In *The ACM Transactions on Computer Systems (TOCS),* 18(1): 1-36, January 2000.

A High-Level Abstraction of Shared Accesses

Peter J. Keleher
keleher@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD 20742

Key words: shared memory, DSM, programming libraries, update protocols

We describe the design and use of the tape *mechanism, a new high-level abstraction of accesses to shared data for software DSMs. Tapes consolidate and generalize a number of recent protocol optimizations, including update-based locks and record-replay barriers. Tapes are usually created by "recording" shared accesses. The resulting recordings can be used to anticipate future accesses by tailoring data movement to application semantics. Tapes-based mechanisms are layered on top of existing shared memory protocols, and are largely independent of the underlying memory model. Tapes can also be used to emulate the data-movement semantics of several update-based protocol implementations, without altering the underlying protocol implementation.*

We have used tapes to create the Tapeworm synchronization library. Tapeworm implements sophisticated record/replay mechanisms across barriers, augments locks with data movement semantics, and allows the use of producer-consumer segments, which move entire modified segments when any portion of the segment is accessed. We show that Tapeworm eliminates 85% of remote misses, reduces message traffic by 63%, and improves performance by an average of 29% for our application suite.

1. Introduction

This paper describes the concept of *tapes* [21]: a new high-level abstraction that unifies the expression and implementation of a number of techniques for improving the performance of software distributed shared memory (SDSM) protocols. SDSM protocols support the abstraction of shared memory to parallel applications running on networks of workstations. The SDSM abstraction provides an intuitive programming model and allows applications to become portable across a broad range of environments. These environments can include clusters of inexpensive PC's and workstations, allowing a much better trade-off between price and performance to be achieved than with most hardware-supported shared memory machines. While SDSM systems have primarily been used as an effective way to obtain cheap cycles, they are also useful in integrating machines with important resources (access to a sensor or a database, for example) into computations running on other machines. Finally, SDSM provides a uniform shared memory abstraction over the small-scale multi-processors that are becoming common in labs and on desktops. However, this level of abstraction prevents the application from improving performance by explicitly directing data movement. While it is relatively easy to get parallel applications working on current DSMs, it can be very difficult to achieve high performance.

Tapes can make this task easier by allowing the data movement to be directed by the application at a high level of abstraction. A tape is essentially an object that encapsulates an arbitrary number of updates to shared data. Tapes are created through calls to the tape library that start and stop recording of updates to shared data made by the local process. Once created, a tape provides a convenient way to manipulate the updates. The data referenced by a tape can be sent to another process. Tapes can be reshaped by changing the set of data to which they refer. Tapes can also be added and subtracted, allowing a single tape to describe any arbitrary set of updates.

As a quick example, Figure 1 shows a simple use of the tape mechanism. We defer detailed description of this example until the next section. Essentially, however, the example shows process P_1 modifying three shared pages while holding lock L_1, followed by P_2 acquiring the same lock and reading the same three pages.

In a traditional invalidate protocol, P_1's modifications would cause all three pages to be invalidated at P_2. The subsequent reads by P_2 would each cause remote page faults. Each fault is satisfied by retrieving a current copy of the faulting page from a remote processor, and hence implies at least one network RPC. After the data is returned and copied to the correct location, page protections are changed to allow the page to be accessed normally.

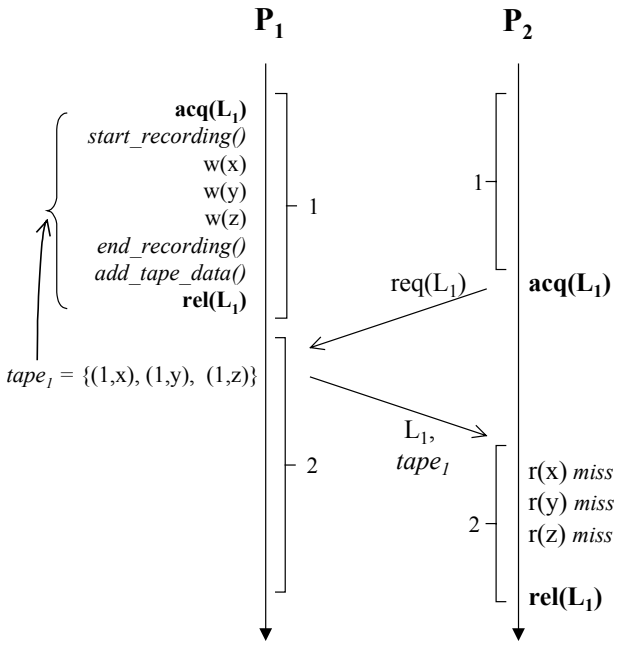

Figure 1: Tapes: *tape*$_1$ describes the writes performed by P_1. Subsequent misses by P_2 can be avoided if the tape, together with the data it describes, is transferred with the lock.

By including the code in italics, however, P_1 can record the accesses automatically, append the modified data to the lock grant message, and *update*, rather than invalidate, P_2*'s* copy of the page. For each page fault thereby avoided, the system eliminates both local fault-handling overhead and network RPC's.

The key points of this example are the following. First, tapes allow sharing behavior to be captured at runtime. The system needs neither compiler cooperation nor extensive user interaction in order to determine exactly which pieces of shared data are accessed by P_1. This is important because we do not assume any explicit associations between synchronization and shared data, just as no such associations are assumed in a typical multi-threaded environment like Pthreads.

Second, moving the data with the lock is only a performance optimization, it can not cause correctness to be violated. No damage is done if P_2 does not access either *x*, *y*, or *z*. Any additional pages accessed by P_2 will be demand-paged across the network when the pages are accessed.

While tapes could be used directly by applications, they are probably more useful when folded into specialized synchronization libraries. Such libraries can reduce the total application involvement to just the replacement of calls to generic synchronization primitives with calls to the corresponding routines in the new libraries. This indirection allows the synchronization implementation to be quite simple, without losing any generality. On the other hand, sophisticated middleware or application programmers can use tapes abstractions to directly improve performance.

The primary claimed advantage of SDSM systems over message-passing programming models is ease of use. By abstracting away any need to specify data locations, SDSM systems allow parallel and distributed applications to be more simply created. Requiring applications to contain additional annotations would seem to run counter to this goal. However, synchronization libraries can hide the mechanism from programmer view. The only change needed to use tape mechanisms in these cases is linking with a different library. Moreover, tape mechanisms can be added to applications incrementally. Applications can be developed and tested without tapes. Since tape mechanisms do not affect correctness, adding tape calls can not break any application that has already been debugged.

We used tapes to implement Tapeworm, a new synchronization library that is layered on top of existing consistency and synchronization protocols in CVM [18], a software distributed shared memory system. The use of tapes allowed us to write Tapeworm in fewer than 400 lines of C++ code. At the same time, Tapeworm is able to track and use very sophisticated data movement patterns. Specifically, Tapeworm augments ordinary locks to include data movement semantics in addition to synchronization semantics. Tapeworm also supports producer-

consumer regions and record/replay barriers. Record/replay barriers use recordings of data accesses from one iteration of an application to anticipate accesses during future iterations.

Overall, Tapeworm eliminates an average of 85% of data misses on our suite of applications. The reduction in misses translates into a reduction in message traffic of 63%, and an average improvement in overall performance of approximately 29%.

The rest of the paper is as follows. Section 2 discusses the high-level semantics of tapes in a protocol-independent fashion. Section 3 describes the protocol-independent interface to Tapeworm, a high-performance synchronization library built using tapes. Section 4 describes the requirements that the tapes abstraction, and Tapeworm in particular, make on the underlying consistency protocols, and Section 5 describes Tapeworm's performance. Section 6 describes the use of tapes in emulating update-based protocols such as home-based LRC [36] and scope consistency [15]. Section 0 describes related work, and Section 8 concludes.

2. Tape semantics

A tape is a recording of shared accesses. Clearly, a tape containing a record of all accesses to shared memory could not be implemented efficiently in software. Hence, accesses must be manipulated in coarser units. We assume that the underlying constancy protocol aggregates accesses by taking advantage of both spatial and temporal locality in the application. Spatial locality is exploited by grouping all accesses to the same object or page into a single unit. Hereafter, we will refer to this unit as a *page*, but it could refer to any systematic grouping of consecutive addresses. Temporal locality is exploited by dividing each process's execution into distinct intervals, each of which is labeled with a system-unique interval id. The exact method by which intervals are defined is not important, although most protocols will probably delimit intervals by synchronization events. For example, each of the processes in Figure 1 has two intervals, delimited by synchronization accesses to lock L_1. These optimizations allow a tape to be constructed from lists of modified pages during distinct intervals, instead of addresses and cycle counts.

More specifically, a tape consists of a set of *events*, each of which is a 3-tuple (x,y,z), where x is an interval id, y is a set of page id's, and z is a processor id. The processor id is not shown in the text below where it can be derived from context. We assume here that such events only correspond to write operations, but we extend the discussion to reads and requests below. Hence, $tape_1$ in Figure 1 consists of the three events $\{(1,1), (1,2), (1,3)\}$. Note that the event (and the tape) consists only of the tuple, it does not contain the actual modifications. The actual modifications are tracked by the underlying protocol.

