Proc. 4th Workshop on Future Trends of Distributed Computing Systems, Lisbon, Portugal, September 1993, pp. 470-476

The PANDA System Architecture — A Pico-Kernel Approach

H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, R. Schwarz

University of Kaiserslautern, Department of Computer Science
Erwin-Schrödinger-Straße, D - 67663 Kaiserslautern, Germany

Abstract

The goal of PANDA is to provide an environment for parallel and distributed programming in C++. The system consists of a small operating system kernel, and a runtime package located in user space. In this paper, the concepts of PANDA are outlined with focus on the system architecture.

1 Introduction

Many research activities on new operating system architectures concentrate on using the object-oriented paradigm. Such systems focus on an object-oriented programming language and class libraries rather than on system call interfaces. An increasing number of operating system mechanisms is provided in non-privileged user space; they are not distinguishable from operations provided by the user. Thus, the system functionality can easily be tailored according to specific application requirements. The use of the system interface is facilitated by language dependency, since there is no impedance mismatch between an application and the operating system. In particular, compiler support is not lost at that interface.

For the development of operating systems, the major challenge imposed by the current hardware trends concerns handling parallelism and distribution. Advanced applications require a platform allowing an efficient use of multiprocessors connected by a high-speed network. Furthermore, support for persistent language-level objects is an issue of particular interest. Ideally, concepts for dealing with parallelism, distribution, and persistence should be integrated into a programming language. However, languages in general use today lack the desired functionality. A pragmatic approach is to support distributed programming within the context of these languages [Chase et al. 89, Shrivastava et al. 91].

The goal of PANDA is to provide an environment for parallel and distributed programming in C++, imposing as few as possible restrictions on the use of the language. PANDA's basic library consists of classes which address the issues of parallelism, distribution, and object persistence. These classes can be used directly for building an application, or, alternatively, as a basis for more specialized programming systems, since the user is free to define new abstractions by customizing the PANDA classes.

The PANDA architecture consists of a small operating system kernel, and a runtime package located in user space. The kernel provides only those functions that are relevant with respect to protection and monopolization. All other services are realized in non-privileged mode. Our operating system kernel is called *pico-kernel* to emphasize its reduced functionality.

The paper is structured as follows. In Section 2, PANDA's application interface is described. The design rationales underlying our system architecture are discussed in the subsequent section. Finally, the pico-kernel realization is outlined.

2 Application level support

PANDA aims at providing the programmer with an object-oriented programming interface based on a C++ class hierarchy. The system supports standard C++; no extensions to the original language are required. This decision was influenced by our goal to remain compatible to existing standards. PANDA has been carefully designed to preserve the high efficiency that C++ offers as an implementation language. Its interface centers around the following abstractions:

- *threads* as units for parallel activity,
- a *global virtual address space* for shared objects,
- *object* and *thread mobility* to deal with distribution,
- *persistence* of objects for permanent storage.

An important objective of the PANDA project has been to develop a unifying framework which realizes all the above abstractions in a consistent manner [Assenmacher et al. 93].

Threads

In an ordinary C++ program there exists only one thread of control. To introduce parallelism into C++, PANDA provides a base class `USER_Thread` containing a virtual member function `code()` which defines the thread's behavior. The programmer has to supply this function. Creating a new object of base class `USER_Thread` will implicitly start a parallel thread of execution performing the object's `code()` method.

Conceptually, threads and objects are complementary to each other. One may think of a thread as an entity that visits objects, enters them by method invocation, and leaves them on return from the call.

PANDA's object and thread abstractions are both mapped on C++ classes. In particular, thread properties are acquired by inheritance from class `USER_Thread`. As an alternative, one might consider to support asynchronous method invocations, as has been done, for example, in the PRESTO system [Bershad et al. 88]. However, PRESTO's approach of passing a method pointer together with appropriate parameters to a specialized asynchronous call operation violates the static type safety of C++, at least in the absence of additional compiler support.

