Replacing Locks by Higher-Level Primitives

Sarita V. Adve, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel

Department of Electrical and Computer Engineering
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Locks are used in shared memory parallel programs to achieve a variety of synchronization objectives. They may provide mutual exclusion for a critical section, or they may provide synchronized access to a task queue. In the former case, no ordering is implied between data operations outside of the critical sections, while in the latter case the operations preceding the enqueue of a task are ordered before the operations following a dequeue of that task.

In this paper we argue that many uses of locks can be replaced by higher-level primitives that directly express the intended behavior, such as, for example, enqueue and dequeue operations on a task queue. This approach not only simplifies the programming model, but also allows a more efficient implementation. By making the intended use explicit, we can tailor the implementation of each primitive accordingly, thereby achieving reductions in latency and communication overhead. Synchronization latency has been shown to be one of the major impediments for achieving high performance, especially on software distributed shared memory systems.

We demonstrate the benefits of our approach by comparing the performance on the TreadMarks distributed shared memory system of lock-based implementations of four applications (TSP, Quicksort, Water, and ILINK) to new implementations using the higher-level primitives. For each of the four applications, the high-level primitives lead to simpler programs with better performance.

This work is supported in part by NSF Grants No. CCR-91163343, CCR-9211004, CDA-9222911, and CDA-9310073, and by Texas ATP Grant No. 0036404013.

1 Introduction

Shared memory programming provides a simple programming model as compared to the alternative of message passing. The latency of synchronization operations, however, is an impediment to the scalability of shared-memory systems, especially for software distributed memory systems.

Synchronization operations are used to mediate access to shared data in a shared memory program. Locks, barriers, and condition variables are the most common primitives provided to the user. The key observation underlying this paper is that the same low-level synchronization operations are often used to implement different high-level synchronization objectives. The thesis of this work is that by allowing the programmer to express synchronization objectives by means of higher-level primitives, we can simultaneously make programming easier and the execution of the resulting programs more efficient.

We illustrate our general approach with a brief example. Locks are typically used to implement critical sections, guaranteeing that only one process at a time can execute inside the critical section. For instance, several processes may accumulate their contributions to the value of a variable inside such a critical section. This use of locks implies no ordering relation between the execution of the different processes. Indeed, a different order of execution of the critical section would result in the same outcome. In a different scenario, when locks are used to implement task queues, the locks again provide atomic critical sections, but, in addition, the task queue imposes an ordering relation between the execution of the different processes. The work done and the modifications made by the process doing an `enqueue` must be visible to the process that does the corresponding `dequeue`.

The synchronization objectives in the above two instances are very different. The first case only requires atomic execution of the critical sections; the second case additionally imposes an ordering relation between different parts of the execution. However, both objectives are normally expressed using the same lock primitives. Therefore, the runtime system (hardware or software) is not aware of the difference and must implement both strategies in the same manner. For example, data protected by a lock is essentially always moved between different processes on each lock access. In the case where locks are simply used to accumulate different contributions to a variable, this is unnecessary. The contributions could be accumulated locally and summed when the processes next synchronize, resulting in much reduced communication and better performance.

The previous discussion illustrates the potential performance benefits of using high-level primitives. They also facilitate programming in a number of ways. For instance, a generic task queue template could be provided that can be customized for each application. More importantly, in our experience with parallel applications, we have seen several examples where the inefficiencies resulting from general-purpose implementations of locks caused the programmer to resort to awkward implementations, more reminiscent of message passing than of shared memory programming. Returning to our example where contributions from different processors to a variable are synchronized by a general-purpose lock, the cost of doing so often leads the programmer to introduce extra local variables to accumulate the contributions locally, followed by an additional step in which the local contributions are summed globally.

The overall goal of our work is to identify a small number of such high-level primitives and provide those to the programmer in addition to the common low-level primitives. In this paper we present three such high-level primitives: non-synchronizing atomic operators, synchronizing atomic operators, and task queues. We evaluate the benefits of these primitives by comparing the performance of lock-based implementations of four applications (TSP, QuickSort, Water, and ILINK) to new implementations using the high-level primitives. This comparison is done in the context of the TreadMarks software distributed shared memory (DSM) system. Representative results of our study are as follows.

For Water (a molecular dynamics simulation), the use of our primitives resulted in a speedup of 4.83 on 7 processors. The original version of the program did not show any speedup on TreadMarks. Some hand-tuning of the original version can be used to improve the speedup of Water with locks; our primitives both preclude the need for such hand-tuning and surpass the performance of the hand-tuned program. For ILINK, the use of our primitives resulted in a speedup of 3.4 with 4 processors compared to a speedup of 2.6 using locks. ILINK can be hand-tuned to give similar performance without our primitives. Thus, in this case, our primitives primarily make it simpler for the programmer to exploit the full potential of the system.

The outline of the rest of this paper is as follows. Section 2 provides some background on release consistency and the TreadMarks DSM system. Section 3 presents the three high-level primitives we have studied so far: non-synchronizing atomic operators, synchronizing atomic operators, and task queues. Section 4 provides a brief description of the experimental platform. Section 5 describes the applications used to evaluate the primitives with particular attention paid to the use of the high-level primitives. Section 6 presents and analyzes performance measurements from an implementation in the TreadMarks DSM system. We discuss related work in Section 7, and conclude in Section 8.

