Appears in "Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)," September 1993

# Panda: A Portable Platform to Support Parallel Programming Languages*

Raoul Bhoedjang Tim Rühl Rutger Hofman
Koen Langendoen Henri Bal
Vrije Universiteit Amsterdam
Department of Mathematics and Computer Science
{raoul, tim, rutger, koen, bal}@cs.vu.nl

Frans Kaashoek
MIT Laboratory for Computer Science, Cambridge MA
kaashoek@amsterdam.lcs.mit.edu

**Abstract**

Current parallel programming languages require advanced run-time support to implement communication and data consistency. As such run-time systems are usually layered on top of a specific operating system, they are nonportable. This paper reports on our early experiences with *Panda*, a portable virtual machine that provides general and flexible support for implementing run-time systems for parallel programming languages.

Panda has two interfaces: a Panda interface providing threads, RPC, and totally-ordered group communication, and a system interface which encapsulates machine dependencies by providing machine-independent thread and communication abstractions. We describe the interfaces, our experience with an initial Unix[1] implementation, and the development of a new, portable, and scalable run-time system for the Orca parallel programming language on top of Panda.

## 1 Introduction

Modern parallel programming languages require advanced run-time support for communication and data consistency. In order to fully exploit a machine's particular features, run-time systems for parallel languages tend to be built directly on top of the host operating system. Our experience with the parallel programming language *Orca* [4] on the *Amoeba* [19, 24] distributed operating system is that this strategy results in a language implementation that is difficult to port to other operating systems.

Orca is based on *shared objects*. As shared objects may be replicated to speed up read accesses, their implementation on distributed memory machines requires advanced run-time support. To investigate the scalability and portability of shared data objects, we aim to port Orca to a variety of parallel architectures. Although operating systems like Amoeba offer a virtual machine abstraction on which shared objects can be implemented,

*This research is supported in part by a PIONIER grant from the Netherlands Organisation for Scientific Research (N.W.O.).

[1]Unix is a trademark of Unix Systems Laboratories, Inc.

they are generally tied to a particular machine architecture and thus nonportable. Also, current operating systems generally provide more functionality than is needed or wanted for parallel processing (e.g., virtual memory management). Some modern operating systems, like Clouds [12], support their own object model. Unless such a model is very simple, flexible, and lightweight, layering another object model on top of it can be troublesome and inefficient.

Higher-level approaches to supporting parallel programming include page-based Distributed Shared Memory (DSM) systems (e.g., Munin [9]). Page-based DSM systems, however, often rely on manipulation of the virtual memory management unit, and therefore also suffer from portability problems.

Highly portable message passing systems exist, but they provide limited functionality. PVM [22] and p4 [8], for instance, provide low-level message passing, but they support neither threads nor totally-ordered group communication.

Instead of relying on page-based DSM, operating systems, or low-level message passing, we have developed a portable virtual machine, called *Panda*. Panda was designed with the portability requirements of parallel languages in mind and is currently used to implement a new Orca run-time system. Panda, however, does not restrict its users to Orca's object model. It provides the following, general abstractions:

- threads,
- Remote Procedure Call (RPC),
- totally-ordered group communication.

Experience with similar Amoeba abstractions has shown that efficient implementations of shared objects can be built on top of them [23]. Threads provide a simple, lightweight unit of activity. RPC [7] is a general mechanism for high-level point-to-point communication between nodes (and thus for the implementation of remote object invocation). Totally-ordered group communication [16] has been successfully employed in previous Orca run-time systems for keeping replicated objects consistent and for the implementation of a distributed checkpointing algorithm [17]. It assures that all members of a group receive all group messages in the same order, which makes many parallel applications easier to implement. Hardware broadcast mechanisms usually do not guarantee this strong semantics. Since Panda's abstractions are language-independent, we believe that Panda can be used to implement run-time systems for languages other than Orca as well.

The Panda architecture, illustrated in Figure 1, consists of two layers that reflect our design goals: portability and support for parallel programming languages. Support for parallel programming languages is achieved by providing high level abstractions in the *Panda interface*. The software that implements the Panda interface is called the *Panda layer*. Portability is achieved by implementing the Panda layer on top of the *system interface*, which encapsulates machine-dependencies. This makes the Panda layer fully machine-independent. An implementation of the system interface, a *system layer*, can be constructed with only some basic operating system support, but can also exploit features of the underlying operating system (e.g., kernel threads, scatter-gather interfaces, or hardware broadcasting and multicasting).