The crucial issue in providing multithreading is performance. The cost of parallelism determines its grain and has thus a strong influence on the programming style. Therefore, PANDA threads have been realized in user space. The underlying operating system kernel is completely unaware of user space parallelism. In particular, thread scheduling is done in non-privileged mode, yielding so-called *user-level threads* [Bershad et al. 88, Marsh et al. 91]. Our implementation offers very fast thread management. Creation and deletion of threads is highly efficient, causing an overhead within only one order of magnitude compared to a procedure call, see Table 1 at the end of this section.

As several threads may concurrently enter the same object, mechanisms for synchronization are required. It is the programmers responsibility to ensure proper parallel object access. PANDA offers synchronization objects such as locks, distributed read/write semaphores, and signals. Being subject to its target language C++ and a conventional compiler, PANDA can hardly support more sophisticated synchronization paradigms that would require language extensions [Gehani and Roome 88, Nierstrasz and Papathomas 90, Saleh and Gautron 91].

Global virtual address space

Typical C++ applications follow the spirit of the C programming language in that they rely heavily on pointer references. This yields a rather "impure" object model due to the lack of encapsulation, but offers maximum efficiency. Restricting the use of pointers would be a major deviation from what is regarded as a C++ standard.

In order to retain the expressiveness of C++, the PANDA system has been based on the abstraction of a global virtual address space spanning a distributed system of homogeneous processors. As a consequence, every pointer throughout an application denotes a uniquely defined memory location on some node participating in the computation. Therefore, pointers may be passed freely across node boundaries as parameters; they have the same "meaning" everywhere in the system. That does, of course, not mean that arbitrary memory addresses can actually be accessed by a local thread. Instead, PANDA statically divides the global virtual address space in several partitions. Whether an object is currently accessible depends on the memory partition in which it resides. On the one hand, there are local partitions whose memory locations can only be addressed by local threads. All local partitions comprise disjoint address spaces, and any attempt to access an address belonging to a remote partition will be flagged out by hardware as an access violation. On the other hand, PANDA offers shared partitions which denote identical address ranges mapped on each node. Whenever a shared partition is accessed, the runtime package ensures that an up-to-date memory image will be presented to the application (see object mobility below). At the language level, there is no difference between an access to a local object, and an object which is remote but resides in a shared partition. Therefore, ordinary, sequential application code may be executed in a distributed environment essentially without any modification. This location transparency is highly desirable with respect to the portability of existing programs, and also for debugging purposes.

When creating a new object in heap memory, the programmer can choose an appropriate memory partition according to his needs by simply specifying a partition name as an additional parameter to the `new` operator. If no such specification is provided, then the object is created in a local partition by default.

Note that a static partitioning of virtual addresses is only feasible if the address space is sufficiently large. With the advent 64-bit architectures which is the emerging technology, populating the address space sparsely in order to obtain a simpler and more efficient realization of the memory management mechanisms seems reasonable [Chase et al. 92, Chase et al. 92b, Garrett et al. 92].

Object and thread mobility

Recall that one of our main goals was to support programming in a distributed environment. As mentioned above, in C++ the most natural solution to the distribution problem is to create the illusion of a single, global object and thread space, i.e., to hide the node boundaries from the application programmer.

A popular approach to location transparency is to redirect object accesses via communication stubs [Birrell and Nelson 84], or proxy objects [Shapiro 86] to the location where the physical representation of the object resides. However, such “software routing” requires one additional level of indirection. This entails overhead, even when a global object resides locally. Besides efficiency considerations, such an explicit communication interface requires extensive compiler support to make it user-friendly, and even then it generally restricts location transparency to method invocations. In C++ where encapsulation is provided only to a limited degree, following a pointer reference can potentially cause a remote object access. Therefore, PANDA employs object and thread mobility based on hardware surveillance to separate location transparency issues from the method invocation mechanism.

Remote object accesses are handled as follows. If a reference to a remote object is encountered by the memory management hardware, a page fault interrupt occurs. Next, the runtime package takes a decision to either migrate the thread to the location where the object currently resides, or to attract the object (or a read copy thereof) to the local node. The first alternative is only allowed if the respective thread is willing to migrate (there might be good reasons for a thread to refuse to migrate because it depends on dedicated local hardware devices). The second alternative may only be chosen if the corresponding object resides in a shared memory partition. If neither of these conditions is met, the page fault is treated as a semantic error, and an access violation exception is raised.