2 Background

The high-level primitives are generally applicable to any shared memory system. In this paper, however, we evaluate them in the context of a lazy release-consistent software DSM system. This section provides the necessary background to understand the performance advantages of the proposed approach in such a system. To that end, we focus on the implementation of general-purpose locks in a software DSM.

2.1 Release Consistency

Release consistency (RC) [11] is a *relaxed* memory consistency model that permits a processor to delay making its changes to shared data visible to other processors until certain synchronization accesses occur. Shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. For example, a lock is an acquire operation and an unlock is a release. Similarly, an arrival at a barrier is a release, and a departure from a barrier is an acquire. Essentially, RC requires ordinary shared memory updates by a processor p to become visible at another processor q, only when a subsequent release by p becomes visible at q.

Programs written for conventional sequentially consistent (SC) memory [14] produce the same results on an RC memory, provided that (i) all synchronization operations use system-supplied primitives, and (ii) there is a release-acquire pair between conflicting ordinary accesses to the same memory location on different processors [11]. In practice, most shared memory programs require little or no modifications to meet these requirements.

RC can, however, be implemented more efficiently than SC. In the latter, the requirement that shared memory updates become visible immediately implies communication on each write to a shared data item for which other cached copies exist. No such requirement exists under RC. The propagation of the modifications can be postponed until the next synchronization operation takes effect.

2.2 Lazy Release Consistency

In *lazy release consistency* (LRC) [13], the propagation of modifications is postponed *until the time of the acquire*. At this time, the acquiring processor determines which modifications it needs to see to preserve the semantics of RC for the programs described above.

To do so, LRC uses the *happens-before-1* partial order [2]. Two memory accesses p and q are ordered by *happens-before-1*, if (i) they occur on the same processor and p comes in program order before q, or (ii) they are on different processors, q is an acquire, and p is the "corresponding" release, or (iii) there exists an r such that p *happens-before-1* r and r *happens-before-1* q.

LRC divides the execution of each process into *intervals*, each denoted by an *interval index*. Each time a process executes a release or an acquire, a new interval begins and the interval index is incremented. The happens-before-1 partial order on memory accesses implies a similar partial order on intervals in an obvious way. This partial order can be represented concisely by assigning a *vector timestamp* to each interval. A vector timestamp contains an entry for each processor. The entry for processor p in the vector timestamp of interval i of processor p is equal to i. The entry for processor $q \neq p$ denotes the most recent interval of processor q that precedes the current interval of processor p according to the partial order. A processor computes a new vector timestamp at an acquire according to the pair-wise maximum of its previous vector timestamp and the releaser's vector timestamp.

LRC requires that before a processor p may continue past a lock acquire, for example, the updates of all intervals with a smaller vector timestamp than p's current vector timestamp must be visible at p. Therefore, at an acquire, p sends its current vector timestamp to the last or current holder of the lock, q. When the lock is available, processor q sends a message to p, piggybacking *write notices* for all intervals named in q's current vector timestamp but not in the vector timestamp it received from p.

A write notice is an indication that a page has been modified in a particular interval, but it does *not* contain the actual modifications. The timing of the actual data movement depends on whether an invalidate, an update, or a hybrid protocol is used (see [9]). Our current implementation uses an invalidate protocol: the arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are obtained for the local copy.

To capture modifications to shared data, write access to shared pages is initially disabled. Upon the first write access, a hardware protection violation causes the operating system to invoke an exception handler. The handler makes a copy of the page, hereafter called a *twin*, and removes the write protection so that subsequent write accesses to the page occur without software intervention. At a later time, when the modifications are sent to another processor, the page and its twin are compared to create a *diff*, a run-length encoding of the modifications. A diff contains the new value assigned to each location for which the page differs from its twin. Upon receipt of a diff, a processor updates its local copy of a page by replacing the contents of each location specified in the diff with the new value from the diff.

3 High-Level Synchronization Primitives

The key thesis of this work is as follows. Locks are used in shared-memory parallel programs to express a variety of synchronization objectives. Since the runtime system does not have any further information about the intended use of these locks, it must implement a "one-size-fits-all" strategy for them. We argue that these locks can be replaced by different higher-level primitives that make explicit their intended use and therefore allow a more efficient implementation, without complicating the task of programming.

The following three sub-sections describe three such higher-level primitives that can be used to replace locks. We do not claim that this set of primitives is in any way exhaustive, but we believe that they can be used in many programs. In particular, we identify the primitives of non-synchronizing atomic operators, synchronizing atomic operators, and task queues. We motivate each primitive with an example, formally specify its effect on the execution order of memory operations, and describe our implementation.

Programmers do not necessarily need to understand LRC or the new implementations to use our primitives. Section 3.4 explains how programmers can exploit our primitives and yet continue to assume the familiar model of sequential consistency.

