Porting Chorus to the PA-RISC: Overall Evaluation

Jonathan Walpole,
Marion Hakanson,
Jon Inouye,
Ravindranath Konuru

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

January 1992

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System kernel to the Hewlett-Packard 9000 Series 800 workstation. This document summarizes the matches and mis-matches between Chorus and the PA-RISC and outlines the general lessons learned during the project.

This document is intended for people who are interested in (a) the separation of machine-dependent micro-kernel code from machine-independent micro-kernel code, (b) the interaction between operating system design and the PA-RISC architecture, and (c) the portability of the Chorus operating system.

The first report in the series, *Porting Chorus to the PA-RISC: Project Overview*, is a prerequisite for reading this document. This document also frequently defers discussion of specific details to the other reports in this series:

- *Porting Chorus to the PA-RISC: Booting*
- *Porting Chorus to the PA-RISC: Virtual Memory Manager*
- *Porting the Chorus Supervisor and Related Low-Level Functions to the PA-RISC*
- *Porting Chorus to the PA-RISC: Building, Debugging, Testing and Validation*

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

1 Introduction

This document is part of a series of reports describing the rationale, goals, technical and organizational issues that arose in porting the Chorus operating system kernel to the Hewlett-Packard 9000/834 workstation which uses the Precision Architecture RISC (PA-RISC) 1.0 processor. This document summarizes the matches and mis-matches between the Chorus kernel design and the PA-RISC architecture. A more detailed discussion of the design decisions made in the project can be found in the other reports in this series [6, 1, 2, 5, 4].

The following two sections discuss the areas in which Chorus and the PA-RISC exhibit particularly good, and particularly bad, interactions respectively. Section 4 identifies some neutral areas in which there was neither good nor bad interaction, but in which the PA-RISC provided some interesting opportunities that we failed to take advantage of in PA-Chorus (the name of our port). In light of our experience porting Chorus to the PA-RISC, section 5 suggests some modifications to enhance the portability of Chorus. Section 6 briefly outlines some enhancements that could be made to our current implementation of Chorus in order to make better use of the facilities offered by the PA-RISC. Finally, section 7 concludes the report.

2 Successful Interactions

This section discusses the specific aspects of Chorus and the PA-RISC that we liked and disliked, and outlines the areas in which there is a good match between the Chorus kernel design and the PA-RISC architecture.

2.1 General Comments on Chorus

Most of our experiences with Chorus were positive. The quality of the Chorus source code is high. It is well organized, well commented and well documented. Chorus' use of C++ and its object-oriented design greatly enhance the understandability of the system and its source code. Modularity is a key aspect of Chorus, and the clear specification of the interfaces that must be implemented in machine-dependent code greatly simplifies the task of porting Chorus. We like the explicit separation of portable code from non-portable code, and even though most of the remainder of this document focuses on specific mismatches between Chorus and the PA-RISC, we found this separation and the interfaces that encapsulate it to be fairly mature.

Chorus has concentrated on developing a Unix-compatible operating system which is both modular and efficient. For efficiency reasons, on other architectures, Chorus has moved some of the servers (the supervisor actors) of its Unix sub-system into the kernel address space. At first glance, this seems to conflict with the design-philosophy of micro-kernels. However, it is important to distinguish between modularity as a design principle, and the *implementation* of modularity in a system. Chorus ensures that the interfaces presented by sub-system actors and the kernel are clearly defined, but does not require sub-system actors and the kernel to be implemented in separate address spaces. We believe that the use of well-defined interfaces is a more important aspect of modularity than the use of separate address spaces. This approach is consistent with architectures, such as the PA-RISC, that advocate the use of a single globally shared address space. The use of such a shared address space does not preclude the benefits of modularity, of course. What is important is for an operating system to clearly identify the module boundaries and their associated interfaces such that the protection and invocation features of the architecture can be used in an appropriate implementation. Our experience indicates that Chorus achieves this.

The modular organization of Chorus is such that it is possible to easily take advantage of the PA-RISC's support for modularity. The extent to which we have done so in our first implementation is fairly limited. However, in section 6 we outline what we consider to be fairly straight-forward modifications to our initial implementation to demonstrate considerably more synergy between the Chorus design and the PA-RISC.

