BOPS – Balancing Objects and Pages in a Shared Space

Technische Universität München
Institut für Informatik
Germany, 80290 München
C. Rehn, M. Pizka
{rehn,pizka}@in.tum.de

Abstract

Distributed Shared Memory (DSM) systems usually employ a number of hardware pages as management units. The gap between the size of application objects and coherence units leads to the undesirable effect of false sharing, resulting in a significant performance degradation for a wide range of applications. To prevent false sharing and reduce the scope of consistency actions some systems introduce objects as the unit of sharing. The size of shared objects is deduced from application objects and may not change during program execution which has usually a negative influence on DSM performance.

We present a distributed memory management which is neither oriented to application objects nor to page size. Object granularity may vary during program execution to adapt the unit of sharing to application requirements which usually change over the time. This leads to a prefetching of the working set of activities and thereby improved performance by reducing the number of messages sent in the distributed system. Our memory management is integrated into a language-based approach to construct structured object-based distributed systems taking advantage of the implicit structural relationships between passive and active objects to further improve the performance.

1 Introduction

The relevancy of distributed computing in practice is still far behind the potentialities of nowadays available distributed computing power, provided by powerful workstations and high-speed interconnection networks. Obviously the reason is tremendous complexity. Distributed computing either burdens the programmer with additional concepts and their effects or it demands tasks from the resource management system that are hard to fulfill. Somehow contradictory to the requirement of *simplicity*, execution *performance* has to be convincing. Distributed execution must not only provide scalability but also low overall management overhead. Performance has to be reasonable compared to sequential and centralized software solutions, as well as it should provide speed-ups if additional computing resources are available.

Peak performance on distributed hardware platforms can be reached by using explicit message passing [GBD+94, BL92]. We argue, that writing parallel or even distributed programs with explicit message passing is a cumbersome and difficult task. The shared memory paradigm obviously provides an easier to use abstraction, since it moves the task of communication from the application-level to the system-level. The idea of a resource management system, that enforces the abstraction of a shared memory in a distributed environment (Distributed Shared Memory) is not very new, but since the first DSM implementation by Li [Li86] in 1986, the performance of most DSM systems is still unsatisfactory [Lu95]. Performance problems of DSM implementations mainly arise from false sharing, diff accumulation, missing hardware support to detect access violations on a fine-grained basis and communication overheads originating in additional messages in contrast to PVM [LDCZ97].

The negative effects of false sharing can be reduced by allowing multiple-writers. The number of messages sent in the distributed system can be decreased in several ways. Objects may be grouped dynamically into larger units of transportation [BS93] and the consistency semantics can be weakened. An implementation of a DSM concept has to consider mechanisms provided by the hardware. The common existing page-fault mechanism should be exploited to detect accesses to locally unavailable objects efficiently.

In this paper we present the concept and implementation of a decentralized distributed virtual memory management that provides simplicity as well as efficiency. The distributed memory management uses hardware properties to identify accesses to locally unmapped objects. Efficiency is gained by combining the advantages of page-based and object-based DSMs introducing the concept of *object clusters* to allow dynamic and flexible determination and modification of the size of shared units while still making use of the page-fault mechanism to provide an application-oriented resource management. In addition we introduce our language-based approach to construct structured object-based distributed systems. This allows the recording of well-defined structural dependencies among all objects of the system which are used by the memory management to improve performance.

The rest of the paper is organized as follows. In the next section we will briefly discuss related work. Section 3 presents our programming model and in section 4 the structural dependencies implied by this programming model are described. In Sections 5 and 6 we will elaborate on the concepts and the implementation of BOPS, before the performance of BOPS is analyzed in section 7. Section 8 summa-

rizes the main features of our approach and gives an outlook on future work.

