The Rthreads Distributed Shared Memory System

Bernd Dreier Markus Zahn
University of Augsburg
Institute of Informatics
D-86135 Augsburg, Germany
{dreier,zahn}@Informatik.Uni-Augsburg.DE

Theo Ungerer
University of Karlsruhe
Institute of Computer Design and Fault Tolerance
D-76128 Karlsruhe, Germany
ungerer@Informatik.Uni-Karlsruhe.DE

Abstract

Rthreads (Remote threads) is a software distributed shared memory system that supports sharing of global variables on clusters of computers with physically distributed memory. Rthreads uses explicit function calls for access of distributed shared data. Unique aspects of Rthreads are: Synchronization primitives are syntactically and semantically closely related to the POSIX thread model (Pthreads). Distributed shared memory is built in a transparent way, because all *global variables of an Rthreads program are shared among the participating nodes and are referenced by their variable name. Moreover, Pthreads and Rthreads can be mixed within a single program. We support heterogeneous workstation clusters by implementing the Rthreads system on top of PVM, MPI and DCE. Our performance results show that the Rthreads system introduces only few overhead compared to equivalent programs in the baseline system PVM. For the evaluated examples, superior performance compared to the DSM system Adsmith and the page based DSM system CVM is achieved.*

1 Introduction

Message-passing models — in practice PVM and MPI — are most commonly used for distributed computing on clusters of workstations due to the physically distributed memory of networked computers. In contrast, the operating systems of single or multiprocessor standard workstations support a shared-memory model based on threads. The POSIX thread model (Pthreads), as established standard, is adopted by most workstation operating systems like e.g. Sun Solaris 2.5 and IBM AIX 4.1. However, POSIX threads cannot be spread over a cluster of workstations due to the mismatch of its underlying shared-memory model and the physically distributed memory within a workstation cluster.

Software distributed shared memory systems (DSM) provide the programmer with the illusion of shared memory on top of physically distributed memory. The first DSM systems extended virtual memory management to maintain coherence on page-level and followed the sequential consistency model [1]. However, applications running on such software DSM systems suffer high communication and coherence-induced overheads that limit performance. False sharing of data can be avoided by an object-based approach that shares data objects instead of memory pages.

The DSM system Rthreads is object-based; it uses primitives to read and write remote data objects and to synchronize remote accesses. The primitives are syntactically and semantically closely related to the POSIX thread model (Pthreads), enabling a precompiler to automatically transform Pthreads (source) programs into Rthreads (source) programs. The automatic generation of Rthreads programs from Pthreads programs is advantageous to other software DSM systems that solely define and implement a programming interface the programmer has to use. Pthreads (POSIX threads) and Rthreads can be mixed within a single application using the Rthreads package. Rthreads may run on different (potentially heterogeneous) machines, while Pthreads are used to exploit the parallelism among multiple processors of a single shared-memory machine.

The Rthreads system introduces a new view of the distributed shared memory: the global variables of a Pthreads program are transformed into shared variables in the Rthreads program. Different schemes must be applied for synchronizing and non synchronizing global data. The DSM scheme and its consequences for the precompiler are described in Section 2 and Section 3. The Rthreads system and the associated precompiler are presented in Section 4, related work is described in Section 5.

A further weakness of most existing DSM systems is the lack of support of heterogeneous systems. Most DSM systems are only implemented for homogeneous environments since their memory access mechanisms are based upon the virtual memory management which is deeply embedded in the operating system and processor hardware. In consequence, even portability is often restricted. Function li-

brary implementations of the Rthreads package exist on top of the message-passing systems PVM and MPI and on top of the RPC-based client-server software DCE thereby providing portability and executability on heterogeneous machines. The top-level implementation introduces a slight overhead that was measured and quanticised by running Rthreads programs versus programs written to the underlying communication system PVM. Performance results are given in Section 6 together with performance comparisons to the DSM systems Adsmith and CVM.

2 Building Distributed Shared Memory

In a traditional parallel program running on a multiprocessor with physically shared memory (e.g. a Pthreads program) all global data of the program is shared between multiple threads of execution. Most existing DSM systems require to declare or allocate all shared data explicitly. After that, shared data is placed in a buffer space that is allocated dynamically or on memory pages in the case of page-based systems. In consequence, a program running on these systems contains two types of global data: Global data shared with other participating processors (the DSM) and global data not visible to other processors.