3.1 Non-synchronizing Atomic Operators

3.1.1 Motivation

Figure 1 shows a code fragment to motivate our first primitive. The code shows processors generating several values to be added into a variable *var*. After all the additions, the processors synchronize at a barrier, and potentially, the variable *var* is read after the barrier. The additions need to be executed atomically; otherwise, some values could be lost. For this purpose, processors first acquire a lock before executing the addition and then release the lock. However, using a general-purpose lock in this situation is overkill since such an implementation will need to ensure that all updates made by processors that previously held the lock be visible to the processor that next requests a lock. Specifically, consider a general-purpose LRC lock implementation. First, a locking processor must see the entire latency to acquire the lock. Second, the processor will receive write notices corresponding to the updates in the critical section and the "work" sections of all processors that earlier held the lock. This requires invalidating the corresponding pages and incurring (possibly false-sharing) misses on subsequent accesses to these pages. In particular, a miss is incurred on the read of *var*.

For the given program, however, the lock is not intended to obtain any information about the shared-memory updates in other parts of the program. Instead, the only intent is to atomically accumulate all increments to the variable *var* before the barrier completes. Thus, the above-mentioned lock latency, write notices traffic, and misses are all unnecessary.

```
repeat until done {
   do some work to generate val

   lock(l)
   var = var+val
   unlock(l)
}
barrier

... = var
```

Figure 1 A lock used for atomic accumulation of increments

A more efficient implementation that adequately captures the programmer's intent is one where each processor performs its increments locally in its local copy of the variable, the processor communicates only the *total increment* to the barrier, the barrier processor accumulates all the increments from all processors in its local copy, and then the barrier processor sends its copy to the next reader. This method eliminates the above-mentioned latency and traffic. The reasons this method gives the correct answers are the following. First, a processor that does an addition need never know the current value of *var*; therefore, there is no need for a processor to take a miss on the read of *var*. Second, since addition is associative and commutative, the order in which the increments are added is not important; therefore, there is no need for a processor to synchronize with other processors that execute the additions. Finally, a subsequent reader of *var* (that uses the value for computation other than the adds) synchronizes with all the adders through the barrier; therefore, communication of the increments can be postponed until the barrier.

The above implementation accurately captures the intent of the programmer; however, a system that only provides general-purpose locks has no means of deducing that intent. We, therefore, introduce a class of primitives called non-synchronizing atomic operators that naturally express this intent. Informally, the intended semantics of such operators are that the specified operations will execute atomically with respect to other such operators, and their effects will become visible only at the next synchronization. We currently support a few non-synchronizing atomic opertors, one of which is an add denoted by `ns_atomic_add`. The entire critical section in Figure 1 can now be replaced with this opertor, which more naturally expresses the intent of the programmer.

3.1.2 Formal Specification

The only guarantee for the non-synchronizing atomic operators is that if such an operator is ordered before an acquire by happens-before-1, then it is performed at the acquiring processor at the time of the acquire.

For example, consider the following execution of the example in Figure 1, where `ns_atomic_add` is a non-synchronizing atomic operator.

```
P1: ... ns_atomic_add(x,1) ... ns_atomic_add(x,2) ... wait_at_barrier(b) ...
P2: ...         ns_atomic_add(x,1) ...       wait_at_barrier(b) ...
```

Assume the initial value of x is 0. After P1 and P2 depart from the barrier, the value of x should be 4, because with arrival at a barrier modeled as a release and departure from the barrier as an acquire, all of the `atomic_add`'s are ordered before the departure from the barrier by happens-before-1 and should be performed before departure from the barrier.

The difference between the above execution and an execution with locks is in the guarantees made about the value of x prior to departure from the barrier. In the version using `atomic_add`'s the only guarantee made is that a processor will see the effect of its own prior operations, because those precede in happens-before-1; updates on other processors do not precede in happens-before -1 and therefore need not be visible. In the lock-based program fragment, instead, intermediate values must reflect all updates made between prior lock-unlock intervals, because locks are modeled as acquires and unlocks as releases.

3.1.3 Implementation

The non-synchronizing atomic operators can be implemented far more efficiently than the lock-based equivalents. Each processor can simply record locally what operators have executed. There is no need to wait for locks to be acquired from other processors or to exchange values with other processors. When the program comes to a synchronization, the locally accumulated operations are merged and applied to the value of the variable.

To implement non-synchronizing atomic operators, we made several changes to the implementation of diffs to support operations in addition to replacement. First, the diff additionally specifies the operation used to update the contents of each modified location. For example, to implement a non-synchronizing `ns_atomic_min`, a processor updates its local copy of a page by comparing the value from the diff to the value from the corresponding location in the page. The value in the page is replaced only if the value in the diff is smaller. Second, the value contained in the diff depends on the type of data stored in a location and the atomic operation performed on the location. For example, to implement `ns_atomic_add`, the value in the diff is determined by subtracting the value in the twin from the value in the page. A processor updates its local copy of a page by adding the value in the diff to the value in the page. Furthermore, how the difference is computed depends on the type of the value, for example, whether it is an integer or floating-point number.

Different atomic operations can be performed on different parts of the same page, producing a diff in which each run specifies a different operation for updating its part of the page. To determine how to construct a diff for such a page, TreadMarks maintains a data structure per page detailing which atomic operations are performed on which parts of the page. By default, replacement is used to update the page.