Tapes can be created in several different ways, but the primary method is that shown in Figure 1, e.g. recording accesses over a period of time. This method of creating tapes enables synchronization protocols to capture dynamic access patterns at runtime, rather than relying on the programmer or compiler to derive complete information statically.

A second method of creating tapes is for them to be generated by hooks into the underlying consistency protocol. While we defer full discussion of the interface to the underlying protocol until Section 3, `hole_tape(Extent *)` is fundamental to some of the interfaces discussed in the next section. Its function is to create and return a tape that describes all updates needed to validate the region of memory described by an *extent*. A shared page is *validated* by applying all updates necessary to bring the page up to date.

Extent is short for "data extent." An extent is merely a list of pages. Extents are useful when the full information encoded in a tape is not needed. For example, if a synchronization interface needs to know the set of pages modified by a process while a lock is held, a tape is created by recording accesses during the synchronized period. The tape is then projected into an extent listing the pages accessed by the tape's events by removing all information from the tape's tuples except page id's.

There are three tape variations. The canonical form is a *write tape*, created primarily by recording write accesses. *Read tapes* are created by recording read accesses. Finally, *request tapes* can be created by recording data requests received by the local node. Request tapes can be used to locally obtain information about the data accessed by other nodes. Unlike read and write tapes, request tapes are not complete. They do not describe all accesses made by a node at any specific time. Nonetheless, they can be a cheap and useful way of obtaining information about remote accesses without explicitly requesting it.

Once a tape has been created, it can be transmitted to remote sites, projected into an extent, pruned to contain only notices that pertain to a given extent, or added to another tape. Most importantly, the tape can be used to request a set of pages or updates that will soon be needed locally, *before* the data is needed.

The approach shown in this example has several advantages over other approaches described in the literature. Simple update protocols push modified data to existing replicas to *update* them, rather than to *invalidate* them. The advantage of such protocols is that subsequent page faults are avoided, but the lack of any selectivity often causes update protocols to move far more data than invalidate protocols. Several

```
class Extent : UniqueIntegerSet {
    Extent();
    void    reset();
    int     empty();
};

class Tape {
    Tape()
    void    reset();
    void    start_reading(), stop_reading();
    void    start_writing(), stop_writing();
    void    start_requesting(), stop_requesting();
    void    pause(), unpause();
    Extent  *project_data();
    Extent  *project_processor();
    Extent  *project_intervals();
    void    operater += (Tape *);
    void    operater += (Extent *);
    void    operater -= (Tape *);
    void    operater -= (Extent *);
    int     populate(char *);
    void    apply();
};

Tape        *hole_tape(char *, int);
Tape        *weak_mods_tape();      /* specific to weak consistency protocols */
void        register_fault_callback(FUNC_PTR func, int type);
void        register_request_callback(FUNC_PTR func, int type);
```

Figure 2: Tape class definition and support routines

researchers have described more selective update protocol variants [3, 32, 33] that might also suffice in this example. However, these protocols effectively encode expected sharing behavior into the underlying protocol. By making such expectations part of the programmable protocol interface, the tape mechanism has far more flexibility.

2.1 Operations

As mentioned above, tapes can be added, subtracted, and projected to extents. We discuss allowable operations in more detail in this section. Recall that a tape, T_a, is an unordered set of 3-tuples, each of which contains an interval id, a page id, and a process id: (v_a, m_b, p_c). Addition is a simple set operation:

$$T_a + T_b = \{x \mid x \in T_a \vee x \in T_b\} \quad (1)$$

Subtraction is similar:

$$T_a - T_b = \{x \mid x \in T_a \wedge x \notin T_b\} \quad (2)$$

Projection is used to extract information from a single tape dimension, such as the set of pages accessed. Projection of a single 3-tuple consists of extracting the appropriate index. We denote extracting the second index of a 3-tuple as follows:

$$\pi_2(a,b,c) = b \quad (3)$$

Projection of a an entire tape into a single dimension (either interval indices, page ranges, or processor lists), can be defined as follows:

$$\Pi_2(T_a) = \{b \mid \exists a,c \text{ where } (a,b,c) \in T_a\}$$

Such a projection defines an extent, either *temporal*, *spatial*, or *processor*.

The main use of extents is in pruning other tapes. For example, consider a static, iterative, three-process application where producer P_0 repeatedly modifies two chunks of data. One piece of data is read during the iteration after it is produced by P_1, and the other by P_2. Assume that P_0 creates a write tape, T_0, and P_1 creates a read tape, T_1, during iteration i. The set of data that will be accessed in iteration $i+1$ by P_1 is a subset of the data named by P_0's iteration i write tape. More specifically, the data that will be needed by P_1 consists of all the data named in T_0 that pertains to the pages mentioned by T_1. We can describe this data formally as follows:

$$\Pi_2(T_0 / T_1) = T_0 - \{(a,b,c) \mid b \notin \Pi_2(T_1)\}$$

The resulting tape can be used to request precisely the page needed by P_1 in iteration $i+1$.

Extents can be added to and subtracted from tapes. Adding an extent to a tape prunes the tape to

only include tuples corresponding to elements of the extent:

$$T + E_{data} = \{(a,b,c) \mid (a,b,c) \in T \wedge b \in E_{data}\} \quad (6)$$

Analogous equations can be defined for adding processor and interval (temporal) extents, but they are not currently defined in our implementation. Subtracting is the converse; the resulting tape is pruned of all 3-tuples corresponding to the extent:

$$T - E_{data} = \{(a,b,c) \mid (a,b,c) \in T \wedge b \notin E_{data}\} \quad (7)$$

2.2 Class definition

Figure 2 shows the generic, protocol-independent interface to the tape abstractions. The API includes class definitions for the Extent and Tape data types, together with two support routines. The `Extent` class is based on a generic set type, with the exception that all elements of an extent are unique scalars. This allows extents to summarize data, processor, or temporal extents equally well.

The `Tape` class consists of routines to start and stop recording each of the three event types, routines to implement the operations defined above, and two routines to handle the data referred to by the tape. `Tape::populate()` copies the tape, together with all data referred to by the tape, to the specified location and returns the number of bytes copied. `populate()` is used to copy data into a message for transmission to another processor. `Tape::apply()` is used to copy the data back out, and to integrate it into the local processor's view of shared data. Note that only local data can be used to populate a tape, so the resulting copy might not contain all of the data described by the tape.

`hole_tape()` is used to create a tape that describes all updates needed to validate a region of the shared segment, and was introduced above. `weak_mods_tape()` is the sole element of this API that is specific to weak consistency protocols. This routine recognizes that tapes, and the data that they describe, are often piggybacked on top of existing synchronization messages. For weak consistency [11] and the many variations of release consistency [14, 19], *release* synchronization messages contain invalidations for the destination. The `weak_mods_tape()` call creates a tape describing these invalidations so that the corresponding data can be included and the invalidations avoided. The routine always returns null if the underlying consistency protocol does not distinguish regular shared accesses from synchronization accesses.

The `register_fault_callback()` and `register_request_callback()` calls are routines used to register functions to be called back when local page faults and requests from remote sites, respectively, occur. The former are used to track local shared accesses, and the latter are used to intercept data requests coming from remote processors. `register_fault_callback()` will normally be used only by the tape support library in implementing the tape recording functions. The latter is used in the implementation of producer-consumer regions discussed in the next section.

3. The *Tapeworm* library

We have implemented a tapes abstraction layer and the *Tapeworm* synchronization library on top of CVM [18], a software DSM that implements a multi-writer form of lazy release consistency (LRC) [19, 36]. CVM supports a single thread or process per machine, a shared segment that can be transparently accessed by any of the processes, and a set of simple synchronization mechanisms. Synchronization is implemented in addition to, rather than on top of, the consistency mechanism.

Tapeworm's application interface consists of three synchronization operations that we found useful for our application suite: *record-replay barriers, update locks*, and *producer-consumer regions*. All three operations have precedent in the literature. Our intent is to show the flexibility, expressiveness, and power of the tapes abstraction. In all cases, we rely on the underlying protocol layer to insure correctness, regardless of when data arrives. Section 4 describes the demands that this requirement places on the underlying protocol. We avoid details specific to any one underlying system whenever possible, but the synchronization routines must interface directly with communication routines in order to avoid redundant messages and extra copying. We tried to make the names of these hooks and interfaces as self-explanatory as possible, and use pseudo-code when it does not obscure important details.

3.1 Record-Replay Barriers

The most simple way in which we expect tapes to be used is in *recording* data movement in the first iteration of an iterative scientific application and *replaying* it in future iterations. Much of the remote latency can be hidden by sending the data before it is needed. Figure 3 shows pseudo-code for a simple grid application. Each process iteratively computes new values for all of the elements that it owns, using barriers and a temporary array to synchronize the read and write accesses to the shared array.

```
while (TRUE) {
        tape_barrier();
        forall i,j {
                temp[i][j] = arr[i-1][j] + arr[i+1][j];
        }
        tape_barrier()
        forall i,j {
                arr[i][j] = temp[i][j];
        }
}
```

Figure 3: Red/black stencil

The only difference between this code and code written for a non-Tapeworm system is that the barrier calls are bound to specialized versions, rather than to the generic `cvm_barrier()`.

