Modularity and Interfaces in Micro-Kernel Design and Implementation: A Case Study of Chorus on the HP PA-RISC*

Jonathan Walpole, Jon Inouye, and Ravindranath Konuru
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(walpole,jinouye,konuru@cse.ogi.edu)

ABSTRACT

The key concept that distinguishes micro-kernel operating systems from their macro-kernel counterparts is modularity. Micro-kernels implement operating system functionality in well-defined modules with clearly identified interfaces between them. Proponents of this modular approach to operating system design claim that it offers advantages in the areas of portability, correctness, protection, extensibility, and reconfigurability for distributed architectures. If micro-kernels are to gain wider acceptance however, it is important to ensure that these benefits of modularity can be attained without incurring significant performance degradation when compared to macro-kernels.

In this paper we explore the relationship between modularity and performance by examining an implementation of the Chorus micro-kernel operating system on the Hewlett-Packard PA-RISC workstation. We outline the key interfaces in Chorus and study the architectural assumptions implicit in these interfaces.

1 Introduction

The key characteristic that distinguishes micro-kernel operating systems from their macro-kernel counter parts is modularity. Micro-kernel operating systems are structured as a collection of cooperating servers running above a minimal kernel. Structuring operating systems in this manner offers a number of potential benefits including ease of distribution, reconfigurability, extensibility, portability, protection and correctness [4, 5].

It has been argued that ease of distribution and reconfigurability result from the separation of system components and the use of message passing as the communication mechanism among them. Similarly, this separation of system components is claimed to improve extensibility by allowing new operating system functionality to be added, in the form of new system servers, without altering existing components or the micro-kernel itself. Arguments about improved correctness are based on the principle that it is easier to avoid design and programming errors if a system is composed from several small modules rather than a single large module. Modular systems also allow protection to be enforced at the boundaries between modules. Finally, it is claimed that micro-kernel operating systems improve portability by localizing machine-dependent code within the micro-kernel.

*This research is supported by the Hewlett-Packard Company, Chorus Systèmes, and the Oregon Advanced Computing Institute (OACIS).

A central research issue in the construction of real-world micro-kernel operating systems is the design of interfaces that allow the above benefits to be attained while achieving performance comparable to less modular, macro-kernel operating systems. This is a major challenge for micro-kernel designers because macro-kernels implement very low-overhead invocation across their internal interfaces by using local procedure calls. In order to achieve all the proposed benefits of modularity, micro-kernels must support a variety of interfaces and invocation mechanisms, many of which are considerably more complex and heavy weight than a local procedure call.

The challenge for micro-kernel designers is to (a) define interfaces that are expressive enough to allow the above benefits to be attained, and (b) to provide implementations of those interfaces that offer acceptable performance. Unfortunately, in making these implementation choices operating system designers are often forced to trade the sought after benefits of modularity for performance. The degree to which this occurs depends on two main factors. First, the anticipated requirements of the target application domain determine the relative importance of modularity and performance. For example, some target application domains may consider runtime protection and dynamic reconfiguration to be critical, and worth sacrificing performance for, whereas others may consider performance to be of paramount importance.

Second, the characteristics of the architecture, or class of architectures, on which the operating system is expected to execute may determine the efficiency with which particular functionality can be supported. Making such implementation decisions based on key architectural assumptions is already common practice in operating system design. However, the success of this approach depends heavily on the accuracy of these assumptions. Consequently, operating system designers must be aware of the trends in computer architecture (and vice versa). This issue is important both for the implementation of efficient interfaces for system structuring, and for defining an appropriate separation between portable and non-portable code.

We believe that the definition of appropriate interfaces, and the implementation decisions that determine the balance between performance and modularity will be critical to the eventual success or failure of micro-kernel operating systems. In this paper we explore this design space by studying our implementation of the Chorus micro-kernel on the Hewlett-Packard Precision Architecture RISC (PA-RISC) 9000/834 workstation.

The remainder of the paper is organized as follows. Section 2 outlines the Chorus approach to modularity and discusses the interfaces defined by Chorus. The trade-off between performance and modularity in the implementation of those interfaces, and the architectural assumptions implicit in the implementation choices, are also discussed. Section 3 outlines the key characteristics of the PA-RISC, discusses the salient features of our implementation of Chorus, and revisits the architectural assumptions implicit in Chorus. Section 4 summarizes the strengths and weaknesses of the Chorus approach, and section 5 concludes the paper.

