Tapeworm: High-Level Abstractions of Shared Accesses

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

We describe the design and use of the tape *mechanism, a new high-level abstraction of accesses to shared data for software DSMs. Tapes can be used to "record" shared accesses. These recordings can be used to predict future accesses. Tapes can be used to tailor data movement to application semantics. These data movement policies are layered on top of existing shared memory protocols.*

We have used tapes to create the Tapeworm prefetching 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 introduces the notion of *tapes*: a new high-level abstraction that allows applications to achieve higher performance on distributed shared memory (DSM) protocols. DSM protocols support the abstraction of shared memory to parallel applications running on networks of workstations. The DSM abstraction provides an intuitive programming model and allows applications to become portable across a broad range of environments. However, this level of abstraction prevents the application from improving performance by explicitly directing data movement. Hence, while it is relatively easy to get parallel applications working on current DSMs, it can be very difficult to achieve high performance.

Tapes 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.

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.

We used tapes to implement Tapeworm, a new synchronization library that is layered on top of existing consistency and synchronization protocols in CVM [1], 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 as well as synchronization. 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 Tapeworm, a high-performance synchronization library built using tapes. Section 4 describes the requirements Tapeworm makes on the underlying consistency protocols. Section 5 describes Tapeworm's performance, Section 6 describes related work, and Section 7 concludes.

2. Tape semantics

Tapes are implemented as a software layer that 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. In practice, tapes are particularly well suited for the relaxed consistency models discussed below.

The primary way in which we expect tapes to be used is in augmenting invalidate protocols in software DSMs. Such systems consist of a single thread or process per machine, a shared segment that can be transparently accessed by any of the processes, and at least a rudimentary

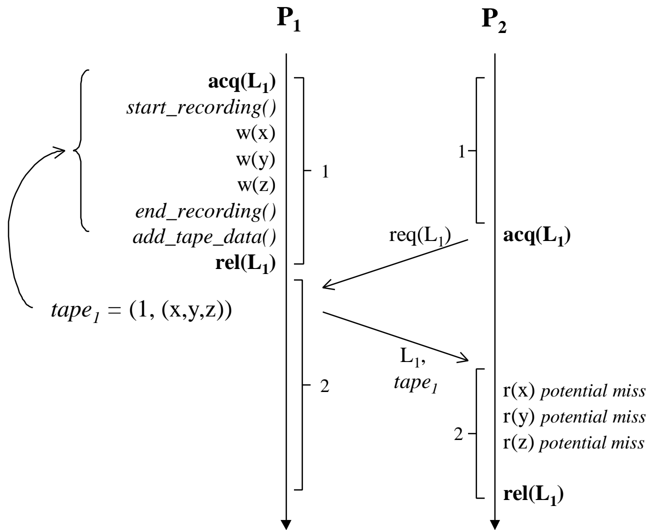

Figure 1: Tapes: $tape_1$ describes the writes performed by P_1. Subsequent misses by P_2 can be avoided if the tape, together with the data it describes, is transferred with the lock.

set of synchronization mechanisms. Synchronization is usually implemented in addition to, rather than on top of, the consistency mechanism. The best page-based consistency mechanisms are based on some form of release consistency (RC) [2] or lazy release consistency (LRC) [3, 4]. Both allow consistency actions to be delayed until subsequent synchronizations. These protocols are ideal for the use of tapes in that they allow considerable freedom in when data actually moves. Tapes could be used with more strict protocols, but more work would be required to provide the necessary hooks.

Figure 1 shows a primitive example of the use of tapes. Process P_1 modifies three pages while holding lock L_1, followed by P_2 acquiring the same lock and reading the same three pages. A traditional invalidate protocol would at least incur remote misses by P_2 on each of the three pages. By including the code in italics, however, P_1 can create $tape_1$, an exact description of the changes that were made to the three shared pages. The tape logically contains a word-by-word description of the modifications; it is not dependent on the modifications fitting some predefined or known pattern. The tape is passed to *add_tape_data()*, which adds the tape's data to the outgoing lock release message. The result is that P_2's remote misses are eliminated because the data arrives with the lock.

We call this a primitive example because we expect tape manipulation to be hidden inside synchronization, rather than being programmed directly in the application code.

Implementation

Tapes are made efficient by hiding the descriptions of actual shared accesses behind two layers of abstraction. First, each process's execution is divide 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. Each of the processes in Figure 1 has two intervals, delimited by access to lock L_1.

