Parallel and Distributed Programming with Pthreads and Rthreads

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

This paper describes Rthreads (Remote threads), a software distributed shared memory system that supports sharing of global variables on clusters of computers with physically distributed memory. Other DSM systems either use virtual memory to implement coherence on networks of workstations or require programmers to adopt a special programming model. Rthreads uses primitives to read and write remote data and to synchronize remote accesses similar to the DSM systems that are based on special programming models. Unique aspects of Rthreads are: The primitives are syntactically and semantically closely related to the POSIX thread model (Pthreads). A precompiler automatically transforms Pthread (source) programs into Rthread (source) programs. After the transformation the programmer is still able to alter the Rthread code for optimizing run-time. 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. We demonstrate that programmer based optimizations may reach a significant performance increase. Our performance results show that the Rthreads system introduces few overhead compared to equivalent programs in the baseline system PVM, and a superior performance compared to the DSM systems Adsmith and CVM.

1. Introduction

Distributed computing on clusters of workstations provides a low-cost alternative to the use of multiprocessors. Message-passing models - in practice PVM and MPI - are most commonly used for distributed programming 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. However, applications running on such software DSM systems suffer high communication and coherence-induced overheads that limit performance. False sharing of data within a memory page can be reduced by a smaller grain size (cache line size or data object size instead of page size). Communication overhead can be further reduced by use of relaxed consistency models, namely the lazy release consistency that maintains coherence of data only when the data is really needed by a remote thread (or process).

Each consistency model introduces a static scheme that cannot adjust to the peculiarities of a specific algorithm. A further major improvement may only be reached by a flexible algorithm-dependent maintenance of consistency by the programmer. Most DSM systems, however, are implemented by changes of the operating system, by modifying a standard compiler, by a function library, or by a combination of some of these features. The programmer is therefore prevented from further optimizations based on his/her knowledge of the algorithm.

Rthreads uses primitives to read and write remote data and to synchronize remote accesses. The primitives are syntactically and semantically closely related to the POSIX thread model (Pthreads). An overview of the Rthread system is provided in Section 2. A precompiler automatically transforms Pthread (source) programs into Rthread (source) programs. After the transformation the programmer is still able to alter the Rthread code for optimizing run-time, in contrast to other software DSM systems based upon compiler changes or virtual memory management. The programmer is therefore able to reach an efficiency level for the distributed program that is beyond the efficiency reachable

by other DSM implementations based on the most advanced consistency models. This is demonstrated in Section 3. The automatic generation of Rthread programs from Pthread 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 Rthread 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.

A further weakness of most existing DSM systems is the lack of support of heterogeneous systems. Today, a workstation cluster very often consists of machines of different manufacturers even running different operating 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.

A reason for the success of the message-passing environments PVM and MPI is the high degree of portability and their support for heterogeneous workstation clusters. Pthreads is the standard for shared-memory systems, that provides source-code level compatibility of various shared-memory systems similar as PVM and MPI do for message-passing systems. Pthread programs cannot be spread over distributed computers - not to mention over heterogeneous machines.

Function library implementations of the Rthread 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 Rthread programs versus programs written to the underlying communication system. The performance results are given in Section 4. Related work is described in Section 5 and Section 6 draws the conclusions.

2. Overview of Rthreads

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 Rthread control and synchronization primitives provide an Rthread equivalent of each Pthread function and of each Pthread type. The Rthread synchronization is implemented to work between Rthreads on different machines and are sequentially consistent. Explicit remote read or write operations are used to preserve the consistency

of shared data. Therefore, we introduce additional functions to exchange shared data between the Rthreads: 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 RThread's runtime system until a *rthread_wflush()* is executed and the aggregated data is sent by a single network transaction. Accordingly, read operations are buffered until the next *rthread_rflush()* is executed.

The Rthread package consists of a precompiler (see next section) that transforms Pthread programs into Rthread programs under control of the programmer and by a supporting C-based function library and run-time system that implements the Rthread primitives.