2 Modularity and Interfaces in Chorus

A Chorus-based operating system is structured as a set of cooperating subsystem servers executing above a Chorus nucleus (henceforth called the micro-kernel) [2]. There are a number of key interfaces in this operating system structure (see figure 1). A high-

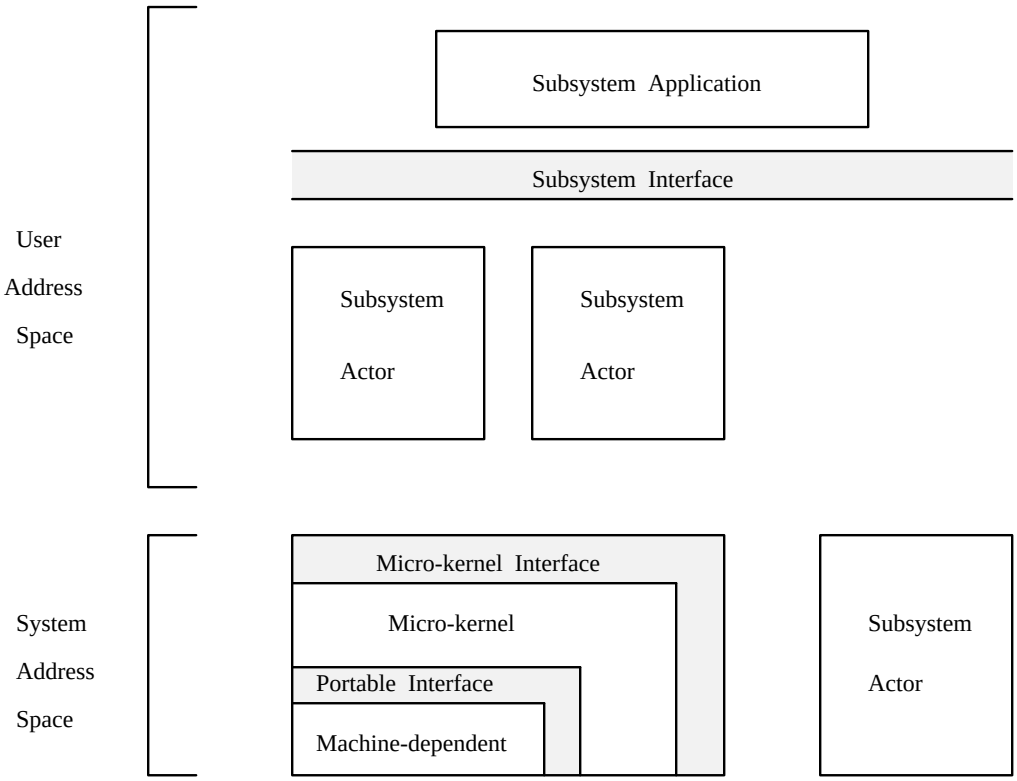

Figure 1: A typical Chorus Operating System Structure

level operating system interface, such as the UNIX system call interface, is presented by the subsystem servers to application programs. We will refer to this as the *subsystem interface*. A lower-level interface, called the *micro-kernel interface* defines the interaction between the micro-kernel and the subsystem servers. The micro-kernel interface exports a number of basic abstractions including IPC which is used to build higher-level interfaces between subsystem servers. An additional interface, defined within the micro-kernel, separates machine-dependent code from portable code. We will refer to this as the *portable interface*. The portable interface is intended to be architecture-independent, and is supported by a new implementation of the micro-kernel's machine-dependent layer for each new architecture on which it executes.

Some of the basic abstractions exported by the micro-kernel interface are ports, messages, threads and actors[1]. Chorus defines two types of actor (*user* and *supervisor*) based on their privilege level and allowed operations [10]. User actors are placed in separate address spaces and system actors share the system address space. In addition to defining operations for the creation and manipulation of the basic micro-kernel abstractions, the micro-kernel interface allows subsystem servers to attach handlers to traps, interrupts, and exceptions.

User and supervisor actors see the same specification of the micro-kernel interface, but use distinct implementations of the interface. These implementations take the form of separate libraries above the interface, and separate vectors of routines below the interface. The use of a common interface specification allows the decision of whether to load an actor into system or user space to be delayed until link time.

