Object Views: Bridging the Performance Gap Between Shared Memory and Message Passing

Ilya Lipkind and Vijay Karamcheti
Courant Institute of Mathematical Sciences
New York University, New York, NY

Abstract

A shared object space simplifies the programming of parallel and distributed applications, but incurs a performance penalty as compared to message-passing programs when supported transparently using software DSM techniques. The penalty results from a mismatch between the sharing granularity and protocols used in the DSM layer and the consistency expectations of computation threads.

This paper describes object views, generalized interfaces that restrict how computation threads can use an object, and implicitly inform the underlying implementation about the potential side effects of object access. This information enables support of a shared object space with message passing-like overheads, improving performance by customizing coherence protocols to application requirements. We discuss language and compiler support for object views, and demonstrate their performance advantages using both parallel and distributed applications.

1 Introduction

Programming models that support a shared object space simplify the programming of parallel and distributed applications. However, these advantages come at a cost: transparently supporting the shared object space using software distributed shared memory (DSM) techniques [1, 15, 7, 3] incurs a performance penalty as compared to an explicit message-passing version of the program.

The fundamental reason for this performance disparity is a mismatch between the sharing granularity and protocols adopted in the DSM layer, and the consistency expectations of application threads accessing the shared objects. Coherence actions in the DSM layer are triggered on demand, and in the absence of knowledge about how the application accesses the object, tend to take conservative actions such as keeping the entire object state consistent after an update and/or serializing potentially conflicting accesses. The consequence is that, most applications do not achieve good performance.

In this paper, we present *object views*, a programming and execution abstraction that conveys consistency expectations of the computation threads to the underlying implementation. Object views serve as generalized interfaces in an object-oriented language, which in addition to advertising object functionality also specify restrictions on its use (e.g., a thread can only invoke a subset of the object's methods). All object accesses are through its views; from a thread's perspective, the latter are indistinguishable from an object. Views inform the implementation about potential side effects of object access, facilitating use of efficient caching protocols customized to application requirements. For example, knowing the invoked method subset allows the implementation to cache a copy of the object that is consistent only on the fields accessed by these methods. In general, object views improve performance by reducing transferred object state, by incurring fewer invalidations/updates, and by increasing the number of concurrent operations. Moreover, they provide an elegant abstraction for expressing domain-specific caching optimizations (e.g., aggregating updates into a local copy) , while retaining the intuitive interface of a shared object space.

Realizing the full performance benefits of object views requires both language and compiler support. We describe extensions to the Java programming language, where the addition of two keywords—**view** and **represents**—facilitates the specification of general object views. Compiler support takes the form of analyses that first derive a model of thread interactions in the program, and then use this model in the context of the program synchronization structure to infer the *minimal* coherence requirements for views accessed by a thread. The language extensions and compiler infrastructure are being developed as part of the InSight system at New York University. Based on application studies involving a parallel application (Radiosity), and a distributed application (Shared Whiteboard), we find that object views are naturally derivable from the program structure and enable overall application performance to match that achieved using low-level programming approaches.

The rest of this paper is organized as follows. Section 2 presents the key concepts behind object views. Sections 3 and 4 describes required language and compiler support for views, and discusses their performance advantages using application studies. Section 5 places our work in context with related efforts and we conclude in Section 6.

2 Object Views

Existing DSM techniques are conservative because they initiate coherence actions at the level of memory accesses *without utilizing any higher-level semantic information about the access*. Consequently, they define object copies to be consistent only if they reflect identical state, maintain globally accessible information (the *directory*) about object state and its copies, query this information prior to satisfying an access request, and conservatively invalidate or update conflicting copies before granting access. Such coupling of activities across different processors degrades performance for a software DSM system.

Object views are a programming and execution abstraction, which provide the underlying implementation with application-level information about what it means for object copies to be consistent, and how to effectively implement the required coherence actions. They serve as generalizations of interfaces in object-oriented languages, providing a restricted handle through which a shared object is accessed. Although indistinguishable from a thread's perspective, object views differ in their consistency requirements.