3. From Pthreads to Rthreads

3.1. Developing Rthread programs

Rthreads concurrency is controlled with the same methods as known from POSIX threads. Furthermore, it is possible to combine both Rthreads and Pthreads, i.e. medium-grained parallel calculations may be processed by Pthreads and coarse-grained parallel tasks may be distributed with Rthreads. Since the POSIX thread library is an extension to the standard C library, ANSI C is the appropriate programming language for programming with Rthreads. Thus, 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 datatypes and arrays of basic data types.

The first step to create a distributed program with Rthreads is to develop a local version using Pthreads. This is demonstrated by innermost code fragment of a Mandelbrot calculation (the rectangle of pixel colors defined by the columns *min_x* to *max_x* and the rows *min_y* to *max_y* is computed as a single task using a bag-of-tasks scheme):

```
for( x = min_x; x < max_x; x++ )
  for( y = min_y; y < max_y; y++ )
    colours[x][y] =
      compute_color( x, y ) % cmax;
```

Although an explicit Pthread version is not necessary, it gives the opportunity to test and debug the newly created code locally. When designing the POSIX threads source

program, the programmer should keep the following restriction in mind to allow a smooth transformation into Rthread code: pointers within global data are not allowed in Rthread programs. Therefore, programmers should avoid global pointers in the Pthread program already. Use of pointers within the single address space formed by local Pthreads is not restricted.

As fine-grained algorithms do not perform very well in a distributed environment, the second step would be to decide which tasks to distribute with Rthreads and which ones to process by local Pthreads. By default, threads with low communication needs should be transformed to remote threads by substituting the Pthreads function calls with their Rthreads equivalents.

3.2. Identifying Shared Data by the Precompiler

In a shared-memory multiprocessor, data items are identified by their address in physical or virtual memory. Due to their architecture, page-based DSM systems are able to use similar mechanisms. In an object-based DSM system, especially in heterogenous environments, a memory address of data items on another computer can not be used to identify the address of the corresponding data item on the local machine.

The most popular solution to this problem is the modification of compilers to insert specific function calls and appropriate identifiers for every read or write access [10]. However, the portability of compiler supported systems depends on the availability of the (modified) compiler. Additionally, for each new compiler version the modifications must be integrated again.

Several object-based DSM systems with explicit read/write operations in the program source code like Rthreads use numerical or textual identifiers, which are specified in explicit declaration of the shared data item. Internally, Rthreads also uses numerical identifiers which is the most efficient way. 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. All global variables in the Pthread program become shared variables, i.e. variables that are global to the Rthread node programs.

The internal identification of shared variables is generated by the precompiler (see Figure 1). First, the precompiler creates 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
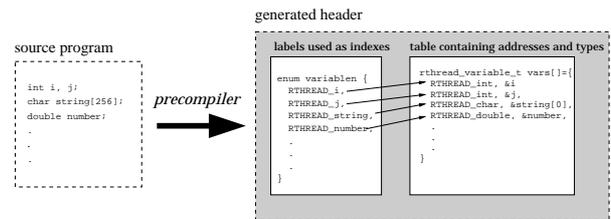

Figure 1. Identifiers generated by the precompiler

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.

3.3. Precompiler Steps from Pthread to Rthread programs

The precompiler starts from a correct parallel program compliant to a Pthread 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 *pthread_*-function is replaced by its equivalent *rthread_*-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 Pthread program. This is demonstrated for the Mandelbrot example as follows:

```
for( x = min_x; x < max_x; x++ )
  for( y = min_y; y < max_y; y++ )
  {
    colours[x][y] =
      compute_color( x, y ) % cmax;
    rthread_wa( colours, x *
      pixels_per_column + y,
      x * pixels_per_column + y, 1 );
  }
rthread_wflush( NULL );
```

The color of each pixel in the task rectangle is marked for remote write immediately after calculation. This is per-

formed by the *rthread_wa*-function that is used to access array elements. The Rthread run-time system buffers pointers to the marked array elements and initiates transfer of the whole rectangle block of pixel values in a single message when the *rthread_wflush*-function is called.