2.2 General Comments on the PA-RISC

Several features of the PA-RISC significantly simplified the task of getting early versions of PA-Chorus up and running. The following features were particularly useful:

- **IODC and PDC:** The I/O-dependent code (IODC) and processor-dependent code (PDC) provided by the PA-RISC raise the level of abstraction at which the operating system writer must deal with the hardware. This greatly simplifies the early phases of porting an operating system to the PA-RISC (see [1] for details).
- **Precise interrupts:** Again, this raises the level of abstraction at which the operating system writer must deal with the hardware. This feature was of major benefit when writing interrupt handling code in the Chorus supervisor (see [5] for details).
- **External device interface:** The interface to external devices via the ITMR, EIRR and EIEM registers is high-level and elegant. The provision of a higher-level, more abstract, interface for masking, enabling, and fielding interrupts is another operating system-friendly feature of the PA-RISC (see [5] for details).

We had mixed experiences with the PA-RISC's memory hierarchy and virtual memory support. Conceptually, we like the PA-RISC's memory management design. However, there are several aspects that conflict with the design of the Chorus virtual memory system. These mismatches, which are discussed in section 3, caused several headaches during the project. The positive aspects of the PA-RISC's memory management design are as follows:

- **TLB-miss handling in software:** The PA-RISC architecture allows for either hardware or software-based TLB-miss handling. The workstation used in the port (HP 9000/834) has software TLB-miss handling. Software refill of the TLB allows significant freedom in page table organization. This is an important feature when the interaction between the operating system's virtual memory system and the memory organization of the architecture is not well understood, as is the case with operating systems such as Chorus and Mach on machines, such as the PA-RISC, with an architecturally visible virtually addressed cache (see [2] for details.

- **Global virtual address space:** We like the PA-RISC's separation of the concepts of protection domain, address space and privilege level. These features provide useful support for modular systems such as Chorus. See sections 5 and 6 for further discussion of these issues.
- **Inverted page table:** The Chorus virtual memory system is well-suited to the use of an inverted page table, primarily because Chorus allows protection for non-resident pages to be maintained at a coarser granularity than pages. The combination of an inverted page table and shared global address space also simplified the implementation of the *mmuContext* class in Chorus. *mmuContext* implements the *context* abstraction in the machine-dependent layers and must normally support per-context (per-process) page tables. Per-context page tables are not necessary on the PA-RISC because its global address space and inverted page table structure are shared by all contexts (see [2] for details).

The *gateway* mechanism is another elegant feature of the PA-RISC. It provides support for efficient changes in privilege level and can be used to implement non-trap-based system calls, hence, avoiding the need to save and restore process state. It was easy to make use of this feature to implement system calls in PA-Chorus (see [5] for details). In fact, we could have made more use of this feature (see sections 5 and 6 for more details).

Overall, we like the PA-RISC. Many aspects of the architecture have been designed with the operating system in mind. The interfaces to the operating system are well defined, fairly high-level, and well documented.

3 Mismatches

This section discusses areas in which there were incompatibilities between Chorus and the PA-RISC. These often resulted from inappropriate assumptions made in Chorus' machine independent interfaces. Some of the mis-matches are relatively superficial, whereas others are the result of fundamental architectural assumptions. Specific mismatches occurred in the areas of thread implementation, synchronization primitives, virtual memory design, and booting. Each of these areas is discussed in the following subsections.

3.1 Thread Implementation

There were four specific aspects of Chorus' thread implementation that conflicted with the PA-RISC. Each of these areas are discussed in more detail in [5].

- **Stack growth direction:** Chorus assumes that stacks grow towards lower addresses. This assumption manifests itself in the portable layers of Chorus which assume that the stack bottom is the highest address in the stack area. On the PA-RISC stacks grow towards higher addresses and hence, the stack bottom is the lowest address in the stack area.
- **Hiding stack growth direction:** Chorus' *threadCreate* call takes the stack bottom as a parameter. This parameter allows users to specify the address of *one end* of the memory

region used for the thread's stack. This address is interpreted as the stack bottom by Chorus. In order to correctly allocate memory for thread stacks the caller must know the stack growth direction of the architecture (its not clear whether this is a bug or a feature). It would be possible to hide the stack growth direction by replacing this parameter with two parameters: one to specify the lowest address in the stack region, and the other to specify the size of the region to be allocated for the stack. This would allow the operating system to determine which end of the region to use as stack bottom according to the stack growth direction of the architecture.