Second, all modifications made to a single page during an interval are combined into a single modification. We can then express tapes in terms of lists of modified pages and intervals, instead of addresses and cycle counts[1].

More specifically, a tape consists of a set of *events*, each of which is a tuple (x,y), where x is an interval id and y is a set of page id's. Hence, $tape_1$ in Figure 1 consists of the single event $(1,(1,2,3))$. 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.

The approach shown in this example has several advantages over other approaches described in the literature. While update protocols [5-7] might also suffice in this example, update protocols are indiscriminant. The tapes version only sends updates of the data modified while the lock is held, and the updates are only sent to the next holder of the lock. Update protocols also have to be trained. The tapes code will eliminate misses even on the first iteration.

Midway [8] would also eliminate P_2's remote misses in Figure 1. However, Midway requires explicit links between synchronization variables. The link in the tapes code is implicit. The data passed to P_2 will adapt if the data modified by P_1 changes.

Clearly, this is not the whole picture. First, P_2 might be missing other modifications to the three pages. In the absence of false sharing, however, P_1 will likely have copies of this data and can supply it along with the lock. Second, application programmers are unlikely to want to reason with tapes. However, all tape references could be encapsulated into modified synchronization routines. The sole application-visible change would be in calling a different version of the lock routines.

This example does not show the full generality of the tape mechanism. Since the tape consists internally of events, the accesses need not be to contiguous pages, or have occurred at similar times. Since tapes consist only of abstract descriptions of shared modifications rather than the modifications themselves, they are relatively lightweight and can be stored, transmitted, and otherwise manipulated.

Tape Creation

Tapes can be created in several different ways, but the primary method is that shown in Figure 1, e.g. recording

[1] The use of the term "page" throughout this paper is a convention. The units can be of any shape that can be tracked by the underlying consistency protocol.

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

Figure 2: Using record/replay barriers.

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 4, *missing_data_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*.

Extent is short for "data extent." An extent is an object that names a list of pages. For example, a tape can be flattened into an extent that contains a list of pages modified by the tapes events. Assume that an extent 'ext' names a large data structure that resides on a set of invalid shared pages. Pages are only invalid if they have been modified by remote processes, and if the remote modifications have yet to be applied locally. A *missing_data_tape(ext)* call returns a tape that lists every such remote update that is needed to re-validate the invalid pages. This tape can be used to request all of the updates at once, possibly before the data is actually needed. The result is greater latency tolerance, and the potential for greater overlap of communication and computation.

Once a tape has been created, it can be transmitted to remote sites, flattened into an extent, pruned to contain only notices that pertain to a given extent, or added to another tape.

While our discussion of tapes has concentrated on write accesses so far, analogous abstractions can be defined for read accesses, and data requests from other processes.

3. The *Tapeworm* Library

The following subsections describe three types of Tapeworm-based synchronization interfaces that we found useful for our application suite: *record-replay barriers, update locks*, and *producer-consumer regions.*

In all cases, we rely on the underlying protocol layer to ensure correctness, regardless of when data arrives. Section 4 describes the demands that this requirement places on the underlying protocol.

3.1 Record-Replay Barriers

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

cvm_read_barrier()
{
    writeTape.stop_writing();
    for proc in (all processes) {
        Tape *out = writeTape + reqExtents[proc];
        if (!out->empty())
            cvm_flush_tape(out);
    }
    reqTape.start_requesting();
}

