View Caching: Efficient Software Shared Memory for Dynamic Computations

Vijay Karamcheti and Andrew A. Chien
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Software distributed shared memory (DSM) techniques, while effective on applications with coarse-grained sharing, yield poor performance for the fine-grained sharing encountered in applications increasingly relying on sophisticated adaptive and hierarchical algorithms. Such applications exhibit irregular communication patterns unsynchronized with computation, incurring large overheads for synchronous (request-reply) DSM protocols that require responsive processing of coherence messages. We describe a new DSM framework, View Caching, that addresses this problem by utilizing application knowledge of data access semantics to enable the construction of low-overhead, asynchronous coherence protocols. Experiments on the Cray T3D show that view caching enables efficient execution of fine-grained irregular applications, reducing both coherence overheads and idle time to improve performance by up to 35% over a weakly-consistent DSM implementation.

1 Introduction

Software distributed shared memory (DSM) provides a shared address space on distributed memory computers without hardware support for cache-coherence. While existing DSM techniques are effective for coarse-grained sharing, they yield poor performance for the fine-grained sharing encountered in dynamic, irregular computations. Such computations are becoming important due to the emphasis on sophisticated adaptive algorithms and irregular pointer-based data structures which significantly reduce sequential work by focusing computational effort where necessary. These computations are frequently expressed using a high-level programming approach (exemplified by concurrent object-oriented [7, 18] and multithreaded [9] systems) as a dynamic collection of light-weight threads operating in a shared object space. The dynamic nature precludes object placement to minimize communication, so efficiency demands effective runtime caching of remote object accesses.

However, the fine-grained and dynamic nature of computation limits the effectiveness of existing DSM implementations. The former rules out relatively high-overhead DSM schemes, while the latter produces unpredictable communication unsynchronized with computation, leading to arbitrary delays in coherence message processing. Current DSM systems [14, 1, 16, 12] rely on weak consistency models to reduce coherence traffic, but still retain the synchronous (request-reply) nature of hardware cache-coherence protocols and require responsive processing of coherence messages for performance. Poor responsiveness degrades performance by increasing the time a processor idles awaiting completion of a coherence operation. Other DSM techniques [8, 15] which optimize synchronous protocols by predicting object access patterns are less effective given the dynamic context. Recent systems [3, 11] which allow application customization of coherence protocols provide mechanisms for improving performance, but offer little insight into how to build protocols for tolerating coherence processing delays arising from unresponsive processors.

We describe a new caching framework – View Caching – that enables the construction of custom asynchronous coherence protocols requiring less synchronization among processors. Our framework utilizes knowledge about data access semantics (the *view*) to infer global information about an object and its copies. Availability of such information eliminates the need to acquire it at runtime, facilitating low-overhead protocols that are robust to delays in processing coherence messages. The framework decomposes coherence actions into three components: `access-grant`, `access-revoke`, and `data-transfer`; the first two manipulate global information about the state of an object and its copies, while the third copies object state. Coherence protocols are constructed by assembling customized implementations of these components; application knowledge drives their customization. To keep customization manageable, our framework allows the selection of a component implementation from among a predefined set by specifying values for a series of parameters.

We contrast the performance of view caching with a weak-consistent DSM (supporting object consistency) for two applications expressed in a concurrent object-oriented language. Measurements using the Cray T3D show that:

1. View caching improves performance by up to 35% over object-consistent DSM, reducing both coherence overhead (up to 10%) and idle time (up to 25%). The latter reflects the impact of asynchronous protocols which decreases the time processors idle awaiting completion of remote coherence actions.
2. Despite the responsive low-overhead data-transfer primitives (put/get) on the T3D, the performance of object-consistent DSM lags view caching without the hardware support, suggesting that low-overhead primitives alone are inadequate to ensure sharing performance robust to unresponsive processors.

The remainder of the paper is organized as follows: §2 discusses limitations of existing DSM approaches to motivate the view caching framework described in §3. §4 presents view caching performance, and we conclude in §5.

2 Background and related work

We focus on efficiently supporting fine-grained sharing in irregular computations expressed using a concurrent object-oriented language – ICC++ [7]. The programming model supports dynamic thread creation in a shared object space (represented by method invocations against concurrent objects), facilitating the natural expression of a variety of synchronization and communication structures. The language semantics permit the use of a weaker consistency model – *object consistency*. Object consistency, a variant of *entry consistency* [1], guarantees that updates made to a shared object on one processor are visible on another at the time of method entry. The irregular application nature combined with multithreading (to support logical threads) means that object sharing is characterized by (i) uneven access distribution, and (ii) unsynchronized computation and communication activities across processors. Together, they result in arbitrary unresponsiveness to coherence requests, either because processors are busy computing or due to contention from several coherence requests.

