Sparks: Coherence as an Abstract Type

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

Abstract

Sparks is a protocol construction framework that treats records of coherence actions as abstract types. Sparks' central abstraction is the coherence history, *an object that summarizes past coherence actions to shared segments. Histories provide high-level access to coherence guarantees. We motivate our work by discussing synchronization design in distributed shared memory systems, and show how histories can be used to cleanly create more efficient synchronization than is currently used.*

1 Introduction

This paper discusses the use of *write-histories* (or just *histories*) to refer to past coherence actions in the context of software Distributed Shared Memory systems (DSMs). A history is an abstract object that summarizes past modification to shared segments. Histories can be compared, added, and subtracted. We advocate using histories in order to address two problems with current DSM systems: (1) a mismatch between system and application semantics, and (2) a lack of any high-level mechanism with which to implement automatic or semi-automatic prefetching.

The first point is mainly concerned with synchronization behavior of shared-memory applications. Most DSM systems provide efficient synchronization primitives that are implemented along side of coherence mechanisms, not on top of them. The reason for the separate implementation is that shared memory and synchronization have very different semantics, and it is therefore difficult to efficiently implement the latter on top of the former.

However, most system typically provide support for only a very limited set of synchronization types, such as barrier and locks. Some systems additionally provide support for reduction types [4], but, in general, application programmers are expected to implement high-level synchronization types on top of low-level synchronization types. This approach is inherently inefficient, because the aggregate semantics of a high-level synchronization type implementation can usually only be approximated by using lower-level synchronization types as building blocks, and the approximation must necessarily be conservative. Hence, such high-level types often have much higher runtime costs than strictly required by application semantics.

By expressing coherence actions as a high-level object, we hope to make explicit coherence actions efficient, yet writable by ordinary mortals. At the same time, histories are a precise enough notion that powerful and varied synchronization types can be expressed in them without unintended consequences.

The second major use of the the history mechanism will be in capturing past shared accesses and using them to predict and anticipate future accesses. This technique uses *read-histories*, a variant of histories that records *access misses* on shared pages rather than shared modifications. Read histories are used to record access misses during one iteration of an outer loop, and to later *replay* an approximation of the data movement initiated by those misses so as to prevent future access misses. Hence, such replay mechanisms will be a a form of prefetch, and will help hide the latency of the shared accesses.

This technique has been used before[9], but only as an ad hoc technique written specifically for a single application. By using histories, the process is semi-automated. Recordings are made essentially by taking snapshots of a process's read history before and after the region to be recorded. The earlier snapshot is *subtracted* from the later, leaving a record that consists only of access misses incurred during the recording period.

When the program or user determines that a similar access pattern is occurring (i.e. the next iteration or the outer loop), any modifications to the shared pages that were missed during the recording are sent to the process that missed on them. Using histories, the entire mechanism can be expressed in only a few lines. However, the mechanism is powerful enough to allow the prefetch to apply only to a single spec-

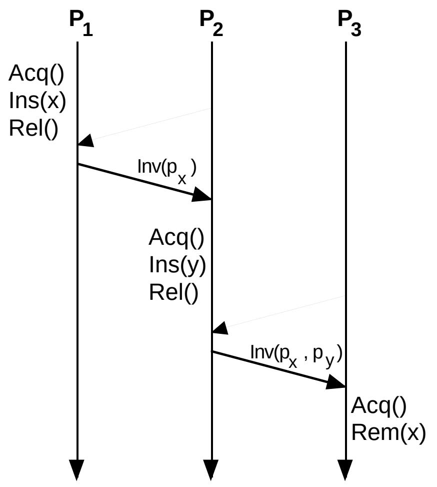

Figure 1. Lock-based queue

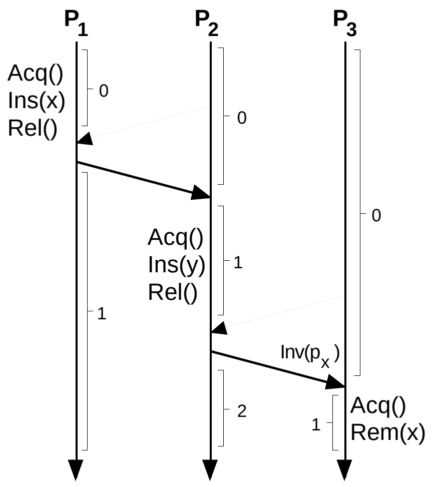

Figure 2. Sparks-based queue

ified range of addresses, the extent of a given object, or to only a single thread of a multi-threaded process. A similar mechanism can be used to initiate bulk movement in producer-consumer interactions.