cvm_write_barrier()
{
    readTape.stop_requesting()
    for proc in (all processes) {
        reqExtents[proc] += readTape.extent(proc);
    }
    writeTape.start_writing();
}
```

Figure 3: Record/replay implementation

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 future iterations. Much of the remote latency can be hidden by sending the data before it is needed. Figure 2 shows pseudo-code for a process of a simple red-black application. Each process iterative calls *cvm_read_barrier()*, computes new values for all of the elements that it owns, calls *cvm_write_barrier()*, and then copies the new values back into the original array. The computation uses two phases in order to synchronize the read and write accesses to *arr[]*.

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

Pseudo-code for the specialized barrier routines is shown in Figure 3. The key to understanding these routines is that we do not record reads directly. Instead, each process records data requests that it has received 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 during the second phase of the computation. Each process uses *reqExtents[]*, an array of extents, to record the data requested by remote processes. In *cvm_read_barrier()*, the writing has been completed, so the routine sends out local writes to other processes that have previously requested modifications to those pages. The set of modifications to be sent to each remote process is created by pruning the tape of new local modifications with the extent for that process. C++'s operator overloading allows addition of a tape and an extent to be in-

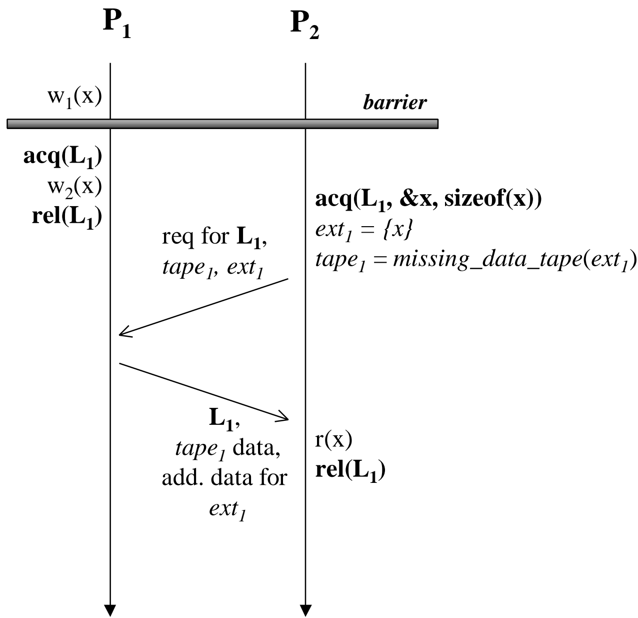

Figure 4: User Locks

terpreted as "create a copy of the input tape, such that only pages described in the extent are included." If the new tape is non-empty, it contains a description of the data that needs to be flushed to that process. The flush is accomplished by *cvm_flush_tape()*, a primitive routine that creates a message containing data named by a tape, and sends it to a remote process.

The *cvm_write_barrier()* routine is less complex. It merely flattens the read tape to an extent specific to each process, adding the new extents to the existing values. These routines have to be extended slightly in order to accommodate dynamic sharing patterns.

Note that for the code in Figure 2, we could have obtained identical results by combining the two types of barriers into a single call and using a pure update protocol. However, there are many reasons why sending all of the updates immediately might not be a good idea. Delaying the updates might allow data to be aggregated. Some updates might be avoided altogether if there are multiple updates to the same location. In general, the use of distinct barrier calls allows the application to have finer control over the data movement, and gives the underlying implementation more opportunities for optimization.

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.

User Locks

Tapeworm contains two versions of update locks. *User locks* augment the standard lock interface with 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 (ext_1 in Figure 4), which provides an efficient and portable representation of the set of pages covered by the region.

The extent is used for two purposes. First, the extent can be used as input to the support layer call *missing_data_tape()*. This call creates a tape that names all updates that are currently needed to validate the pages named by the extent. In Figure 4, for instance, P_2's copy of the page containing *x* is presumably invalidated at the global barrier. While acquiring L_1, P_2 creates extent ext_1, which names page *x*. Extent ext_1 is then used as input to *missing_data_tape()*, which returns $tape_1$, a tape specifying that the result of $w_1(x)$ is needed to validate the pages named by ext_1.

This tape is not necessarily sufficient, however, because the synchronization transfer may allow the acquiring process to learn about other updates that need to be applied. In this example, the synchronization transfer will also allow P_2 to learn of $w_2(x)$. This write is not named by the tape because P_2 does not yet know about it. However, by appending ext_1 to the request, the protocol allows P_1 to infer that the result of $w_2(x)$ will be needed as well, and to include it in the lock grant message.

Auto-locks

Auto-locks differ from user locks only in that they do not require the programmer to explicitly name the shared region associated with the lock. Instead, the region is created automatically by recording read and write accesses while the lock is held. The most immediate implication is that the initial lock acquisitions will be performed without any knowledge of the corresponding data. Therefore, auto-locks will require a warmup period, much like caches.

Second, subsequent accesses might be *more* accurate in anticipating data accesses than user locks. Programmers are often inaccurate, and locks may guard accesses to non-contiguous regions of shared data. Auto-locks can also accommodate slowly changing access patterns by using only recent data to inform subsequent lock requests.

Auto-locks are implemented by maintaining a per-lock extent. While the lock is held, shared reads and writes are recorded into a per-lock tape. The tape is flattened into an extent when the lock is released. The resulting extent is retained as input for the next acquisition of that lock.

Figure 5 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. As