Distributed shared memory of Rthreads is built without this separation. There is no explicit declaration of shared data and all global data is part of DSM automatically. Beyond that, Rthreads don't use additional buffers for shared data management. All operations on shared data are executed at the locations of the global variables of each node. The necessary information for data transfer and conversion in heterogeneous environments is retrieved from the source code by the Rthreads precompiler.

The resulting architecture is shown in Figure 1. The distributed shared memory consists of the memory regions, where the global variables of each node reside. The global variables themselves can be considered as (processor-)local copies of the shared data items. Hence, each node participating in an Rthreads program accesses global variables like a traditional parallel Pthreads program. However, these accesses are local and don't affect local copies of other nodes. Consistency of multiple local copies has to be ensured by explicit read/write operations in the program source code, which can also be inserted by the precompiler. These explicit operations are initiated by the application program, however, the call of the relevant procedures does not lead to any exchange of shared data values between the application program and the Rthreads software. The application program only indicates that a specific local copy of a shared variable has to be read or written. This situation is illustrated in Figure 1 by the arrows between the different nodes, which symbolize the data exchange between global variables of different nodes initiated by the application program.

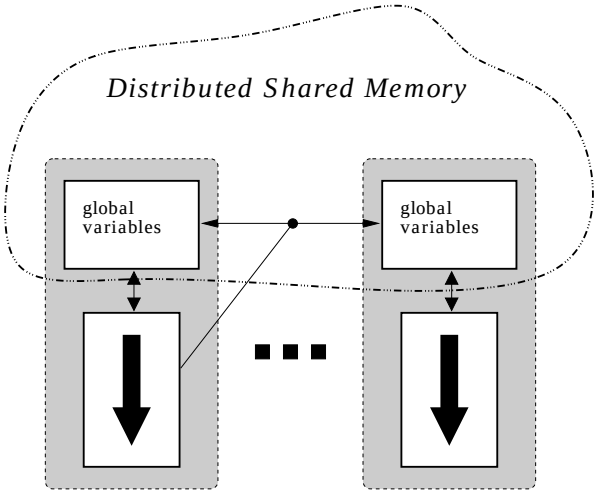

Figure 1. Distributed shared memory of Rthreads

The application program itself works on shared data through the global variables as in a traditional shared memory program.

Rthreads is a multithreaded DSM system, every participating Rthreads node program may consist of several Pthreads itself. The Pthreads of a single node are synchronized by the according primitives of Pthreads.

For the synchronization of multiple nodes of an Rthreads program we provide an Rthreads pendant (*rthread_**) to each Pthreads synchronization function (*pthread_**), the interfaces of these functions are equivalent. However, the Rthreads synchronization functions work on synchronization variables, which are part of the distributed shared memory. Because synchronization variables are accessed through these synchronization functions only, synchronizing accesses to the distributed shared memory are distinguished from ordinary accesses.

Most consistency models recommend sequential consistency [1] or processor consistency [2] for synchronizing accesses as basis for more relaxed consistency models (e.g. [3], [2], [4]). Rthreads guarantees sequential consistency for synchronization accesses. The consistency of ordinary accesses is not fixed by Rthreads. The precompiler generates sequential consistent Rthreads programs. However, the programmer may optimize the precompiler output due to any other more relaxed consistency model for better performance.

The new situation is shown in Figure 2: Every participating process contains several threads of execution. In contrast to the ordinary accesses, where the application program only indicates a DSM access to the Rthreads software, synchronizing accesses lead to immediate data exchange

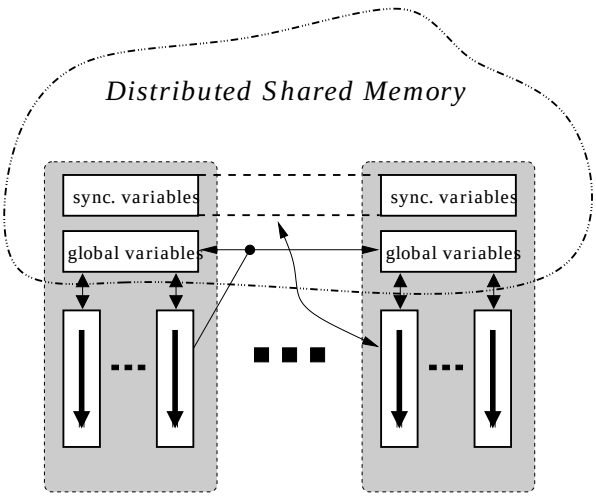

Figure 2. Multithreading and synchronization with Rthreads

and local copies of synchronization variables are always consistent during program execution. Therefore, all nodes have a uniform consistent view of synchonization data symbolized through dashed connecting line.