Pseudo-code for each process's barrier routine is shown in Figure 4. The purpose is to selectively send updates to remote processes before they are requested. Each process identifies data to be flushed to other processes by crossing the set of locally-created modifications with the set of data requested by other processes, and assuming that sharing patterns are static.

Each process records locally-created modifications, as well as data requests from other processes. This allows a process to directly track the data that will be needed by other processes during the next iteration. Tracking writes allows a process to identify new local modifications. Crossing such requests with the tape of local modifications allows us to create descriptions of the data that needs to be sent to other processes.

In more detail, each process uses `writeTape` to record local writes and `reqTape` to record requests during any single iteration. The `reqExtents[]` array is used to hold the set of all pages that each process has ever requested. The barrier procedure starts by projecting the remote request tape to sets of pages requested by each remote process. Each such set is unioned with all previous pages requested by that process. The tape of local writes is then crossed with each such set to create a new tape naming the set of modifications that needs to be flushed to the corresponding process. Each tape is created by adding the write tape to the extent describing the pages that have been requested by that processor, as defined by Equation 6. For each such tape that is non-empty, a message is created, populated with the tape, and sent to the corresponding process.

This code assumes static access behavior. Applications with dynamic sharing patterns will only benefit to the extent that there is overlap between the sets of data accessed by consecutive iterations. Record/replay barriers for dynamic sharing patterns would only maintain extent information about recent iterations, rather than about all as in the static case.

```
Tape        reqTape;      /* records requests from other procs */
Tape        writeTape;    /* records local writes */
Extent      reqExtents[NUM_PROCESSES];

tape_barrier()
{
    writeTape.stop_writing();
    reqTape.stop_reading();

    for proc in (all processes) {
        reqExtents[proc] += reqTape.project_data(proc);

        Tape *out = writeTape + reqExtents[proc];
        if (out) {
            create flush message
            int     len = out->populate(msg->curr_buf);
            msg->add(len);
            send flush message to proc
        }
    }

    barrier();

    reqTape.reset();     reqTape.start_reading();
    writeTape.reset();   writeTape.start_writing();
}

void flush_handler(Msg *msg)
{
    Tape    *tape = msg->get_tape();
    tape->apply();
}
```

Figure 4: Record/replay implementation

3.2 Update locks

Update locks are modifications of the globally exclusive locks common to many parallel programming environments. Update locks use tapes and extents to combine data movement with synchronization transfers. Rather than using separate protocol transactions for synchronization and for data, update locks attempt to piggyback the data movement on top of existing synchronization messages. Tapes and extents are used to identify and communicate the updates that are needed to validate shared data. This section discusses two variants: *auto-locks*, which gather and use access information automatically, and *user locks*, where an explicit API is presented to the user.

Auto-locks

Auto-locks attempt to exploit static access patterns by using past behavior to predict and eliminate memory faults during later lock synchronizations. The assumption is that the set of pages accessed during the $n+1^{th}$ acquire of any lock is similar or identical to the set of pages accessed during the n^{th} acquire of the same lock. Hence, we can avoid remote faults by ensuring that the pages accessed during the last lock

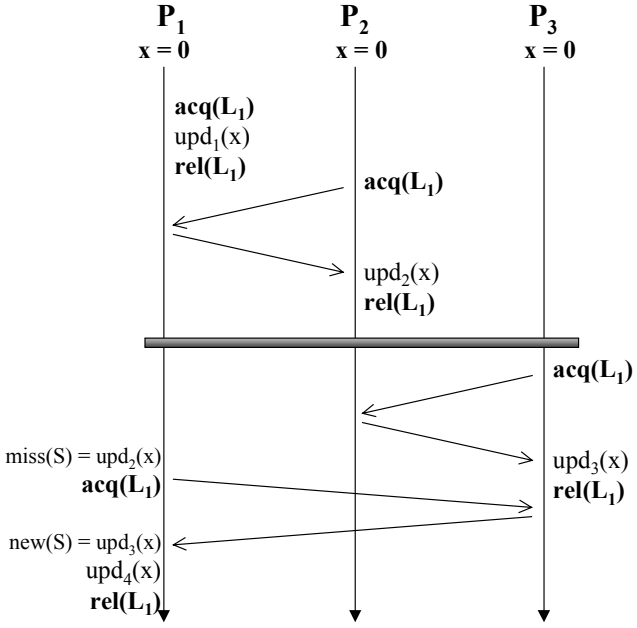

Figure 5: Auto-locks

acquisition (of the same lock) are valid when the lock acquisition is accomplished.

There are two sets of updates that need to be retrieved in order to prevent these remote faults. Let *S* be the set of pages that the requestor will access while holding the lock. This is the set of pages that the auto-lock mechanism will attempt to validate. The necessary updates can be divided into `miss(S)` and `new(S)`. `Miss(S)` consists of updates known (but not present locally) before the lock grant returns, whereas `new(S)` consists of new updates learned from information piggybacked on the lock grant. The former set is empty if the pages in S are all valid when the lock acquisition begins.

Consider the example in Figure 5. For the sake of simplicity, assume that *x* is a single page. Prior to performing its second lock acquisition, P_1's copy of page x is invalid because the preceding barrier disseminated an invalidation resulting from P_2's update. P_1's `miss(S)` therefore consists of $\texttt{upd}_2\texttt{(x)}$.

The `new(S)` set is needed because weakly-consistent protocol implementations often append consistency information to existing synchronization communication. In Figure 5, the lock grant at P_1's second lock acquisition returns knowledge of a third update, $\texttt{upd}_3\texttt{(x)}$. Hence, this latter update constitutes `new(s)` at the lock grant. All of the updates in either set are needed in order to validate the pages in *S*.

Figure 6 shows the code used to implement auto-locks in Tapeworm, lacking only comments and error-checking code. Each of the five routines is an upcall from the underlying implementation into the protocol code. The first four execute on the requestor's side, the last is executed by the previous holder of the lock. Tapeworm is implemented as part of a tapes protocol that specializes the default multi-writer LRC protocol. Therefore, all upcalls from CVM first call the Tapeworm routines, and then fall through to the corresponding LRC routines that maintain memory consistency.

The data structures consist of `writes`, a tape used to record local modifications to shared memory, and `lockExtent`, an extent used to remember the set of pages accessed the last time the lock was held. The code starts recording modifications in `lock_entry()`, and stops in `lock_release()`.

The `add_to_lock_request()` routine is called just before the lock request messages are sent. The auto lock routine adds to this message an extent and a tape. The extent is derived from the `writes` tape created during the previous lock access. The tape, created by `hole_tape()`, names all updates needed in order to validate the region covered by the extent. In other words, if page *x* of the extent's region is currently invalid, the tape specifies all updates that need to be applied to *x* in order to re-validate it.

The routine `add_to_lock_grant()` is called by the lock granter. This routine first retrieves `miss(S)` from the message and then creates `new(S)` by pruning `weak_mods_tape()`the extent sent in the request. These two tapes are added together, potentially resulting in a tape that includes modifications from several different processes. Finally, `populate()`is used to load the tape data into the reply.

The requesting process uses `apply()` to read and apply all updates from a message. If all has gone well, `apply()` will also re-validate the entire shared region named by `lockExtent`.

User Locks

The second type of update locks, *user locks*, replace the implicit arguments of auto-locks with explicit buffer and length arguments. User locks are useful when the shared data accessed while a specific lock is held will change in some well-known manner. The interface to user locks include a simple buffer pointer and length. These parameters allow the program to specify a single contiguous section of shared memory that is likely to be accessed while the lock is held. Inside the lock operator, the region is converted to an extent, which provides an efficient and portable representation of the set of pages covered by the region.

```
Tape      writes;              /*  per-lock tape */
Extent    lockExtent;

/* executed by prospective holder of the lock */
void Tapeworm::lock_entry(int id)
{
      writes.reset(); writes.start_writing();
}

void Tapeworm::add_to_lock_request(Msg *msg, int id)
{
      Tape *empty = tape->hole_tape(lockExtent);
      msg->add(tape, (char *)empty, empty->size());
      msg->add(type_extent, (char *) lockExtent, lockExtent ->size());
}

void Tapeworm::read_from_lock_grant(Msg *msg, int id)
{
      Tape *in = msg->get_tape();
      in->apply();
}

void Tapeworm::lock_release(int id)
{
      writes.stop_writing();
      lockExtent = writes.project_data();
}

/* executed by last holder of the lock */
void Tapeworm::add_to_lock_grant(Msg *msg, int pid)
{
      Tape              outTape;

      outTape = *(Tape *)msg->retrieve(type_tape)) {
      if (extent = msg->retrieve(type_extent)) {
            outTape += weak_mods_tape(pid, extent);
      }

      int    len = outTape.populate(msg->curr_buf);
      msg->add(len);
}
```

Figure 6: Auto-lock implementation

This extent is used to create `miss(S)` as above. It is also appended to the lock request in order to identify `new(S)`. These quantities are handled similarly to the corresponding quantities in auto-locks.