Our approach to object mobility is generally known as *shared virtual memory* or *distributed shared memory* (DSM) [Nitzberg and Lo 91]. As it is based on hardware support by a memory management unit, it completely avoids overhead as long as objects are locally accessible. This should be by far the most common case in a well-structured distributed application. It has, however, the disadvantage that DSM is provided on a per-page basis, disregarding any object boundaries. Thus, it may suffer from false sharing if objects are allocated too carelessly. Nevertheless, we strongly believe that it is important not to penalize fine-grained local accesses in a language like C++. Measurements which confirm our claim have also been reported from other environments, see, for example, the figures presented in [Levelt et al. 92].

An alternative to object mobility is thread migration — i.e., function shipping instead of data shipping. Actually, both object and thread mobility have proved their value in practical implementations [Bennett et al. 90, Chase et al. 92b, Dasgupta et al. 91, Garrett et al. 92, Jul et al. 88]. A programmer *may* explicitly migrate a thread to a target host, but typically migration is triggered by page faults as described above. Note that in PANDA thread migration is not subject to false sharing. Furthermore, threads are copied in a “lazy” fashion, i.e., only the relevant parts of the execution stack are transferred by default. Additional local data of the thread is only migrated on demand, i.e., if it is actually referenced during the execution.

By managing threads at user level, and by applying data compression techniques to both thread and object transmission, PANDA is able to support mobility very efficiently. Some preliminary performance measurements are listed in Table 1. The measurements were carried out on Sun SPARC-2 workstations connected by a 10 Mb/s Ethernet on top of SunOS 4.1.1. Although we expect to substantially improve our DSM access times in the future, the measurements show that thread migration is still a reasonable alternative to object migration. Whether it is better to transfer the thread or the object (provided that both is feasible) is a matter of a particular migration policy, and the decision is taken by the PANDA page-fault handler at user level.

Benchmark operation	**Time**
Thread creation, null execution, deletion (with / without lazy stack allocation)	15.1 µs / 10.7 µs
Context switch	17.7 µs
Local memory allocation and deletion	1.5 µs
Remote distributed shared memory access (page at origin, page not at origin)	5.3 ms / 9.6 ms
Thread migration and reactivation	8.9 ms
Null procedure call (with / without register window overflow)	7.1 µs / 0.31 µs

Tab. 1. Performance of the PANDA prototype

Our solution is very attractive from a programmer’s point of view. In principle, a program could be completely unaware of node boundaries. In most application domains, this view is, of course, too optimistic — efficiency considerations typically preclude a thoughtless distribution and require a careful decomposition of the system into (relatively independent) components. Nevertheless, location transparency is at least a first step in the direction of this ideal, and it helps to reduce the complexity of the application code.

Persistence

In conventional environments, persistence is typically realized with the help of file systems or database interfaces. Unfortunately, traditional file or database systems suffer from impedance mismatch. Object-oriented database systems are currently being developed which address this issue [Lamb et al. 91, ODeux et al. 91]. The integration of persistence mechanisms into the run-time environment of the language seems to be a reasonable alternative. In particular, such an approach has been pursued in systems designed for fault tolerant computing [Ahamad et al. 90, Liskov 88, Shrivastava et al. 91].

In PANDA, a persistence mechanism has been realized. In essence, PANDA provides a dedicated, shared memory partition which is kept persistent, even in the presence of failures. The content of the persistent partition is mirrored on a stable storage device. Persistent objects are created within that partition and they are globally accessible. From the programmers perspective, persistent objects are accessed in the same way as ordinary, transient objects. The instrumentation required to support object persistence is added by simple preprocessor macros. To indicate persistence, the `new` operator has to be invoked with an appropriate parameter. Symbolic names can be assigned to persistent objects, and PANDA provides operations to retrieve objects by their names.