While several DSM solutions have been proposed, few efficiently support fine-grained sharing in a dynamic context. Systems such as TreadMarks [14], Midway [1], SAM [16], and CRL [12] rely on weak consistency models for performance. However, such systems all use request-reply protocols derived from hardware cache coherence protocols, requiring prompt servicing of coherence requests (similar to dedicated hardware cache controllers). The unresponsiveness described above degrades the performance of such systems by increasing the time processors idle awaiting completion of remote coherence actions.

Some DSM systems [8, 15, 5] predict access requests and eagerly create an object copy on the requesting processor. These solutions eliminate coherence operations from the critical path and replace synchronous protocols with asynchronous ones. However, such optimizations are less effective in our dynamic context where predictive information about future object accesses is hard to obtain.

More promising are systems such as Munin [3] and Tempest [11] which allow application-specific customization of coherence protocols. Munin allows custom coherence actions for shared regions by selecting from among a set of predefined coherence protocols that reflect common application sharing scenarios. While such an approach improves performance in certain cases, Munin associates the custom protocol with the shared region, disallowing optimizations which exploit use-site specific consistency semantics. The Tempest approach is more general, permitting arbitrary user-defined coherence handlers to handle access requests. However, Tempest only provides customization mechanisms, offering little insight into how to build protocols that tolerate coherence processing delays. Further, use-site specific protocols incur additional overheads, requiring the user handler to disambiguate among use sites.

3 View caching

In this section, we first describe why existing DSM systems use synchronous protocols that couple processor activities. We then describe the use of application knowledge to construct asynchronous protocols. We finally describe the view caching framework and show examples of its use.

3.1 Synchronous DSM protocols

Existing DSM techniques utilize synchronous protocols because they adopt a *black-box* perspective of coherence. Coherence actions are determined by a single parameter, the consistency model, which reflects the lowest common denominator of application consistency requirements. However, consistency models only specify the coherence granularity, and when data needs to be made consistent, leaving it to the DSM implementation to determine *what it means for copies to be consistent* and *how coherence is actually implemented*. Given the black-box perspective, coherence has to be defined only in terms of observable *effect* of accesses on the *state* of a shared object. Consequently, most implementations conservatively define object copies to be consistent only if they reflect identical state. As for the problem of implementing coherence, most protocols

- maintain globally accessible information (the *directory*) about object state and its copies (i.e., where it resides and the access privileges of its copies), and
- query this information to satisfy an access request, invalidating or updating any conflicting copies.

Querying directory information requires a request-reply protocol, coupling activities on the requesting processor and the processor storing the information. Similarly, invalidating (updating) conflicting copies requires synchronous protocols that couple activities across the requesting processor, the processor initiating the invalidates (updates), and the processor holding the copy which needs to be invalidated (updated). With the black-box constraint, synchronization and coupling is inevitable. While this coupling does not degrade performance for hardware cache coherence protocols with responsive cache controllers, such is not the case for a software DSM system. Coupling activities on processors that are unresponsive to coherence messages degrades performance due to an increase in processor idle times.

3.2 Application-specific custom protocols

Our solution for low-overhead coherence protocols relies on application knowledge to supply information that is not provided by the consistency model, namely what it means for object copies to be consistent, and how is object coherence implemented. This approach has two motivations:

1. Different program objects often require different consistency semantics (based on application use) and exploiting these differences reduces coherence traffic.
2. Knowledge of data-access semantics can often provide information about the state of an object and its copies, whose decentralized availability eliminates the need for acquiring it at runtime and facilitates protocols that require reduced synchronization.

A natural unit of shared object use (in a concurrent object-oriented programming context) is a user-defined method. We define an object "view" to be the object state accessed by a method and consider it as the coherence unit. Application knowledge about data-access semantics as well as inferred knowledge about object copies is considered at this level. This view-specific knowledge is used to construct customized coherence protocols for object caching (hence the name, *view caching*). Two kinds of application knowledge are useful for customizing protocols:

- **Local knowledge** is information local to the method which can be automatically analyzed by a compiler. Examples include the subset of object state that the method accesses, its side-effects (e.g., accumulation of a variable, etc.), and its interaction with executions of other methods (e.g., commutativity, etc.).
- **Global knowledge** is information about global consistency requirements and concurrent program activities. Examples include knowing that threads can compute with out-of-date values of object state, information about program synchronizations, and the effect of concurrently-active threads on shared state.