2 Related Work

Most software realizations of Distributed Shared Memory are using conventional virtual memory management hardware and local area networks. Li's Ivy system [Li86] was the first implementation of a page-based DSM. Newer implementations are Mirage+ [FHJ94], TreadMarks [ACD+96] or Odin [Pea96]. All of these implementations have one thing in common: the size of management blocks of the DSM is bottom-up oriented, equal to hardware page sizes ignoring the needs of applications.

Other implementations try to avoid these problems. The implementation of Midway [BZS93] is an example for a DSM which is not bound to hardware pages. All store operations are done by the Midway library. The coherence protocol runs without triggering a hardware page-fault. This necessitates the execution of additional operations even if the accessed object is locally available and leads to performance disadvantages. BOPS uses objects while still exploiting the advantage of hardware support. In Midway the library has to be entered each time a write operation is necessary, in BOPS writes can be done without additional management, if the object is locally mapped. In addition, object-based DSMs burden the programmer with an object-based synchronization model. The granularity of the unit of sharing in object based systems is usually determined by application objects which may lead to performance losses if two different processes try to access different parts of an application object. BOPS allows objects of varying granularity which is not determined by application objects but by the dynamically changing application needs.

The Shadow DSM [GPR97] also tries to exploit the page fault mechanism for an object based memory at the expense of an additional indirection for memory accesses.

The performance advantage of TreadMarks compared with the region-based protocol as described in [BK98] stems from reduced message traffic because of the prefetching effect and the spatial locality of many applications motivated the introduction of object clusters in BOPS. Finding the working sets of activities, this prefetching effect can be exploited by the system management and reduce the DSM page fault rate and network traffic.

Just as in Orca [Bal94], BOPS is interwoven with a distributed object-based programming language. Compiler and runtime system are used to enable static and dynamic analysis supporting resource management.

3 Programming Model

The idea of BOPS is based on a distributed system architecture [PE97], which is featured by an object-based, top-down driven and language-based approach combined with structuring facilities to efficiently bridge the gap between application programmers and hardware. The structural dependencies between the objects are exploited by the memory management to gain efficiency. This section gives a brief overview about the main concepts of the distributed system architecture as far as they are relevant.

We distinguish between *named* objects which are known at compile time and *anonymous* objects which are dynamically created in the path of execution. Pointers to anonymous objects can be duplicated and passed between objects in the system whereas the creation of references to named objects is not supported.

Objects can be either *passive* or *active* and can be created dynamically at run time. Active objects serve for the explicit specification of parallelism on a high level of abstraction and are called *actors*. The creation of an actor establishes a new, concurrent flow of control. Actors may execute subprograms as in other procedural programming languages.

As opposed to many other object-based languages, deletion of objects is automatically handled by the runtime system rather than explicitly by the programmer. *Termination dependencies* guarantee that an object exists as long as it is accessible by other objects.

Waiting for the termination of the forked actors (join operation) is implicitly performed by the runtime system, since an activity is not allowed to be deleted prior to the termination of all its forked child-actors. Passive objects are deleted with the termination of the associated actor, method or subprogram.

We call the collection of passive objects together with the actor they depend on with respect to its existence an *actor sphere*. Each actor sphere is assigned to exactly one node in the distributed system. If an actor tries to access a passive object belonging to a different actor sphere located on a remote host it gets a copy of the page(s) the passive object is mapped to. Figure 1 illustrates termination dependencies by showing a snapshot of a program in execution.