```
Tape      writes;
Extent    lockExtent;

void Tapeworm::lock_entry(int id)
{
        tape.start_writing();
}

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

        msg->add(type_extent, (char *) lockExtent, lockExtent ->size());
}

void Tapeworm::add_to_lock_grant(Msg *msg, int pid)
{
        if (tape = (Tape *)msg->retrieve(type_tape)) {
                tape->add_data(msg);
        }
        if (extent = msg->retrieve(type_extent)) {
                iMan->new_interval();
                tape = get_new_tape(pid, extent);
                tape->add_data(msg);
        }
}

void Tapeworm::read_from_lock_grant(Msg *msg, int id)
{
        read_data(msg);
}

void Tapeworm::lock_release(int id)
{
        writes.end_writing();
        lockExtent = writes.get_extent();
}
```

Figure 5: Auto Lock Implementation

Tapeworm is implemented as part of a tapes protocol that specializes the default multi-writer LRC protocol, 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 used are quite simple. A single tape, writeTape, suffices to capture shared accesses while the lock is held. Recording is started in the *lock_entry()* routine, which is the first code executed in a lock acquisition. 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 past the writes tape from the previous lock access. The tape, created by *missing_data_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 reads the tape and extent from the request, and then uses the *add_data()* call to load the updates named by the tape into the return message. In the case of the extent, the routine *get_new_tape()* is used to create a tape naming "new" updates for the region covered by the extent. These "new" updates are updates named by consistency information known to the granter, but not yet by the grantee. This is basically a hook for LRC-like implementations that append consistency information to synchronization grants. Without this hook, the underlying consistency protocol might still invalidate pages in the region named by the lock's extent, causing demand fetches. The *add_data()* call is then used to load the data named by this new consistency information into the reply as well.

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