The above knowledge relaxes the consistency requirements (as compared to object-consistency) by dictating the required visibility of updates, the semantics of view consistency, and allows inference of global information about the states of object copies, allowing construction of asynchronous protocols. For instance, the knowledge that a view is never modified past a program synchronization point allows one to infer the information that (i) the copy of the view at the home node is up-to-date, and (ii) all object views acquired since that time have read-only access. This information simplifies the coherence actions required to obtain a new copy of the object view.

3.3 Framework for custom protocols

Coherence protocols include protocol processing components which manipulate the *directory* (which stores location information of object copies) and the *cache* (which maps global names to local copies), as well as data transfer components. While most DSM systems bundle these components into a single generic protocol, view caching distinguishes among three components: `access-grant`, `access-revoke`, and `data-transfer`. The first consists of processing (at the cache and directory) to satisfy the requested access, including revoking access that was previously granted. `access-revoke` disables further use of the copy, and `data-transfer` transfers object state between processors. Coherence protocols are built by assembling customized implementations of these components; application knowledge about data access semantics drives this customization, focusing particularly on decreasing message traffic and required synchronization. Reducing synchronization enables use of one-sided messages that can be efficiently supported in software (see [13]), or directly in hardware using the put/get support available in several current-day parallel machines.

To keep customization manageable, our framework only permits selection from among a predefined set of component implementations by specifying values for a series of parameters. As we shall see in §4, view caching achieves good performance despite this lack of flexibility. The customizable parameters for each component and their allowed values are described in Table 1. *requester* refers to the node caching the copy while *home* refers to the node storing the directory information. `access-grant` can either be performed on the requester (if the directory state can be inferred using application knowledge), or on the home node. In both cases, based on whether or not directory update requires an acknowledgement, asynchronous protocols can be utilized. The third `access-grant` parameter controls whether or not existing copies need to be invalidated. Processor access to a cached copy may be revoked either voluntarily (e.g., with the requester determining that it has no fur-

PARAMETER	VALUES
access-grant	
where is the access decision made?	requester/home
when are existing copies invalidated?	always/read-invalidate on write/ write-invalidate on read
update of directory state?	no update/ asynchronous/synchronous
access-revoke	
who initiates the invalidation?	home/requester/other
acknowledgment for invalidation?	required/not required
data-transfer	
who initiates the object state transfer?	home/requester/other

Table 1. Parameters and values for customizing coherence protocol components.

ther need for the copy) or on demand from the home node. Both situations permit utilization of asynchronous messages for updating directory state. Finally, data transfer can be implemented either in a requester- or home-initiated fashion, and can exploit hardware support for one-sided communication. Custom view caching protocols are implemented by predefined handlers which execute on the requester and the home node. Requester-initiated `access-revoke` is implemented by maintaining, for each synchronization point, a list of views that need to be invalidated.

3.4 Examples of protocol customization

Examples of using application knowledge to construct custom protocols (see Table 2) are described below:

Example 1 Knowledge that the accessed view is currently consistent and unchanging allows the requester to infer that all potential copies only have read access. The custom protocol thus consists of requester-initiated `access-grant` and `data-transfer`, and an empty `access-revoke`.

Example 2 Knowledge that the view is currently consistent and will not be modified in the current phase allows the requester to infer that all potential copies have read access (so, no invalidation is required), and that invalidation requests will not appear until phase completion. This allows a custom protocol with requester-initiated `access-grant` that asynchronously records the requester as a reader to permit subsequent invalidation. An asynchronous protocol suffices because reader information is required only after a program synchronization. As in example 1, data-transfer can be requester-initiated and take advantage of put/get support.

Example 3 Knowledge that a thread can compute with a view consistent up to the last synchronization allows the requester to infer that the home node copy satisfies the desired consistency requirement. The custom protocol consists of requester-initiated `access-grant` and `data-transfer`, and performs `access-revoke` by having each processor invalidate cached copies at the next synchronization. Self-invalidation ensures that cached copies satisfy the desired view consistency requirement.