- **Boot actor stack initialization:** During the initialization of stacks for user-level actors installed at boot time, Chorus' portable layers push a frame onto the stack by subtracting a fixed amount from the address of the stack. There are two problems here: (a) it assumes that stacks grow towards lower addresses (not the case on the PA-RISC), and (b) the size of the stack frame, which is machine dependent, is hard-coded in the portable layers. This functionality should be moved into Chorus' machine-dependent layers by passing the stack address directly to the thread constructor and performing the frame adjustment in *SupCtx-Init()*. Note, that this problem does not occur for regular stack initialization (its limited to boot actors), and is more of an oversight than a fundamental design flaw.
- **System stack allocation:** During stack allocation, Chorus lazily allocates stack regions (using the *regionCreate()* system call) by mapping the region copy-on-reference. Since traps are handled on the system stack of the running thread, it is essential to ensure that system stacks are actually allocated initially (i.e., lazy allocation could result in recursive tra handling). Chorus ensures system stack allocation by referencing the page(s) of the system stack. However, the portable layers of Chorus assume that the size of the system stack is a single logical page. In our implementation of Chorus on the PA-RISC the system stack is four pages. System stack size should be exported as a constant.

3.2 Synchronization Primitives

Chorus makes two assumptions in its implementation of mutexes that conflict with the PA-RISC's *load and clear word* instructions.

- **Mutex alignment:** Chorus allows mutex variables (structures) to be declared, with no alignment restrictions, in user address spaces. These mutexes are then manipulated using kernel-exported functions. The PA-RISC's *load and clear word* instructions, *ldcwx* and *ldcws*, which are used in the underlying implementation of mutex operations, require their target address to be 16-byte aligned. These architectural constraints require a redefinition of the mutex data structure in Chorus to ensure that the *lock* field of the mutex structure is correctly aligned. In PA-Chorus we ensure 16-byte alignment by replacing the integer *lock* field in the Chorus mutex structure with a four element integer array.
- **Mutex values:** The portable layers of Chorus assume that a value of 1 in the *lock* field of the mutex structure indicates that the lock is held, and that a value of 0 indicates that the lock is free. This conflicts with the use of the PA-RISC's *load and clear word* instructions which atomically clear the target word (set it to 0) and load the previous value into a register. In this case, a value of zero indicates that the lock is already held.

3.3 Virtual Memory Design

The PA-RISC's architecturally visible virtually addressed cache resulted in several mismatches with Chorus' virtual memory system. The PA-RISC requires software to maintain cache consistency in the presence of address aliases. Address aliases can occur when more than one virtual address is mapped to the same physical address (we call this *virtual-virtual aliasing*) and when physical addressing is used to access a memory location that is non-equivalently mapped to a virtual address (we call this *virtual-physical aliasing*).

Chorus is built around the assumption of a separate address space per process, and makes widespread use of copy-on-write and copy-on-reference memory mapping techniques. These techniques result in virtual-virtual aliases which, on the PA-RISC, must be managed by operating system software. The specific aspects of Chorus that result in virtual-virtual aliases are:

- **The global map:** The Chorus virtual memory system supports the concept of a global map. When the virtual memory system must perform an operation on a physical page it creates a mapping for the page in its own global map and uses the resulting virtual address to access the page. This technique means that Chorus does not need to run in physical mode. However, it is used regardless of whether the page is already mapped. If the page is already mapped then virtual-virtual aliasing results.
- **Region mapping:** when the same segment is mapped read-only (to support shared text for example) or read-write by multiple regions, multiple virtual addresses map directly to common physical addresses (i.e., each region's virtual pages map to physical pages of the segment cache). This results in virtual-virtual aliasing.
- **Mapping via history objects:** When a segment is mapped copy-on-write or copy-on-reference into more than one region, Chorus uses *history objects* to maintain temporary protection and sharing information for the shared pages. This type of region mapping results in virtual-virtual aliasing.

The specific source of these virtual-virtual aliasing problems can be traced to the following methods of the *mmuPage* class in Chorus' machine-dependent layers:

- *getAddr*: This method is used to generate a mapping (virtual address) to a physical page. Following the call, the virtual address can be used to access the page. Virtual-virtual aliasing occurs if a mapping already exists for the page.
- *map*: This method is used to load pages into *contexts*. Virtual-virtual aliasing occurs if the page is loaded into more than one context.