We look at user locks primarily in order to have a point of comparison when evaluating how well the auto-locks are able to correctly predict and anticipate shared accesses.

3.3 Producer-consumer regions

Many applications exhibit producer-consumer interactions. In these applications, one process *produces* a region of memory that is *consumed* by another process at an arbitrary time later. These types of communication are difficult to anticipate because the producer-consumer connections are often dynamic and can have low locality. If such regions are multiple pages, the consumer usually must fetch updates to each page separately, as the pages are accessed.

Tapes and extents can be used to aggregate these transfers by recording writes at the producer end, projecting the resulting tape to an extent, and storing it with the region pointer. When a process subsequently consumes the data by removing the pointer from the central repository, it also retrieves the corresponding extent.

Figure 7 shows the implementation of producer-consumer regions in Tapeworm. The application registers the region by bracketing its writes with `start_produce()` and `end_produce()` calls. In addition to stopping the recording, the latter enters the resulting tape into an ordinary queue. CVM first vectors page fault requests to the tape protocol, pro-

```
start_produce()
{
      tape.reset();     tape.start_writing ();
}

end_produce()
{
      tape.stop_writing ();
      queue.add(tape);
}

producer_region(int pg_id, Msg *msg)
{
      if (Tape *tape = queue.search(pg_id)) {
            int len = tape->populate (msg->curr_buff);
            msg->add(len);
      }
}

register_request_callback(producer_region-region, 0);
```

Figure 7: Producer-consumer regions

viding an opportunity to search the queue for a tape that contains the requested page. If the page is found, the entire region's data is appended to the reply message. While the total data transferred is the same as if the pages were transferred one at a time, the benefits of aggregating multiple requests into one can be significant.

4. Tape implementation

While tapes are conceptually independent of both the programming model and the particular protocol implementation, the underlying consistency protocol and system architecture must provide basic support. Our tapes implementation is layered on top of CVM [18], a software DSM that supports multiple protocols and consistency models. CVM is written entirely as a user-level library and runs on most UNIX-like systems. CVM was created specifically as a platform for protocol experimentation.

New CVM consistency protocols are created by deriving classes from the base *Page* and *Protocol* classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized commercial system running a similar protocol. Although CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration, we use neither of these techniques in this study. Tapeworm is a subclass of LmwProtocol, which is derived from the base Protocol class. LmwProtocol is the base multi-writer LRC protocol used by both CVM and TreadMarks [3].

Function	Explanation
Interval specification:	API for specifying interval boundaries.
Access recording:	Between start() and end() calls, return a list of all pages either read, written, or requested.
"Hole" tape creation:	Create a list of data/pages needed to validate a given address range.
Data instantiation:	Retrieve local data/pages corresponding to a tape.

Table 1: Required protocol functionality

4.1 Interactions with the consistency protocol

A tapes layer logically resides on top of existing consistency and synchronization protocols. Conceptually, at least, the tape mechanism is independent of both the underlying protocol implementation, and of the precise application access orderings that are being captured. Tape semantics have been carefully crafted to accommodate multi-writer relaxed consistency models. However, they also work well with less complicated single-writer protocols.

Tapes place four requirements on consistency protocols, as summarized in Table 1. First, process executions must be divided into the intervals discussed above. The actual division into intervals is best left to the tape mechanism, as the synchronization libraries built using tapes will tend to have more information about application semantics than the underlying protocols. However, the consistency protocol must provide a means for a tapes layer to inform it that a new interval has begun.

Second, the consistency protocol must provide a way to record shared accesses. We divide this process into *write trapping* and *write collection* [2]. Write trapping refers to the process of tracking shared accesses. SDSM systems have to track shared accesses in order to determine when to make pages valid, and when to move data. Most page-based systems, including CVM, use the virtual memory system in order to trap accesses. Address ranges for which there is no valid local copy are marked unreadable. Any access to such a page leads to an SDSM handler being called, which notices that the page is being accessed. Similarly, writes are trapped by making pages unwritable by default. The first write access to any such page causes an SDSM handler function to be called. The handler sets things up so that the write can proceed, but also notes that the write took place. A second approach to write trapping is through the use of software dirty bits [5, 26]. This approach consists of modifying all shared writes to also set a bit that can later be used to track which locations were

written. Write collection refers to the mechanism used to marshal shared modification into a network message, and unmarshalling the data at the other end. Many systems marshal and unmarshall data merely by copying the entire page or object into and out of messages. However, multi-writer protocols usually create some form of *diff*, which encodes only those portions of the page or object that were modified. Diffs are usually smaller than the entire object, and allow multi-writer protocols to allow simultaneous concurrent write accesses to the same object.

For a given interval, the consistency protocol must be able to provide a list of all pages that have been written, and all pages that have been read. Assuming a consistency protocol based on virtual memory, most of the accesses can be tracked with little or no overhead. An ordinary read fault implies that the corresponding page is being read, and an ordinary write fault indicates that a page is being written. For example, we know that any readable, but not writable, page is not being written. When recording only write accesses, therefore, we merely need to note any such page that takes a write fault. The tape support layer can track these by using the `register_fault_callback()` routine described in Section 2.

Only tracking existing faults will not catch accesses to pages for which permissions are sufficient. In order to record all read accesses, for example, read permission must be removed from all currently-readable pages. The first subsequent read access to a given page will generate a fault, which allows the access to be recorded. These *recording faults* are handled by noting the page access and restoring the original page protection; no network communication occurs, and subsequent accesses proceed without faults. When recording ceases, the original page protections must be restored for any page that did not have a recording fault. The overhead of recording is therefore two protection changes, and potentially one local recording fault, per page that is already in the desired mode. Note that protection changes can often be combined in order to aggregate the cost of kernel calls. Request tapes are created by recording the sequence of incoming requests.

Recording faults can be avoided entirely by recording only normal consistency faults, i.e. not invalidating pages that are already in the target protection. This approach is less complex, and can be implemented with little or no runtime overhead. This is the approach used in CVM's tape support. The disadvantage, of course, is that it provides less complete information. Note, however, that recording is usually used to predict and anticipate future misses. If no miss occurs for a page during iteration *i*, a miss is unlikely to occur during iteration *i+1*. If it does, the faulting page can easily be added to the pages that caused misses during the previous miss. The only cost is that a single miss for that page went unanticipated.

The third functionality required of the consistency protocol is to provide a means of describing the data needed to revalidate a "hole" in the shared segment, i.e. all data in a consecutive range of virtual addresses. For example, the programmer might know that a given object will be read soon. The base and length of this object is passed to the consistency protocol, which returns a list of tuples describing the modifications needed to validate the object.

Fourth, the consistency protocol must provide a means of instantiating the updates described by a tape, and of applying this data at a remote location. For example, the above interface can be used to create a tape describing all updates needed to validate an object. This tape can be sent to the process that created those updates. This creating process needs a means of turning a description of the updates, i.e. the tuples in the tape, into the actual data, which is then returned to the requester in a message. At the requesting site, the incoming data needs to be reintegrated into the consistency protocol's view of the shared segment.

Finally, the producer-consumer interface defined in Section 3 requires a callback when a page request arrives. The first row of Table 2 shows the low-level hooks, or callbacks, required by the tapes layer. `Protocol::fault()` and `Protocol::page_request()` are upcalls from the SDSM to a protocol, in this case Tapeworm. Tapeworm specializes these calls to track accesses to shared pages. `Protocol::fault()` is called at local accesses to pages with the wrong permissions, i.e. reading an invalid page or writing a page without permission. Tapeworm uses this call to track reads and writes to shared pages. The `Protocol::page_request()` function is called when a remote site requests local data. This is used both for tracking requests (as with record/replay barriers), and for identifying and handling accesses by a consumer to producer/consumer regions.

4.2 Interactions with the message subsystem

Table 2 summarizes the required interfaces to the messaging subsystem. Independent of the consistency protocol, Tapeworm must also have access to the messaging layer in order to add and retrieve data to existing messages, as well as to create Tapeworm-specific messages. The calls `msg->add()` and `msg->retrieve()` allow arbitrary data to be added and retrieved from CVM *Msg* objects. While *Msg* objects are specific to CVM, the same function-

Interface	Description
*msg->add(msg_type, char *, int)* *msg->retrieve(msg_type, char **, int *)*	Allows arbitrary data to be added and retrieved from Msg objects. 'curr_buf' assumed if ptr omitted.
Protocol::fault(int pg) *Protocol::page_request(Msg *, int)*	Upcalls to Tapeworm for local page faults and requests for local data from remote sites.
*Protocol::add_to_lock_request(Msg *, int)* *Protocol::add_to_lock_grant(Msg *, int)* *Protocol::read_from_lock_request(Msg *, int)* *Protocol::read_from_lock_grant(Msg *, int)*	Allows data to be piggybacked on top of existing synchronization messages. Barrier routines are analogous.

Table 2: Low-level hooks and messaging support

ality could be made available without reference to specific message objects. However, this method would be less clear, and we have therefore left the interface unchanged.

The last row of Table 2 shows upcalls from the SDSM system to the consistency protocol. These are intercepted by CVM to provide hooks into existing messages. By adding data to these messages, Tapeworm can often avoid creating messages itself.