The rest of the paper proceeds as follows. Section 2 describes the relaxed consistency models from which the history mechanism grew. Sections 3 and 4 describe the use of histories in building efficient high-level synchronization types, and Section 5 concludes.

2 Background

DSM systems support the abstraction of shared memory for applications running on loosely-couple distributed systems, i.e. workstations on a general-purpose network. While early systems strictly emulated the *sequentially consistent* [8] programming model of tightly-coupled multiprocessors, most recent systems support relaxed consistency models such as lazy release consistency (LRC) [5], a close relative to the *eager* release consistency (ERC) [3] memory model. DSMs that implement ERC delay propagating modifications of shared data until they execute a release, and then the modifications are performed globally. Under LRC protocols, processors further delay performing modifications remotely until subsequent acquires by other processors. Additionally, the modifications are only performed at the processor that performed the acquire. The central intuition of LRC is that competing accesses to shared locations in correct programs will (almost) always be separated by synchronization. Since coherence operations are deferred until synchronization is acquired, we can piggyback consistency information on the existing synchronization messages. In general, LRC performs better than ERC by eliminating consistency messages and further hiding the effects of false sharing.

The Sparks class library can be used to build high level synchronization objects that accurately reflect the synchronization objects' coherence semantics. Our approach is related to the causality annotations of CarlOS [7], but Sparks will provide a much richer set of mechanisms and finer control over the scope of consistency actions. Sparks will replace the top layer of CVM. Since coherence in LRC systems like CVM is driven by synchronization, it is also entirely proper to view Sparks as a toolkit with which to write DSM protocols.

3 Synchronization Support

DSMs typically separate synchronization support from shared address space support in order to achieve good performance [1, 2, 6, 4]. Such systems provide a limited set of synchronization primitives (locks, barriers), and expect application programmers to build sophisticated synchronization constructs in terms of them.

However, building high level synchronization objects out of synchronization constructs supplied by the DSM system (such as locks or barriers) is often inappropriate, because the coherence constraints implied by the DSM constructs may be more strict than those needed by the high level object. Figures 1 and 2 show lock-based and Sparks-based distributed queue implementations in an LRC environment. In both cases, process P_1 creates and inserts item x, P_2 creates and inserts item y, and P_3 retrieves item x. LRC systems transitively require the acquirer of a lock to see all shared updates seen by the last releaser. In the lock-based queue of Figure 1, both P_2 and P_3 see all updates seen by P_1, and P_3 sees all updates seen by P_2. More to the point, P_2 invalidates its copy of the page containing x and P_3 invalidates its copy of the pages containing both x and y. However, P_2 never needs to see x. It merely transfers knowledge of x's creation from P_1 to P_3. Similarly, P_3 does not need to know about y. Therefore, neither P_2's invalidation of the page containing x, nor P_3's invalidation of the page containing y are necessary. In general, applying unnecessary coherence operations can waste bandwidth, create extra CPU overhead, and cause unnecessary page faults, especially in the presence of false sharing.

3.1 Histories

History objects allow users to express and manipulate coherence constraints. By applying one node's current history at another node, the second node's view of shared state is brought up to date with respect to events seen by the first.

More formally, a history is a partially ordered set of *intervals* [5], where an interval describes a portion of the execution of a single processor. Intervals contain *write notices*, which are generally just indications that a given page has been modified. Applying such a notice usually invalidates the associated page. However, a write notice may also contain the newly written data, and hence application of the write notice updates the page instead of invalidating it. Intervals represent a logical unit of time; they have no correspondence with real time. In a distributed system, new intervals typically start at each non-local synchronization event.