One major problem concerning persistent objects is how to maintain their consistency in the case of concurrent access or node failure. PANDA offers distributed transaction support, restricted to persistent objects. Currently, nested transactions are not supported; nested invocations are, of course, feasible. If a failure occurs, always the outermost transaction bracket is aborted. Only persistent objects are restored.

3 System architecture

The PANDA architecture consists of an operating system kernel and a runtime package (RTP). The kernel comprises a fundamental set of privileged functions, described in more detail in the subsequent section. Two basic notions — *virtual processor* and *protection domain* — are provided to abstract from the available processors and from memory. On top of the kernel, the RTP implements all those functionality that may run in non-privileged mode. Application software is based on the RTP interface. Both application and RTP execute in user mode (Figure 1).

The RTP is realized as a hierarchy of C++ classes. They are divided into several class families reflecting the main entity types which constitute our distributed system model. For example, there are hierarchies for persistent or garbage-collectable objects. Another hierarchy — the thread family — models the different kinds of system and user activities such as migratable and immobile threads. Yet another inheritance tree comprises synchronization classes for concurrency control. The strength of PANDA's object-oriented approach is that a well-chosen collection of these classes is made available at the application level which may be tailored to the needs of the user by means of derivation.

There is a one-to-one mapping between virtual processors and runtime packages. A distributed application spans several RTPs. Since only one virtual processor executes within each RTP, the user-level thread scheduler is simplified. Moreover, a substantial performance gain is achieved because there is no need to synchronize scheduling activities [Anderson et al 89]. On the other hand, if an application wants to exploit the processing power of a multiprocessor platform, several RTPs sharing the same protection domain must be employed on each node, one for each physical processor. Since each RTP operates on its private system objects, frequent cache invalidation caused by system activities such as thread scheduling is avoided. Maintaining only a loose coupling between the processing units of a multiprocessor is a key to exploit its full computational power [Misra 91]. In order to achieve multiprocessor scheduling, threads have to be migrated, either explicitly or by a load balancer. If an application decides to run on top of a single RTP, true parallelism is not exploited, even on multiprocessor machines.

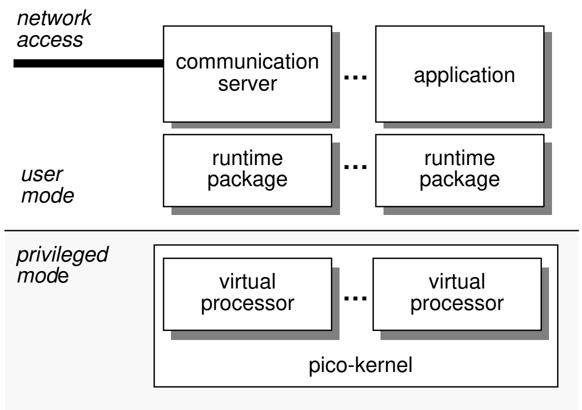

Fig. 1. The PANDA system architecture

Performance can be increased by scheduling threads in a non-preemptive fashion. By interrupting activities at well-defined points during execution, only a small fraction of the thread context need to be saved. In comparison, preemptive scheduling implies storing the complete thread context and hence slows down context switches. An even more serious problem occurs if a thread holding a lock which is central to the computation is preempted. In this case, all succeeding threads are likely to be blocked immediately after their continuation — the computation is starved by the preempted lock holder. This is a common scenario in applications consisting of closely cooperating threads. In PANDA, a preemptive as well as a non-preemptive user-level scheduler may be supplied. We prefer the non-preemptive strategy because of its efficiency. Furthermore, experience has shown that in our applications threads are rather fine-grained, and would be preempted at a rate of less than one percent.

Most PANDA services are realized at user level. Services handling unique resources shared by several applications must not be included into any of the user RTPs. They have to operate in the context of a private runtime package. Consequently, calling such services requires interaction between local RTPs which entails kernel intervention. In particular, this applies to the remote communication handler which is responsible for network access. Therefore, sending or receiving messages requires one additional context switch at kernel level compared to a highly optimized communication service integrated into the kernel. On the other hand, our approach has several advantages. By separating services from the kernel, better modularity is achieved, and thus extending and maintaining the system is facilitated. Moreover, service implementation can profit from PANDA functionality. Currently, the implications of a user-level service realization are being studied, and we are trying to devise means to further reduce the potential performance penalties of the current scheme.