```
start_produce()
{
        tape.start_recording();
}

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

Tapeworm::page_request (int pg_id, Msg *msg)
{
        if (Tape *tape = queue.search(pg_id)) {
                tape->add_data(msg);
        }
}
```

Figure 6: Producer-consumer regions

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, flattening 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 6 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, providing 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

Interface	Type	Description
*Tape *missing_data_tape(Extent *)*	c	Return tape describing updates needed to validate extent.
*Tape *get_new_tape(Extent *)*	c	Return tape describing "new" updates (see Section 3.2).
*Tape::add_data(Msg *),* *Tape::read_data(Msg *)*	c, m	Add updates described by tape to message, read any such updates from message and incorporate into shared data.
*msg->add(msg_type, char *, int)* *msg->retrieve(msg_type, char **, int *)*	m	Allows arbitrary data to be added and retrieved from Msg objects.
Protocol::fault(int pg) *Protocol::page_request(Msg *, int)*	c, m	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)*	m	Allows data to be piggybacked on top of existing synchronization messages.

Table 1: Low-Level Support for Tapes ('c' – consistency, 'm' – message)

as if the pages were transferred one at a time, the benefits of aggregating multiple requests into one can be significant.

4. Low-Level Support for Tape Protocols

Our tapes implementation is layered on top of CVM [1], 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.

While tapes are conceptually independent of both the programming model and the particular protocol implementation, the underlying consistency protocol and system architecture must provide some basic support. Table 1 summarizes the required interfaces to the consistency mechanism and to the messaging subsystem. Those rows marked with a 'c' are requirements specific to the consistency protocol itself. Those marked with an 'm' are other hooks for creating and handling communication, or message, events.

4.1 Interactions with the consistency protocol

Tapeworm is layered on top of a consistency protocol, it is not a consistency protocol itself. 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 [6]. All protocol calls other than those listed in the last two rows of Table 1 are passed directly through to LmwProtocol. Those in the last two rows are handled first by Tapeworm, and then passed down to the lower level. Porting Tapeworm to another protocol would require changing the base class, and re-implementing the functions in the first two rows of Table 1.

Missing_data_tape() and *get_new_tape()* are functions provided by the underlying protocol to Tapeworm, and were discussed in Section 3. The primary requirement that they impose on the underlying protocol is a *versioning* capability. This is the ability to generate update summaries, to apply them at remote sites, and to ensure that consistency is not violated throughout. In LRC, shared updates are summarized as *diffs* [9], and can easily be added to messages and applied at remote sites. Consistency correctness is preserved because diffs carry enough consistency information to determine when and where they should be applied.

This requirement can be problematic in the case of single-writer protocols like sequentially consistent page-based protocols [10]. Such protocols provide no way to determine whether a given copy of an object is current or not. However, single-writer protocols with support for object versions, i.e. the single-writer LRC protocol [1], provide the necessary basic mechanisms.

Tape::add_data() and *Tape::read_data()* are also functions provided by the underlying protocol to tapeworm. They allow Tapeworm to copy the data named by a tape into or out of network messages.

Protocol::fault() and *Protocol::page_request()* are upcalls from the DSM 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 identifying and handling accesses by a consumer to producer/consumer regions.

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 2: Application Summary

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 3: Water

4.2 Interactions with the message 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 functionality could be made available without reference to specific message objects. However, this method would be less clear, so we have left the interface unchanged.

The last row of Table 1 shows upcalls from the DSM 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.

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 [11]) 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.

Table 2 summarizes the maximum performance improvements on each of our applications. 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 steady state behavior was measured. As with any performance numbers, these should not be viewed out of context.

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 4: TSP

Protocol	Speedup	Remote Misses	Updates Created	Updates Used	Comm Kbytes	Msgs				
						Lock	Barrier	Flush	Data	Total
Default	3.62	32677	4845	76%	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 5: Spatial

5.3 Application performance

Water

The 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 3 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 versus 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. In at least one place, the region actually passed to the lock is only a guess at the data that will end up being modified.

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 missed 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 4, TSP is almost exclusively lock-based. Barriers are used only during initialization and cleanup. We investigated both user locks and auto-locks. The results are shown in Table 4.

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 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 5 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

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 6: QS

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

Table 7: Barnes

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 3.3.

Table 6 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 are the entire centralized queue structure. As this structure is updated frequently, the 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 3.3. 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 to contain only barrier synchronization. Because of this modification, fine-grained tasks such as *make-tree* are now performed sequentially.

Table 7 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. 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 8. All remote misses are eliminated. However, overall speedup is still mediocre because the last iterations have too little computation to make parallelism worthwhile.

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

Table 8: Gauss

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 grows too large.

As far as the effectiveness of the specific synchronization library discussed in this section, Table 2 shows 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 3 through Table 7. 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 3.1, 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. 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.

Finally, one possible critique of this work is that it is too closely tied to the multi-writer LRC protocol used to generate the above results. Actually, this protocol is not an ideal substrate. The reason is that the tape protocol relies on noticing all shared modifications that occur when recording. However, multi-writer LRC uses lazy diffing, meaning that once a process has started writing a page, the page remains writable until any of the modifications are requested elsewhere. Hence, a tape that starts recording writes *after* a page has already been made writable will not necessarily notice additional writes to the same page. This problem could be avoided by removing write permission from all pages when any recording of writes is started. This will result in more complete information (and possibly fewer residual remote misses), but incur higher overheads for the actual recording.

6. Related work

Record/replay barriers were first implemented by the Wind Tunnel project [12]. 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 [13]. 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. Also, 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-DSM interfaces [14]. The *missing_data_type()* routine is essentially the information-

gathering phase of the TreadMarks [6] *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 [15].

Tapes are not a consistency mechanism, but they can be used to tune the interface of a CVM-like system so that it behaves as if a dissimilar underlying protocol were used. For example, our work could be used to build a synchronization interface that would closely approximate the data communication characteristics of Midway's [16] or CRL's [17] update protocol. While both systems differ from CVM in many ways, one of the key differences is that both Midway and CRL use update protocols. Unnecessary updates are avoided by limiting the updates to shared regions that are explicitly associated with synchronization. The auto-locks described in Section 3.1 would approximate these data movement patterns, modulo excess invalidations caused by false sharing.

Similarly, tapes can be used to build a synchronization interface that superficially mimics *scope consistency* (ScC) [18]. 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. Since the existing ScC implementation is home-based, all updates are constrained 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.

This paper has discussed the use of tapes to improve performance of software DSM 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 [19] implement user-initiated data movement on top of the underlying consistency protocols. Other work [20] generalized these primitives to allow the destination of pushes to be specified either by runtime copyset management or by specific calls initiated by application programs. By augmenting these primitives with the ability to read and store copyset information for future iterations, 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.

Shared memory systems with dedicated protocol processors [21, 22] 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.

7. Conclusions

This paper has described the tape mechanism, and its use in implementing high-performance synchronization libraries. Tapes can be used to tailor 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. Future work will investigate more sophisticated automatic interfaces, and the use of tapes in creating debugging libraries [23].

8. References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software Write Detection for Distributed Shared Memory," in *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, November 1994.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

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