[1] An actor is the unit of resource allocation in Chorus, similar to a task in Mach [1].

During a micro-kernel interface call, stubs in the library for user actors load the call number into a temporary register and use an architecture-specific instruction to cause a change in privilege level[2]. In contrast, the stubs in the library for supervisor actors take advantage of the fact that supervisor actors are in the same address space as the micro-kernel by using a global data structure, called the ROOT structure, to locate the appropriate vector of routines in the micro-kernel. The call stub for a supervisor actor thus reduces to little more than a procedure call to the required micro-kernel routine.

The supervisor actor concept offers several important advantages. First, it allows heavily used IPC paths between subsystem servers to be streamlined by using IPC calls in the optimized implementation of the micro-kernel interface. Second, it allows trap-based subsystem interfaces to be implemented efficiently by directly calling subsystem call handlers rather than passing control back up to an emulation library in the application's address space.

For example, the Chorus MiX subsystem[3] presents a UNIX compatible subsystem interface by using a supervisor actor (called the process manager (PM)) to attach system call handlers to UNIX-specific trap numbers. When an application makes a MiX system call a trap is generated and the micro-kernel calls the handler attached by the PM. Since the PM is a supervisor actor and runs in the system address space it can access the call parameters directly in the application's address space. Once the system call has been identified, it is either executed directly within the PM, or passed via IPC to another subsystem actor (such as the object manager[4] (OM)). Other MiX subsystem actors also attach handlers to hardware events. For example, the OM connects handlers to disk interrupts, i.e., the disk driver is part of the OM.

2.1 Architectural Assumptions in Chorus

In all of its interfaces, Chorus is fairly successful in achieving the benefits of modularity without sacrificing too much performance. This is largely a result of good design decisions relating to the separation of specification from implementation in its interfaces. However, there are several key assumptions implicit in the design decisions that led to the current interface definitions. These assumptions are aimed at optimizing performance for some "common class" of architectures.

First, the decision to load supervisor actors into the system address space is based, in part, on the assumption that invocation between user-level entities is considerably more expensive than invocation between entities residing in the system address space. Second, because some supervisor actors need to perform privileged instructions, to access hardware for example, Chorus places them in the system address space. The basic assumption here is that an actor can be privileged only if it runs in the system address space.

The placement of supervisor actors in the system address space in order to support the attachment of handlers to traps and interrupts involves similar assumptions, i.e., that efficient handler invocation requires the handler to reside in the system address space. This kind of assumption is related to the relative cost of cross-address space and intra-address

[2]On most architectures this is a trap-based instruction.

[3]MiX is a UNIX System V compatible subsystem that runs above the Chorus micro-kernel. We had access to Chorus MiX 3.2.

[4]The object manager implements file service and acts as a default mapper.

space communication and is implicit in the decision to include the disk driver in the MiX 3.2 object manager.

Finally, the portable interface specification involves a number of architectural assumptions. Of specific interest in our experiments with Chorus is the extensive use of memory mapping in and above the portable interface. This assumes that memory mapping is inexpensive, as is the case on architectures with physically addressed caches.

The design and implementation decisions discussed above are not unusual. In fact, they are based on "common knowledge" about traditional computer architecture. In the following sections we argue that the characteristics of current-generation architectures have begun to change, and that operating system designers may need to re-evaluate some of these basic architectural assumptions.

3 Supporting the Chorus Interfaces on the PA-RISC

The PA-RISC is the framework for HP's 3000/900, 9000/800, and 9000/700 series computer systems. During 1991 we ported the Chorus v3.3 micro-kernel to the Hewlett-Packard 9000/834 workstation, and studied the interaction between Chorus and the PA-RISC architecture. This section discusses the issues involved in implementing the Chorus interfaces on the PA-RISC and revisits the architectural assumptions implicit in those interfaces.

The PA-RISC provides a 64-bit global address space that is shared between all processes and the operating system. Virtual memory is partitioned into segments, called *address spaces*, each containing 2^{32} bytes. Our implementation of Chorus uses the first of these 32-bit address spaces for the system address space which is further divided into partitions for the micro-kernel, supervisor actors, and the ROOT structure. Virtual addresses consist of two components: a space identifier and a space offset. In short pointer (32-bit) addressing mode the space identifier is held in a space register which is identified using the two most significant bits of the 32-bit offset. In long pointer (64-bit) addressing mode both parts of the address are specified explicitly. The PA-RISC also provides specific instructions for intra-space and inter-space branching.