5. Performance evaluation

This section describes the performance of several applications, both with and without the use of Tapeworm's new synchronization primitives. Section 5.1 describes our experimental environment and Section 5.2 gives an overview of our application suite. The rest of the subsections describe the impact of Tapeworm on performance. Since each application was chosen to provide a different challenge to the synchronization library, we describe our results one application at a time rather than all at once.

5.1 Experimental environment

We ran our experiments over CVM's lazy multi-writer protocol on an eight-processor IBM SP-2. Each node is a 66.7 MHz POWER2 processor. The processors are connected by a 40 MByte/sec switch. The operating system is AIX 4.1.4.

CVM runs on UDP/IP over the switch. Lock acquires are implemented by sending a request message to the lock manager, which forwards the request on to the last requester of the same lock. This takes either two or three messages, depending on whether the manager is also the last owner of the lock. Two-hop lock acquires take 779 μsecs, while three-hop lock acquires take 1185 μsecs. Simple page faults across the network require 1576 μsecs. Page fault times are highly dependent on the cost of mprotect calls, 15 μsecs, and the cost of handling signals at the user level, 120 μsecs. Minimal 8-processor barriers cost 1176 μsecs.

App.	Input Set	APIs Used	Improvement			
			Speedup	Msgs	Misses	Bytes
Water	5 iters, 512 mols	lock, bar	14%	42%	83%	0%
TSP	18 cities	lock	7%	79%	94%	9%
Spatial	5 iters, 1024 mols	lock, bar	41%	96%	100%	15%
QS	$1x10^6$	lock, p-c	49%	53%	88%	0%
Gauss	1024 x 1024	flush	25%	67%	100%	2%
Barnes	8192 bodies	bar	40%	75%	48%	-2%

Table 3: Application Summary

5.2 Application suite

Our application suite consists of one branch-and-bound lock application, TSP, one producer-consumer divide-and-conquer application, QS, two applications that combine both locks and barriers, Water (Water-Nsquared from SPLASH-2 [34]) and Spatial (Water-Spatial from SPLASH-2), one tree-structured barrier application, Barnes (also from SPLASH-2), and gauss (gaussian elimination with partial pivoting). While these applications are meant to be in some sense "representative," their more important common attribute was that each had characteristics that illustrate one or more facets of tape behavior. Note that there certainly exist applications for which tapes do not improve performance. Performance can even degrade if the access patterns assumed by the tape mechanisms called by an application do not match the actual sharing patterns in the application.

Table 3 summarizes our applications and the maximum performance improvements on each. Details of the algorithms are deferred until the discussion of each application's performance. Overall, the best combination of options for each application eliminated an average of 85% of all remote page misses, 63% of all messages, and an average increase in speedup of 29%. For iterative programs, e.g. Barnes, Spatial, and Water, only the second and subsequent iterations were measured, in order to eliminate effects caused by the initial data distribution.

Protocol	Speedup	Remote Misses	Lock Pages	Updates Used	Comm KBytes	Messages				
						Lock	Barrier	Flush	Data	Total
Default	5.66	4852	0	-	6697	2786	196	0	4878	7860
Rec/Rep	5.78	3405	0	71%	6761	3016	196	420	3415	7047
User Locks	5.93	4336	1579	60%	6852	2642	196	0	4348	7186
User + Rec/Rep	6.14	1874	1550	64%	7736	2720	196	924	1950	5790
Auto-locks	6.16	3200	1566	70%	6683	2550	196	0	3200	5946
Auto + Rec/Rep	6.43	841	1535	68%	6655	2592	196	924	852	4564

Table 4: Water

Protocol	Speedup	Remote Misses	Lock Pages	Updates Used	Comm KBytes	Messages			
						Lock	Barrier	Data	Total
Default	7.02	6058	0	-	6860	1124	28	6060	7212
User	7.22	4297	6161	88%	6648	1142	28	4272	5442
Auto-locks	7.48	387	6120	68%	6249	1134	28	387	1549

Table 5: TSP

5.3 Application performance

Water

Our first application is Water, an iterative molecular simulation. Water alternates phases in which locks are used and phases in which barriers are the only synchronization. Table 4 shows the performance of Water with no tape optimizations, with record/replay barriers, with user locks, with automatic locks, and with both types of locks plus record/replay barriers. “Speedup” is relative to the single-processor time without CVM overhead. “Remote Misses” is the number of remote page faults incurred. “Lock Pages” is the number of pages that are re-validated by data moved as a result of one of the tape mechanisms. The “Updates Used” column shows the percentage of updates moved by the tape mechanism that are used at the destination. This column is omitted in some of the other application tables because it is near one hundred percent. “Comm KBytes" shows the total amount of data communicated during the measured portion of the application. Again, this column is omitted in some later tables because it is essentially unchanged across different runs. Finally, the last five columns show lock, barrier, flush, data (data request), and total messages.

Several trends are clear. First, auto-locks perform better than user locks. The reason is that the user locks are difficult to specify statically. For example, in one place, the region passed to the user-lock is an entire molecule, which may extend over two pages. The user-lock implementation therefore sends all modified data on both pages, including data modified as the result of false sharing. The auto-lock implementation is able to determine that only the first half of the molecule is while the lock is held, and the falsely-shared modifications in the second page are not sent.

Second, the sets of misses addressed by the lock and barrier mechanisms are disjoint: the number of misses eliminated with both mechanisms is almost exactly the sum of the misses eliminated by the mechanisms individually. Simple update protocols would perform similarly to the record-replay barriers, but be less effective at eliminating misses that are addressed by the update locks.

TSP

TSP is a branch-and-bound implementation of the traveling salesman problem. The central data structure is a global queue that contains partially completed tours. Processes alternately retrieve tours from the queue, split them into sub-tours, and put them back into the queue.

As shown in Table 5, TSP is almost exclusively lock-based. Locks are used to guard access to the central queue and to current minimum tour values. Barriers are used only during initialization and cleanup. We investigated both user locks and auto-locks. The results are shown in Table 5.

The first row shows the default TSP application. The second row shows performance with user locks. User locks are used to avoid misses when updating

Protocol	Speedup	Remote Misses	Updates Created	Updates Used	Comm KBytes	Messages				
						Lock	Barrier	Flush	Data	Total
Default	3.62	32677	4845	-	21727	764	518	0	65354	66636
Auto-locks	3.63	32494	4847	76%	21746	780	518	0	64988	66286
Rec/Rep	4.98	158	8950	98%	18924	762	518	1588	316	3184
Auto+Rec/Rep	5.12	11	8943	98%	18885	734	532	1589	22	2877

Table 6: Spatial

Protocol	Speedup	Remote Misses	Lock Pages	Messages				
				Lock	Barrier	Data	Tape	Total
Default	4.23	4499	185	3804	28	9110	0	12942
User	5.86	3377	1064	3830	28	6890	0	10748
User + PC	6.32	539	1563	3806	28	142	2096	6072

Table 7: QS

the “best” tour variable and when accessing the work queue. However, user locks can not specify the data that will be returned by a request for new work to perform, because the specific work has yet to be identified.

The auto-locks perform better because they retain a history of the last data that was accessed when the lock was held. This history is not an accurate predictor of future accesses (witness the low “updates used” value), but is relatively complete.

Spatial

Spatial solves the same problem as Water, differing primarily in that the molecules are organized into three-dimensional “boxes." The sizes of the boxes are set so that molecules in one box interact only with molecules in neighboring boxes. The box structure allows synchronization and sharing to be done at the level of boxes rather than individual molecules, effectively aggregating much of the synchronization. This gain is partially offset by the overhead of maintaining the box structure.

Table 6 shows the performance of Spatial. The “Updates Created” column describes the number of separate per-page updates that are constructed by the underlying LRC system. The number of updates doubles with record-replay barriers because the default version is able to lazily create updates only every other barrier.

Other than the overhead of creating and applying updates, this problem ends up having little impact on the Spatial’s performance. The multiple updates usually do not overlap, and therefore do not consume any more space or bandwidth than single updates. Second, few additional flush messages are sent because there are usually other updates destined for the same site. Therefore, the messages would need to be sent even if the excess updates were not produced. The flush versions actually send less data than the non-flush versions because the large flush messages have less system overhead than individual update requests.

Auto-locks have little effect on Spatial’s performance. The reason is that locks are used mainly to arbitrate access to the linked lists that tie molecules to boxes. The auto tape mechanism only prefetches the pages containing these pointers, not the pages containing the molecules themselves. Nonetheless, the overall impact of the flush mechanism is to improve performance by over 41%.

QS

QS is a parallel implementation of QuickSort. Again, the central data structure is a global queue that contains partially computed values, which are iteratively removed, refined, split, and inserted back into the queue until all are complete. QS differs from TSP in that the chunks of data that are taken out of the queue are merely pointers to the actual data. Hence, we use the producer-consumer regions that were discussed in Section 3.