3.4. Programmer Optimizations of Rthread Programs

Even after the precompiler run, the developer of the distributed application is able to optimize the automatically created Rthread source code to improve its performance. Potential programmer optimizations concern:

1. Communication aggregation by reducing number and size of messages by combining *rthread_rflush*- or *rthread_wflush*-functions, or by eliminating consecutive *rthread_w**- and *rthread_r**-functions to the same variable.

2. Shunting *rthread_r**-functions and its corresponding *rthread_rflush* as far as allowed by the chosen consistency model to the beginning of a critical section, respectively moving *rthread_w**-functions and its corresponding *rthread_rflush* as far as allowed to the end of a critical section.

3. Using aggregation by the *rthread_ra*- and *rthread_wa*-functions.

The latter is demonstrated with the Mandelbrot example as follows:

```
for( x = min_x; x < max_x; x++ )
{
  for( y = min_y; y < max_y; y++ )
    colours[x][y] =
      compute_color( x, y ) % cmax;
  rthread_wa( colours,
    x * pixels_per_column + min_y,
    x * pixels_per_column + maxy - 1,
    1 );
}
rthread_wflush( NULL );
```

The optimized version combines the marking of remote writes within each row. Network transfer is again triggered by the *rthread_wflush*-function as in the previous sample version.

4. Performance Evaluation

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

In Figure 2, we present the performance results of the already described Mandelbrot calculation. We compare five different implementations of the algorithm: two Rthreads-based implementations, a message passing version based on PVM, an implementation in the page-based DSM system CVM [6] (Version 0.2), and an implementation in the DSM system Adsmith [9] (Version 1.8f) which is closely related to the Rthreads system.
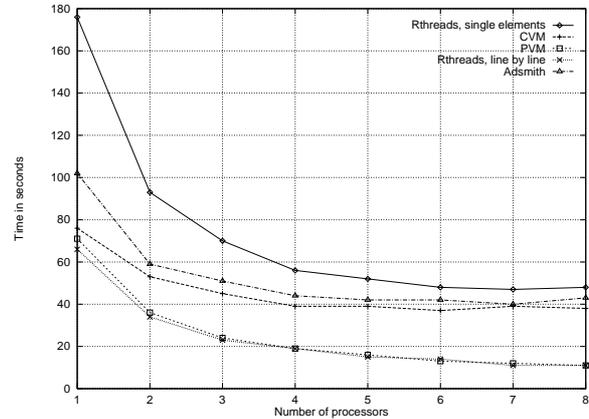

Figure 2. Performance evaluation of the distributed Mandelbrot calculation

The graphs in the figure demonstrate three different implications that were supported by further performance measurements not shown here:

The first implication concerns the two different Rthreads-versions. One of the two Rthreads-based implementations is implemented without array optimization, i.e. each color entry is marked separately to be written to the DSM, the other one uses the aggregation of marking functions as described in the previous section. The latter method makes it possible for the Rthreads system to minimize descriptive data that has to be exchanged over the communications network to allow heterogeneity. Thus, a noticeable performance gain is reached by Rthread's array operations. All benchmarks below use Rthreads array operations.

A second implication concerns the overhead that is introduced by Rthreads compared to the underlying PVM system. The performance comparisons show that the Rthread 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 systems show.

A third implication is the comparison with related software DSM systems namely CVM as representative for page-based DSM systems and Adsmith which is an object-based system and therefore more closely related to Rthreads (see also next section). 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 the most efficient DSM operation with Adsmith, to move newly calculated blocks to the distributed memory. With Adsmith, as with all object-based DSM systems known to us, access to groups of DSM data items is only possible if they were formerly allocated together within the DSM. Therefore, we had to rearrange allocation and access of global data to improve performance of the Adsmith version.

The CVM performance is in between the optimized Rthreads and the Adsmith versions.