Note that locations accessed by atomic operators can also be accessed by ordinary data operations (e.g., for initializing or reseting the location). As long as the data operation does not form a race with the atomic operators (i.e., they are ordered by happens-before-1), the normal LRC protocol will ensure that the updates are propagated correctly.

3.2 Synchronizing Atomic Operators

3.2.1 Motivation

The non-synchronizing atomic operator of the previous section is useful when the values of the locations accessed by the operators are guaranteed to be used only after adequate synchronization, and when the involved operations are associative and commutative. However, some of the performance advantage of such operators can be exploited even when the above conditions are not true.

Consider, for example, the code in Figure 2. It illustrates a common class of applications that use a branch and bound algorithm to search the best solution from a tree of possible solutions. As shown in the figure, such algorithms involve a global bound based on which the search space is pruned. When a solution that beats the current bound is found, the bound and the current-best solution are both updated. The comparison of the old and new bounds, and the two updates need to be executed atomically. Again, locks are typically used to ensure atomicity.

As for Figure 1 in the previous section, the locks in Figure 2 are not intended to order any operations outside the critical sections. A simple LRC system, however, has no means of discerning this intent and must necessarily be conservative. To better express the intent of the programmer, we again propose replacing the locks and the critical section with a new atomic operator. In contrast to the previous section however, for programs illustrated by Figure 2, the value of the current_bound is often used later in the program without further synchronization. We therefore additionally require a limited form of synchronization for the new operators, and call them *synchronizing atomic operators*. Informally, the intended semantics for these operators is that they will take effect atomically with respect to each other. In addition, they are also serialized with respect to each other, so that any values returned by these operators are the "latest" values.

The specific operator in this class that we support is an `atomic_compare&swap`, where the

```
until done {
search for new solution
lock(l)
if (new_bound is better than current_bound)
update bound
update current_best_solution
unlock(l)
}
```

Figure 2 A lock used for atomic updates in branch-and-bound algorithms

compare is a "less than" comparison and the operator has the obvious parameters. The critical section in the above example can now simply be replaced by the above operator, again a more natural expression of the programmer's intent.

3.2.2 Formal specification

The system guarantee for synchronizing atomic operators is simply that such operators appear to take effect atomically with respect to each other, in a total order consistent with the happens-before-1 order.

The presence of the above total order is the reason we call these atomic operators "synchronizing." Thus, the order of execution of a pair of atomic operators (that access a common location) can impose an order on the execution of other atomic operators. Specifically, an atomic operator always sees the updates of other atomic operators "serialized" before it. This is in contrast to the non-synchronizing operators described in the previous sub-section.

Note, however, that the atomic operators do not impose an order between ordinary data operations. Thus, the *happens-before-1* relation no longer contains any arcs due to the order of execution of atomic operators of different processors.

3.2.3 Implementation

The implementation of synchronizing atomic operators is simpler and faster than the implementation of locks. The primary reason is that a lock transfer must convey the write notices for the (potentially many) writes that happened before the lock release and apply them (using invalidate or update) at the acquiring processor. In contrast, the atomic operator must only insure that it is serialized with respect to other (relatively few) atomic operators in the execution. To achieve this serialization, the processors communicate with a fixed processor that serves as a manager for the synchronizing atomic operators.[1] To perform such an operation, a processor sends a request to the manager specifying the relevant variables, values, and the operation. The manager replies with the new values of the variables and a *serial number*. The processor updates its local copy of the variables and stores the serial number in the TreadMarks directory structure. When another

[1]For greater efficiency, we can employ different managers for different sets of operators, based on any of several criteria. For example, if an atomic operator is restricted to access variables from only a single page, then each page may be allocated a different manager. The current implementation uses a single manager for all operators.

processor requests the modifications made to a page containing such variables, the serial number is encoded in the diff in addition to the variables' value. The processor receiving the diff checks its serial number for the variable and updates its local copy only if the diff's serial number is larger.

3.3 Task Queues

Many programs that rely on dynamic parallelism rely on a task queue to achieve dynamic load balance. Again, such a task queue is most commonly implemented with a lock surrounding a section of code in which the queue data structures are updated. The ramifications of this are illustrated in Figure 3, which shows an execution of such a program.

Processor P1 enqueues some work, followed by another enqueue by processor P2, followed by a dequeue by processor P3. Assuming a FIFO queue, P3 will get the work enqueued by P1. Typically, P3 will require that all the data generated by P1 pertaining to work1 be visible to it at the time of the dequeue. It does not matter what updates P2 did while generating work2 since typically work1 and work2 will be independent. Thus, although P3 accessed the queue after P2 did, there is no intended synchronization between P2 and P3. An LRC system in which queues are implemented through the use of locks has no means of determining this. Therefore, an LRC system will unnecessarily pass information about all preceding updates from P2 to P3. In the presence of false sharing, this can unnecessarily force P3 to later communicate with P2 and possibly other processors (see Section 6.1.2 for an example).