2.1 View Composition

An object can be associated with three kinds of views: a *base view*, a *subset view*, and an *arbitrary view*. Figure 1 shows the coherence traffic requirements for these three, described in additional detail below.

Base views serve as a baseline, where the underlying DSM maintains coherence making only conservative assumptions about thread behavior (e.g., that any thread can invoke any method on the object).

Subset views restrict the behavior of a thread by exposing only a subset of the publicly available functionality of the object. This information enables a compiler to determine the minimal transfer of object state needed for consistency. The intuition is that a cached copy supporting a particular view need only be invalidated (or updated) if there is concurrent access to the object by another thread using a conflicting view. In addition to providing an intuitive abstraction, subset views allow the underlying implementation to aggressively cache the object while maintaining coherence at sub-object granularity.

Arbitrary views augment subset view functionality by allowing view-local fields and methods. These views permit the expression of application-specific caching protocols where threads can obtain a snapshot of the object, compute with respect to this snapshot, and finally merge their changes into the object. In contrast to explicit management of cached copies, arbitrary views still present the programmer with the abstraction of accessing a single shared object. Moreover, reconciliation of view state is triggered automatically whenever another thread accesses the object using a conflicting view. View acquisition and reconciliation can be specified in a flexible fashion using extract and merge callbacks.

Example 1: Consider the natural parallelization of a five-point stencil computation on a 2D grid. If each grid partition is a different object, then a thread operating on a partition needs to access the row and column values contained in the objects to its north, south, east, and west. In the absence of views, caching the entire object is expensive, particularly since the cached values are updated each iteration. Accessing the object using views that permit access to the appropriate rows and columns alone drastically reduces the amount of data that must be transferred.

Example 2: Arbitrary views allow consistent sharing of complex objects in heterogeneous environments. The challenge here arises from the fact that client machines may not be capable of retrieving and storing the entire object, yet still benefit from caching a view of it. For example, consider a rich-text document being accessed both from a workstation and a PDA; the

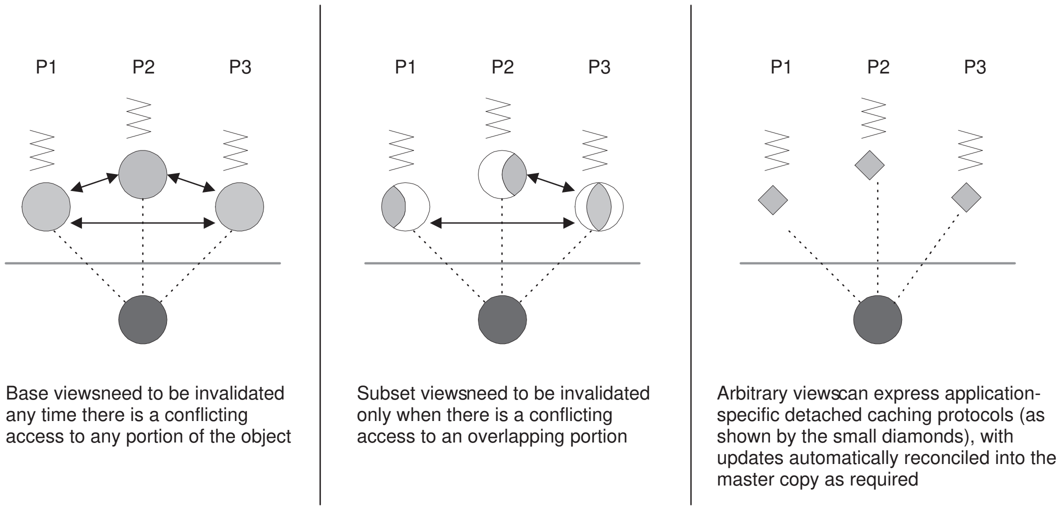

Figure 1: Coherence traffic requirements of different kinds of views. The gray circles represent cached copies of the object, with arrows indicating conflicts that require invalidation.

latter is only capable of operating against a distilled view of the document. Arbitrary views can be used to conveniently specify domain-specific methods for extracting this distilled view from the original document and reconciling any updates.