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 distribution CVM to Rthreads. During each iteration, every element of a two-dimensional input grid is updated by the average of four of its neighbours two times (from the RED to BLACK matrix and vice versa). After each half step, processors are synchronized by Barriers. For the measurements shown in Figure 3 we used the native implementation of CVM Version 0.2 (i.e. the socket-based, not the MPI-based one). Figure 3 shows that the optimized Rthreads version is noticeable faster as the CVM and the Adsmith versions for 600×400 matrices. Due to the hand optimization possibilities with Rthreads data exchange can be reduced to the elements really accessed by neighbor nodes.
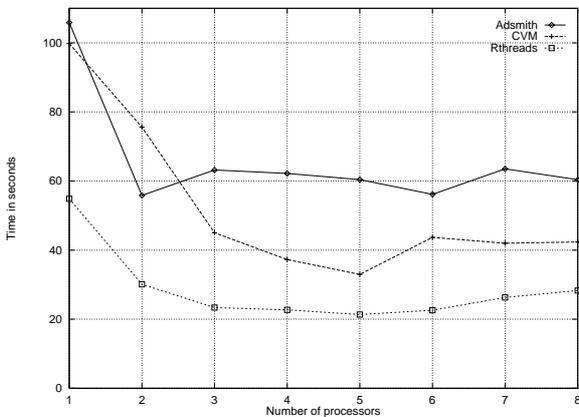

Figure 3. SOR evaluation, 600×400

We assume that the performance loss of Adsmith is caused by its complex buffer handling. Buffer handling in Adsmith must be implemented separate from the application processes, because Adsmith is not multithreaded. Adsmith dæmons on each machine coordinate the exchange of messages with other nodes. Consequently, each message is handled five times: From the application process to the Adsmith dæmon, from this dæmon to the local PVM dæmon,

from the PVM dæmon to another machines PVM dæmon, from there to the corresponding Adsmith dæmon, and from the Adsmith dæmon to the receiving application process. Rthreads is multithreaded itself and can therefore handle incoming and outgoing messages by a separate thread within the application process. Moreover, buffer handling is simple as we use global variables as buffers.

Both CVM and Rthreads don't scale well. We identify two reasons: The used data sets are too small and consequently communicating dominates computation 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.

CVM and Adsmith show noticeable performance loss compared to Rthreads with one processor already. With Adsmith, all DSM accesses have to be performed with respect to the dæmon of the local machine. This leads to process communication resulting in the obeyed performance loss of the one processor version compared to Rthreads. In the case of CVM the performance loss is caused by overhead for the consistency protocol. An unsynchronized one processor version (no consistency information has to be evaluated) reaches almost the performance of Rthreads. Rthreads itself does not suffer from any of the mentioned disadvantages.

A third example is the 3-D FFT code of the NAS benchmark suite [2]. With this application, 1-D-FFTs are applied to the three dimensional working matrix in different phases. The matrix is implemented as a one dimensional array. In each 1-D FFT, the array represents a three dimensional matrix with changed order of dimensions. The 1-D FFT is distributed to the processors along the first dimension of the matrix. Therefore, the distribution of shared data to the processors differs from phase to phase.

With Adsmith, it is not possible to rearrange the grouping of DSM data after the initial allocation. As efficient handling of communication for *all* phases is prevented and expected performance is poor, we omit an Adsmith version of this application.

The impossibility of an efficient data distribution for all phases holds for Rthreads, too. Independent of data distribution, Rthreads' flexible array operations allow suitable access of data in each phase, but access is more complicated than for the previously shown applications.

Figure 4 illustrates the results of the CVM- and Rthreads-based FFT implementations. CVM reaches the minimal execution time. However, the qualitative behaviour of Rthreads is more stable and scaling is slightly better.

Due to the complicated array accesses with changing dimensions the performance gain of Rthreads vs. CVM like in the SOR application can not be found with FFT. Another reason for this can be found in the less frequent synchro-
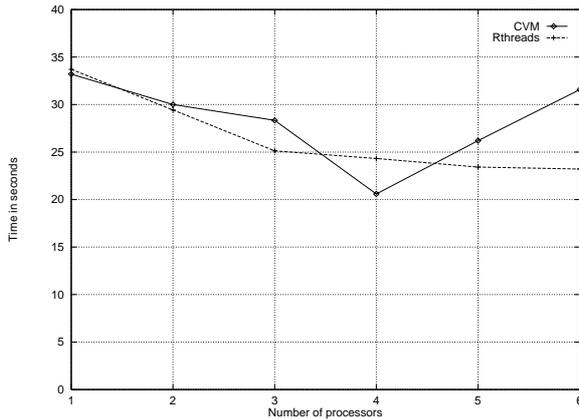
Figure 4. FFT, 16 × 64 × 64

nization in FFT compared to SOR leading to less consistency protocol overhead. This can also be noticed in the execution times of the one processor versions of CVM and Rthreads for FFT, which are almost identical.