We propose to provide primitive operators `enqueue_task()` and `dequeue_task()`, with the obvious meaning. The actual function to manage the queue may be provided by the programmer

```
P1                  P2                 P3

...                 ...

/*enqueue work1*/

lock(l)
code to enqueue
unlock(l)

                    /*enqueue work2*/

                    lock(l)
                    code to enqueue
                    unlock(l)
                                       /*dequeue*/
                                       lock(l)
                                       code to dequeue
                                       unlock(l)
```

Figure 3 Work queues

as a parameter to the above operators; the runtime system uses the operators to ensure that only the necessary consistency information is communicated on a dequeue, as determined by the entry that is dequeued.

3.3.1 Formal Specification

The only ramification of the enqueue and dequeue primitives is that they are now part of the happens-before-1 relation. An enqueue operation is ordered before a dequeue operation by happens-before-1 if the dequeue returns the entry inserted by the enqueue. The transitivity property of happens-before-1 extends this order to other operations as well. Thus, if a write to a field of an object precedes an enqueue of a pointer to the object, then the write will be seen by any processor that dequeues the pointer (but not by a processor that dequeues other entries in the queue).

3.3.2 Implementation

Like a lock, each queue has a fixed, well-known processor that serves as the manager. Processors send enqueue and dequeue requests to the manager. The manager maintains a queue containing a record for each enqueue it receives. The element enqueued is kept at the enqueueing processor. In other words, the enqueueing processor only announces that it is enqueueing an item to the manager to establish the order of enqueue and dequeue operations. No data movement to the manager occurs. On a dequeue, the manager forwards the request to the processor whose element is being dequeued. This processor responds to the dequeueing processor with the data element and the consistency information so that the dequeueing processor will see all modifications that happen before the enqueue.

3.4 Using the Primitives With Sequential Consistency

We have proposed three primitives: non-synchronizing atomic operators, synchronizing atomic operators, and task queues. As with LRC, there exists a simple interface for programmers who wish to use the above primitives and yet retain the familiar model of sequential consistency.

LRC effectively requires that there should not be data races; in other words, all conflicting data operations should be ordered by the happens-before-1 relation. With the new primitives, we first need to consider an enqueue and a dequeue as a release and an acquire respectively. (Atomic operators are not modeled as releases or acquires.) Now, the conditions that the programmer has to obey are the following (for any sequentially consistent execution of the program).

- Two conflicting data operations should either be ordered by happens-before-1 or should be parts of an atomic operation.
- Non-synchronizing atomic operators should be commutative and associative, and any shared variables accessed by such operators should be accessed only by other non-synchronizing atomic operators or by ordinary reads.

Executions of programs that obey the above constraint will appear sequentially consistent, and the atomic operators will appear to be atomic with respect to each other.[2]

[2]The proof that this restriction ensures sequential consistency can be done along the same lines as in [3].

4 Experimental Environment

Our experimental environment consists of 8 DECstation-5000/240's running Ultrix V4.3. Each machine has a Fore ATM interface that is connected to a Fore ATM switch. The connection between the interface boards and the switch operates at 100-Mbps; the switch has an aggregate throughput of 1.2-Gbps. The interface board does programmed I/O into transmit and receive FIFOs, and requires fragmentation and reassembly of ATM cells by software. Interrupts are raised at the end of a message or a (nearly) full receive FIFO. All of the machines are also connected by a 10-Mbps Ethernet. Unless otherwise noted, the performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

The minimum roundtrip time using send and receive for the smallest possible message is 500 μseconds. The minimum time to send the smallest possible message through a socket is 80 μseconds, and the minimum time to receive this message is 80 μseconds. The remaining 180 μseconds are divided between wire time, interrupt processing and resuming the processor that blocked in receive. Using a signal handler to receive the message at both processors, the roundtrip time increases to 670 μseconds.

The minimum time to remotely acquire a free lock is 827 μseconds if the manager was the last processor to hold the lock, and 1149 μseconds otherwise. In both cases, the reply message from the last processor to hold the lock does not contain any write notices (or diffs). The time to acquire a lock increases in proportion to the number of write notices that must be included in the reply message. The minimum time to perform an 8 processor barrier is 2186 μseconds. A remote page fault, to obtain a 4096 byte page from another processor takes 2792 μseconds.

5 Applications

In this section, we describe the applications used in our study and motivate the efficiency and ease of use of the proposed synchronization primitives.

5.1 ILINK

Our first application is a parallel version of ILINK [10], which is part of the standard LINKAGE package [15] for genetic linkage analysis that is widely used by geneticists (and therefore a "real" application). We started with a version from the FASTLINK package [7], in which the sequential algorithms have been sped up by roughly one order of magnitude.

Genetic linkage analysis is a statistical technique that uses family pedigree information to map human genes and locate disease genes in the human genome. The fundamental goal in linkage analysis is to compute the probability that a recombination occurs between two genes, called the recombination probability. The ILINK program searches for a maximum likelihood estimate of the multilocus vector of recombination probabilities (called the recombination vector) of several genes. The sequential algorithm is summarized in Figure 4. The program involves a nested loop that, for a given value of the recombination vector, iterates over each pedigree and each nuclear family (consisting of parents and child) within each pedigree to update the probabilities of each genotype [15] for each individual. The probabilities are stored in an array `genarray`. The update involves an addition of a value, which is computed in the inner loop.