Histories have three types of extent: a *temporal* extent, a *segment* extent, and a *thread* extent. The temporal extent specifies the interval of time for which events are summarized. A limited temporal extent can be used to name only those events that occurred during part of an execution, such as between two synchronizations. Temporal extents are described by using *version vectors* to summarize the earliest and latest included intervals of each processor in the system. The temporal extent of P_3 after the lock acquire in Figure 2 could be written as:

$$\{\bot, \bot, \bot\} \;\; \{0, 1, 1\} \tag{1}$$

meaning that the history summarizes all intervals from the start of execution to i_1^0 (interval 0 of P_1) on P_1, to i_2^1 on P_2, and to i_3^1 on P_3.

The segment extent names the segment of shared memory that may be affected by the history's write notices, i.e. all those pages for which the history might carry write notices. The primary purpose of the segment extent is to limit the scope of a history's consistency actions to a subset of shared memory. While a segment in a page-based DSM consists of a set of pages, segments could also be composed of arbitrarily-shaped objects in distributed object systems such as Midway [1], or CRL [4].

```
class History {
     TemporalExtent temporal;
     SegmentExtent  segment;
     ThreadExtent   thread;

     void           register(int on_or_off);

     void           operator += (History *);
     void           operator -= (History *);
     void           apply();
     UpdateData     *get_data();
};
```

Figure 3. `History` Class

The thread extent names the set of threads whose write notices may be contained in the history. Usually this includes all threads in a system. For example, the thread extent of H_3 is P_1, P_2, and P_3. However, limiting the thread extent has several uses, including limiting the information passed to a global barrier by each node (each may wish only to inform the barrier master about local intervals), and integrating prefetching with thread scheduling on multi-threaded nodes.

A history's write notices are contained within the intersection of the temporal, segment, and thread extents. In Figure 2, interval i_1^0 contains a single write notice for the page containing x.

3.2 Operations on Histories

History semantics allow for addition, subtraction, and application. *Adding* histories H_i and H_j results in a new history that contains all intervals named in either H_i or H_j. *Subtracting* histories can be use to limit temporal scope. Subtracting H_i from H_j limits the temporal scope of the resulting history to the interval of time seen by H_j but not by H_i. History subtraction can be used to create a compact representation of all shared updates to the extents covered by a history during a specific interval of time. Finally, *applying* a history at a node takes consistency action corresponding to the notices named by the history, usually invalidation.

4 Programming with Sparks

The initial prototype of Sparks is being written as a C++ class library. Later versions may migrate to a language-based approach as we expand the scope of the research to include compiler-based analysis of synchronization and automatic protocol verification.

A simplified definition of the History class is shown in Figure 3. This definition allows histories to be added, subtracted, and applied. Additionally, some protocol implementations of `get_data()` will return all data present locally whose creation is described by the history's write notices. The `apply` routine can be used to update pages when the history is applied elsewhere. The `register` routine is used to tell Sparks to begin recording shared writes in a given history.

Adding histories H_i and H_j results in a new history that contains all intervals named in either H_i or H_j. For example, the coherence operations that take place in a lock acquisition on an LRC system can be expressed by:

```
    $H_{acq}$ += $H_{rel}$;
    $H_{acq}$.apply();
```

where H_{rel} refers to the current history of the last releaser of the lock, and H_{acq} refers to the current history of the acquirer. The existence of a history detailing modifications to shared memory does not imply that any coherence operation has taken place. Consistency action only occurs when a history is `applied` to the local version of shared memory. In the above example, the first line merely creates a description of shared modifications seen by either the acquirer or releaser. No action is performed until the resulting history is applied in the second line. All three extents may be modified by an addition.

Histories may also be *subtracted*. Subtracting H_i from H_j limits the temporal scope of the resulting history to the interval of time seen by H_j but not by H_i. History subtraction can be used to create a compact representation of all shared updates to the extents covered by history H_{in} during a specific interval of time:

```
    History             $H_{save}$;
    extern History      $H_{local}$;

    void begin_record () {
        $H_{save}$ = $H_{local}$;
    }

    History * end_record () {
        return $H_{local}$ - $H_{save}$;
    }
```

where we assume H_{local} is registered (recording is turned on) and has been tracking local accesses. The history returned by `end_record` contains a complete record of the intervals that were created or learned about between the calls to `begin_record` and `end_record`. The next section presents possible uses of this type of construction.