2.2 View Coherence Requirements

Object views improve performance by enabling the use of custom protocols to implement required view coherence actions. The intuition for these custom protocols stems from the fact that the consistency expectations of a thread accessing object state are determined by the synchronization structure of the program, and more generally by the specific thread interactions implicit in the program specification. Object views are an appropriate granularity for either specifying or automatically inferring these consistency requirements. For example, in the grid computation described above, the synchronization structure implies that even though row and column views are updated every iteration, a thread does not need access to these updates until the next iteration. This information enables the use of a custom protocol (e.g., producer-consumer) for caching such views.

In general, custom protocols can broadly be classified into three categories—*synchronous*, *asynchronous*, or *detached*— distinguished by how the consistency of the corresponding view is coupled with the state of the home node copy:

- *Synchronous views* are tightly coupled with the home node copy: all object views have state identical to that at the home node.
- *Asynchronous views* are loosely coupled: object views differ for a period of time but eventually become consistent. This provides the underlying implementation the freedom to decouple protocol actions from data transfer, and optionally delay updates to coherence meta-data structures.
- *Detached views* have no coupling with the copy at the home node: they are essentially a snapshot of the current state there. Detached protocols require only data transfer actions.

In addition to reducing the frequency of coherence and required message traffic, asynchronous and detached protocols have the additional advantage of being able to effectively exploit advanced capabilities such as remote memory access becoming available in current-day networks like Myrinet and VIA. These capabilities require separation of data transfer from associated synchronization, exactly the information provided by the custom protocols. For example, a detached view can be obtained using a "get" operation without any remote processor involvement.

3 Language and Compiler Support

In this section, we describe language and compiler support for object views, being developed as part of the NYU InSight project. Language support is used to identify *view composition*, leaving the compiler to determine *view coherence requirements*, i.e., how best to implement the coherence actions required to cache the view. Such a partitioning allows a programmer to express views that naturally suit the problem, additionally incorporating any domain-specific optimizations. The compiler analyzes global program properties to infer use-dependent view coherence protocols.

3.1 VJava: Language Support for Views

VJava extends the Java programming language with language support for object views by defining two new keywords: **view** and **represents**. The **view** keyword is used to define a view, matching Java class definitions with an argument reflecting the view's underlying class. Views constrain how different computation threads can use objects of this class, serving as a generalized interface whose instances provide a restricted handle for object access. The **represents** keyword specifies the restriction, remapping object fields and methods into view fields and methods (with optional side-effect information). The syntax of a VJava view definition is as follows:

```
view ViewName represents ClassName
                 extends SuperViewName {
  // remapped object fields, with side-effect info
  [final] remapFld1 represents type objFld;
  // view-local fields
  type viewFld2;
  // remapped object methods
  remapMeth1 represents type objMeth( signature );
  // view-local methods
  type viewMeth2( signature );
  // callback methods for arbitrary views
  [ void extractView(); ]
  [ void mergeView(); ]
}
```

Notice that **final** indicates read-only access to a field, while the **represents** keyword both associates a view with the class and maps class fields and methods into renamed view fields and methods. The **mergeView** and **extractView** methods provide the callback functionality to support coherence of arbitrary views, as described in Section 2.

Subset views are implicitly specified by remapping a subset of the class methods into the view. This presents a natural interface to the programmer, while still allowing a preprocessor to conveniently infer the object state subset accessed by the view and the corresponding mode of field access (e.g., read-only or read-write). The side-effect information determines the coherence requirements of the view. For example, copies of views that read a field are invalidated upon object access using a view that writes that field.

Arbitrary views, typically used to express domain-specific caching optimizations, are specified by including view-local fields and methods in addition to those remapped from the object. View-local methods can only operate on view-local fields and remapped object fields. The former are initialized from and reconciled with global object state by overriding the **extractView** and **mergeView** methods. The coherence requirements of an arbitrary view is specified in terms of its side-effects on object state. The programmer can override default side-effects by specifying modes of access for the remapped object fields.

3.2 Compiler Analyses and Optimizations

An optimizing compiler can improve performance by replacing synchronous coherence protocols with detached and asynchronous protocols based upon an analysis of global program behavior. Because these latter protocols minimize remote processor involvment, they can take advantage of capabilities such as remote memory access being encountered in high-performance networks like Myrinet and VIA.