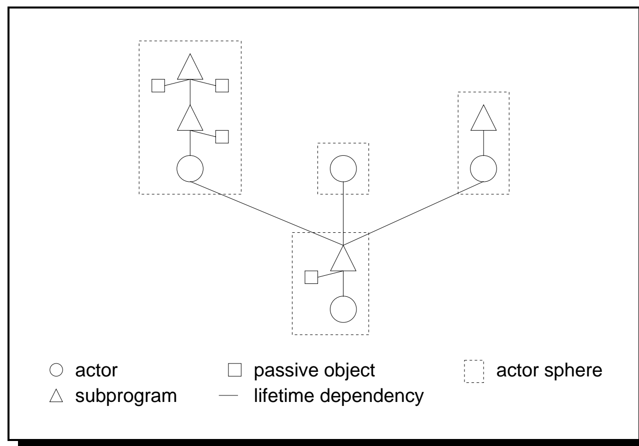

Figure 1: Termination dependencies of an example program

Each object is created as an instance of a class describing component, called *generator* and has a declaration and a possibly empty statement part. The declaration part might contain declarations of local objects, methods or nested generators. The statement part can be compared with a constructor in common objectoriented languages. An object can only be used through one of its methods.

Another important feature is the principle of *nesting* which is well known from languages such as Ada or Pascal. Nesting enables the programmer to specify wellstructured applications which is advantageous especially for big applications. Based on the nesting of classes different dependencies between objects are implicitly established.

Actors may cooperate in a client-server style by syn-

chronous method invocations or by using shared passive objects. Hence, we support message passing as well as shared memory paradigm. The invocation of an actor method is handled in the same way as a method invocation of a passive object except that the caller and callee synchronize using operation oriented rendezvous semantics. The caller is blocked and has to wait for the callee to accept the request. When the method returns both the caller and the callee continue their computation in parallel.

The operation oriented rendezvous concept is the only explicit synchronization mechanism available in our approach since other low-level mechanisms like semaphores or barriers are error prone and aggravate distributed programming. Implicit *start-synchronization* takes place between the generating actor and a newly generated actor. At start-synchronization a coherence event happens to asserts that a newly created actor has the same view on memory as its creator at the time of creation. *Stop-synchronization* is also an implicit synchronization and occurs between a terminated actor and its creator. The coherence event connected to stop-synchronization asserts that all memory modifications done by an actor are visible by its creator after termination. Between synchronization points two actors sharing a passive object cannot make any assumptions about the order of operations performed on the object. This allows the delay of write operations and update of existing copies to synchronization events and changes the semantics of update operations since they do not have immediate effect.

4 System Structures

In this section we describe the system structures which describe dependencies between active and passive objects that are implicitly determined by the programmer. These dependencies can be exploited as described in later sections to make memory management more efficient.

The nesting of generators and objects implies a relation between objects called definition dependency. An object O is *definition dependent* on object P – $\delta(O,P) \Leftrightarrow$ the generator for O is contained in the declaration part of P.

Along with the creation of a new actor B by an actor A, a new flow of control is established that executes the statement part of B in parallel to the computation of A. This relationship is recorded in the π relation. An object B *operates in parallel* to object A – $\pi(B,A) \Leftrightarrow B$ is an actor and was created by A.

To describe the communication dependencies between actors that synchronize with rendezvous semantics as described in section 3 we introduce the κ relation. An actor A *communicates* with an actor B – $\kappa(A,B) \Leftrightarrow A$ requests a service from B, B has accepted the service and both, A and B are synchronized to perform the requested service.

We call an object O local to an object P – $\lambda(O,P) \Leftrightarrow O$ is a named object and is declared in the declaration part of P.

The dependency between an anonymous object and the location of its generator is expressed by the γ relation. An object O is $\gamma-$*dependent* on object P – $\gamma(O,P) \Leftrightarrow O$ is an anonymous object and P is the location where the generator that is needed to create pointers to O is declared.

Combining the λ and γ relation we can define the termination dependency already described in section 3. An object O is called termination dependent on object P – $\epsilon(O,P) \Leftrightarrow$ $\lambda(O,P)$ and O is a named object or $\gamma(O,P)$ and O is an anonymous object.