Protection between processes sharing the global virtual address space is supported using protection identifiers, access identifiers, and access rights. Protection identifiers are associated with processes, whereas access identifiers and access rights are associated with virtual memory pages. Access rights specify the types of access that are allowed at different privilege levels, and access identifiers are used in combination with protection identifiers to implement capability-style protection. During execution, four protection registers are available to store some of the protection identifiers associated with a process. In order to make a successful access, the requested operation and privilege level must pass the access rights check, and one of the four protection registers must contain a protection identifier that matches the page's access identifier[5].

In our Chorus implementation we use a special access identifier (0) for the code and data pages of the micro-kernel and supervisor actors. This has special significance on the PA-RISC, in that it matches all protection identifiers. Access rights are used to prevent accesses by code running at user privilege level. However, since the micro-kernel and supervisor

[5]It is possible to associate more than four protection identifiers with a process if a handler is provided to manage the protection fault generated during a match failure at access time.

actors all run at the highest privilege level this does not prevent them from accessing each others' pages.

In common with other architectures, promotion of privilege level can be triggered during a trap or hardware exception. However, the PA-RISC also provides more efficient support for privilege promotion and reduction. Privilege promotion can occur, without causing a trap, by executing a *gate* instruction on a specially protected page, called the *gateway page* which is mapped, execute-only, into a pre-defined location in the system address space. In our implementation, the access rights on the gateway page are set such that the execution of a gate instruction will cause promotion to the highest privilege level.

Efficient privilege reduction is supported using a two-level instruction address queue which supports delayed branching. The two least significant bits of the instruction address are not needed since instructions lie on 4-byte boundaries. These bits are used instead to keep track of the current execution privilege, and can be set during a branch instruction in order to *reduce* the current privilege level. This feature is used during the return from a system call.

3.1 The Micro-Kernel Interface

The implementation of the micro-kernel interface for user actors makes use of the above features in the following manner:

- A thread in a user actor makes a micro-kernel interface call by executing a stub in the micro-kernel interface library for user actors (**chorusLib.a**).
- The call stub loads a subsystem number and a call number into temporary registers. This allows the micro-kernel to marshal calls intended for a subsystem interface rather than the micro-kernel interface. In order to distinguish micro-kernel interface calls, the micro-kernel is assigned a special subsystem number.
- The call stub then executes an inter-space branch instruction to the gateway page in the system address space.
- A gate instruction is executed in the gateway page, which causes privilege promotion, and an intra-space branch instruction is executed into the appropriate entry point in the system address space.
- The micro-kernel switches to the system stack and saves the necessary registers. If the subsystem number indicates a micro-kernel interface call the micro-kernel copies the parameters from the user stack and calls the appropriate routine in the call vector for user actors. Otherwise, a handler attached by a specified subsystem actor is called. The subsystem actor is then responsible for copying parameters from the user stack and performing the subsystem-specific call.
- After the call has been serviced by the micro-kernel, or by a subsystem actor, the micro-kernel completes the call by switching stacks and using the delayed branch instruction to return to user space and re-establish the original privilege level. Note that no trap was necessary to handle the system call.

The implementation of the micro-kernel interface for supervisor actors differs from that for user actors in the following way:

- A thread in a supervisor actor makes a micro-kernel interface call by executing a stub in the micro-kernel interface library for supervisor actors (**chorusSvLib.a**).
- The call stub loads the address of the micro-kernel's call vector for supervisor actors from the ROOT structure[6]. The stub uses the call number to calculate the address of the appropriate routine in the call vector for supervisor actors and uses it to call the routine. This does not require a change of stacks because supervisor threads always execute on a system stack.

Note that this implementation of the micro-kernel interface is only an optimization rather than a change in the interface specification. Supervisor actors may still use the user actor micro-kernel interface library without a loss in functionality.