Our compiler infrastructure extends interprocedural concurrency [11, 9] and side-effect [5] analyses for SPMD programs previously reported in literature. These analyses, optionally augmented with run-time profiling information, infer the *minimal* coherence requirements for each view access. Specifically, three kinds of analyses are performed (see Figure 2):

1. *Thread-family analysis* identifies code segments that are executed by each program thread. Distinct portions of the program control-flow graph (CFG) rooted at different thread creation sites correspond to different *thread families*.
2. *Concurrency analysis* uses the program sychronization pattern to fragment thread-family CFGs into subportions that may execute concurrently from those that do not.
3. *Side-effect analysis* complements user-supplied view information to compute the set of elements (object state) that are read and written by the identified fragments, summarizing this information for each user-defined object type.

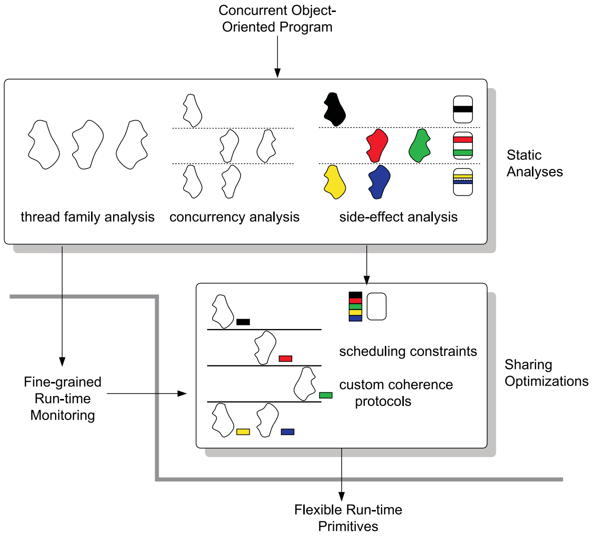

Figure 2: Compiler analyses develop a model of thread interaction, which drives the construction of optimized view coherence protocols.

Given thread-family CFGs and their side-effects, the compiler next derives per-fragment custom protocols which satisfy consistency requirements implied by fragment interactions.

Synchronous protocols are used when object updates should be visible to conflicting accesses in the same phase, and are implemented identical to object DSM protocols, albeit at a smaller granularity.

Asynchronous protocols are used when an access is separated from both the last conflicting view access as well as the next conflicting view access by intervening synchronization points: this enables the protocol actions to be decoupled from data transfer actions. The two synchronization points define the lifetime of a cached view, or alternately the asynchrony allowed in protocol actions. For example, if a phase only contains read accesses without an intervening write access, an asynchronous protocol can obtain read views without consulting the home node for the existence of any writers. Moreover, informing the home node about the readers can be delayed till the end of the phase, and can be completely avoided if the cached copies are self-invalidated. In some cases, synchronous protocols can be converted to asynchronous protocols by controlling which thread fragments are concurrently scheduled at run time.

Detached protocols are a special case of asynchronous protocols, and are used when the current access is not followed by any conflicting access.

These custom protocols are implemented using a toolkit of flexible, composable object coherence primitives. The reader is referred to [8] for additional details about these primitives.

4 Application Studies

We evaluate object views using a parallel C++ application (Radiosity) running on the SGI Origin, and a distributed Java-based application (Shared Whiteboard) running on a Myrinet-connected cluster of PCs. In both cases, the shared object space is synthesized using a lean software DSM layer [8] built on top of Illinois Fast Messages (FM) [12].

4.1 Radiosity

The radiosity method computes illumination in a scene made up of diffusely reflecting surfaces. It iteratively computes patch radiosity as a linear combination of radiosities of patches on an interaction list, hierarchically subdividing patches for accuracy. The above requires precomputation of an interaction's visibility and formfactor, accessing potentially remote patch state. The computation terminates when the change in total radiosity falls below a threshold.