4 pico-Kernel architecture

PANDA employs a small operating system kernel. One of our main design goals was to reduce the frequency of kernel calls. As a consequence, the kernel interface offers only those abstractions which are critical in respect to protection and monopolization considerations. All other functionality usually found in operating system kernels has been moved into user space. By minimizing the amount of kernel intervention, the overhead induced by the need to cross the trap boundary can be kept very low. The kernel interface is “skinny” and intended to be used only by system programmers. We chose the name *pico-kernel* to stress the characteristics of our approach.

The hardware abstractions provided by the kernel are *protection domain* and *virtual processor*. A protection domain offers functionality to control access to address spaces on a per-page basis. A virtual processor is an abstraction of the processor and has the functionality to deal with interrupts, exceptions, and to communicate with other virtual processors on the same node. Every virtual processor is assigned a protection domain, but a single domain may be shared by several processors.

Kernel interface

The kernel interacts with the runtime package in three different ways. First, the functions of the kernel interface are accessed via traps from user space into privileged mode. Second, the kernel informs the runtime package about exceptions and certain interrupts using an upcall mechanism (software interrupt). Third, read-only access from user space to dedicated kernel data allows efficient information flow without crossing the trap boundary.

All kernel calls are non-blocking. No thread will ever block within the kernel, thus stopping the whole runtime package. Alternative approaches to prevent user-level threads from blocking the thread package may be found in [Anderson et al. 92, Draves et al. 91, Marsh et al. 91].

Interrupt handling

The kernel provides an interrupt forwarding facility. All interrupts are passed to user space. Interrupts are classified into *immediate* and *delayed* interrupts. These notions indicate whether an interrupt should be handled instantly after its occurrence, or whether it should be recorded and processed later. Synchronous program exceptions — e.g. page fault or division by zero — are always treated as immediate interrupts, while asynchronous events — such as timer or I/O interrupt — may be labelled either as immediate or delayed.

Handling of an immediate interrupt is depicted in Figure 2. First, the kernel saves the execution context of the virtual processor in a dump area in kernel memory (1). Next, an upcall to the runtime package is issued, transferring interrupt type and a pointer to the saved context region as parameters (2). Interrupt handling — e.g. acquiring pages after page faults — is done in user space (3). Finally, the context which is identified by the dump area pointer may be restored by a kernel call, if the runtime package wants to resume the interrupted thread (4).

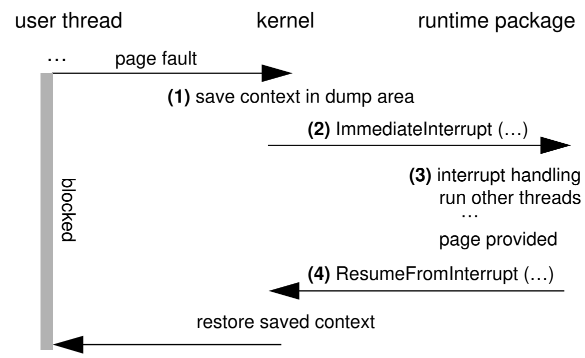

Fig. 2. Immediate interrupt handling

Delayed interrupts are delivered via shared data structures. The runtime package has read-only access to a table maintained by the kernel. This table contains counters which are incremented whenever a delayed interrupt occurs. By comparing these counters with private copies, a runtime package may identify new interrupts. This method is very efficient because it avoids context switching and kernel traps.

Migration support

As described above, PANDA is capable of migrating threads between runtime packages that may reside on different nodes of a network. Since the kernel has no knowledge of threads, this must be done completely under control of the user-level thread system. Provided a thread issues a synchronous migration request, no kernel intervention is needed. If, however, a thread that is interrupted — e.g. due to a page fault — has to be migrated asynchronously to a different runtime package, kernel support is required. The execution context controlled by the kernel has to be made accessible at the user level for transmission. To this end, two kernel calls are available: `SaveUserContext()` extracts the relevant data from the dump area in a user-readable data structure; `RestoreUserContext()` generates a new dump area on the basis of a given context data and returns a pointer to this region. To continue an interrupted thread after migration, the normal handling of immediate interrupts applies.