A straightforward method of parallelizing this program is to split the iteration space among the processes and surround each addition with a lock to do it in place (ILINK-Lock). This approach was deemed far too expensive both on a hardware shared memory multiprocessor and on TreadMarks,

```
For each pedigree
    For each nuclear family
        For double loop over possible genotypes for each parent
            genarray[i] += computed value
```

Figure 4 Sequential Linkage Computation

and instead the approach in Figure 5 was initially used for ILINK (We will refer to this version as ILINK-Local). In this version, a local copy of genarray, called `gene` in Figure 5, is created. After all the processes have completed their local iteration spaces, the updates are merged into the final global copy after synchronization.

It is clear that the additions can be done in a non-synchronizing manner because they are commutative and associative and any other reads of `genarray` occur only after the next barrier. The non-synchronizing atomic operators allow the additions to be performed in place, thereby allowing the programmer to use the simpler programming style of Figure 4 where the updates are performed in place without loss in efficency. Figure 6 illustrates their use. We refer to this version as ILINK-Add.

5.2 Quicksort

Quicksort sorts an array of 256K integers using a quicksort algorithm. Quicksort uses a single global task queue on which pieces of the array to be sorted are placed. The array is recursively divided into smaller pieces until a small enough (1K elements) subarray is split off, which is then sorted locally using bubblesort.

In the original version of the program, the queue was implemented using locks, implying that each queue operation synchronized with all previous queue operations. This version is called Quicksort-Lock. What the program really requires is only that the processor dequeueing a piece of work synchronize with the processor that enqueued the specific piece of the array. It is easy to see that the use of the `enqueue_task()` and `dequeue_task()` primitives of Section 3.3 to implement the queue allows the knowledge of the necessary synchronization to be exploited and simplifies the programmer's responsibility in terms of queue implementation. We have implemented such a version and refer to it as Quicksort-Queue.

```
For each pedigree
    For each nuclear family
        Split up double loop over possible genotypes for each parent
        For rows assigned to each processor
            Sum updates in local gene array
        Barrier synchronization
        Sum updates from gene into genarray
```

Figure 5 Localized Parallel Linkage Computation- ILINK-Local

For each pedigree
    For each nuclear family
        Split up double loop over possible genotypes for each parent
        For rows assigned to each processor
            `ns_atomic_add`(genarray[i], computed value)
        Barrier synchronization

Figure 6 Parallel Linkage Computation - ILINK-Add

5.3 TSP

TSP solves the travelling salesman problem. The program uses a branch-and-bound algorithm (described in Section 3.2) to generate a minimum-cost tour consisting of 18 cities.

The bound in this case contains the length of the best tour generated at the given point in the execution. The execution mainly consists of each processor picking a partial tour from a work queue and doing one of two things. If the partial tour is still too short (i.e., there is extensive computation left to complete the tour), the processor expands the tour further to create multiple partial tours and enqueues these tours in the task queue for other processors to examine. If the partial tour is long enough, the processor itself expands the partial tour into all possible full tours. Each time a processor generates a full tour that is shorter than the current best tour, it updates the bound and the current best tour.

A lock is used to ensure the updates to the bound and the current best tour occur atomically. Since no ordering of other data operations is implied, the entire critical section can naturally be replaced with the provided `compare&swap` operation. This eliminates any consistency information transfer at that critical section. The two versions of TSP are referred to as TSP-Lock and TSP-Min.

5.4 Water

Water is a molecular dynamics simulation from the SPLASH suite [17]. The program evaluates forces and potentials in a system of water molecules for a user-specified number of time steps. Each time step consists of several phases, where consecutive phases are separated by a barrier. One of the computationally intensive phases concerns updating the intermolecular forces exerted by every pair of molecules on each other. Each processor updates a subset of all pairs of molecules. The operation for each pair involves reading the force vectors of the two molecules, reading the positions of the two molecules (which are not updated in the current phase), computing the new values for the force vectors (based on the positions), and finally writing the new values to the force vectors. The above operation must be done atomically, and so is protected by a lock. In this case also, the lock is not used to ensure any ordering, and this operation is simply an `atomic_add`. Furthermore, it is clear that these adds can be done in a non-synchronizing manner because adds are commutative and associative and any other reads of the force vectors occur only after the next barrier. Thus, we can replace the entire critical section by a non-synchronizing `ns_atomic_add`.

As with most of the other programs, our primitives also allow more natural programming for Water in the following manner. In the original program, each time a pair of molecules is examined, a lock has to be accessed for both of the examined molecules (so that the forces can be updated atomically). Thus, a lock for a given molecule is accessed by a processor several times in a phase. Since locks are expensive, Water was modified (Water-Local) so that each processor created a local copy of the force vectors for each molecule. All the updates were first done on the local copy. Finally, at the end of the computation, the actual copies of the force vectors of all molecules were updated, requiring only a single lock access per molecule. The use of the non-synchronizing `atomic_adds` precludes the need for the above modification and eliminates the extra copy from local to global data.