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 Rthread programs even leads to superior or similar performance compared to the page based system CVM for all evaluated applications. 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.

5. Related work

Software-based DSM systems for workstation clusters are an active area of research starting from the late eighties. The original idea dates from Kai Li's IVY system [8] 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, Munin [10], TreadMarks [1] and CVM. TreadMarks and CVM employ the lazy release consistency model [7].

Rthreads do not fall in the class of page-based systems (i.e. DSM systems with page- or cache-line-sized coherence blocks). Granularity of data sharing is on basis of objects, i.e. variables of simple data types or variables of combined data types as e.g. arrays and structures. Other object-based DSM systems, Adsmith and Phosphorus [4], provide a similar administration of data on the basis of simple or combined data types. In contrast to Rthreads, however, in both systems an identificator (string or integer), type and, in the case of Phosphorus, node-local source and destination addresses have to be explicitly specified for each remote read or write access. The Rthread system generates these informations by the precompiler. Only the names of the concerned variables have to be specified.

In page-based DSM systems the memory accesses are triggered by the virtual memory management. In object-based systems remote accesses are either introduced by an modified compiler (e.g. Midway [3]) or must be set explicitly in the program. The latter is the case with Adsmith and Phosphorus, where the accesses cannot be introduced automatically by a precompiler.

By page-based DSM systems, distributed memory is seen as a linear array of bytes. Therefore the correct data passing in heterogeneous systems cannot be guaranteed. Mermaid is the only page-based DSM system that tackles the problem of heterogeneous systems: only data of the same type are allowed within the same page. Besides the Rthread system, Phosphorus is the only object-based DSM system that supports heterogeneous networks.

It is specific to the Rthread system that Rthreads and Pthreads can be mixed and synchronization of Pthreads within a Rthread can be done by the same Rthread function calls as for Rthread synchronization. This explicit support of node local multithreading is essential due to the increasing importance of SMP workstations interconnected by a LAN or high performance networks, e.g. used by the IBM SP series. SoftFLASH [5] is the only DSM system with similar possibilities.

Rthreads reaches a high degree of portability by its implementation on top of PVM, MPI, DCE and soon Active Messages. Other systems usually use operating system or even hardware-based communication mechanisms. Exceptions are Adsmith, Phosphorus, both based upon PVM. Besides its native implementation, CVM offers an MPI-based version, too.

6. Conclusions

The software DSM system Rthreads implements an object-based approach to distributed shared memory. POSIX thread programs – often developed for multiprocessor workstations – are automatically transfered to Rthread programs and explicit remote read and write functions are introduced.

The precompiler is implemented such that type information and header file are generated automatically. *pthread_*-to *rthread_*-function transformation and introduction of the remote read and write functions is still done by hand. Elimination of unnecessary remote reads and writes proved easy, but hand optimizing beyond the level provided by the software buffer and flush-functions of the Rthread implementation is not always possible. In principle, parallelizing

compilers for shared-memory machines (e.g. SUIF) already generate Pthread programs, thereby rendering a fully automatic generation of Rthread programs from a sequential program possible. However, we did not yet evaluate if such compiler back-ends would provide us with transformable Pthread programs.

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 Rthread package. The enhancements concern support for complex user-defined distributed data types and an underlying lazy-release consistent implementation.

References

[1] 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.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, Aug. 1991.

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

[4] 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, Sept. 1995.

[5] 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, Oct. 1996.

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

[7] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Computer Architecture News*, 20(2), May 1992.

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

[9] W.-Y. Liang, C.-T. King, and F. 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.

[10] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *First Symposium on Operating Systems Design and Implementations*, 1994.