The Chorus portable layers do not generate virtual-physical aliasing. However, virtual-physical aliasing does occur in the machine-dependent layers. The source of Chorus' virtual-physical aliasing problems can be traced to the following methods in the *mmuPage* class in the machine dependent layers:

- *mmuPage - third constructor*: This constructor initializes a target page (this page) from a source page. If the target page is not yet mapped then physical addresses are used to perform the initialization. This results in virtual-physical aliasing. For simplicity, in our initial implementation we also use physical addresses when the page is mapped. This is not a requirement.
- *getPhysAddr*: This method returns the physical address of the page. The use of that physical address on non-equivalently mapped pages results in the generation of virtual-physical aliases.

In our implementation of Chorus on the PA-RISC we solved the virtual-virtual aliasing and virtual-physical aliasing problems using a technique called *pseudo-aliasing*. Pseudo-aliasing involves flushing the appropriate address range from the cache whenever aliases can be generated. In other words, the machine dependent layers only allow a single address translation to exist for each physical page at any point in time. Accesses to a page via a different virtual address result in a *pseudo page-fault*. In handling this fault the machine-dependent virtual memory layers remove the existing translation, flush the appropriate address range from the cache, and establish the new translation. This enables the portable layers of the virtual memory system to work under the assumption that aliasing is allowed by the architecture.

While pseudo-aliasing allows a *correct* implementation of Chorus on the PA-RISC, it is potentially very inefficient. It is possible to reduce the number of cache flushes by making judicious changes to the virtual memory design. For example, by recognizing when pages are being shared read-only, or by performing operations on physical pages using virtual addresses rather than physical addresses if the mapping of the target page has been determined by the time the method is called. However, the real performance problems result from some basic architectural assumptions made by Chorus regarding the suitability of per-process address spaces, the efficiency of memory mapping, and the efficiency of using both virtual and physical addressing to access pages. We are in the process of making a quantitative evaluation of the performance of pseudo-aliasing and related techniques. The results of this study will be presented in [3].

The "right" way to use the PA-RISC's virtual memory system is to make use of the shared global virtual address space. We did not attempt to do this in our initial implementation because the modifications required to Chorus' virtual memory system appeared to be potentially large. In retrospect, this was probably the correct decision. While Chorus' basic virtual memory abstractions, *contexts*, *regions*, *pages* and *segments*, seem to be suitable for supporting a globally shared address space, there are some specific assumptions built in to the Chorus source code that complicate such an implementation. In particular, Chorus stores some pointers as 32-bit *unsigned long* values and generally assumes that it is possible to pass *void** to procedures. These assumptions limit a context to a 32-bit address space. A better implementation approach would be to hide these architectural dependencies by making more use of the abstract data types supported by C++.

The use of an inverted page table was, for the most part, a success. The only mismatch we noticed concerned Chorus' implementation of context deletion (the deletion of an address space). Context deletion in Chorus on the PA-RISC is expensive because it requires unloading all physical pages from a specified virtual address range. This is expensive with an inverted page table because inverted page tables are optimized for accesses using physical address ranges rather than virtual address ranges.

Another memory management-related problem concerned the inability of Chorus to correctly support the semantics of the PA-RISC's *probe* instruction. The semantics of the *probe* instruction are to return true when the specified memory access is valid, and false when it is not valid. Operating systems, such as Chorus, that temporarily change memory protections to support techniques such as copy-on-write and copy-on-reference memory, can not support these semantics. When the TLB protections for a specified address are obtained, there is no way of knowing whether the observed protections are temporary protections set up by the virtual memory system to cause copy-on-write or copy-on-reference faults. Consequently, under Chorus the *probe* instruction returns true when the specified access is valid, and false when it *may not be* valid.

Finally, non-access TLB miss faults, such as the *probe* instruction, should not result in page-faults that cause data to be transferred. Non-access faults generally require meta-information about pages, such as protection information, and do not require the actual data of the page to be brought into memory. To support this functionality, the Chorus portable layers need to be able to distinguish between regular faults and *non-access* faults. This is not currently supported in Chorus.