3.5 Summary

View caching takes advantage of application knowledge of data-access semantics to construct custom coherence protocols which require less synchronization and efficiently support fine-grained sharing in a dynamic environment. View caching differs from other DSM systems supporting protocol customization in several key aspects. Unlike Munin [3], view caching associates custom protocols with views (methods), allowing different accesses of a shared region to be subject to different coherence protocols. As we shall see in §4, treating each view as an independent coherence unit provides additional opportunities for optimization. Unlike Tempest [11] which only provides mechanisms, our approach makes the case for a specific kind of customization – reducing synchronization requirements of coherence protocols – as being important for high performance DSM in the context of dynamic computations. Finally, although our customization procedure resembles Munin's, it permits a larger variety of coherence protocol optimizations based on the high-level access semantics of user-defined datatypes.

4 View caching performance

We compare performance of view caching with an object-consistent DSM for two irregular ICC++ [7] applications — Grobner and Radiosity.[1] All experiments were performed on the Cray T3D and use the Illinois Concert System [6] which consists of an optimizing compiler and a high-performance runtime. Programmer annotations guide selection of view caching protocols.

We consider four alternative DSM implementations:

1. *Object-consistent DSM (messaging)* (**OC-M**): protocols use a messaging interface to the hardware [13].
2. *Object-consistent DSM (put/get)* (**OC-P**): protocols utilize T3D's remote memory access capabilities.
3. *View caching (messaging)* (**VC-M**).
4. *View caching (put/get)* (**VC-P**).

All implementations use identical messaging and put/get interfaces, so performance differences are entirely attributable

[1] We focus on the performance advantages of asynchronous protocols facilitated by view caching. For the applications considered, false sharing reduction due to view caching is negligible.

VIEW REQUEST	APPLICATION KNOWLEDGE	COHERENCE PROTOCOL COMPONENTS		
		access-grant	access-revoke	data-transfer
Read view	Value is consistent and unchanging	requester decides, no home involvement	none required	requester-initiated
Read view	Value is consistent, no concurrent updates	requester decides, asynchronous directory update to record reader	home performs read-invalidate on write (when phase ends)	requester-initiated
Read view	Can use value consistent up to last synchronization	requester decides, no home involvement	requester invalidates on next synchronization	requester-initiated

Table 2. Examples of using application knowledge to customize coherence protocols.

to coherence protocols. Comparing put/get and messaging implementations quantifies the incremental performance benefits of low-overhead communication primitives.

4.1 Grobner

Grobner basis [2] calculation is a fundamental tool for symbolic algebra. Our algorithm, previously implemented by Chakrabarti [4], repeatedly augments the basis by evaluating polynomial pairs (to isolate irreducible terms) formed from the existing basis members. Polynomial pair evaluation, expressed as a dynamic thread in ICC++, is priority scheduled and requires access to the basis as well as the basis polynomials (shown in Figure 1). These accesses are cached using the underlying DSM system. Pair evaluations can take widely differing execution times, leading to arbitrary unresponsiveness to coherence messages.

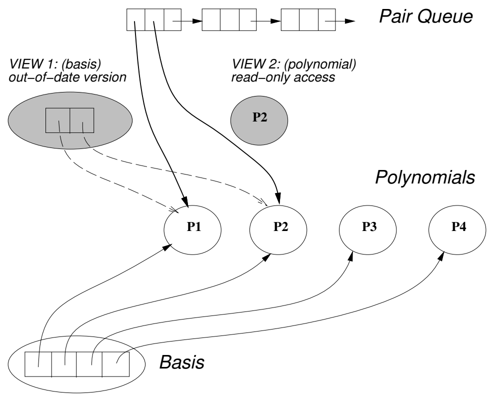

Figure 1. Pair evaluation in Grobner accesses the basis and polynomials. Two kinds of views permit low-overhead caching (shown shaded).

The view caching implementation makes use of two kinds of views (shown alongside cached objects in Figure 1). The first accesses the basis, utilizing knowledge about its monotonic growth to permit pair evaluation to proceed with out-of-date basis copies which are periodically updated to ensure that pair evaluation does not require increased work. The custom protocol completes `access-grant` on the requester, interacting with the home node to asynchronously register the requester as a reader. The asynchronous protocol decouples the requester from the home node. The home node sends asynchronous invalidations to registered readers to implement `access-revoke`, and `data-transfer` utilizes one-sided communication primitives if available. The second view accesses polynomials that are known not to change after creation. The corresponding custom protocol completes `access-grant` entirely on the requester, does not require `access-revoke`, and implements `data-transfer` using one-sided communication.