3.2 The Portable Interface

In addition to affecting the implementation of the micro-kernel interface, the PA-RISC's architectural features also had a significant impact on the implementation of the portable interface. In fact, the majority of the work involved in our port of Chorus to the PA-RISC was associated with building a new implementation of the portable interface. There were a number of interesting aspects to this work, particularly in the area of cache management. The PA-RISC uses a virtually addressed cache to improve performance by allowing TLB look-up and cache access to be performed in parallel [9]. As with other architectures that use virtually addressed caches (such as the IBM RS-6000 and the MIPS 4000) the PA-RISC relies on the operating system to maintain *address translation consistency*. In other words, address aliases falling in different cache sets must be resolved by the operating system. Such aliases are generally created by mapping multiple virtual addresses to the same physical address. However, since the PA-RISC also uses the cache when executing in physical addressing mode, aliases can arise when the cache index generated by a physical address differs from the index produced when accessing the same data using a currently mapped virtual address. In either case, the operating system must prevent any loss of consistency due to the concurrent placement of the same data item in different cache lines.

Unfortunately, the Chorus portable interface contains several functions that can generate aliases, and the portable layers of Chorus assume that such aliases are inexpensive (both to set up and maintain). In order to support aliasing at the portable interface, the machine-dependent layers must maintain address translation consistency. In our implementation we achieved this by using a technique called *pseudo-aliasing*. Pseudo-aliasing guarantees that for any physical page, only one mapping can exist in the cache and TLB at a time. When an access is attempted via a virtual address that is logically valid, but for which the mapping has been removed from the cache and TLB, a *pseudo page fault* occurs. During pseudo page fault handling any existing mapping to the physical page is invalidated, the cache lines and TLB entry associated with it are flushed, and a new mapping is established. This approach presents a semantically correct implementation of the portable interface. However, it significantly changes the relative costs of certain primitive virtual memory operations. In particular, maintaining multiple mappings to the same page, and unmapping a page, become expensive (see [7] for more details).

[6] This is set up during kernel initialization.

3.3 Revisiting the Architectural Assumptions in Chorus

Our initial implementation of Chorus did not take full advantage of many of the more interesting features of the PA-RISC such as the global address space, protection, and privilege manipulation facilities. This was largely due to caution: we wanted to avoid making any radical departures from the standard Chorus approach in our first implementation. Consequently, we implemented Chorus as if the architectural assumptions discussed in section 2 held for the PA-RISC. This section revisits those assumptions and discusses alternative approaches to implementing Chorus on the PA-RISC (see [11, 6, 8, 12] for a more detailed discussion of our implementation and possible alternative approaches).

First, Chorus tends not to distinguish between issues of address space, protection domain and privilege level. Actors either run as supervisor actors at the same privilege level, and in the same address space and protection domain as the micro-kernel, or they run as user actors at the lowest privilege level, and in their own address space and protection domain. On the PA-RISC the concepts of address space, protection domain and privilege level are orthogonal. Therefore, it is feasible to use arbitrary combinations of these features in the implementation of different types of actors. This approach leads to a range of possible invocation costs depending on the specific boundaries between actors.

In order to gain a better understanding of these costs we profiled our Chorus implementation. Table 1 illustrates costs for a cross address space call at the same privilege level and protection domain, and null system calls using both the user and supervisor implementations of the micro-kernel interface[7].

For comparison, the cost of a null procedure call is also presented. Contrary to Chorus' assumptions, these figures show that the cost of an inter-space, intra-protection domain call (3.4 μ seconds) is not dramatically higher than the cost of an intra-space, intra-protection domain call (0.9 μ seconds). This suggests that, on the PA-RISC, groups of actors which communicate heavily do not necessarily have to be placed in the same address space in order to exhibit good performance.

Table 1: Basic invocation costs

Call-type	Time (in μ sec)
null procedure call	0.9
supervisor system call	2.2
user-mode system call	12.3
cross address space	3.4

Chorus also places supervisor actors in the system address space to optimize the performance of invocations across the micro-kernel interface. Chorus assumes that micro-kernel entry and exit costs from user space are high relative to the cost of the supervisor implementation of the micro-kernel interface. Our figures for the PA-RISC show that micro-kernel calls from supervisor actors are about one fifth the cost of micro-kernel calls from user actors.

[7] The 9000/834 provides a timer with a resolution of 1/15 of a μ second.