Table 7 shows three versions of the QS program, with statistics as for TSP. The only new statistic is the “tape” message type. The first row shows the default implementation. The second row shows the results of a run in which all accesses to the central queue are through user locks. The regions passed to the user locks comprise the entire centralized queue structure. As this structure is updated frequently, the

Protocol	Speedup	Remote Misses	Updates Used	Comm KBytes	Messages			
					Barrier	Flush	Data	Total
Default	3.88	4177	-	15767	140	0	31826	31966
Rec/Rep	5.43	2157	87%	16047	140	576	7266	7982

Table 8: Barnes

Protocol	Speedup	Remote Misses	Updates Used	Comm KBytes	Messages			
					Barrier	Flush	Data	Total
Default	3.45	14294	-	32280	7160	0	14294	21454
Flush	4.31	0	100%	31673	7160	0	0	7160

Table 9: Gauss

user locks eliminate all misses on the pointer data structures, about one fourth of all remote misses.

The row labeled "User+PC" contains statistics reflecting the producer-consumer tape functions discussed in Section 3. The number of remote misses is reduced six-fold over the version with just user locks. The total number of messages is reduced by 53%, and speedup is increases by 49%.

Barnes

Barnes is the n-body galactic simulation from SPLASH-2, modified by Rajamony [27] to contain only barrier synchronization. Because of this modification, fine-grained tasks such as *make-tree* are now performed sequentially. This modification effectively increases the synchronization granularity. Note that while tapes can reduce or limit data movement during a parallel make-tree phase, they can do nothing to affect the direct costs of fine-grained synchronization.

Table 8 shows that Barnes differs from the other applications in that use of the tape mechanism is only able to eliminate about half of the remote misses. This is primarily because there is little locality across iterations. Processes access new pages during each iteration, and the system is therefore unable to anticipate all accesses. Nonetheless, 87% of updates flushed at barriers are eventually used, and total messages sent drops by a factor of four.

Gauss

Gauss is an implementation of gaussian elimination with partial pivoting. Essentially, it consists of a 2-D grid, with rows assigned to processes in chunks. At the beginning of each iteration, a new row is chosen as the "pivot", and all processes update all rows *after* the pivot row. The pivot row and column index need to be propagated to all other processes.

This method of updating plays havoc with standard update protocols. The problem is that each pivot is only flushed once, meaning that historical information can not be used to determine that the data needs to be broadcast. Application input is essential. We used tapes to build two new routines called "cvm_start_flush()" and "cvm_stop_flush()". These routines use a tape to record all shared modifications, and to broadcast them to all other processes.

Gauss's performance is shown in Table 9. All remote misses are eliminated. However, overall speedup is still mediocre because the last iterations have too little computation to make parallelism worthwhile.

5.4 Discussion

The tape mechanism's advantages are performance and simplicity. In evaluating performance, we distinguish between the performance of the tape layer itself, the performance of Tapeworm, the specific synchronization library discussed in this paper, and the potential performance improvements of other synchronization libraries that could be built using tapes.

The tape layer itself adds very little overhead. Recording page reads and writes adds only a few instructions to the page fault handlers. The runtime cost of manipulating tapes and extents is also small. Extents are implemented as bitmaps in our current prototype. They are therefore fast, but reasonably expensive in terms of memory consumption. Since the constituent elements of extents are pages, the size of an extent is proportional to the number of shared pages. Currently, the largest applications we run share on the order of thirty-two megabytes. Assuming 8k pages, this results in a bitmap of 512 bytes. On the other hand, water uses less than 500k of data, resulting in bitmaps of only eight bytes. If the current representation becomes unacceptable, extents could be implemented as sets of bitmaps, and would have

size proportional to the working set of pages. Tapes are currently implemented as sequential records of events, and are therefore of size proportional to the number of recorded events. Similar to extents, more sophisticated representations for tapes are possible in the event that their size or runtime cost grows too large. Most of the tape operations involve only a single linear pass through a tape, with random access to the bitmaps used to represent extents. The result of tape addition (used in auto-locks) is the set of all unique elements of the union of two unordered sets. Our current implementation is $O(n^2)$ in the size of the tapes, but more sophisticated implementations are possible.

As far as the effectiveness of the specific synchronization library discussed in this section, Table 3 showed that Tapeworm eliminates an average of 85% of all remote access misses. The percentage of access misses eliminated can be termed the coverage of the protocol. The accuracy of the protocol can be characterized by the number of updates sent but not used. These updates are pure overhead, but do not affect correctness. This quantity is given by the "Updates Used" column in Table 4 through Table 8. Tapeworm's average accuracy is 91%. Assuming a uniform distribution of diff sizes, this implies that the average bandwidth overhead is only nine percent. However, the number of extra messages is likely to be a much smaller percentage. Most of these extra updates are sent in messages that would have to exist for other updates or synchronization, even if the useless updates were not sent.

One last aspect of this effectiveness is whether Tapeworm results in a significant number of extra updates being created and applied. This occurs only in Spatial. However, it does not result in either extra messages or data, so we conclude that the effect on Spatial's performance is negligible. This effect could be significant in other applications. We expect that specializing barriers, as described in Section 3, would minimize this effect.

Mechanisms such as auto-locks and record/replay barriers also incur overhead in that they need to be trained before being used. Faults incurred during the initial use of these mechanisms can be termed cold misses. Faults avoided during subsequent synchronizations are always conflict misses for our implementation because CVM relies on the underlying virtual memory system to handle capacity problems. All results presented in this paper represent the steady state execution of applications after the cold misses are complete. Assuming static sharing behavior, however, the percentage of potential faults that are cold misses can never be higher than *1/n*, where '*n*' is the number of iterations timed. Hence, cold misses are unlikely to be important for realistic runs.

The second claimed advantage of tapes, simplicity, has two parts: simplicity of use and simplicity of support. Using the modified synchronization routines merely consists of replacing existing synchronization calls. Determining *whether* to use the routines is more difficult. Ideally, the system itself would recognize iterative access patterns and either automatically invoke modified routines or inform the user. This is a subject of future research. However, the access patterns that our current mechanisms exploit are quite simple and common. Parallelizing an application in the first place requires far more detailed knowledge of data movement than is required to select among these mechanisms.

Our claim of simplicity for support is based on the amount of code needed to build the tape mechanism. The total size of the CVM system is about 15,000 lines of commented code, including debugging statements. The tapes support layer consists of less than 500 lines of C++ code, and the Tapeworm synchronization library is an additional 400 lines.

6. Tapes and other memory models

The tapes concept has been carefully designed in order to accommodate a range of underlying consistency protocols, including those implementing weak memory models like LRC. However, tapes can be efficiently implemented on top of conventional single-writer-multiple-readers (SWMR) protocols as well. The salient feature of these sequentially consistent protocols is that a single virtual page can be writable at one node, or readable at one or more nodes, but not both. This implies that any valid copy of a page is as up to date as any other copy in the system. This is very different than with multi-writer LRC implementations, where a single page may be simultaneously modified by any number of processors. Much of the complexity in CVM's support for tapes arises from the need to identify, locate, and retrieve the data needed to create an up-to-date copy of a page.

The support needed from a SWMR protocol is essentially a subset of the support in CVM described above. Section 4 listed four requirements from the underlying system: interval specification, access recording, "hole" tape creation, and data instantiation. The first two are handled as with LRC protocols. Taking the last two in reverse order, SWMR protocols do not validate local copies of pages by applying diffs; all that is required is to create a local copy. Since all copies are guaranteed to be up to date, any such copy will contain all previous updates. Hence, instantiating the data referenced in a tape consists of making local copies. "Hole" tapes, therefore, consist of a single 3-tuple per invalid page; the interval and process id's of each tuple are irrelevant.

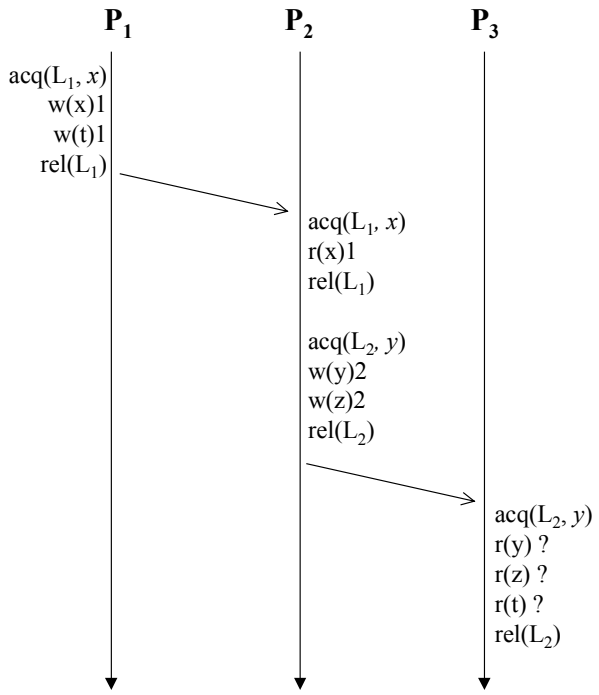

Figure 8: The parameters to the *acq()* calls are the lock, and any explicit data associations that would be specified in the case of entry consistency. Assuming that all data was initially zero, the P_3's reads of *y*, *z*, and *t* would return 2,2,1 for LRC, 2,2,0 for scope consistency, and 2,0,0 for entry consistency.