Figure 2 shows the speedup for the four DSM implementations for the `pavelle` data set. In contrast to **OC-M** which achieves a speedup of 5.75x on 16 T3D nodes, **VC-M** achieves a speedup of 7.75x which is comparable to the performance achieved by Chakrabarti [4] using explicit data replication. The execution time breakdowns (shown normalized with respect to the ideal speedup) provide an explanation for performance improvement. View caching reduces both coherence overhead (up to 10%) as well as processor idle time (up to 20%). The latter shows the extent to which asynchronous protocols reduce the time processors wait for completion of remote coherence actions. In addition, unlike object-consistent DSM which shows negligible performance improvement from put/get support, **VC-P** improves speedup to 8.25x by directly using one-sided communication primitives facilitated by asynchronous view caching protocols. In contrast, **OC-P** still requires coherence requests to be processed in software.

4.2 Radiosity

Radiosity computes the global illumination in a scene by expressing the radiosity of an elemental surface patch as a weighted linear function of radiosities of all other patches. Our algorithm, due to Hanrahan [10], computes light transport between potentially visible pairs of patches, hierarchically subdividing a patch as needed to ensure accuracy. Subdivided patches acquire their own interaction lists (of visible patches) and contribute a weighted term to their par-

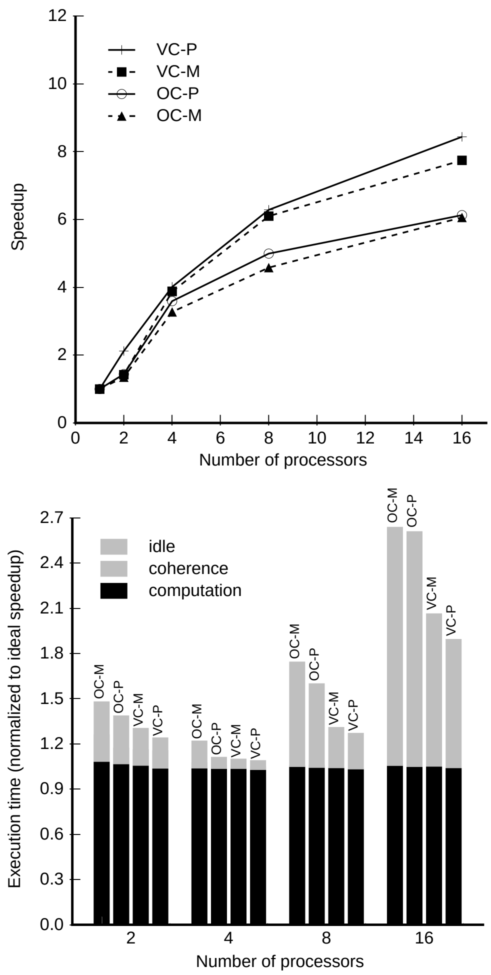

Figure 2. Grobner speedup and execution time breakdown (into computation, coherence and idle time). VC improves performance by up to 35%.

ent's radiosity. The visibility calculation, expressed as a dynamic ICC++ thread, accesses a pair of patches (which are cached by the underlying DSM system) and can require widely differing amounts of time, leading to arbitrary delays in processing coherence requests.

The view caching implementation makes use of two kinds of patch views (shown in Figure 3). The first accesses patch state which is known not to change past a program synchronization point. The custom protocol completes `access-grant` on the requester, does not require `access-revoke`, and implements `data-transfer` in a requester-initiated fashion using one-sided communication if available. The second view accesses patch state whose value is required to be consistent only up to the last iteration. This information allows a custom protocol which eliminates protocol processing at the home node. `access-grant` is completed on the requester,

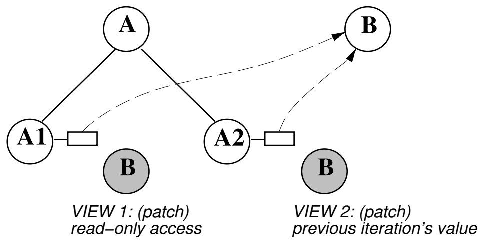

Figure 3. Visibility calculations in Radiosity accesses patch objects which are cached (shown shaded) according to two kinds of views.

and `data-transfer` can utilize one-sided communication. To ensure the desired consistency requirement, `access-revoke` is requester-initiated with each processor invalidating any cached copies on iteration completion.