3.4 Booting

The organization of Chorus' boot procedure seems to assume a more static I/O configuration characteristic of small personal computers rather than larger machines with flexible and extensible I/O systems. Chorus encourages the placement of device initialization code in a device manager that runs after kernel (specifically virtual memory) initialization. However, on architectures, such as the PA-RISC, which support automatic configuration, support for device initialization is more appropriately placed in the boot code, before virtual memory initialization. This is mainly due to the fact that the procedure to determine which devices are present, and the initialization of those devices, requires the machine to be in physical addressing mode. If this is work is performed in the early phases of kernel initialization the I/O address space can be configured during virtual memory initialization.

4 Missed Opportunities

There were several features of the PA-RISC that we considered to be very good matches with Chorus, but which we failed to take advantage of in our initial implementation. This was usually due to our quest for simplicity in trying to get something (anything!) up and running as early as possible. In particular, we failed to take advantage of the true power of the PA-RISC's virtual memory and protection facilities, and we didn't make as much use of the gateway page as we could have for implementing the system call interface for supervisor actors. Each of these areas is discussed in the following subsections.

4.1 Modularity

Chorus is designed in a modular manner, with well-defined interfaces between the kernel and the various sub-system actors that comprize the MiX sub-system. In mapping this modular design onto an efficient implementation, Chorus makes several architecture-related assumptions regarding the efficiency of various different implementation approaches. For example, for reasons of efficiency, existing implementations of Chorus place sub-system actors, such as the process manager and object manager, in the same address space and protection domain as the kernel, and run them at the kernel privilege level. Such actors are called supervisor actors. This implementation approach makes the tacit assumption that address space, protection domain, and privilege level are not orthogonal issues. It further assumes that cross address space invocations, changes in privilege level, and cross protection domain invocations are prohibitively expensive. These assumptions are not appropriate for the PA-RISC.

On the PA-RISC it is feasible to place supervisor actors in separate protection domains, with separate space identifiers, and running at different privilege levels. Unfortunately, we didn't have time to experiment with any of these ideas, and hence we view this as a missed opportunity. The following subsections elaborate on the implementation assumptions mentioned above.

4.1.1 Using The PA-RISC's Global Address Space

Chorus is based on the concept of a per-process (per-actor) address space and specifies that there is a unique system address space per site. All supervisor actors share the system address space and share a single global variable *KernelContext* that identifies their context. Each user actor has its own private context (address space) and shares the kernel context, i.e., its threads execute within the kernel address space following a system call.

There are several reasons for Chorus' placement of supervisor actors and the kernel in the same address space. First, it allows supervisor actors to connect uniquely identifiable handlers for hardware traps and interrupts by simply registering the address (within this address space) of the handler. Second, for the implementation of sub-systems with trap-based interfaces it allows supervisor actors that are part of such sub-system to execute in the virtual address space (context) of the user actor causing a trap (or exception). Third, it allows efficient invocation between supervisor actors, and between supervisor actors and the kernel, using Chorus' featherweight RPC which is based on an intra-address space procedure call.

On architectures with a shared global address space, such as the PA-RISC, the above arguments are not appropriate. On the PA-RISC a supervisor actor could reside anywhere in the shared global address space, yet still connect a handler for a hardware interrupt or a trap, without any problem, by specifying the global address for the handler. The global address space also means that all supervisor actors and user actors conceptually execute in the same address space. Finally, the PA-RISC's procedure calling facilities allow for efficient invocation between actors regardless of their location in the global address space.

There are two alternative approaches to making use of the PA-RISC's global address space. One requires the use of 64 bit pointers, and the other does not. The degree of modification required to

Chorus varies for each approach. The use of 64 bit pointers requires fairly extensive modifications to Chorus. These extensions are discussed in section 6.

In order to make use of the gobal address space without using 64 bit pointers the interface for connecting handlers would need to be modified to allow the space identifier of the actor, as well as the address of the routine, to be passed as a parameter to the system call connecting the handler. Several other system calls would also need to be modified to accept a space identifier as an additional parameter. Furthermore, the *btChorus* utility that generates the boot archive would require modification to manipulate the space identifiers of the boot actors and store them as part of the *rootStructure*. See section 6 for further discussion of this approach.