6.1 Emulation of update-based protocols

Having described the interaction of tapes with different underlying memory model implementations, we next describe how tapes can be used to emulate the runtime behavior of several high-performance memory model implementations. While a full-fledged performance comparison between tapes-based code and these other implementations is beyond the scope of this paper, we argue that tapes can be used to express both the data capture and data movement functionality needed to emulate the other protocols. In all cases, we assume that the underlying protocol is LRC.

Tapes can be used to construct *data-movement protocols* that offer the performance advantages of SDSM consistency algorithms such as home-based LRC [36], entry consistency [5], and scope consistency [15]. These algorithms differ from standard, or *homeless*, LRC [19] in both consistency semantics and in data movement. However, the difference in consistency semantics usually has a negligible effect on performance.

To see this, note that the differences essentially boil down to the question of what writes seen by the releaser of a synchronization variable need to be seen by the next subsequent acquirer. Figure 8 illustrates the differences in consistency semantics between (homeless) LRC, scope consistency, and entry consistency. Home-based LRC implements the same memory model as homeless LRC with a different underlying page replication scheme. Greatly simplified, entry consistency requires locks to be explicitly identified with the data that they guard. The acquirer of a lock must see the same versions of all data associated with the lock as were seen by the last releaser when its release was performed. At the time of its first lock acquire in Figure 8, for example, p_2 sees the update to *x*, but not the update to *t*. Likewise, p_3 sees only p_2's update to *y*, not the updates to *x* and *t*.

Scope consistency is similar, but differs in that the associations of data to synchronization variables need not be explicit[1]. Hence, the lock transfer from p_1 to p_2 would transfer notification of the updates to both *x* and *t*. The other lock transfer would transfer notification of the updates to *y* and *z* between p_2 and p_3.

Finally, LRC requires all modifications made prior to a release to be made visible at the next subsequent acquire. Hence, p_3 sees p_2's modifications to both *y* and *z*, and even sees p_1's modification to *t*. This is despite the fact that p_1 and p_3 have not accessed any synchronization variables in common, or communicated directly.

[1] There are other differences, but they are not relevant to this discussion.

None of these differences in consistency semantics translates to a significant performance difference. Transferring notifications of additional writes uses no additional messages, and consumes a negligible amount of space on existing messages. These additional notifications may cause pages to be erroneously invalidated if pages are falsely shared, and if write trapping is performed through page-based virtual memory mechanisms. However, this still only has a performance impact if the erroneously invalidated pages are subsequently accessed *before* being invalidated through true sharing. Various studies have shown that relaxed consistency models largely eliminate the effects of false sharing in page-based systems [4, 18].

Aside from differences due to the use of page-based or diff-based write collection mechanisms, the primary performance differences between scope, entry, and lazy release consistency are due to the degree to which faults are eliminated by moving updates prior to their use [9]. We discuss below how tapes can be used to emulate this data movement for each protocol. Our discussion assumes an underlying protocol implementing homeless LRC, but the data-movement protocols will work with a variety of underlying protocols.

Home-based LRC

Home-based LRC [36] protocols implement the same consistency model as homeless LRC, differing only in the underlying implementation. The primary difference is that pages in home-based protocols each have a designated *home*. Any modification made to a page is flushed to the home at the next synchronization release, while page faults are satisfied by retrieving a complete new copy from the page's home. This has two advantages. First, the protocol *updates* the home, rather than invalidating it. A consumer does not have to fault data across the network if it is the home for the desired pages, because they are automatically flushed to it soon after they are created. Second, home-based protocols have lower memory overheads because copies of the modified data can be discarded after being flushed to the home node. Homeless protocols need a garbage collection mechanism in order to identify diffs that can be discarded.

Emulating home-based protocols essentially consists of providing a means of designating each page's home and of flushing updates to the home at synchronization points. The former is trivial and omitted here. The latter is implemented by flushing updates made while the lock is held to the updates' homes, and is implemented by the code in Figure 9. An array of extents, `homes`, names the pages for which each node is "home". A single tape captures all updates created while a lock is held, and another track all other updates. This code assumes that locks are not nested, but extension to the nested case is straightforward.

Recording into `barrierWrites` is paused while a lock is held. The `flush()` routine is used to flush new updates to their homes. For each node, we create a new tape containing the tape of all updates filtered by the pages for which that node is "home". The result is flushed to that node.

Entry and scope consistency

Tapes could also be used to build a synchronization interface that would closely approximate the data communication characteristics of Midway's [5] or CRL's [16] update protocol. While both systems differ from CVM in many ways, one of the key differences is that both Midway and CRL use update-based protocols. Unnecessary updates are avoided by limiting the updates to shared regions that are explicitly associated with synchronization. The auto-locks described in Section 3.2 would approximate these data movement patterns, modulo excess invalidations caused by false sharing.

Similarly, a tape is not a *scope,* but they can be used to build a synchronization interface that superficially mimics *scope consistency* (ScC) [15]. The two would differ in that ScC is a consistency model, whereas any interface built using tapes is merely a data movement mechanism that exists on top of the underlying consistency model. Hence, whatever claims are made as to the relative benefits of ScC and LRC as a consistency model still apply. However, tapes can be used to greatly reduce communication traffic in either case. The canonical ScC implementation is home-based [15], so all updates are constrained to move through the home node. Therefore, data communication between processes p_1 and p_2 must involve the home nodes of any data communicated. The tapes-based approach can therefore move less data, and certainly use fewer messages, than the home-based approach for all cases where the home nodes are not one of the communication endpoints. We plan to investigate the performance of a tapes layer on top of ScC in the future.

Although entry consistency allows either invalidate or update implementations, Midway's canonical implementation uses an update protocol for lock synchronization transfers. Acquiring a lock guarantees that all of the data explicitly associated with that lock are present and valid. Recall that this is very close to the semantics of the user locks discussed in Section 3.2. The only major difference is that the user lock mechanism is not guaranteed to have all diffs in the

```
Tape            lockWrites;
Tape            barrierWrites;
Extent          homes[MAX_NODES];    /* already initialized */

void HomeBased::lock_entry(int lockId)
{
        lockWrites.reset();
        lockWrites.start_writing();
        barrierWrites.pause();
}

void HomeBased::lock_release()
{
        flush(lockWrites);
        barrierWrites.unpause();
}

void HomeBased::barrier_exit()
{
        barrierWrites.reset();
        barrierWrites.start_writing();
}

void HomeBased::barrier_entry()
{
        flush(barrierWrites);
}

void flush(Tape *tape)
{
        tape.stop_writing();
        Extent *ex = tape.project_data();
        for proc in (all processes) {
                if (proc == ME) continue;
                if (Tape *perNodeTape = tape + homes[proc]) {
                        populate perNodeTape and send to proc;
                }
        }
}
```

Figure 9: Implementing home-based data movement.

presence of false sharing. Likewise, data movement under scope consistency is similar to the auto-lock mechanism discussed in Section 3.2.

7. Related work

Software distributed shared memory (SDSM) has been an active field of research for over a decade. Early page-based systems, such as Ivy [24] and Clouds [10], established the basic ideas of page-based VM support. Relaxed consistency models were introduced by Munin [6], which pioneered the use of release consistency and multiple protocols, and Treadmarks [20], the first highly-portable implementation of lazy release consistency and the system on which CVM is closely modeled.

Work on alternatives to VM-based write-trapping began in earnest with Midway [5] and Blizzard [31]. Midway uses a modified compiler to generate code that modifies dirty bits after each shared write. Blizzard-S (and later, Shasta [30]) uses a binary-rewriter to insert code that invokes a state machine at each access. Blizzard-E is similar, but uses manipulation of the ECC bits of the CM-5 to invoke the state machine. In all cases, the method of write trapping and collection can easily be accommodated by the tapes abstraction. Furthermore, all of these protocols are single-writer-multiple-reader, so support for tapes would be quite simple. However, one of the key points of these protocols is that they avoid false sharing by maintaining consistency at fine granularity, either at a cache block or program object size. This potentially poses scalability problems for tape implementations because tapes contain 3-tuples for each modified object, and cache lines and program objects are usually much smaller than virtual memory pages. Tape data structures would therefore be much larger on such systems than on VM-based systems.

There are a number of ways that this scalability problem might be addressed. For example, fine-grained objects might be aggregated at the tape layer, either in consecutive ranges or by dynamically observing objects that are accessed together. Shasta performs a great deal of analysis in order to minimize the amount of inserted code. This analysis could conceivably be extended to statically identify and aggregate objects.

Record/replay barriers were first implemented by the Wind Tunnel project [13]. This work focused on providing support for irregular applications by coding application-specific protocols, one of which implemented a record/replay barrier. Later work in the same project resulted in a protocol-implementation language called Teapot [7]. This work is similar to ours in that both are trying to expose protocol handles to application or library builders. However, the Teapot language is more complex. More lines of Teapot code are required to implement a sequentially consistent invalidate protocol than the corresponding protocol written in C++ on CVM. Part of the reason for this additional complexity is that Teapot protocols perform both data movement and maintenance of correctness, whereas consistency can not be violated in any synchronization library built on top of tapes. One major advantage of Teapot is that it leverages existing cache protocol verifiers to automatically verify Teapot programs.