Local communication

The pico-kernel offers communication primitives to enable interaction between different virtual processors. Kernel-level message passing is only possible between virtual processors on the same node (note that remote communication is handled by a communication server). Virtual processors are directly addressed. Incoming messages generate a logical interrupt (`MessageInterrupt`). Two arbitrary

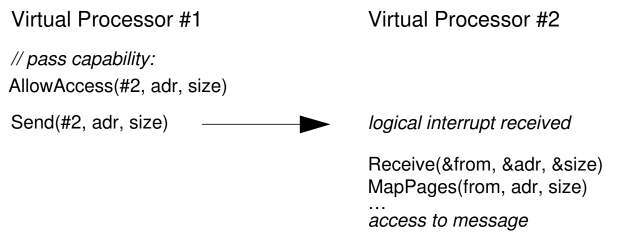

Fig. 3. Virtual processor communication

parameters can be passed; they typically denote address and size of a memory region. Virtual processors running in the same protection domain share all data so that message exchange is straightforward. Virtual processors in different protection domains can share data based on capabilities (Figure 3). Virtual processors may grant or revoke capabilities for memory access. Capabilities are used on a per-page basis, allowing efficient data sharing with support of the memory management unit. Therefore, passing data in a truly secure fashion requires page alignment.

5 Conclusions

The PANDA design was driven by our intention to provide as much as possible system functionality in user space. Such an architecture has definite advantages with respect to flexibility and performance. The pico-kernel approach requires, however, somewhat more effort to ensure timely and fair access to critical devices compared to architectures where device drivers such as communication links are an integral part of the kernel. Evaluating the trade-offs of our design, and optimizing the interplay between external device drivers and the kernel is an area of our particular interest.

The class hierarchy of the runtime package has proved its usefulness and flexibility in a number of system extensions. PANDA has the capability to serve as a base for various concurrent and distributed programming models. For example, we implemented a runtime support layer for COIN [Buhler 90], a programming language especially designed for parallel and distributed applications. As expected, no major performance penalties had to be payed due to interface mismatch.

There is an increasing demand for environments supporting parallelism and distribution compliant with standard C++. By meeting this requirement, PANDA attracted the interest of an industrial partner who is employing it as a platform for telecommunication software. Thus, we are able to obtain valuable feedback regarding our design choices.

In distributed systems, partial malfunction and temporary disconnection have to be taken into account. In respect to these characteristics, an important goal of our current work is to enhance our environment in the direction towards increased robustness.

Acknowledgments

We would like to thank Ronja for her patience and for lending us an ear whenever we were in doubt.

References

[Ahamad et al. 90] M. Ahamad, P. Dasgupta, and R.J. Leblanc, Jr. *Fault-tolerant Atomic Computations in an Object-Based Distributed System.* Distributed Computing, Vol. 4, pp. 69-80, 1990.

[Anderson et al. 89] T.E. Anderson, E.D. Lazowska, and H. Levy. *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.* IEEE Trans. Comput. 38, 12 (Dec.), pp. 1631-1644, 1989.

[Anderson et al. 92] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, Vol. 10, No. 1, Feb. 1992.

[Assenmacher et al. 93] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *PANDA — Supporting Distributed Programming in C++.* Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, Germany, July 1993.

[Bennett et al. 90] J.K. Bennet, J.B. Carter, W. Zwaenepoel. *Adaptive Software Cache Management for Distributed Shared Memory Architecture.* Proc. 17th Annual Int. Symposium on Computer Architecture, Seattle, Washington, pp. 125-134, May 1990.

[Bershad et al. 88] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D. Wagner. *An Open Environment for Building Parallel Programming Systems.* Proc. of the ACM SIPLAN PPEALS - Parallel Programming: Experience with Applications, Languages, and Systems, (July 19-21), pp. 1-9, July 1988.

[Birrell and Nelson 84] A.D. Birrell and B.J. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1), Feb. 1984.