5 BOPS Concepts and Design

To provide distributed memory management functionality for units of flexible and dynamic granularity the concept of *object clusters* is introduced. An object cluster comprises the working set of an actor. All objects accessed while executing the statement part of an actor or subprograms called by this actor must be within its object cluster at the time of access. In this paper the word *object* denotes a compound of virtual addresses. Objects are oriented to application needs and may vary over the time. Programmer defined encapsulated data structures are called *application objects*. Each path of execution i.e. each actor has its own object cluster. As shown in figure 2 virtual addresses are bound to a set of objects O by a function $vo_t : VA \rightarrow 2^O$. The function $oc_t : O \rightarrow OC$, binds objects to object clusters where OC is the set of object clusters. All these function are dependent

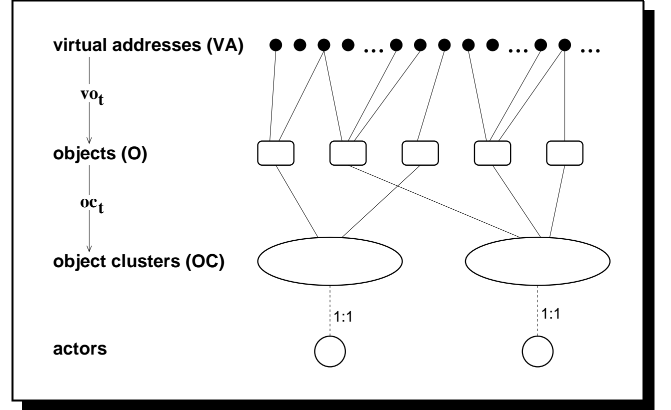

Figure 2: Building of object clusters

on time and are specific to a single actor. The clustering is achieved by the integrated management of the distributed system through static analysis done by the compiler and dynamic analysis done by the runtime system which observes memory access to enrich the information used for future management decisions. The assignment of virtual addresses to objects to object clusters may change dynamically during program execution according to changing access patterns. A virtual address can be bound to one or more objects at a given time whereas an object belongs to exactly one object cluster. An object clusters is a possibly empty set of objects.

Object clusters are introduced to reduce communication in the distributed system. On *object cluster fault*, which is triggered whenever an actor tries to access an object O not available on the local node N_i, the memory management locates the node N_j running the actor belonging to the same actor sphere as O. BOPS asserts that all objects belonging to an actor sphere are available on the node running the connected actor. This node sends not only the object that caused the object cluster fault but the values of all virtual addresses belonging to the object cluster that are available on N_j but not on N_i. In addition to reducing the number of messages the prefetching characteristics will decrease the number of object cluster faults.

To obtain a maximum degree of concurrency and reduce communication we allow multiple copies of objects with read and write access. All modifications to objects within an

object cluster are recorded and propagated on *object cluster release* events.

This raises the question, when are object clusters released. It can be answered by looking at the implicit and explicit synchronization concepts as described in section 3. The memory management has to assert the coherence events linked with start-, stop- and rendezvous-synchronization. Thus all pending modifications done by an actor must be propagated when a new actor is forked or when an actor terminates. Likewise all pending modifications done by the caller have to be propagated on rendezvous-synchronization and the modifications of the callee when the called method returns. Because transitive dependencies may exist between different callers it is not enough to propagate pending caller modifications to the callee. By the return from the synchronizing method the modifications of both, the caller and the callee must also be distributed among the other nodes in the distributed system. The propagation of modifications can be delayed for start- and stop-synchronization if the creating and created resp. terminating and joining actor run on the same node. Only if an actor is forked on a remote host the changes done by its π predecessors running on the local host in the π predecessor chain down to the the first one not running locally need to be distributed. This delay has no impact on the semantics of start-synchronization. Likewise the propagation triggered by stop-synchronization can be delayed if the joining π predecessor and the terminating actor are located on the same host.

Changing application requirements force the adaption of the functions vo_t and oc_t. To enlarge object clusters by adding objects or objects by adding virtual addresses nothing special has to be done except the adaption of the functions *vo* and *oc*. In contrast to the extension the reduction of an object cluster resp. object is more complicated. Both the functions *vo* and *oc* are changed and the memory management has to remember modified removed objects resp. virtual addresses until the next object cluster release. The termination of an actor is combined with the reduction of its object cluster to the empty set.