3 Identifying Data of the DSM

In an object-based DSM system, especially in heterogenous environments, a memory address of data items on another computer cannot be used to identify the address of the corresponding data item on the local machine. Several other object-based systems use numerical or textual identifiers, which are specified in the explicit declaration of a shared data item. Internally, Rthreads also uses numerical identifiers, which is the most efficient way for others than page-based systems. However, these identifiers are not visible to the programmer. The programmer simply specifies the same textual name as it is used in the definition of the corresponding global variable.

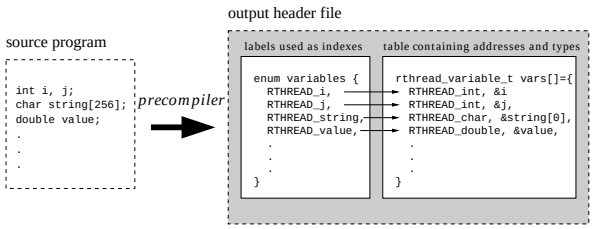

Figure 3. Identifiers generated by the precompiler

This identification of shared variables is made possible by the Rthreads precompiler, that generates a label for every defined global variable of the program and combines these labels to an enumerated type. Second, it generates an array with one entry for each global variable, the array is sorted corresponding to the enumerated type. At this point the most important information in each entry is the type and the address of the global variable on the node the program runs on. An explicit read/write function called with an identifying label as parameter is now able to use the label as an index to the array.

In the case of a write call, a value of the type specified by the array element is read from the address specified by the element and is then transfered (after data conversion) over the network to the home node of the data item together with the identifying label. On the home node, the label transfered with the incoming call can be used to retrieve the datatype and the address of the variable on this node again by use of the label as index in the precompiler generated array.

To avoid the use of the "RTHREAD_*" labels in the program source code, we provide C-preprocessor makros, which prefix the parameters of explicit memory accesses with "RTHREAD_". Now, the programmer is allowed to specify the variables by textual name as parameter for memory accesses.

4 The Rthreads System and its Precompiler

The software DSM system Rthreads (Remote threads) provides primitives for control and synchronization of remotely executed threads and specific primitives for remote access of scalar variables, distributed arrays and structures.

The Rthreads control and synchronization primitives provide an Rthreads equivalent of each Pthreads function and of each Pthreads type. The Rthreads synchronization is implemented to work between Rthreads on different machines.

Explicit remote read or write operations are used to preserve the consistency of shared data: By *rthread_r(var)* and *rthread_w(var)*, the shared variable *var* in the local buffer is marked to be read or written from or to the shared data space.

To access parts of an array, two further functions are available:

```
rthread_ra(array, first, last, stride)
rthread_wa(array, first, last, stride)
```

For efficiency reason we decided that *rthread_w()*- and *rthread_wa()*-operations are collected by the Rthreads' runtime system until a *rthread_wflush()* is executed and the aggregated data is sent in a single network transaction. Accordingly, read operations are buffered until the next *rthread_rflush()* is executed.

The Rthreads package consists of a precompiler that transforms Pthreads programs into Rthreads programs under control of the programmer and by a supporting user-level library that implements the Rthreads primitives.The Rthreads precompiler is able to process ANSI C source codes to provide the necessary information about the distributed shared memory data, including type information of basic data types and arrays of basic data types.

The programmer creates a distributed program with Rthreads by developing a local version using Pthreads first. Although an explicit Pthreads version is not necessary, it gives the opportunity to test and debug the newly created code locally.

The precompiler starts from a correct parallel program, compliant to a Pthreads model that satisfies the only restriction: exclusion of pointers for data that will be distributed. All precompiler actions take place within the source code. The precompiler performs a type analysis of the global variables and places type and naming information in a header file, as described in the previous section, and automatically generates the shared data information.

Then, each Pthreads function is replaced by its equivalent Rthreads function. Next, the already described remote read-/write marking functions and flush-functions are inserted to access the distributed shared memory. A read marking function is inserted preceeding each occurrence of a global variable as *rvalue*, and a write marking function succeeds each *lvalue* use of a global variable in the Pthreads program.