[Buhler 90] P. Buhler. *The COIN Model for Concurrent Computation and its Implementation.* Microprocessing and Microprogramming 30, No. 1-5, North Holland, pp. 577-584, 1990.

[Chase et al. 89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. *The Amber System: Parallel Programming on a Network of Multiprocessors.* Proc. of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, 1989.

[Chase et al. 92] J.S. Chase, H.M. Levy, M. Baker-Harvey, and E.D. Lazowska. *How to Use a 64-Bit Virtual Address Space.* TR 92-03-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1992

[Chase et al. 92b] J.S. Chase, H.M. Levy, E.D. Lazowska, and M. Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System.* TR 92-03-09, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1992.

[Dasgupta et al. 91] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. *The Clouds Distributed Operating System.* IEEE Computer, pp. 34-44, Nov. 1991

[Draves et al. 91] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. *Using Continuations to Implement Thread Management and Communication in Operating Systems.* Proc. of the 13th ACM Symposium on Operating Systems Principles, Oct. 1991.

[Garrett et al. 92] W. E. Garrett, R. Bianchini, L. Kontothanassis, R.A. McCallum, J. Thomas, R. Wisniewski, and M.L. Scott. *Dynamic Sharing and Backward Compatibility on 64-Bit Machines.* TR 418, University of Rochester, Computer Science Department, Rochester, NY, April 1992.

[Gehani and Roome 88] N.H. Gehani and W.D. Roome. *Concurrent C++: Concurrent Programming with Class(es).* Software — Practice and Experience 18(12), pp. 1157-1177, Dec. 1988.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. *Fine-grained Mobility in the Emerald System.* ACM Trans. Comp. Syst. 6(1), pp. 109-133, Feb. 1988.

[Lamb et al 91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb. *The Objectstore Database System.* Comm. ACM, Vol. 34, No. 10, pp. 50-63, Oct. 1991.

[Levelt et al. 92] W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. *A Comparison of two Paradigms for Distributed Shared Memory.* Software — Practice and Experience, Vol. 22, No. 11, pp. 985-1010, Nov. 1992.

[Liskov 88] B. Liskov. *Distributed Programming in ARGUS.* Comm. ACM, Vol. 31, No. 3, pp. 300-312, March 1988.

[Marsh et al. 91] B. D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P.Markatos. *First-Class User-Level Threads.* Proc. of the 13th Symp. on Operating Systems Principles, Pacific Grove (California), pp. 110-121, Oct. 1991.

[Misra 91] J. Misra. *Loosely-Coupled Processes.* Proc. Parallel Architectures and Languages Europe (PARLE'91), Springer-Verlag, LNCS 506, E.H.L. Aarts, J. van Leeuwen, and M. Rem (eds.), pp. 1-26, Eindhoven, The Netherlands, June 1991.

[Nierstrasz and Papathomas 90] O.M. Nierstrasz and M. Papathomas. *Viewing Objects as Patterns of Communicating Agents.* Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 38-42, 1990.

[Nitzberg and Lo 91] Nill Nitzberg and Virginia Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms.* IEEE Computer, pp. 52-60, Aug. 1991.

[ODeux et al. 91] O. Deux et al. *The O_2 System.* Comm. ACM, Vol. 34, No. 10, pp. 34-48, Oct., 1991.

[Saleh and Gautron 91] H. Saleh and P. Gautron. *A Concurrency Control Mechanism for C++ Objects.* Proc. of the ECOOP 91 Workshop on Object-Based Concurrent Computing (Geneva, Switzerland, July 15-16), LNCS 612, Springer-Verlag, pp. 95-210, 1991.

[Shapiro 86] M. Shapiro. *Structure and Encapsulation in Distributed Systems: The Proxy Principle.* Proc. 6th Int. Conference on Distributed Computer Systems, pp. 198-204, Cambridge, MA, May 1986.

[Shrivastava et al. 91] S.K. Shrivastava, G.N. Dixon, G.D. Parrington. *An Overview of the ARJUNA Distributed Programming System.* IEEE Software, pp. 66-73, Jan. 1991.