6 Implementation

In this section some implementation issues of the BOPS concepts are discussed. The hardware configuration chosen for the implementation consists of 14 SUN UltraSparc 1 workstations, that are interconnected via a 100MBit/s Fast Ethernet and run Linux (UltraPenguin-1.1.9 distribution). The use of Linux allows us to modify the kernel where necessary and even implement the kernel related parts of BOPS as a module.

The implementation of BOPS is based on the imperative, object-based and type-save programming language INSEL (Integration and Separation Language) [Win96] which is derived form the concepts described in section 3. An INSEL compiler called gic has been implemented by adapting the GNU C compiler gcc [Piz97]. The structural relationships between INSEL objects are automatically managed by gic generated code as part of the runtime system. For example displays, which are normally used for compiling programming languages that allow the nesting of functions, have been expanded by host-identifiers. This additional field makes it possible to find the hosts running the δ predecessors. To implement BOPS the compiler analysis and code generation is enhanced by object cluster management. In addition code to call functions executing object cluster releases and the maintenance of pending write accesses is generated.

For network communication between the hosts of the distributed system we use the TCP/IP protocol stack since it is reliable and available.

To gain efficient memory management the hardware support provided by the page fault mechanism is exploited by BOPS. The functions vo_t and oc_t as described in section 5 imply an assignment between object clusters and virtual memory pages. If a virtual address $v \in VA$ belongs to an object $o \in O$ which belongs to an object cluster $c \in OC$ the virtual memory page $p \in VP$, v belongs to, is assigned to c by the function $pc_t : VP \rightarrow OC$. This is shown in figure 3.

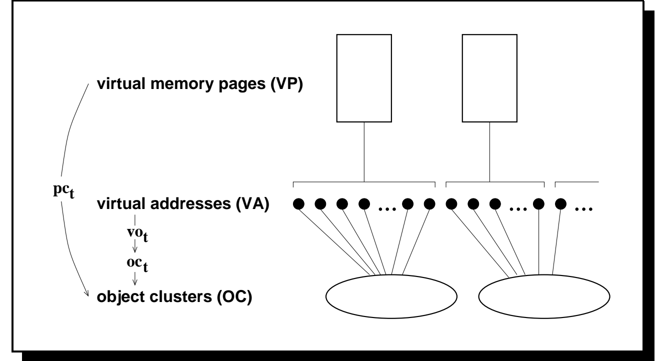

Figure 3: Object clusters and memory pages

Associated with each host is a *pending write queue* which keeps track of pages altered by local actors. The queue is needed on object cluster synchronization events. All modifications to pages listed in both, the queue and the object cluster of the releasing actor are propagated to nodes holding a copy and deleted from the pending write queue.

To find the owner of a page the concept of *home nodes* is introduced. If a host needs a copy of a page it always applies to the home node. At each point in time a virtual memory page has exactly one home node which owns that page and keeps track of nodes holding copies. To assert the uniqueness of the node owning a page some restrictions to the memory layout must be taken into account. These are described for named and anonymous objects below. The home node is not only useful to get a copy of a page but also consulted by an actor when releasing an object cluster to find hosts holding a copy.

The home node of pages containing named objects is the node running the actor this object is termination dependent on. To ensure the uniqueness of home nodes, named objects belonging to different actor spheres may not share the same memory pages. This is not a hard restriction and can easily be satisfied by assigning each actor its own amount of memory pages. All objects which are termination dependent on this actor are realized using those pages. To find the actor sphere and thus the home node of the pages a named object O belongs to, the λ and δ relations as described in section 4 are used. Assuming an actor A tries to access a named object O, O must be either a local object of A or a non-local object belonging to one of the δ predecessors of A. If O would be not local to A and not local to any of A's predecessors, O would not be in the scope of A. If O is local to A it must be present on the node running A since this

would be the home node of O's virtual memory page(s). If O is not local to A we can find the owner by following the δ relation chain to find the youngest incarnation of an actor A' with $\epsilon(O, A') \equiv \lambda(O, A')$. The node running A' must be the home node of the page(s) belonging to O.