Figure 4 shows speedup for the four DSM implementations on the SPLASH-2 data set. Unlike object-consistent DSM which achieves a speedup of 30x on 64 T3D nodes, **VC-M** achieves a 34x speedup which compares favorably with speedups reported elsewhere on a cache-coherent shared memory machine [17]. As with Grobner, view caching reduces both coherence overhead (up to 10%) and processor idle time (up to 25%) the latter resulting from the reduced synchronization of custom protocols. In addition, unlike **OC-P** which shows virtually no improvement even with put/get support, **VC-P** improves speedup to 39x indicating that asynchronous custom protocols facilitate better use of lower-overhead communication primitives.

5 Conclusions

We have described a software DSM framework for dynamic computations, view caching, that achieves robust performance despite unresponsive coherence request processing. Our framework utilizes application knowledge of data-access semantics to infer global information about an object and its copies, which facilitates the construction of low-overhead custom coherence protocols that require reduced synchronization. Results for two irregular applications – Grobner and Radiosity – show that view caching improves performance by up to 35% over a weak-consistent DSM implementation, reducing both coherence overhead and processor idle time. Further, this improvement cannot be matched by lower-overhead communication primitives alone, highlighting the need to redesign coherence protocols for robust sharing performance.

This work is part of an ongoing effort looking at hardware and software support for latency-tolerant coherence protocols that can efficiently support fine-grained sharing in diverse applications.

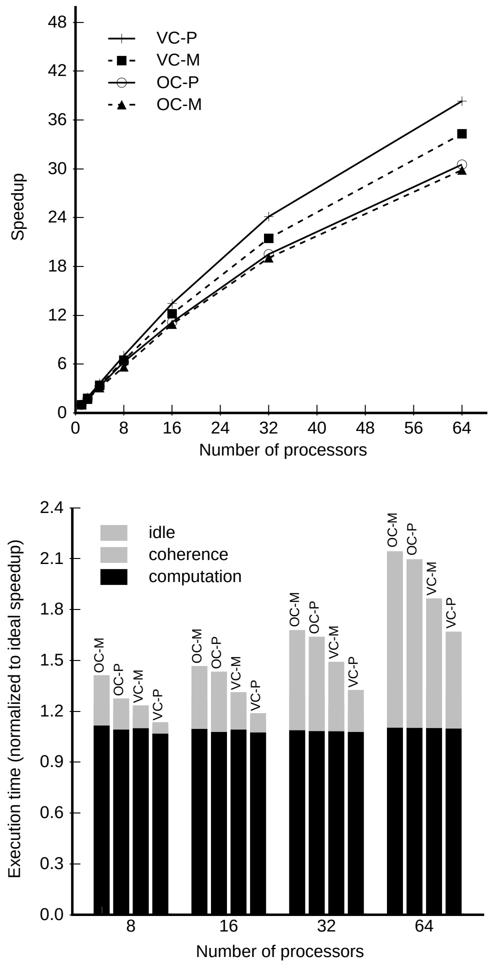

Figure 4. Radiosity speedup and execution time breakdown (into computation, coherence and idle time). VC improves performance by up to 33%.

Acknowledgements

The research described in this paper was supported in part by DARPA Order #E313 through USAF Rome Laboratory Contract F30602-96-1-0286, NSF grant MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and supercomputing resources at the Jet Propulsion Laboratory. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Vijay Karamcheti is supported in part by an IBM Computer Sciences Fellowship.

References

[1] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON*, Mar. 1993.

[2] B. Buchberger. Grobner basis: an algorithmic method in polynomial ideal theory. In *Multidimensional Systems Theory*. D. Reidel Publishing Company, 1985.

[3] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Symposium on Operating Systems Principles*, 1991.

[4] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, May 1993.

[5] H. Cheong and A. V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[6] A. Chien, V. Karamcheti, and J. Plevyak. The Concert system—compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Tech. Rep. UIUCDCS-R-93-1815, University of Illinois, June 1993.

[7] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high-performance parallel computation. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.

[8] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the International Symposium on Computer Architecture*, May 1993.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, 1996.

[10] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc Siggraph)*, 25(4):197–206, July 1991.

[11] M. D. Hill, J. R. Larus, and D. A. Wood. Tempest: A substrate for portable parallel programs. In *Proceedings of COMPCON*, Mar. 1995.

[12] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Symposium on Operating Systems Principles*, 1995.

[13] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[14] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenopol. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, Jan. 1994.

[15] T. Mowry and A. Gupta. Tolerating latency through software controlled prefetching in shared-memory multiprocessors. *J. Parallel and Distributed Computing*, 12:87–106, 1991.

[16] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First Symposium on Operating Systems Design and Implementation*, 1994.

[17] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–56, July 1994.

[18] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Object-oriented concurrent programming – modelling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.