After the precompiler run, the programmer is able to optimize the automatically created Rthreads source code to improve its performance. Precompiler steps and programmer optimizations are described in more detail in [5].

5 Related Work

The first software-based DSM system for workstation clusters is Kai Li's IVY system [6] that provides virtually distributed shared memory as extension of the virtual memory management within a single machine implementing sequential consistency. Further page-based DSM systems are Mermaid [7], Mirage [8], Munin [9], CVM [10], SoftFLASH [11], and TreadMarks [12]. TreadMarks and CVM employ the lazy release consistency model [4]. Due to the architecture of page-based systems access to distributed memory is triggered by the virtual memory management.

Object-based DSM systems, like Midway [13], Shasta [14], Adsmith [15], CRL, Phosphorus [16], and Rhtreads, provide an administration of distributed data on the basis of simple or combined data types. In object-based systems remote accesses are either introduced by a modified compiler or must be set explicitly in the program.

All DSM systems known to us require the programmer to declare data of the DSM explicitly. Memory is allocated at runtime. Consequently, a sometimes complex buffer handling is introduced and the programmer has to access global variables using pointers returned by the DSM system at declaration. With Rthreads no additional declaration of global variables is required. Global variables of the source program are recognized by the precompiler, which retrieves all information necessary for Rthreads. The memory region of the global variables in each participating process is used as buffer thereby rendering complex buffer handling superfluous. Furthermore, the programmer identifies data in the DSM by the use of variable names like he is used to do.

Adsmith is similar to Rthreads regarding explicit memory accesses and the implementation on top of existing communication systems (PVM, in the case of Adsmith). However, Rthreads is more flexible in data accesses. Several concepts to vary the granularity of data sharing are devised. An example was already given in section 4 by the grouping of array elements. Further concepts for data structures and blockwise grouping of array elements are already implemented with Rthreads, but not yet evaluated. Adsmith only allows access to single data items or to *complete* arrays.

Further characteristics that differentiate Rthreads from other DSM systems are full support of heterogenous systems, multithreading by mixing Rthreads with Pthreads and portability.

	Granular.	Acc.	Consistency	Het.
IVY	P	O	SK	–
Mermaid	P	O	SK	×
Mirage	P	O	SK	–
Munin	P	O	SK, RK	–
CVM	P	O	RK	–
SoftFLASH	P	O	RK	–
Treadmarks	P	O	RK	–
Midway	O	C	EK, PK, RK	–
Shasta	O	C	RK	–
Adsmith	O	T	SK, RK	–
CRL	O	T	—	–
Phosphorus	O	T	SK, RK	×
Rthreads	O, chang.	T	—	×

Figure 4. Classification of DSM Systems

Figure 4 gives an overview over several software DSM systems with respect to the granularity (P for page-based, O for object-based), the modes of access to shared data (O for operating system-based, C for compiler-based, and T for

use of data access functions explicitly in program text), the consistency model (SK for sequential consistency, PK for processor consistency, EK for entry consistency, and RK for release consistency), and the support for heterogeneous workstations.

The "O, changing" entry with Rthreads means that data granularity is determined by data objects, i.e. all global variables. The part of a data object (e.g. an array) and consequently the size of affected data items can change from one access to another. This aspect is unique among all other systems and allows very flexible data handling during a program run.

6 Performance Evaluation

Our experimental environment consists of 15 IBM RS/6000 (Model 220W, 32 MB main memory) workstations running IBM AIX 4.1. The machines are connected by a 10-Mb/s Ethernet.