The home node of pages containing anonymous objects can be determined using the γ relation. The uniqueness of the home node of virtual memory pages belonging to anonymous or named objects requires that anonymous and named objects are never mixed on the same pages. This is no severe restriction since anonymous objects are located on the heap and named object on the stack. Thus we only have to claim that anonymous objects which are instances of generators belonging to different actor spheres must not be located on the same pages. This can easily be enforced by the memory management since it is responsible for the allocation and mapping of memory for anonymous objects. The method to find the node running the actor an anonymous object P belongs to is analogous to the one described above for named objects. P is only usable for an actor A if the generator for P is local to an actor A' and A' is a δ predecessor of A. So A only has to follow the chain of δ relations down to the actor A' with $\gamma(P, A')$. The host running A' is the home node of P.

To obtain a maximum degree of concurrency and reduce communication we allow multiple copies of pages with read and write access. Initially shared memory pages are write-protected. When an actor tries a write access to an object located on a protected page, write-protection is removed, a local copy(*twin*) is made and an entry is added to the pending write queue. This twin is later used to create a *diff*, which describes the local modifications of that page. This is forced when an object cluster is released and the distribution of modification cannot be delayed (cf. section 5). BOPS performs a word-by-word comparison of all pages entered in both, the pending write queue and the object cluster and their twins. Using the diffs, other hosts are able to reproduce the local changes and update their pages accordingly.

The existing page-fault mechanism is used to trigger an object cluster fault when a hardware page belonging to that cluster is missing. The page-fault is handled by the kernel and not handed on to a user level signal handler. It resolves the problem by locating the home node of the page and demanding a copy of that page resp. all locally not available pages belonging to the object cluster according to the function pc_t. The home node adds entries to the copysets of the locally owned pages and sends their contents to the demanding host. In addition to reducing the number of messages the prefetching characteristics induced by object clusters will decrease the number of object cluster faults.

The multiple writer protocol has the disadvantage of needing twice the amount of memory for modified pages. If there is no more local RAM for creating a twin, the operating system may force the release of pages recorded in the pending write queue to free the space needed by copies of pages participating in these clusters.

7 Performance Analysis

This section compares BOPS to page-based and object-based systems. Since the implementation is not yet finished we are not able to present any measurements.

As described in [LDCZ97] low performance of DSM systems compared to PVM is mainly caused by additional communication costs. More messages and more data are sent because of the separation of synchronization and data transfer, extra messages to handle access misses caused by invalidations, false sharing and diff accumulation.

To reduce the number of messages and improve performance, BOPS does not only transfer a single page at a time but all available pages belonging to the working set of an actor. This enlargement of messages has hardly any negative effects on performance, because in typical networks of workstations, sending large data packages is not much more expensive then sending small ones [LH89] mainly due to the software protocols. As a side effect future page-faults are prevented. In contrast to the idea of transferring more than one adjacent pages at a time we analyze the applications access patterns and aggregate the pages accordingly.

Experiments with software DSMs releasing the consistency model and modifying coherence granularity [ZIS+97] show that two combinations generally do a good job. The sequential consistency protocol and fine granularity units of sharing or the multiple writer protocol with coarse grain coherence units perform good for most applications. Because commodity workstations offer no hardware access control on a fine-grained basis we decided to choose the multiple writer approach.

In contrast to object-based DSMs we use the page-fault mechanism and therefore have no additional overhead when locally mapped memory is accessed. Object granularity is not oriented on application objects but on current application needs and adapts continuously over the time. For example, if the application object is a matrix A and every thread of a distributed algorithm works on a single line, each $A_{i\bullet}$ can be made up in one object cluster c_i.