4.1 High-Level Synchronization: Queues

As discussed above, unintended consequences can result from using constructs as powerful as Locks to build high level synchronization types. In the case of the lock-based queue in Figure 1, the unintended consequences are processor P_2's invalidation of page p_x, and P_3's invalidation of p_y. The only intended consequence is P_3's invalidation of p_x.

The Sparks-based queue implementation in Figure 2 stores the history of the data producer with the object in the queue. When the data is consumed by P_3, P_1's history is applied P_3.

4.2 Reductions and Mutual Exclusion

Many operations in parallel programs can be described as reductions, or operations that are associative and commutative. The semantics only require mutual exclusion between consecutive reducers. However, reductions are typically implemented using locks. Locks are stronger than necessary because their implementation updates later reducers with all coherence actions taken by prior reducers. The only coherence actions that need to be performed are those to the data modified by the reduction.

Reductions can be implemented in Sparks similarly to locks, except that temporal and segment extents limit the scope of the histories transferred between consecutive reducers. The below code presents the relevant aspects of a reduction:

```
    reduce(SegmentExtent * object) {
        send request for object to current owner
        extract history $H_{obj}$ from reply
        $H_{obj}$->apply();
        $H_{obj}$->register(TRUE);

        compute reduction()

        $H_{obj}$->register(FALSE);
        $H_{obj}$->segment = object;
    }
```

The first two lines fetch the reduction token, together with H_{obj}, the history of previous reductions on that object. Next, H_{obj} is applied, and then registered in order to record new actions into the reduction object. After the reduction has been computed, H_{obj} is un-registered, and the segment extent is limited to the object. This last step is necessary because the reduction may have modified shared data outside of `object`. H_{obj} is then ready to be passed to the next reducer of the same object.

4.3 Prefetch Playbacks

Prefetch playbacks is a technique that allows us to *record* access misses taken during one iteration, and to *play back* the next update to the same data as an update during a subsequent iteration.

The coherence histories described so far are essentially records of write faults. We can use a similar mechanism to record read faults. If we assume routines analogous to `begin_record` and `end_record` for `ReadHistory` objects, the following code could pass a record of a compute phase's read misses to synchronization routines for replay during the next iteration:

```
for (...) {
        ...
    barrierX();
    begin_read_record();
        compute();
    ReadHistory *$H_{local}$ = end_read_record();
    barrierX− >attach($H_{local}$);
        ...
}
```

We are assuming that `begin_read_record()` and `end_read_record()` allow a history of read misses to be captured, that `barrierX()` is a global barrier operation, and that the `attach` operation lets us inform the barrier of the misses that we took subsequent to leaving it. During the next iteration, `barrierX` will use H_{local} to disseminate the read miss information on barrier releases, allowing other processors to stream in data they produce *before* the misses occur again.

Recording and playing back data transfers was first used by the Mukherjee [9] in the context of a sequentially consistent DSM. Our work differs in two ways. First, our recording mechanisms will be part of the synchronization type definitions. The playbacks will be initiated by automatic heuristics, making them more reliable and easier to apply. All of the above mechanism could have been hidden inside special-purpose barrier routines provided by library builders. We pulled much of it outside the barrier routines for explanatory purposes. Second, our technique will be used for prefetching, not to maintain coherence. We will not violate correctness if subsequent iterations access different data.

5 Conclusions

Parallel systems are clearly reaching a point where increasing affordability is making their widespread acceptance possible. However, this transition will not take place unless parallel machines are easy to program, and perform well. Current DSM systems handle the first problem, but do less well with the second.

Our research will bridge the gap between loosely-coupled and tightly-coupled systems by using the Sparks abstractions to reduce and optimize data movement in DSM systems. As large-scale systems increasingly resemble multiprocessor nodes connected by DSM, we expect our techniques to become common not only in clusters of stock workstations, but in the most powerful systems as well.

An implementation is underway.

References

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

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

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

[4] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. To appear in *The Proceedings of the 15th ACM Symposium on Operating Systems Principles*.

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

[6] 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*, pages 115–131, January 1994.

[7] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-driven relaxed consistency in a software distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 75–86, November 1994.

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

[9] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the 1995 Conference on the Principles and Practice of Parallel Programming*, July 1995.