Radiosity admits two object views. The first, a detached view, corresponds to a patch object portion that is written once and not subsequently modified (e.g., patch geometry). This view results in an object copy inconsistent at the object level, but consistent for the relevant portion. The second view, an asynchronous view, exploits a race in the application specification permitting concurrent accesses and updates to the patch radiosity field within each iteration. The view results in an object copy with the radiosity field consistent up to the end of the previous iteration. Caching such a view reduces invalidations, and permits overlap of read accesses with updates.

Figure 3 shows the performance impact of object views for various communication interfaces on the Origin. The MSG, PUT/GET, and LD/ST interfaces correspond to support for messaging, non-atomic remote memory-access capabilities (e.g., VIA), and remote atomic operations (e.g. Cray T3E), respectively. CC-LD/ST represents Origin-like support for hardware cache-coherence. The speedup plots and execution time breakdowns show that caching at view granularity improves performance across a range of net-

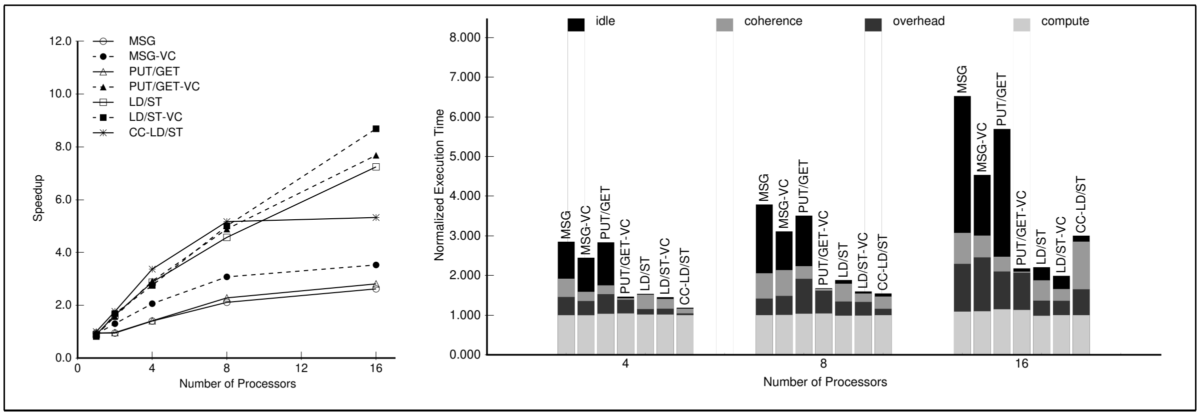

Figure 3: Performance of the Radiosity application using object views (denoted by *-VC) on the Origin.

work capabilities, enabling software DSM to achieve efficiency comparable to hardware DSM. Also, as seen for PUT/GET, object views can effectively exploit network remote memory access capabilities.

4.2 Shared Whiteboard

A whiteboard provides users the ability to concurrently inspect and modify the same document, and see changes made by others reflected in their own view. Such an application naturally supports multiple object views, corresponding to different operations. For example, views can capture differences between superficial and detailed object inspections, and modification operations that change object position from those that modify its internal attributes. These multiple views reduce overall coherence traffic, while still maintaining object consistency. Note that additional improvements are possible using views that encapsulate domain-specific caching optimizations.

To characterize the performance impact of object views, we built a VJava-based whiteboard application, and compared its message traffic against three other approaches. The first two represent *multicast*- and *event*-based approaches, where each update is propagated to all interested parties, accesses incur no traffic, and no consistency guarantees are provided. The third comparison point, *object caching*, maintains coherence at full object granularity.

Figure 4 shows the message traffic incurred in the four approaches for a whiteboard randomly accessed by different numbers of threads on a Myrinet-connected PC cluster. Our results, normalized to the multicast traffic, show that object views reduce both data and control message traffic as compared to traditional object caching by as much as a factor of 5. Moreover, views incur less data traffic than even multicast- and event-based approaches, despite the latter's lack of consistency guarantees. Although this reduction comes at the cost of increased control message traffic, the latter are much smaller than data messages. With the availability of high-performance FM-like messaging layers, small messages can be supported with very small overheads, and consequently, message size (rather than number) becomes the dominant contributor to communication overheads.