In order to avoid these modifications in our initial implementation of Chorus on the PA-RISC we used a single space identifier for the kernel context, i.e., the kernel and all supervisor actors shared a single context and were accessed using the same space identifier. Our initial assumtion was that this would allow us to base invocation between supervisor actors on simple procedure calls, and that the identification and execution of connected handlers would be simple. However, since each of the supervisor actors is a separately executable image, each one has its own *global* to which the data pointer must be initialized before any code that accesses data can be executed (this is a requirement of the PA-RISC's procedure calling conventions). Therefore, invocations between supervisor actors and the kernel, and between the supervisor actors themselves, require manipulation of the data pointer and hence, can not take place via a simple procedure call. Furthermore, during interrupt handling, the correct data pointer for the actor that connected the handler must be set up before the handler can be executed. Had we recognized these problems earlier we would have started out by "simply" implementing separate space identifiers for each supervisor actor. This would have required a modification of the system call interface for supervisor actors to pass the space identifier of the actor as an additional parameter, which would affect the system call routines in the kernel portable layers. It would also require recoding of the kernel event handling layers to make use of the PA-RISC's external procedure call.

4.1.2 Protection

In placing supervisor actors in the same address space as the kernel, Chorus associates a high-level of trust with such actors. In other words, the kernel is not protected from supervisor actors, and there is no protection between supervisor actors. In our initial implementation we adhered to this philosophy and placed the kernel and supervisor actors in the same protection domain and ran them at the same privilege level. However, this level of trust is not strictly necessary on the PA-RISC.

An alternative approach would be to allow supervisor actors to share the same kernel context but place them in different protection domains by using different sets of protection identifiers for each supervisor actor. We believe that this would not require major modification to Chorus' portable layers. However, it would require modifications to the machine dependent virtual memory layer to manage these protection domains. For other alternative approaches see section 6.

4.1.3 Privilege Levels

The PA-RISC provides two interesting features for running supervisor actors at different privilege levels. First, the PA-RISC supports four different privilege levels, making it possible to run user actors, the kernel, and supervisor actors at distinct privilege levels. Second, the PA-RISC's *gateway* mechanism allows efficient transfers between privilege levels without incurring the overhead of a trap-based transfer.

Running supervisor actors at a different privilege level to the kernel would require Chorus' supervisor system call interface to be extended to make use of the *gateway* mechanism. Specifically, one *gateway page* could be used for each of the privilege levels 0, 1, and 2. This would protect the kernel from supervisor actors without a significant degradation in performance (we have not performed a quantitative evaluation of the performance effects of such an approach). In addition to the above, access to the *RootStructure* by supervisor actors could be implemented using gateway-based system calls to ensure controlled access and robustness.

5 Enhancing the Portability of Chorus

This section summarizes the areas in which the portability of Chorus could be enhanced. Many of the issues presented below have been discussed in more detail in earlier sections or in other reports in this series.

Summary of changes:

- Thread Implementation:
  - To account for different stack growth directions, the threadCreate call should take parameters to indicate stack size and location. This would allow either end of the stack to be located and used as stack bottom.
  - More aspects of stack initialization, such as frame-size and the requirement to push a frame on initialization, should be submerged in the machine-dependent layers.
- Synchronization: To avoid the architecture-specific dependencies of mutex implementation, a machine dependent mutex header file should be used.
- Coding Conventions: Since Chorus uses C++, more use could be made of the abstractions object-oriented programming allows. For example, Chorus should abstract out its use of *long* and *void**. This may require extensive use of typecasts in the standard Compaq release of Chorus, but it would allow those typecasts to be turned into machine dependent functions for other architectures.

6 How to do it Better

Much of the time we have spent, to date, porting Chorus to the PA-RISC has been focussed on *just* getting things working. Every time we were faced with a major design decision, we took the approach that we thought would minimize development time and allow us to reuse as much existing code as possible (from Tut, HP-UX, or Chorus). This often led us to avoid the more elegant solutions made possible by Chorus' modularity and the advanced protection, privilege, addressing and invocation features of the PA-RISC. This section presents a brief outline of how we would architect a future implementation of Chorus on the PA-RISC in order to take advantage of these matches, and discusses a number of performance optimizations that would be made possible by revising Chorus' separation between machine-dependent and portable code.

The first general design approach, involving relatively straightforward extensions to our current implementation, would involve assigning separate space and protection identifiers to supervisor actors. The Chorus supervisor actor system call interface would be modified to use the gateway mechanism, and where possible, supervisor actors would run at an intermediate privilege level between that of the kernel and user actors. The Chorus root structure would also be encapsulated (not read or written directly) and moved to a separate protection domain. We believe that each of these modifications would be relatively easy to make because of the well-defined interfaces in Chorus.