Panda takes a layering approach towards portability. Although layering is an effective way to abstract from machine-dependencies, it bears with it the danger of poor performance. Thoughtless layering may well result in a loss of information that is essential to

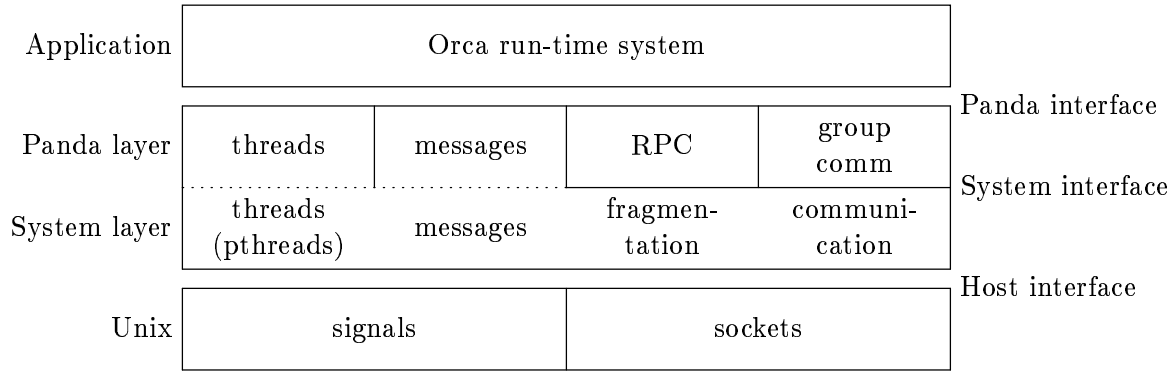


Figure 1: The Panda Architecture on Unix.

achieving good performance [20]. Therefore, we have identified Panda's performance-critical parts: threads, message manipulation, and the nature of the underlying network. These performance-critical parts are all implemented in the system layer, where they have access to low-level, operating-specific features.

The main elements of the system layer are threads, message manipulation primitives, and communication primitives. By implementing threads and messages in the system layer, we can benefit from operating system-specific features and thus achieve better performance. Communication takes place between virtual processors, called *platforms*, which are identified by *platform identifiers*. The communication primitives provide unreliable point-to-point communication and multicasting between these platforms.

Porting Panda to a new architecture requires porting the system layer only. The minimal support required from the operating system for implementing the system layer consists of a facility for unreliable message passing, and a facility for handling signals generated by incoming messages and expired timers. If the host operating system offers no more, all of the system interface has to be implemented from scratch on top of this operating system. However, most current operating systems offer threads and usable communication facilities. Implementing the system layer on such systems is easier.

We have constructed an initial implementation of Panda for a collection of SPARC-based workstations, running Unix, and connected by a 10 Mbit/s Ethernet. We intend to port Panda in the near future to a T9000-based parallel machine, the Alewife [1], and the CM-5 [25].

Section 2 describes the Panda and system interface in more detail. The machine-independent implementation of the Panda interface is outlined in Section 3. In Section 4, we describe our experience with an initial implementation of the system interface on Unix. Section 5 discusses the implementation of a new, portable Orca run-time system on top of Panda. In Section 6, Panda is compared with related systems. Finally, in Section 7, we present our conclusions.

## 2 The Panda and System Interface

In this section, we describe the relevant parts of the current[1] Panda and system interfaces. For reasons of efficiency threads and messages are implemented in the system layer (and part of the system interface), but most of the primitives associated with them are also visible

[1]Based on further experience with Panda and Orca these interfaces may evolve.

```
void thread_create(thread_p thread, void * (*func)(void *arg), void *arg,
                   long stacksize, int priority);
void thread_exit(void *result);
void thread_yield(void);
int thread_getprio(thread_p thread);
void thread_setprio(thread_p thread, int priority);
```

Table 1: The thread interface

in the Panda interface.To distinguish between Panda layer and system layer functions, each Panda layer function name is prefixed by *"pan_"*, and each system layer function by *"sys_"*. Functions that are part of both interfaces are not prefixed.

