executed MP3D and Water assuming weak ordering. In Table 5 we show the execution time for the MP3D. There is no significant difference in the execution time between the three protocols. The large fraction of read latency comes from the fact that most data objects in MP3D are migratory [8] which implies that the data objects are only shared by two processors at the same time. The difference in write latencies does not influence significantly on the execution time since the write operations almost always result in the invalidation of only one copy.

| | full-map | STP | linear-list |
|---|---|---|---|
| Busy time | 12.6% | 12.6% | 12.6% |
| Read latency | 86.7% | 86.7% | 86.7% |
| Write latency | 0.7% | 5.0% | 1.9% |
| Total time | 100.0% | 104.3% | 101.2% |

Table 5: The busy time, read latency, and write latency for MP3D.

In Figure 9 we show the execution time for Water. For Water, the execution time for the three protocols is roughly the same. The execution time is 6% and 3% longer for the STP and the linear-list protocol relative to the execution time of the full-map protocol. The ratio of communication and computation is smaller in Water than in MP3D. Most data objects in Water are

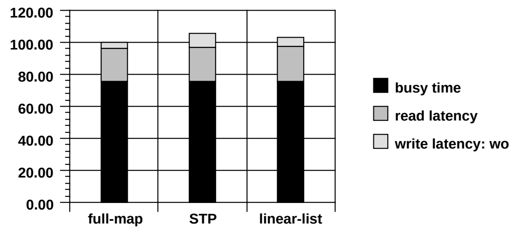

Figure 9: The busy time, read latency, and write latency for Water.

also migratory objects [8], and as a result most writes invalidate only one copy. The execution of MP3D and Water shows that the STP performs slightly worse than the other protocols. The reason is that the tree construction cannot be hidden completely since the migratory objects are updated through read-modify-write operations. For example, in MP3D the particles are read and written in the same C-statement (e.g. `Part->x += u;`).

In summary, we note that for applications with small sharing sets, similar to MP3D and Water, it is insignificant which of the cache coherence protocols the architecture support. In this case, the implementation cost is of greater importance. Link-based cache coherence protocols constitute a promising approach in this case.

## 5 Concluding remarks

In this paper we have evaluated the implementation and performance tradeoffs between three directory-based cache coherence protocols. The evaluation is based on program-driven simulation and three benchmark programs. The protocols investigated are a full-map, a tree-based, and a linear-list protocol.

Link-based protocols, such as the STP and the SCI, show a significantly lower implementation cost as compared to full-map protocols. A drawback, however, is their write latency which grows as $O(\log n)$ and $O(n)$ for tree-based and linear-list protocols, respectively. We have shown that this latency is especially pronounced for linear-list protocols under sequential consistency even for small system configurations containing only 16 caches. For large system configurations the write latency is expected to be prohibitive. Note that tree-based protocols can tolerate this latency by using larger branching factors. In the experiments, we assumed a branching-factor of 2.

An important issue for the STP is the tree construction which can impact the write latency. We observed that if the distance between a cache miss and a subsequent write operation is small, such as in read-modify-write operations, the write operation may take longer time due to the tree construction. We have seen that this effect is fairly small and, more importantly – the tree construction time is constant and independent of the number of caches.

We did not address the impact of cache replacements on the performance. As discussed in our previous work [11], replacements involve more actions for tree-based protocols than for full-map and linear-list protocols. As a result, if the replacement misses dominate over coherence misses, the STP may suffer. For COMA-architectures [15], where the replacement miss rate is low, STP is a promising approach.

For applications where the sharing set is small, the performance difference between the three protocols is negligible. Since link-based protocols have a considerably smaller implementation cost for large system configurations, we believe that they are preferable compared to full-map protocols.

## Acknowledgements

We want to thank Mats Brorsson and Fredrik Dahlgren for providing feedback on an earlier draft of this paper. This work was supported by the Swedish National Board for Industrial and Technical Development (Nutek) under the contract number 9001797.

## References

[1] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[2] W.C. Brantly, K.P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proc. of the 1985 International Conference of Parallel Processing*, pages 782–789, 1985.

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

[4] 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.

[5] 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.

[6] 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.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. of the ACM conference ASPLOS-IV*, pages 245–257, 1991.

[8] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

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

[10] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[11] H. Nilsson and P. Stenström. The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, December 1992. To appear.

[12] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[13] 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.

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

[15] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proc. of 19th International Symposium on Computer Architecture*, pages 80–91, May 1992.