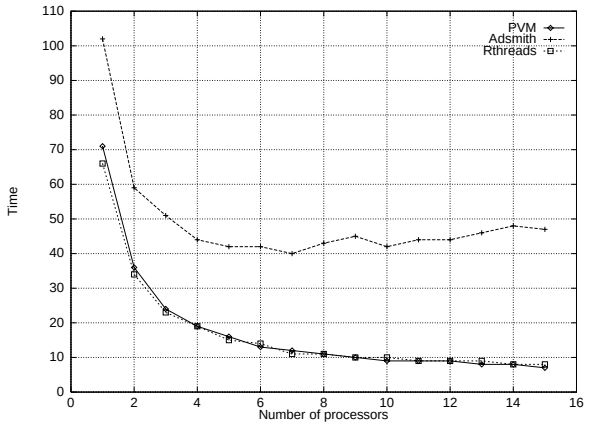

Figure 5. Performance evaluation of the distributed Mandelbrot calculation

In Figure 5, we present the performance results of a Mandelbrot calculation. We compare three different implementations of the same algorithm: One message passing version based on PVM, an Adsmith version and an Rthreads version. The performance comparisons prove that the Rthreads overhead measured versus an application programmed in the original underlying system is nearly not measurable. This is not a matter of course as the performance measurements for the related Adsmith DSM system show. Adsmith uses the same underlying communication system (PVM) and implements the shared memory also by explicit remote reads and writes that have to be set by the programmer. The Mandelbrot version based on Adsmith already uses bulk transfer, i.e. the most efficient DSM operation with Adsmith, to move newly calculated blocks to the distributed memory. We assume that performance loss of Adsmith is caused by the complex buffer handling. With Adsmith being not multithreaded itself, buffer handling had to be separated from the slave processes. There is a dæmon on each machine, which coordinates sending and receiving messages from other nodes. Consequently, each message has to be handled three times: From the slave to the dæmon, from the local dæmon to another machine's dæmon and from that dæmon to the receiving slave. Rthreads is multithreaded itself and can therefore handle incoming and outgoing messages within the slave process. Moreover, buffer handling is simple as we use the global variables as buffer.

	Adsmith	CVM	Rthreads
1st read access	1033.6 ms	140.6 ms	97.1 ms
1st write access	101.1 ms	289.7 ms	60.2 ms

Figure 6. Duration of local access to 100000 array elements of DSM data

Figure 6 shows the reasons for Adsmith's inefficiency stated above. The first read access is about ten times slower compared to CVM or Rthreads. The use of the data attributed *AdsmDataCache* allows caching of DSM data within the application process. Therefore, with no other processors involved, the afterwards measured write access is relatively cheap. The same holds for Rthreads accesses, where the native caching mechanisms lead to a cheap second (first write) access. The measurements of CVM show the overhead for the consistency protocol: The first read access is significantly slower in CVM than in Rthreads, but stays within the same order of magnitude. The costs of a first write operation are very high, which is caused by the creation of *twins* or *diffs* necessary for the lazy release consistency protocol.

As a second example, we show the results of SOR (Successive Over-Relaxation). To obtain another comparison to an existing DSM-system, we ported an algorithm included with the CVM package to Rthreads. During each iteration, every element of a two-dimensional input grid is updated by the average of four of its neighbours. For the measurements shown in Figure 7 we used the native implementation of CVM (i.e. not the MPI-based one). The figure shows that the optimized Rthreads version is noticeable faster as the CVM version for 600×400 matrices. Due to the hand optimizations possible with Rthreads data exchange can be reduced to the elements really accessed by neighbor nodes, i.e. to the very minimum necessary.

However, both CVM and Rthreads don't scale very well. We identify two reasons: The used data sets are too small and consequently communication dominates computation

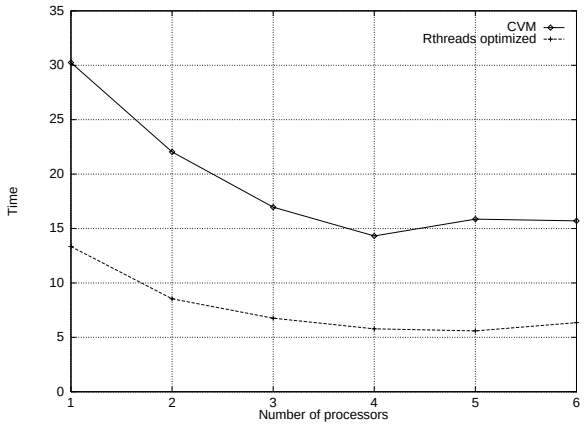

Figure 7. Performance evaluation of the distributed SOR algorithm for 600×400 matrices

due to the simple averaging update function and the slow underlying 10 Mb/s Ethernet connection of the test environment. Unfortunately, memory consumption of CVM for page management seems to be too aggressive for higher dimensional inputs. Therefore we show only the results of Rthreads computation on 2000×500 matrices in Figure 8. Scaling behavior is better than with the small data sets. We assume that further scaling is prohibited by the use of a centralized memory manager scheme in the evaluated Rthreads software. Use of global barriers in the evaluated program increases this disadvantage by concentrating communication at synchronization points. The possibility of data distribution with Rthreads is already implemented, but not yet evaluated.