## 2.1 The Panda Interface

The Panda interface provides the RPC, totally-ordered group communication, and thread abstractions with which Panda applications can be built.

### Threads

The thread interface (see Table 1) is based on the Pthreads [15, 18] and C Threads [11] interfaces. Since threads are implemented in the system layer (see Figure 1), thread primitives do not have a *pan_* prefix.

From experience with Amoeba threads we have learned that a thread package for parallel programming languages should support priority scheduling to handle incoming messages immediately when they arrive. This automatically implies preemption of running threads when a new message arrives. Priorities are supported by the operations *thread_getprio* and *thread_setprio*, which return and set priorities. Since we do not specify a scheduling policy among threads with the same priority, we also provide the function *thread_yield* that tries to run another runnable thread with the same priority.

Synchronization between threads is based on mutexes and condition variables. Together these provide strong enough semantics to construct monitors [14].

### RPC

The RPC interface (see Table 2) is based on the notion of a *service* that provides a number of operations. A service is implemented by one or more servers. A server can register its services with Panda's name server using *pan_export_service*, giving as arguments the number of operations it supports and an array of pointers to these operations. Before an operation can be called, the client must get a handle to a server (*pan_import_service*). This handle can be used to identify the server that must handle the RPC request (*pan_do_rpc*).

When a request message comes in, a thread is started. This thread calls the registered function for the specified service and operation. This function has three parameters: an operation index number, an input message, and a reply message.

```
int pan_export_service(char *name, int nr_operations,
                       void (*func)(int operation,
                                    message_p request, message_p reply)[ ]);
int pan_import_service(char *name);
int pan_do_rpc(int handle, int operation_no,
               message_p request, message_p *reply);
```

Table 2: RPC interface

### Totally-Ordered Group Communication

The group abstraction of Panda (see Table 3) supports totally-ordered, closed groups [16]. The total ordering assures that every group member receives all group messages in the same order. A group is closed if only its members can send messages to the group. This makes an efficient implementation possible.

Each group is identified by a character string, which is registered with Panda's name server. A platform that wants to join the group calls *pan_group_join*, which initializes a group structure. If the group does not exist, it will be created. Group messages are handled asynchronously by an upcall to a specified receive routine, which handles incoming messages one by one to ensure total ordering.

## 2.2 The System Interface

The system interface hides machine dependencies by providing three abstractions: threads, messages, and communication primitives. As explained before, threads are implemented in the system layer; the system and Panda interface are identical with respect to threads.

### Communication

The communication facilities are divided into two parts: send primitives (see Table 4) and addressing. At startup time each platform gets a unique platform identifier (pid), which is an integer number ranging from 1 to the number of platforms. This pid is used as a point-to-point address. Pids can be grouped together in a *platform set (pset)* that serves as a logical multicast address.

The send primitives provided by the system layer are *sys_unicast*, for (unreliable) point-to-point communication, and *sys_multicast*, for (unreliable) one-to-many communication. When the Panda layer initializes itself, it registers a message receive handler with the

```
int pan_group_join(group_p group, char *name,
                   message_p mesg_join, void (*receive)(message_p mesg_in));
void pan_group_leave(group_p group, message_p message);
void pan_group_send(group_p group, message_p message);
```

Table 3: Totally-ordered group communication interface

```
void sys_unicast(int pid, message_p message);
void sys_multicast(sys_pset_p pset, message_p message);
```

Table 4: System communication primitives

system layer. All platforms run a system layer receive daemon. Each time a (unicast or multicast) message arrives, this daemon makes an upcall to the message receive handler in the Panda layer.

### Messages

At the interface level, messages look like stacks. To construct a message, senders push data fields of a specified size and alignment onto a message's stack; these fields are popped in reverse order by the receivers (see Table 5). *message_look* is similar to *message_pop*, but it does not pop the data field off the message.

Although the communication primitives hide machine dependencies, they do not handle messages with a length larger than the underlying system supports. Instead, the system interface provides primitives to fragment messages so that they can be handled by the communication primitives. This fragmentation is based on a *common header*, a header that is placed in front of every fragment.