Table 2: A breakdown of user-mode system call costs

Stage	Time (in μ sec)
kernel entry	1.6
kernel processing	10.5
kernel exit	0.2

A further breakdown of the component costs involved in a user-mode system call is presented in table 2. The cost of promoting privilege level via the gateway page, followed by a branch into the system address space is illustrated in the kernel entry figure (1.6 μ seconds). The kernel exit figure (0.2 μ seconds) illustrates the cost of privilege reduction using the delayed branch instruction. The remaining time (10.5 μ seconds) is taken in kernel processing which includes saving and restoring registers[8], switching stacks, and making the system call.

The difference in cost between a user-mode system call and a cross address space call is due to the cost of changing privilege level and protection domain [3]. Since the cost of changing privilege level using the gateway page is known to be around 1.6 μ seconds, we can deduce that the remainder of the cost is associated with the protection domain transfer. Further investigation of this costs shows the dominant cost in protection domain transfer to be associated with switching execution stacks. Consequently, the difference in cost between supervisor and user calls across the micro-kernel interface is due mainly to the use of a protected call rather than the use of different privilege levels or address spaces.

Chorus also assumes that the execution of privileged instructions requires a supervisor actor to be in the system address space. Since address space and privilege level are orthogonal issues on the PA-RISC, it is possible to place supervisor actors outside the system address space and still allow them to execute privileged instructions. Such actors need not execute continually in privileged mode since the gateway mechanism allows changes in privilege level that are considerably more efficient than trap-based mechanisms. On the PA-RISC the cost of increasing privilege level via the gateway page is around twice the cost of a null procedure call (i.e., 1.6 μ seconds) and the cost of lowering privilege is only around 0.2 μ seconds.

A related assumption is that an actor must be in the system address space to handle hardware events efficiently. This is based on the principle that the identification of a virtual address for a handler in another address space requires significant overhead for memory context set-up. The PA-RISC's global address space ensures that every virtual address can be uniquely identified. Furthermore, the cost of an inter-space call (3.4 μ seconds) is not dramatically higher than the cost of a null procedure call (0.9 μ seconds). Thus, supervisor actors need not reside in any specific 32-bit address space in order to handle hardware events.

Consequently, there are a number of feasible options for implementing supervisor actors on the PA-RISC. For example, they could be given their own private 32-bit address

[8]Our measurements of user system call cost do not include the cost of saving and restoring co-processor context.

space, or they could reside in the system address space. If they reside in the system address space they could be given distinct protection and access identifiers, or they could use the same protection domain and privilege level as the kernel. Each of these possible variants would exhibit slightly different characteristics with respect to modularity and performance, and each would require a slightly different implementation of the micro-kernel interface.

The specification of the Chorus portable interface also involves some assumptions that are inappropriate for the PA-RISC. In particular, software maintenance of address translation consistency causes the relative costs of certain primitive memory management operations to alter significantly. For example, cache flushing during an unmapping operation for a 2 K-byte page can increase the cost of the operation by between 150% and 1000% depending on the state of the cache [7]. Changes in the cost of primitive operations supported in the portable interface can, in turn, lead to inappropriate design decisions in higher-level, portable code. A classic example is the use of memory mapping, rather than copying, to implement IPC. On architectures with a physically addressed cache, mapping is a clear win over byte-copying for page-sized, page-aligned data. However, the expense of cache flushing on a virtually addressed cache architecture can decrease the performance of IPC implementations based on mapping to the extent that, under certain circumstances, byte copying implementations can be as fast as, or even faster than, mapping implementations (For an in-depth discussion of the effects of virtually addressed caches on virtual memory design and performance, and for more details of the performance results discussed here, see [7]).

A better long term solution would be to make use of the PA-RISC's global address space and long pointer addressing in order to avoid address aliasing problems. This would require a major redesign of Chorus to remove the notions of private per-process address spaces, and to share memory only via the global address space using unique 64-bit virtual addresses rather than by mapping multiple virtual addresses to the same physical page.

4 Strengths and Weaknesses of the Chorus Approach

One of the major strengths of the Chorus approach is its separation of interface specification from implementation. The interfaces between operating system components are specified in terms of IPC which is supported in the specification of the micro-kernel interface. An interesting feature of Chorus is its use of two distinct implementations of the micro-kernel interface: the supervisor actor interface and the user actor interface. Each implementation makes a different trade-off between modularity and performance. The supervisor actor interface is implemented in a manner that preserves dynamic reconfigurability, but trades runtime protection for performance. User actors, on the other hand, maintain protection at the expense of performance by using a more heavy weight invocation mechanism when crossing the micro-kernel interface.