Our work has similarities to work performed at Rice University on compiler-SDSM interfaces [12]. The `missing_data_type()` routine is essentially the information-gathering phase of the TreadMarks [3] `validate()`. Some of the update work we describe is similar in spirit to the TreadMarks `push()` command. However, our work not only provides ways to manipulate data, as with TreadMarks, but it also provides ways to gather this information dynamically through tapes. While the TreadMarks work assumes all information is provided by the compiler, our work provides a way for the user or synchronization library to gather this information at runtime. For instance, our tapes allow us to dynamically determine the extent of the data being accessed, while this information is assumed to be known by the compiler in TreadMarks. Our work also allows the user to manipulate discover and manipulate shared modifications at a high level. Recent work at Rice has investigated automatic determination of extent-like objects in shared memory applications [4].

We have concentrated our discussion on software SDSM systems, but it may also be relevant in the context of hardware shared memory systems. For instance, the prefetch and poststore primitives of the KSR-2 [1] implement user-initiated data movement on top of the underlying consistency protocols. Other work [28] generalized these primitives to allow the destination of pushes to be specified either by run-time copyset management or by specific calls initiated by application programs. By augmenting these primitives with the ability to read and write copyset information, tapes could be supported on top of this type of system with only a minimal runtime layer. Even with an efficient implementation, however, such a system would probably only be useful with large cache lines, i.e. 128 or more bytes. Kagi [17] categorizes a number of techniques for increasing synchronization speed on SMP's, including collocation (putting locks and the data they guard on the same cache lines) and QOLB, a distributed lock implementation. The most directly relevant to SDSM systems is synchronous prefetch, or compiler-directed prefetches of synchronization variables. However, this technique could only be implemented with extensive cooperation from the underlying protocol, and so would not generally be implemented at the tapes level.

Several papers have used predictive techniques to accelerate hardware coherence protocols directly. Zhang [35] described a technique similar to our producer-consumer regions, but in the domain of cache lines. Their technique allows users and compilers to explicitly create arbitrary groups of cache lines, which are fetched as a group. This is useful even for regular applications (where the groups will usually consist of contiguous lines), but is especially useful for irregular applications. Lebeck [23] describes *self-invalidating* caches, where the directory controller identifies cache lines to be self-invalidated based on prior behavior. There is no direct tapes analog because the point of self-invalidating caches is to eliminate coherence messages. Tapes affect only data movement, not coherence. Mukherjee [25] described the use of branch predictors in anticipating coherence operations. Similar techniques could be used to direct tape operations. However, they are likely to be less useful because SDSM systems have larger, and therefore fewer, operations. Statistical techniques usually work best with large populations.

Shared memory systems with dedicated protocol processors [22, 29] might turn out to provide the best possible platform for tapes implementations. Tape code executing on the protocol processors could track data and synchronization accesses without ever involving the application processor.

8. Conclusions

This paper has described the tape mechanism, and its use in tailoring data movement to application semantics. Tape-based synchronization libraries are layered on top of existing consistency protocols and synchronization interfaces, meaning that incorrect choices

(whether by heuristics or programmers) affect only performance, not correctness.

The tape mechanism is ideally suited to direct data movement because it allows shared accesses to be recorded, grouped, and manipulated at a very high level. These tapes can be used to predict future data accesses and to eliminate subsequent misses by moving data before it is needed.

We used the tape mechanism to build Tapeworm, a new synchronization library that uses information gathered at runtime to reduce access misses. Tapeworm's interface consists of auto-locks, producer-consumer regions, and record/reply barriers. Auto-locks pre-validate data that is accessed while locks are held. Producer-consumer regions use the first access to a region as a hint to request the rest of the region before it is needed. Record/replay barriers allow accesses to be recorded during one iteration and then played back during future iterations. The combination of these mechanisms allows Tapeworm to eliminate an average of 85% of remote misses for our applications, 63% of all messages, and to improve overall performance by an average of 29%. We conclude that the tape mechanism is a promising approach to creating high-performance synchronization libraries.

We also describe how tapes can be used to mirror the data movement in recent update-based protocols, including home-based LRC [36], entry consistency [5], and scope consistency [15]. These protocols differ from generic LRC in terms of the programming model, the memory model, and the flow of data. Tapes-based implementation can be used to separate the performance effects of the latter factor from the effects of the former two factors. Such implementations can also be used to provide "front-ends" to a SDSM system, somewhat analogously to the front end of a compiler.

In general, Tapes have at least two major advantages over optimizations of specific protocols. First, tapes provide a high-level abstraction of shared accesses, and are protocol-independent. Tapes make few requirements on the underlying protocol, providing a terse, powerful approach to managing data movement. Second, tape mechanisms can be implemented and used incrementally. Applications can be completely debugged before any tape mechanisms are added. One by one, tape mechanisms can be used to improve data movement at inefficient points in application executions.

Our future work with tapes will center on two areas. First, we are exploring the use of compilers to automatically generate tapes interfaces. This work is complementary to recent work in parallelizing compilers [8, 32]. Tapes improve performance by exploiting repetitive access patterns. Identifying such patterns *with high degree of probability* in the compiler is much easier than generating explicit message-passing code for the data movement. Hence, compiler heuristics that might not be rigorous enough to generate verifiably correct message-passing code could be used by tapes to direct data movement in a SDSM system.

Second, we are looking at the expressiveness of the tapes interface, and attempting to identify other functionality that tapes layers could be used to support. One current deficiency is the inability of tapes to track dynamically changing access patterns. Our tapes support only tracks accesses that generate page faults. We currently have no way of telling (at the tapes level) whether data that is pushed to another node by a tapes layer is actually used. Hence, data might continue to be pushed to nodes that long ago ceased consuming the data. There are several obvious ways to address this, such as periodically resetting all mechanisms and rebuilding access information from scratch. However, this approach would entail overhead even in the common case of static applications, so we are looking for a less expensive option. We are also looking at the use of tapes in debugging. A tapes-based approach could conceivably be used to build an online race-detection mechanism [26] for single-writer protocols. However, such a mechanism, while expressible at the tapes level, would likely be protocol-dependent. Race detection on multi-writer protocols requires the underlying protocol to check for sharing at all granularities less than a virtual memory page.

9. References

[1] "Kendall Square Research. Technical Summary," 1992.

[2] S. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "A Comparison of Entry Consistency and Lazy Release Consistency Implementations," in *Proceedings of the Second High Performance Computer Architecture Conference*, February 1996.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18--28, February 1996.

[4] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.

[5] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared

Memory System," in *Proceedings of the '93 CompCon Conference*, 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*, October 1991.

[7] S. Chandra, B. Richards, and J. R. Larus, "Teapot: Language Support for Writing Memory Coherence Protocols," in *SIGPLAN Conference on Programming Languages Design and Implementation*, 1996.

[8] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers," in *Proceedings of the International Parallel Processing Symposium*, 1997.

[9] A. L. Cox, E. d. Lara, Y. C. Hu, and W. Zwaenepoel, "A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory," in *Proceedings of the 5th High Performance Computer Architecture Conference*, January 1999.

[10] P. Dasgupta, R. C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. L. Jr., W. Applebe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wileknloh, "The Design and Implementation of the Clouds Distributed Operating System," in *Computing Systems Journal*, Winter 1990.

[11] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," in *IEEE Computer*, February 1988.

[12] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System," in *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[13] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. H. J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *Supercomputing 94*, 1994.

[14] 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*, May 1990.

[15] L. Iftode, J. P. Singh, and K. Li, "Scope Consistency: a Bridge between Release Consistency and Entry Consistency," in *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[16] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[17] A. Kagi, D. Burger, and J. R. Goodman, "Efficient Synchronization: Let Them Eat QOLB," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[18] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[19] 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*, May 1992.

[20] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the 1994 Winter Usenix Conference*, January 1994.

[21] P. J. Keleher, "Tapeworm: High-Level Abstractions of Shared Accesses," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[22] J. Kuskin and D. O. e. al., "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[23] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *22nd International Symposium on Computer Architecture (ISCA)*, June 1995.

[24] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[25] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *25th Annual International Symposium on Computer Architecture (ISCA)*, June 1998.

[26] D. Perkovic and P. Keleher, "Online Data-Race Detection via Coherency Guarantees," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[27] R. Rajamony and A. L. Cox, "Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependency Analysis," in *Proceedings of the Sigmetrics'97 Conference*

on the Measurement and Modeling of Computer Systems, June 1997.

[28] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors," in *Supercomputing*, 1995.

[29] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[30] D. Scales and K. Gharachorloo, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[31] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain Access Control for Distributed Shared Memory," in *The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[32] C.-W. Tseng and P. Keleher, "Enhancing Software DSM for Compiler-Parallelized Applications," in *11th International Parallel Processing Symposium*, 1997.

[33] L. D. Wittie, G. Hermannsson, and A. Li, "Eager Sharing for Efficient Massive Parallelism," in *Proceedings of the 1992 International Conference on Parallel Processing (ICPP '92)*, August 1992.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[35] Z. Zhang and J. Torrellas, "Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," in *22nd International Symposium on Computer Architecture (ISCA)*, June 1995.

[36] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October, 1996.