With *sys_message_mark* the Panda layer can specify the end of the data part and the start of the common header. Every data field pushed after the mark belongs to the common header. *sys_message_fragment* initializes a new *fragment* message containing the common header and part of the data of the original message. This function takes as a parameter an offset indicating the start of a fragment's data in the original message, and it returns the offset of the next fragment's data. After getting a fragment from a message, some fields in the common header part can be filled in with information that identifies this fragment.

At the receiving side, *sys_message_assemble* is used to reassemble the original message. These primitives resemble the x-kernel primitives for fragmenting messages [20].

A fragment message need not contain copies of the common header and data fields of the original message; pointers may be used instead. To support sharing of the common

```
void *message_push(message_p message, int length, int align);
void *message_pop(message_p message, int length, int align);
void *message_look(message_p message, int length, int align);
void message_copy(message_p message, message_p copy);

int sys_message_data_len(message_p message);
void sys_message_mark(message_p message);
int sys_message_fragment(message_p message, message_p fragment, int offset);
void sys_message_assemble(message_p message, message_p fragment);
```

Table 5: The message interface

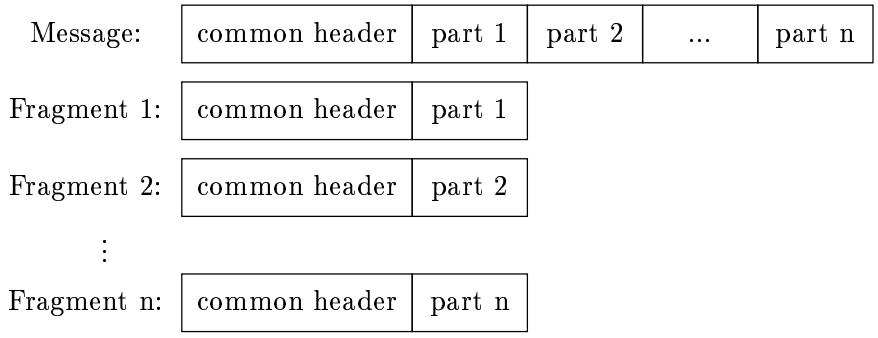


Figure 2: Fragmentation with a Common Header

header among fragments, only one fragment may exist at a time (i.e., before creating the next fragment, the predecessor fragment must be released). Now it is always clear to what fragment the identification information in the common header refers. Using pointers to the original message in the fragment message avoids unnecessary copying.

### 2.3 Portability and Efficiency

Both the Panda and the system interface have been designed to allow efficient implementations. Some of the abstractions present in the system layer may seem high-level, but providing these abstractions rather than low-level primitives gives us the opportunity to exploit advanced features offered by many modern operating systems. Among these features are efficient, user-level thread packages or kernel threads, scatter-gather message transmission, access to hardware broadcasting and multicasting, etc.

We have decided to make message passing in the system interface unreliable. Reliable message passing would have prohibited an efficient RPC implementation on architectures that only provide unreliable message passing, since sending a message reliably requires at least two network packets.

## 3 Implementation of the Panda Interface

The Panda interface is implemented using the primitives provided by the system interface. Therefore, this code is entirely machine-independent.

### 3.1 Group Communication Structure and Protocol

The group communication implementation is based on [16], which describes an efficient, totally-ordered, and atomic group communication protocol. Since we are not concerned with fault tolerance, we have implemented this protocol in a non-resilient way, thereby loosing atomicity (all-or-none delivery, even in the presence of processor crashes). It is possible, however, to do synchronous checkpointing on top of totally-ordered group communication without atomicity [17].

Totally-ordered group communication is achieved by having a special member in each group, the *sequencer*, which assigns a sequence number to each group message. This gives two possibilities for a group send [16]. The first method is to send a point-to-point message to the sequencer, and the sequencer then broadcasts the message after filling in the sequence number (the so-called PB method). The second method is to let the sender itself do

the broadcast. When the sequencer receives this broadcast message, it assigns a sequence number to it, and broadcasts a short acknowledgement message containing this sequence number (BB method). This method saves network bandwidth (because the data is transmitted only once), but it generates more interrupts. A choice between the two methods is made dynamically, based on the message size and on information from the system layer. Either way, when a message arrives its sequence number is checked against the last sequence number received. If the sequence number indicates this is the next message, the message can be delivered to the application level, otherwise the receiver asks the sequencer for the missing messages.