The provision of these two separate implementations of what is essentially the same interface allows Chorus to support multiple different implementations of the same modular operating system. The main problem, however, is that there are only two choices: a system component must be either a supervisor actor, which uses an efficient but unprotected implementation of the micro-kernel interface, or a user actor which uses a protected but relatively slow implementation of the interface. For some application domains neither of these choices may be appropriate. Similarly, the loss of modularity in exchange for performance, which is implicit in these choices, may be unnecessary for some architectures, particularly if the

architectural assumptions on which the choices are based are inappropriate.

It would be useful to generalize the approach taken in Chorus by allowing arbitrarily many implementations of a single interface specification. This would allow operating systems to be customized for different architectures and application domains. In order to support this level of customization it must be possible to localize the code representing a particular implementation of an interface. Above the interface this code is generally localized in a system call library. For example, Chorus provides two separate micro-kernel interface libraries, one for linking with supervisor actors and the other for user actors. Below the interface, the code associated with a particular implementation must also be localized if customization is to be supported efficiently. In Chorus such code is localized in system call vector modules: one for handling calls originating in the supervisor actor implementation of the micro-kernel interface, and the other for user actor calls.

Another strength of Chorus is its separation of portable and machine-dependent code. Chorus defines a portable interface, internal to the micro-kernel, that is implemented using machine-dependent code. However, this interface is so close to the underlying hardware that it is difficult to avoid building architectural assumptions into the interface specification. This can have the unfortunate effect of fostering inappropriate assumptions in higher-level code that uses the portable interface. Our experience with Chorus on the PA-RISC indicates that even though we are able to support a semantically correct version of the portable interface, the relative performance of some of the operations within it put into question various design decisions made in the portable layers.

5 Conclusion

This paper has discussed the relationship between modularity and interface design in micro-kernel operating systems. Modularity is a major potential strength of micro-kernel based systems. However, in order to gain wider acceptance, micro-kernels must exhibit performance comparable to monolithic operating systems. An important principle in achieving this goal is the separation of interface specification from implementation. Interface implementation decisions are then largely concerned with establishing an appropriate balance between modularity and performance. Since this balance varies for different application domains and computer architectures, the most promising approach for micro-kernel designers is to offer customization by allowing multiple different implementations of the same interface specification. Chorus has begun to apply this principle by offering two different implementations of its micro-kernel interface. We believe that this approach should be generalized.

This paper has also outlined some of the architectural assumptions that are implicit in the design and implementation of operating system interfaces. Some of these assumptions, particularly those relating to cache design, are becoming out-dated given recent trends in computer architecture, and can lead to a relative decline in operating system performance.

6 Acknowledgements

Many people participated in the port of Chorus to the PA-RISC. We are particularly grateful to Marion Hakanson of OGI; Bart Sears of Hewlett-Packard Laboratories; Pascal Dietrich and Philippe Voisin of the University of Nancy; Vadim Abrossimov, Jean-Jacques Germond, Frédéric Herrmann, Olivier Giffard, and Marc Rozier of Chorus Systèmes.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, Atlanta, Georgia, 1986.

[2] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, October 5-6 1989.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, December 3-6 1989.

[4] Michel Gien. Micro-Kernel Design. *UNIX REVIEW*, 8(11):58–63, November 1990.

[5] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1986 Summer USENIX Conference*, pages 87–95, Anaheim, California, 1990.

[6] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Virtual Memory Manager. Technical Report CSE-92-005, Oregon Graduate Institute, January 1992.

[7] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. Technical Report CSE-92-010, Oregon Graduate Institute, March 1992.

[8] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting the Chorus Supervisor and Related Low-Level Functions to the PA-RISC. Technical Report CSE-92-006, Oregon Graduate Institute, January 1992.

[9] Ruby B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[10] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrman, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Chorus Distributed Operating Systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[11] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Project Overview. Technical Report CSE-92-003, Oregon Graduate Institute, January 1992.

[12] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Overall Evaluation. Technical Report CSE-92-008, Oregon Graduate Institute, January 1992.