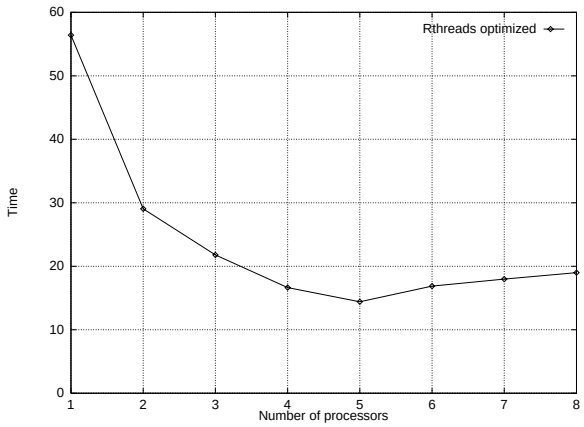

Figure 8. Performance evaluation of the distributed SOR algorithm for 2000×500 matrices

The most important results of the shown performance experiences are: Rthreads introduce only very little overhead compared to programs written in the underlying communication system. Rthreads outperforms the closely related DSM-system Adsmith. The possibility of hand optimizations with Rthreads programs even leads to superior performance compared to the page based system CVM. Finally, we expect that data distribution will further increase scalability of Rthreads and applying a lazy release consistent protocol will help to make the precompiler generated programs more efficient.

7 Conclusions

The software DSM system Rthreads implements an object-based approach to distributed shared memory with explicit accesses to shared data. Due to the close relationship to Pthreads, porting of Pthreads programs is straightforward. Besides the precompiler the Rthreads system relies upon already implemented function libraries based on PVM, MPI or DCE. We also develop an Active Message based implementation. The Rthreads system can be used on all parallel computers and even on heterogeneous workstation clusters that support one of these underlying platforms.

Performance evaluations showed that the Rthreads package does generate only a slight overhead in communication by the top-level implementation compared to a PVM system. This is not a matter of course as the performance measurements for the related Adsmith DSM systems showed.

We are working towards the implementation of an enhanced Rthreads package. The enhancements concern support for complex user-defined distributed data types and an underlying lazy release consistent implementation.

References

[1] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, pages 690–691, 1979.

[2] Kourosh Gharachorloo, Daniel Lenoski, et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer architecture news*, 18(2):15–26, June 1990.

[3] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Commitee, 1989.

[4] Pete Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, January 1995.

[5] Bernd Dreier, Markus Zahn, and Theo Ungerer. Parallel and distributed programming with pthreads and rthreads. *Workshop on High-Level Parallel Programming Models and Supportive Environments at the*

joined IPPS/SPDP International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing, Orlando, Florida, March 30 – April 3 1998.

[6] Kai Li. IVY: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, pages 94–101, 1988.

[7] S. Zhou, M. Stumm, and T. McInerney. Extending Distributed Shared Memory to Heterogeneous Environments. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, May 1990.

[8] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP-12)*, pages 211–223, December 1989.

[9] Matthew Zekauskas, Wayne Sawdon, and Brian Bershad. Software write detection for a distributed shared memory. In *First Symposium on Operating Systems Design and Implementations*, 1994.

[10] P. Keleher. *CVM: The Coherent Virtual Machine*. University of Maryland, Department of Computer Science, 1996.

[11] A. Erlichson, N. Nuckolls, G. Chesson, and J. L. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 210–220, October 1996.

[12] 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*, 29(2):18–28, February 1996.

[13] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. In *COMPCON*, pages 528–537, 1993.

[14] Daniel J. Scales, Kourosh Ghrachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. *ASPLOS VII*, pages 174–185, October 1996.

[15] Wen-Yew Liang, Chun-Ta King, and Feipei Lai. Adsmith: An efficient object-based distributed shared memory system on PVM. *Proceedings of the 1996 International Symposium on Parallel Architecture (ISPAN 96)*, pages 173–179, June 1996.

[16] I. Demeure, R. Cabrera-Dantart, and P. Meunier. Phosphorus: A Distributed Shared Memory System on Top of PVM. In *Proceedings of EUROMICRO'95*, pages 269–273, September 1995.