Incoming group messages are handled by a single daemon thread, which upcalls to the receive handler specified by *pan_group_join*. To prevent loosing the ordering of the group messages by unpredictable thread scheduling, we use only one daemon thread per group.

Since the underlying architecture may have stronger semantics than we actually need, the system layer can define the following two compilation flags: RELIABLE_MULTICAST to specify that multicast messages are never lost, and ORDERED_MULTICAST, which specifies that all multicast messages arrive in total order. The code of the group implementation is adapted by these parameters.

### 3.2 RPC Structure and Protocol

The RPC protocol is based on Birrell and Nelson[7]. An RPC requires three messages during normal execution: a request, a reply, and an acknowledgement. On some architectures (e.g. a network of T9000 Transputers) reliable message passing is provided already, so the acknowledgement is not necessary. Therefore, the system layer can define a compilation flag RELIABLE_UNICAST, which implies that messages are reliable. When compiled with this flag set, no acknowledgements are sent.

## 4 Experience with Panda on Unix

We have implemented Panda on Unix (SunOS 4.1.2). The following subsections describe the implementation of the system layer and the performance of Panda. Not all parts of the implementation have been tuned yet.

### 4.1 Implementation of the System Interface

In contrast to modern operating systems for parallel computers, Unix provides neither threads nor multicasting. Nevertheless, we have selected Unix as our initial target operating system, because it provides a complete programming environment and because it is widely available. To avoid writing a large amount of software that we expect to be provided by future target platforms, we have used public-domain software for our threads and (unreliable) multicast implementation. We have implemented our threads interface with Pthreads [18], a POSIX-conformant, user-space threads implementation. We have extended the kernels of our SPARC workstations with IP multicast, a kernel extension for multicasting [13]. Point-to-point message passing has been implemented on top of UDP [21].

Pthreads provides all the functionality we need, including priority scheduling, and runs entirely in user-space. User space threads are more efficient than (pure) kernel-based implementations, because thread context switches do not involve trapping to the kernel. However, they suffer from poor integration with virtual memory management and blocking I/O [2].

| Test case | Performance |
|---|---|
| Thread switching | 300 $\mu$s |
| Unicast message passing latency | 2.1 ms |
| Multicast message passing latency | 2.3 ms |
| Null RPC latency | 5.9 ms |
| RPC throughput | 435 Kbyte/s |
| Group communication latency | 6.7 ms |

Table 6: Performance figures

Virtual memory makes performance less predictable: a page fault will block all threads while the missing page is brought in from the disk. Blocking network I/O is a more serious problem. The thread that waits for incoming messages should not block all other threads contained in the same process. Therefore, each platform's receive daemon thread uses Unix's asynchronous and nonblocking I/O options to prevent blocking the entire process when reading from the network. If it finds no pending messages, it waits for a signal. Since Pthreads supports signals on a per-thread basis, only the receive daemon is blocked, not the entire process. Incoming messages generate SIGIO signals that cause the receive daemon to be rescheduled immediately (since it has the highest priority).

Since UDP has no support for Panda's stack-based message manipulation and fragmentation routines, most of our system layer code is devoted to the implementation of these routines. This code is machine-independent and need not be changed when Panda is ported. However, it may be beneficial to adapt the code to platform-specific features (e.g., scatter-gather facilities). The system interface was designed to allow such modifications in its implementation. No changes need to be made to the interface itself.

### 4.2 Performance

To compare the overhead of our protocols, Table 6 gives an overview of the performance of the communication primitives in the system and the Panda layer. These performance figures were obtained on a collection of diskless SPARCstations SLC, running at 20 Mhz, and connected by a 10 Mbit/s Ethernet. Also included is the overhead of thread context switching. The message passing latencies were measured with two platforms (running on two different machines), one sending 10,000 messages and the other sending acknowledgements.

The null RPC latency is measured with 10,000 empty request and reply messages to an empty server routine, and the throughput with 1000 RPC messages with a request message size of 8000 bytes and an empty reply message.