An alternative and more appropriate long-term approach would involve extending Chorus to use a shared global address space for the kernel, user and supervisor actors. This would help to avoid the virtual-virtual aliasing problems caused by the PA-RISC's virtually addressed cache, however, it would require significant modification to Chorus. Ideally, such an implementation would make use of 64 bit pointers. The virtual memory system would be redesigned to share memory only using unique addresses in the global address space, i.e., memory sharing would not be implemented by mapping multiple virtual addresses to the same physical address. The global map concept in the virtual memory system would also require modification to ensure that mapped pages are accessed using their current virtual address rather than remapping them into the global map.

In addition to the general design changes mentioned above, several performance optimizations would be made possible by revising the separation of functionality between the portable and machine-dependent layers of the Chorus virtual memory system. Chorus supports the abstractions of *segment, context, region,* and *page* in its portable layers, but only supports the abstractions of *page* and *context* in its machine-dependent layer. There are cases in which it would be useful to implement the region and segment abstractions in the machine dependent layers.

For example, if protection modifications in the machine-dependent layer could be performed on regions as well as pages it would be possible to optimize support for certain region mapping operations which involve modifying the protections for groups of pages. High-level operations, such as fork, cause entire regions to be mapped copy-on-write. A common characteristic of these operations is that they result in large regions (containing many pages) being shared by a small number of threads. Chorus currently requires the machine-dependent layers to reset the protections, page at a time, for every page in the region.

The PA-RISC provides hardware support for an interesting alternative to the approach mentioned above. It allows read-only protections to be set on groups of pages, with the same access identifier, by setting the *protection identifier write-disable bit* for all threads whose protection identifier matches that access identifier. This approach requires the operating system to maintain enough information to be able to find all the threads that can access a particular region, and hence would require modifications to Chorus' portable code. The performance effects of this optimization depend on the trade-off between the cost of (a) maintaining this thread information and performing per-thread protection identifier modifications, and (b) the cost of performing per-page protection modifications for the entire region.

In addition to the above, knowledge of the *segment* abstraction in the machine-dependent layers could also be used to improve the performance of Chorus' virtual memory system on the PA-RISC. The approach Chorus currently uses to share text is to map multiple regions onto the same segment. Since the machine-dependent layers do not know about segments, the portable layers must specify the mappings to be created on a per-page basis. This results in virtual-virtual aliasing which could be removed by the machine-dependent layers if they could choose an appropriate space identifier in setting up the mapping (i.e., the machine-dependent layers could ensure that text was always shared using the same space identifier). This would require the machine-dependent layers to be informed about (or be able to inquire about) higher-level segment sharing.

7 Conclusion

The modular design of Chorus gives it the potential to exercise many of the advanced features of the PA-RISC. This document has summarized specific matches and mismatches between Chorus and the PA-RISC, and has outlined the key opportunities that we missed in our initial implementation. While we believe that Chorus and the PA-RISC are fairly well matched, there are a few fundamental assumptions implicit in the Chorus design that conflict with the PA-RISC. These assumptions are mostly concerned with the virtually addressed cache architecture and the use of a 64 bit address space, and are generally not restricted to Chorus alone (i.e., other operating systems, such as Mach, make similar assumptions). Since these architectural features are characteristic of the latest generation of high performance processors we believe that it is time for operating system designers to reconsider some of the basic architectural assumptions on which their design and implementation decisions are based.

8 Acknowledgements

Several people contributed their time and energy to help us complete this project. We are particularly grateful to Jean-Jacqués Germond, Frédéric Herrmann, Vadim Abrossimov and Olivier Giffard of Chorus Systèmes, and Bart Sears, Jean Gascon, Ahmed Ezzat, Howard Stateman, and Charlie Roberts of Hewlett-Packard Company.

References

[1] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Booting. Technical Report CSE-92-4, Oregon Graduate Institute, 1992.

[2] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Virtual Memory Manager. Technical Report CSE-92-5, Oregon Graduate Institute, January 1992.

[3] Jon Inouye, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Improving Memory Management Performance. In preparation.

[4] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting Chorus to the PA-RISC: Building, Debugging, Testing and Validation. Technical Report CSE-92-7, Oregon Graduate Institute, January 1992.

[5] 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-92-6, Oregon Graduate Institute, January 1992.

[6] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Project Overview. Technical Report CSE-92-3, Oregon Graduate Institute, 1992.