6 Results

6.1 Lock-Based Implementations Versus High-Level Operations

Figures 7 to 10 present the speedups achieved for ILINK, Quicksort, TSP, and Water on TreadMarks using the lock-based implementations and the high-level operations. The TreadMarks speedups are relative to the single processor DECstation run times without TreadMarks. Table 1 details the number of messages and the amount of data movement per second on TreadMarks for each of the applications on up to 7 processors.[3] Sections 6.1.1 to 6.1.4 discuss the results for each application in detail.

6.1.1 ILINK

Figure 7 shows the speedups for ILINK for the version using `ns_atomic_add` vs. locks and the CLP input data set [12]. As can be seen, the use of `ns_atomic_adds` greatly improves performance. This is mainly because of the reduction in the number of off-node messages, as borne out by the message counts at 7 processors. In addition, the lock version results in a larger amount of code executed per atomic addition, and hence slower execution even on a small number of processors.

6.1.2 Quicksort

Figure 8 shows the speedups for Quicksort sorting 256K integers. The version of Quicksort using the synchronization queues outperforms that using locks to implement the queue. This is because of the reduction in the number of messages/second.

	Messages/second	Kbytes/second
ILINK-Add	222.8	133.3
ILINK-Local	340.7	131.3
Water-Lock	5518.8	601.5
Water-Add	957.6	906.6
Water-Local	2277.2	735.5
Qsort-Lock	567.9	657.5
Qsort-Queue	462.7	786.7
TSP-Lock	369.1	121.3
TSP-Min	355.2	119.6

Table 1 7-processor TreadMarks execution statistics

[3] We are unable to report results for 8 processors because of a hardware failure.

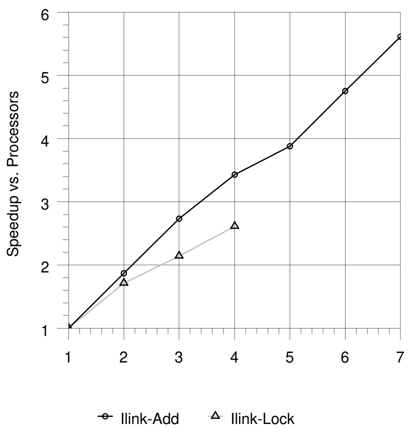

Figure 7 ILINK: CLP.

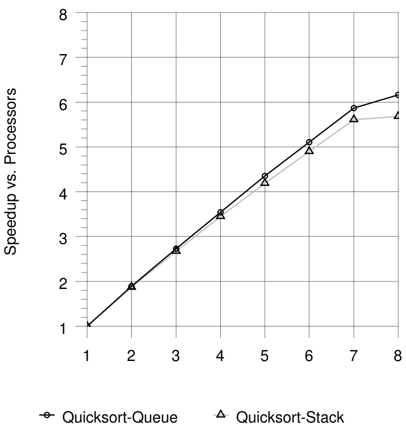

Figure 8 Quicksort: 256K integers.

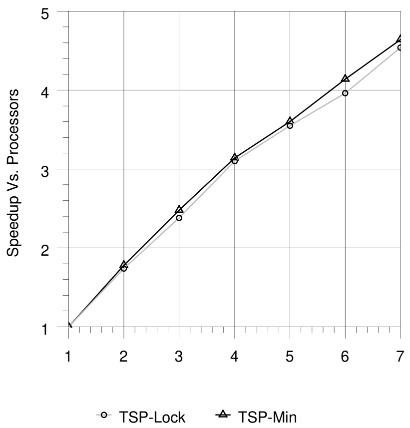

Figure 9 TSP: 18 cities.

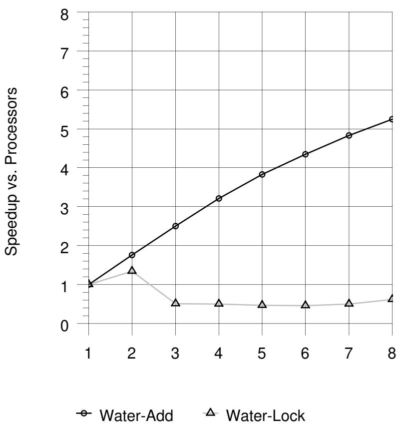

Figure 10 Water: 5 steps on 288 molecules.

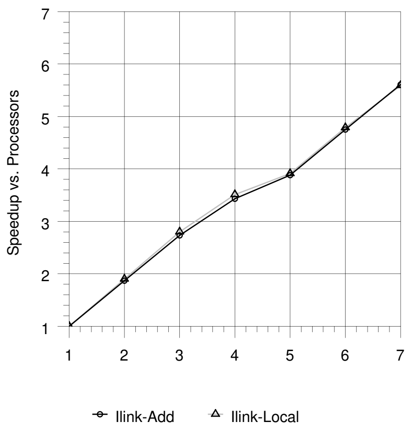

Figure 11 ILINK: CLP.

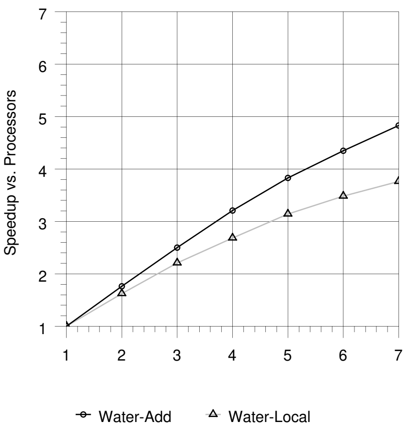

Figure 12 Water: 5 steps on 288 molecules.