If we modify our approach to have exactly one page bound to an object cluster for each point in time, BOPS would correspond to a page-based DSM resulting in a loss of the prefetching characteristic usually achieved by the clustering.

BOPS exploits structural dependencies to find the owner of a memory page very efficiently in contrast to alternative approaches. For example doing a broadcast interrupts each processor, using a centralized-server has the effect of serializing queries, reducing parallelism and being a single point of failure. The probable owner algorithm may send $n - 1$ messages in the worst case if there are altogether n nodes in the distributed system.

The programming model proposed in section 5 allows to defer a huge amount of object cluster releases to further reduce the amount of messages.

8 Conclusion

In this paper we presented the distributed memory management BOPS based on a distributed system architecture featured by an object-based, top-down driven and language-based approach combined with structuring facilities. The desired efficiency is attained with BOPS by dynamic and alterable determination of memory management units according to the working set of activities. Although BOPS manages clusters of any size, efficiency is reached by exploiting the page-based faulting mechanism provided by the hardware instead of choosing an all in software implementation. Implicit structural dependencies resulting from our language-based approach are exploited for an efficient localization of page owners.

According to [ZIS+97] the multiple writer protocol does not work well when synchronization frequency of applications is high. Thus we intend to enlarge the flexibility pro-

vided by BOPS. In addition to the dynamic and flexible determination of object clusters we will enable dynamic choosing of coherence protocols conform with application needs. Beside the multiple writer, a single writer protocol will be implemented and the compiler will be enhance to decide between an invalidate or update strategy.

In a second project we are investigating distributed load balancing techniques. The impacts of load management on memory management and vice versa will influence further development of BOPS.

References

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[Bal94] Henri E. Bal. Report on the programming language Orca. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1994.

[BK98] B. Buck and P. Keleher. Locality and performance of page- and object-based DSMs. In *Proc. of the First Merged Symp. IPPS/SPDP 1998)*, pages 687–693, March 1998.

[BL92] Ralph M. Butler and Ewing L. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. Technical report, Mathematics and Computer Science division, Argonne National Laboratory, Argonne, Illinnois, 1992.

[BS93] W. J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. *Proc., Fourth Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE CompCon Conference*, 1993.

[FHJ94] B. D. Fleisch, R. L. Hyde, and N. C. Juul. MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. *Software—Practice and Experience*, 24(10):887–909, October 1994.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical report, Engineering Physics and Mathematics Division, Oak Ridge Laboratory, Oak Ridge, Tennessee, September 1994.

[GPR97] S. Groh, M. Pizka, and J. Rudolph. Shadowstacks – a hardware-supported dsm for objects of any granularity. In A. Goscinski, M. Hobbs, and W. Zhou, editors, *1997 3rd International Conference on Algorithms And Architectures for Parallel Processing (ICA3PP'97)*, pages 225–238, dec 97.

[LDCZ97] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between pvm and treadmarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Dissertation, Department of Computer Science, Yale University, New Haven, CT, October 1986.

[Lu95] H. Lu. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. Master's thesis, Department of Computer Science, Rice University, May 1995.

[PE97] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, January 1997. IEEE CS Press.

[Pea96] A. N. Pears. Odin: Implications and Performance of a Novel DSM Design. In *11th Int'l Conf. on Systems Engineering (ICSE'96)*, January 1996.

[Piz97] Markus Pizka. Design and implementation of the gnu insel-compiler gic. Technical Report TUM-I9713, Institut für Informatik Technische Universität München, April 1997.

[Win96] H.-M. Windisch. The Distributed Programming Language INSEL - Concepts and Implementation. In *High-Level Programming Models and Supportive Environments HIPS'96*, 1996.

[ZIS+97] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Singh, Kal Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 193–205, New York, June 18–21 1997. ACM Press.