The latency of an empty group message is 6.7 ms. Since the protocol uses negative acknowledgements, this latency is almost independent of the number of platforms [16].

RPC and group communication performance of our initial Panda implementation is within a factor 4 of Amoeba, which has RPC and group communication built into its microkernel. (On comparable hardware, Amoeba does a null RPC in 1.7 ms; a null group message on a collection of 20 MHz MC68030s takes 2.7 ms.)

# 5 Implementing the Orca RTS using Panda

Orca is a type-secure parallel and distributed programming language. Orca programs consist of processes that communicate solely through shared objects, which are instances of abstract data types. To speed up read accesses to shared objects, such objects may be replicated. The replication strategy is based on a combination of compile-time and run-time techniques [3]. The Orca run-time system (RTS) is responsible for keeping replicas in a consistent state.

Orca is being re-implemented to obtain a programming system that is portable, efficient, and scalable. A new Orca compiler generating fast and portable ANSI-C code has already been implemented, and we are now reimplementing the run-time system on top of Panda. The new RTS makes heavy use of all Panda facilities:

**Threads** The Orca RTS uses Panda's threads for the implementation of Orca processes. Threads are also created implicitly as a result of incoming RPC requests and group messages.

**RPC** RPC is used by the RTS for performing operations on remote, nonreplicated objects and for transmitting objects when they are migrated or replicated.

**Group Communication** When a shared, replicated object is simultaneously updated by two Orca processes, then *all* replica holders of this object must apply the updates in the same order. To achieve this, the RTS uses totally-ordered group communication. All RTSs belong to a single group and simply send their update messages to this group. Since communication in this group is totally-ordered, all RTSs receive and process the update messages in the same order, thus keeping the replicas consistent.

In contrast with previous Orca implementations based on Amoeba, machine dependencies are now hidden from the RTS by Panda, thus making the RTS portable. Moreover, as can be seen from the interface descriptions, the primitives in the Panda interface are language independent and can be used for the implementation of other language run-time systems.

# 6 Related Work

Panda differs from many existing parallel programming platforms in that it has been designed with the requirements for run-time systems for parallel programming languages in mind. As previous Orca implementations have demonstrated, such run-time systems can benefit from high-level support in the form of RPC and totally-ordered group communication. In this section we compare Panda with other systems that can be used for implementing parallel programming languages; some of these systems are language-based themselves, whereas others come in library form. We consider portable message passing systems (p4, PVM), Distributed Shared Memory systems (Munin and Midway), ARCADE, and ISIS/HORUS.

Like Panda, PVM [22] and p4 [8] provide portable communication primitives. PVM and p4, however, provide message passing primitives only, and neither provides high-level communication in the form of RPC or totally-ordered group communication. In our experience, RPC and group communication simplify the implementation of complex run-time systems. Neither PVM nor p4 supports lightweight threads. Because of their high context-switching overhead, processes are not suitable for hiding communication latencies.

Both PVM and p4 provide visualization tools and support for heterogeneity. Work on extending Orca and Panda with performance debugging tools is in progress. Unlike PVM and p4, Panda does not support heterogeneity.

DSM systems like Munin [9] and Midway [5] support parallel programming by providing a shared memory abstraction that hides all message passing from the programmer. Although Panda does not provide such an abstraction by itself, it gives sufficient support to layer a shared memory model on top of it.

Munin programmers annotate shared variables with their expected access pattern. These shared variables are kept consistent through a release consistency protocol. The Munin implementation of this protocol relies on the Memory Management Unit (MMU) to detect writes to pages containing shared data, thus rendering the implementation machine-dependent.

In the Midway system, shared variables are associated with their synchronization objects and kept consistent through a memory consistency protocol called entry consistency. Although Midway does not rely on MMU manipulation to enforce entry consistency, it does need the MMU to implement stronger memory consistency models (release consistency and processor consistency) [5].

Munin and Midway support parallel programming by providing a shared memory abstraction and weak consistency models. We consider this support too low-level for application programming: programmers should not have to annotate their variables or use low-level locking. Munin (and sometimes Midway) needs to manipulate the MMU, while Orca implementations guarantee sequential consistency, which is stronger than all previously mentioned forms of consistency, without MMU manipulation. Thus, layering an Orca run-time system on top of Panda results in a portable implementation of sequential consistency.