The reduction in the number of messages/second in the queue-based implementation compared to the lock-based implementation occurs in part because false sharing causes less communication in the queue-based implementation. Consider the case where sections of the array that fall within the same page are modified and enqueued by two different processors before either is dequeued. In the queue-based implementation, the processor performing the dequeue synchronizes directly with the processor that performed the corresponding enqueue. As a result, it only sees the modification to one part of the page, and must only obtain a diff from that processor. In contrast, under the lock-based implementation, the dequeueing processor will see both modifications and will obtain diffs from both enqueueing processors even though it does not intend to access the other section of the array.

6.1.3 TSP

Figure 9 shows the speedups for TSP solving an 18 city problem. The speedups improve slightly using synchronizing atomic operators instead of locks to update the minimum bound. This is due to the reduction in consistency information generated as well as the reduction in the number of messages used to perform the atomic operation.

6.1.4 Water

Figure 10 shows speedups for Water executing 5 steps on 288 molecules. The lock-based implementation gets no speedup on TreadMarks, except on 2 processors, because the high rate of synchronization (1,540 remote lock acquires/second) causes many messages (6,161 messages/second). The majority of this synchronization are locks used to implement the atomic add.

The speedup for the version using `ns_atomic_adds` is 5.25 on 8 processors. The reduction in the number of messages/second and in the off-node synchronization rate contributes to the vastly improved speedups.

6.2 Hand-Optimized Versus High-Level Operations

6.2.1 ILINK

ILINK-Local is the hand-tuned version of the program that uses local variables to collect local accumulates, which are then summed into the global variables. Figure 11 shows speedups for ILINK-Local and compares them with speedups for ILINK-Add. As can be seen, ILINK-Add performs comparably with the hand-tuned version of the program.

6.2.2 Water

Water-Local is the hand-tuned version of Water. Similar to ILINK-Local, Water-Local also uses local variables for local accumulates, which are then summed into the global variables. Figure 12 shows the speedups for Water-Local and compares them with Water-Add. On 7 processors, Water-Local shows a speedup of 3.76, while Water-Add shows a speedup of 4.83. Thus, Water-Add outperforms even the hand-tuned Water-Local program. The improvement is due to the elimination of the local to global copy in addition to the removal of the lock operations required to perform these copies.

7 Related Work

There has been a lot of previous work on using programmer-provided information to lead to memory system optimizations (e.g., [1, 3, 4, 6, 8, 11, 16]). This paper builds on previous work to get more information from the programmer.

We replace the use of locks with high-level aggregate operations that make the intended use of the lock more explicit. In contrast, the work in [1, 6, 8, 11, 16] is at a lower-level, focusing on distinguishing individual memory operations based on their behavior in the execution.

Two previous studies that look at higher-level semantics are the following. The Midway system [5] and the entry consistency model [4] associate locks with data, thereby reducing the information that needs to be transferred on synchronization. Our synchronizing atomic operators can be interpreted as a similar construct. However, the key difference between Midway and our approach is that we do not restrict the programmer to use locks with the above semantics. Thus, Midway might sometimes require more synchronization than necessary (as in the task queue example where locks are intended to actually order operations not in the critical section). We do not require any more synchronization than would be needed in a sequentially consistent system.

Adve [3] discusses how system performance can be improved by making explicit many well-defined high-level programming constructs that have not been fully exploited by previous memory models. Some of those constructs are lock-based. That work, however, mainly focuses on developing a framework for when programmers can use such constructs while continuing to assume the model of sequential consistency; it does not include quantitative evaluations.

8 Conclusion

We advocate the replacement of locks in shared memory programs, to the extent possible, by higher-level primitives. The advantages of this approach are two-fold: it leads to simpler programs and to better performance. In particular, in software DSM systems, synchronization latency and communication cost are the primary sources of overhead. It is exactly these overheads that we are reducing by tailoring the implementation of the higher-level primitives according to their intended use.

We have identified three high-level primitives, non-synchronizing atomic operators, synchronizing atomic operations, and task queues. While by no means an exhaustive collection, these primitives can be used in a variety of programs. We have then evaluated the benefits of replacing locks by these primitives for a number of applications by measuring their performance on the TreadMarks software DSM system. The measurements confirm our hypothesis: the version of the applications written using the high-level primitives are simpler and perform better.

Future work will focus on a number of areas, including identification of other high-level primitives, the use of these primitives in other applications, and the implementation and benefits of using high-level primitives on architectures other than software DSM systems.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] S.V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin, Madison, December 1993.

[4] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[5] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[7] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[8] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

[9] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

[10] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

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

[12] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and S. P. Daiger. Non-syndromic cleft lip and palate: No evidence of linkage to hla or factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[15] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proc. Natl. Acad. Sci. USA*, 81:3443–3446, June 1984.

[16] David Probst. Programming, compiling and executing partially-ordered instruction streams on scalable shared-memory multiprocessors. In *Hawaii International Conference on System Sciences*, pages 504–513, 1994.

[17] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.