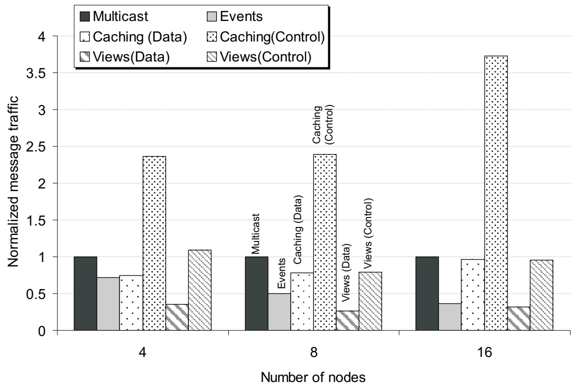

Figure 4: Performance of the Shared Whiteboard application on a Myrinet-connected PC cluster.

5 Discussion and Related Work

Programming models that provide a shared object space abstraction simplify programming, but have traditionally been considered to incur higher overhead than explicit message passing. We have described how object views can bridge the performance gap between software shared memory and message passing, without compromising on programmability.

Several researchers have looked at improving performance of software coherence schemes, motivated by their wider applicability and flexibility as compared to hardware solutions. Starting from Ivy [10], a number of systems support a shared object space at either fixed granularity (e.g., TreadMarks [1], Shashta [14], and AURC [6]) or variable (object) granularity (e.g., SharedRegions [13], Midway [3], SAM [15], and CRL [7]). Recent research uses weak consistency protocols such as lazy release consistency [1] and entry consistency [3] that postpone propagation of object updates until program synchronization points. Although these protocols achieve good performance on small applications, their scalability is limited by the fact that, in the absence of information about application behavior, they are forced to rely on conservative decisions. For example, most schemes maintain coherence at entire object granularity, propagating updates for all accesses or disallowing potentially conflicting operations.

A few systems (e.g., Munin [4] and Orca [2]) have exploited object usage information (e.g., an object is read-only) to reduce coherence traffic. However, their reliance on global object properties limits optimizations to cases when *all* object accesses satisfy the desired property. Object views address the same goal but relax this latter restriction considerably. Our work is most similar to database schemas; however, unlike the structured nature of schemas, object views can be used to express and enforce very general consistency requirements. Moreover, we are not aware of previous work supporting coherence in a non-flat shared object space, crucial in heterogeneous environments.

6 Conclusion

In this paper, we have introduced the concept of object views, a programming and execution abstraction that provides a restricted handle for accessing the object, enabling the underlying implementation to efficiently support a shared object space using coherence protocols customized to application requirements. We have shown that object views are convenient to specify and use, and with appropriate compiler technology can produce coherence protocols that yield efficiency comparable to message-passing programs. Application results verify that object views can yield substantial performance improvements in both parallel and distributed applications.

Acknowledgements

The authors would like to thank Igor Pechtchanski for his assistance in implementing the VJava run-time. Vijay Karamcheti was supported in part by an NSF CAREER Award CCR-9876128.

References

[1] C. Amza et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), February 1996.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Software Engin.*, 18(3), 1992.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *COMPCON*, pages 528–537, Mar. 1993.

[4] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP*, pages 152–164, 1991.

[5] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.

[6] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *HPCA*, pages 14–25, February 1996.

[7] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *SOSP*, pages 213–228, 1995.

[8] V. Karamcheti and A. A. Chien. Architectural support and mechanisms for object caching in dynamic multithreaded computations. *Journal of Parallel and Distributed Computing*, 1999. To appear.

[9] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 1996.

[10] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[11] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *PPoPP*, pages 129–138, May 1993.

[12] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and mpps. *IEEE Concurrency*, 5(2):60–73, 1997.

[13] H. S. Sandhu, B. Gamsa, and S. Zhou. The shared region approach to software cache coherence on multiprocessors. In *PPOPP*, pages 229–238, July 1993.

[14] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS*, pages 174–185, 1996.

[15] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *OSDI*, pages 101–114, 1994.