Like Panda, ARCADE [10] supports the implementation of parallel programming languages through high-level abstractions. The ARCADE abstractions, however, are based on language-independent data units rather than communication mechanisms. A data unit is an abstraction of a typed region of memory that can be named, moved, and shared across multiple nodes in a distributed environment. Language-specific objects can be mapped onto ARCADE's data units.

Like Orca, ISIS [6] is currently being reimplemented for reasons of portability and scalability. The new ISIS system, HORUS [26], has a core interface that provides reliable, causal multicasting (CBCAST). Other services are implemented on top of this interface. This interface is somewhat like the Panda interface, although the ordering semantics of CBCAST is weaker than that of Panda's group communication — totally-ordered group communication and RPC are implemented on top of CBCAST. The CBCAST layer is implemented on top of a portable operating system abstraction, MUTS (Multicast Transport Service), that is similar to Panda's system layer.

# 7 Conclusion

This paper described the motivation, design, and implementation of Panda, an implementation platform for parallel programming languages, that combines portability with flexibility and efficiency.

Panda achieves portability by defining a machine-independent system interface in addition to the Panda interface. The Panda interface is implemented on top of this system interface and is thus machine-independent. The implementation of the system interface requires only basic operating system support for the context switching of threads and unreliable message passing. Most of the current system interface implementation is machine-independent and can be easily reused. Its careful interface design and modular implementation allow for the incorporation of efficient, native thread packages and communication facilities (e.g., scatter-gather message passing and hardware broadcasting and multicasting). Porting the system layer to other parallel architectures should be straightforward.

Panda provides its users with three flexible abstractions that have been effectively employed for the implementation of several Orca run-time systems: threads, RPC, and totally-ordered group communication. We use these abstractions to implement Orca's object model.

Early experience with a SPARC/Unix implementation of Panda has shown the feasibility of a layering approach towards support for parallel programming languages.

## Acknowledgements

We would like to thank Gerard Kok and Anil Sukul for testing Panda's RPC implementation. We also greatly appreciate the helpful comments of Ceriel Jacobs and Leendert van Doorn on earlier drafts of this paper.

## References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowics, K. Hurihara, B-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS TM-454, MIT, 1991.

[2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 95–109. ACM, 1991.

[3] H.E. Bal and M.F. Kaashoek. Object Distribution in Orca using Compile-time and Run-time Techniques. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, Washington D.C., 26 September–1 October 1993. To be published.

[4] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[5] B. Bershad, M. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Computer Conference*, 1993.

[6] K. P. Birman and T.A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

[7] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] R. Butler and E. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. *Journal of Parallel Computing*. (submitted).

[9] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th Symposium on Operating Systems Principles*. ACM, 1991.

[10] D.L. Cohn, A. Banerji, M.R. Casey, P.M. Greenawalt, and D.C. Kulkarni. Basing Micro-kernel Abstractions on High-Level Language Models. In *Proc. of the Autumn 1992 OpenForum Technical Conference*, pages 323–336, Utrecht, Holland, November 1992.

[11] E.C. Cooper and R.P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1988.

[12] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, 3, 1990.

[13] S.E. Deering and D.R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 17(1), January 1991.

[14] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 12(10):549–557, October 1974.

[15] IEEE. *Threads Extensions for Portable Operating Systems P1003.4a*, draft 6 edition, February 1992.

[16] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit Amsterdam, 1992.

[17] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent Fault-Tolerance in Parallel Orca Programs. *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.

[18] F. Mueller. Implementing POSIX Threads under UNIX: Description of Work in Progress. In *Proc. of the 2nd Software Engineering Research Forum*, pages 253–261, November 1992.

[19] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba–A Distributed Operating System for the 1990s. *IEEE Computer*, 1990.

[20] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer*, pages 23–33, May 1990.

[21] J. Postel. User Datagram Protocol. Internet Request for Comments RFC768, September 1981.

[22] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[23] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[24] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(2):46–63, December 1990.

[25] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[26] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proccedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 269–283, April 27–28 1992.