Porting Chorus to the PA-RISC: Virtual Memory Manager

Jon Inouye, Marion Hakanson,
Ravindranath Konuru and Jonathan Walpole

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-005

January 1992

Porting Chorus to the PA-RISC: Virtual Memory Manager

Jon Inouye
Marion Hakanson
Ravindranath Konuru
Jonathan Walpole

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

January 1992

This document describes the port of the Chorus virtual memory manager to the Hewlett-Packard Precision Architecture RISC (PA-RISC) workstation. The information contained in this paper will be of interest to people who:

- intend to port the Chorus virtual memory section.
- intend to port a virtual memory design to the Hewlett-Packard PA-RISC.

The reader is strongly encouraged to read the following PA-Chorus documents before reading this document:

- Technical Report CSE-92-3, *Porting Chorus to the PA-RISC: Project Overview*

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

1 Introduction

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System to the Hewlett-Packard 9000 Series 800 workstation. This document describes the design and implementation of the virtual memory manager port.

In addition to this report, certain other sections of the virtual memory implementation are described in other PA-Chorus technical reports. The virtual memory initialization routines are described in [8] and the page fault interface is described in [11].

Section 2 gives an overview of the PA-RISC memory management unit (MMU). Section 3 gives a very brief overview of Chorus virtual memory management. Section 4 presents the tasks involved in porting the VM system to the PA-RISC. The machine-dependent interface is shown in section 5. The implementation of the machine-dependent interface is described in section 6. Finally, section 7 evaluates our approach.

2 Overview of the PA-RISC Memory Management Unit

This section describes the memory management unit of the PA-RISC. The material in this section is covered in greater detail in the *Precision Architecture and Instruction Set Reference Manual* [7].

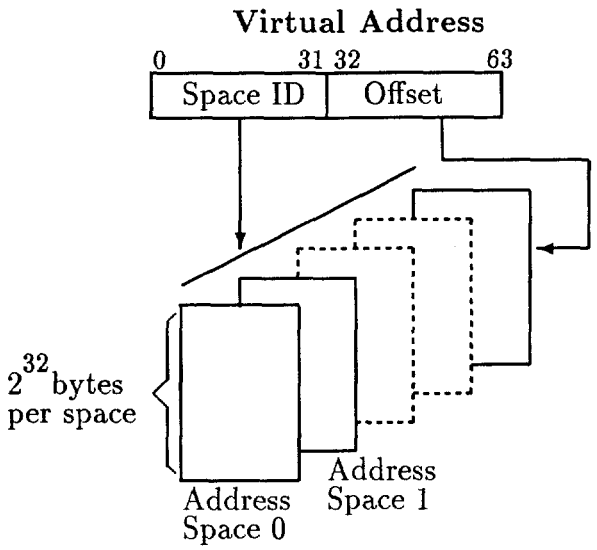

Figure 1: Address Space Partitioning

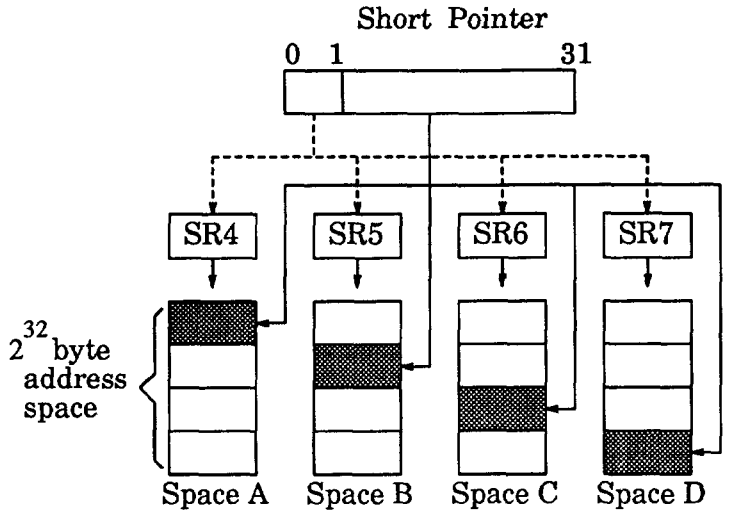

Figure 2: The upper two bits of a short pointer are used to select one of four space registers. This space register selects which 32-bit address space is used. The contents of the short pointer are then used as an offset into the 32-bit address space. Since the upper two bits are already known, the remaining 30 bits are used to address a single quadrant within the address space.

2.1 Address Space

The PA-RISC supports the concept of a large virtual memory space that can be shared by all processes. Virtual memory is partitioned into segments, called *address spaces*, each containing 2^{32} bytes. The number of *address spaces* is dependent on the level of the architecture. Level 1 systems have 2^{16} address spaces, level 1.5 systems support 2^{24} address spaces, and level 2 systems support 2^{32} address spaces. Figure 1 shows how the virtual memory space is partitioned. Each virtual address consists of two components: an address space identifier and a space offset. The space identifier (SID) is used to select an address space and the space offset determines the location within the address space. The space identifier can be stored in any of the PA-RISC's 8 space registers. The offset is stored in a general purpose register. The PA-RISC allows addressing using both long (> 32-bit) pointers and short (32-bit) pointers. Instructions using long pointers specify two 32-bit registers (1 space register and 1 general purpose register) used to generate a 64-bit address. Short pointers are generated by using segments in the following way. The two most significant bits of a 32-bit pointer (stored in a general purpose register) are used to select one of four space registers (SR4–SR7). The contents of the selected space register are concatenated with the short-pointer to form a 64-bit address (see figure 2). Note that when using short pointers, each space register (SR4–SR7) can only address a single quadrant (1 gigabyte) of its address space, i.e., SR4 addresses the first quadrant, SR5 the second, SR6 the third, and SR7 the fourth.

Figure 3: Hash Table Entry

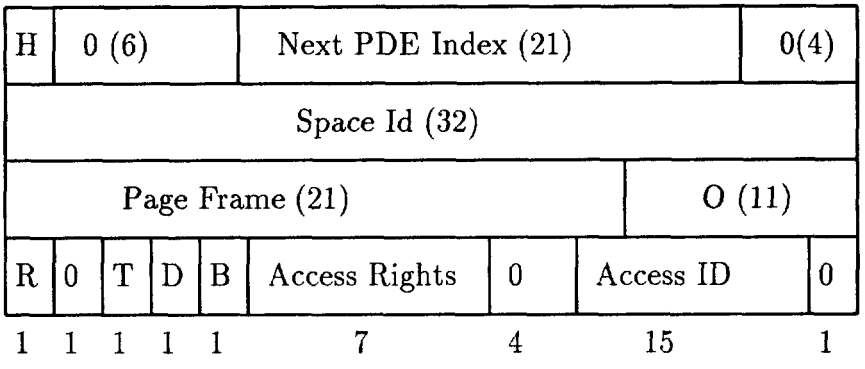

Figure 4: PDIR Entry

2.2 Address Translation

Like other architectures, the PA-RISC uses a translation look-aside buffer (TLB) to support virtual to physical address translation. Unlike other RISC implementations, the PA-RISC requires that TLB misses must be handled by software.[1]

In order to manage a larger virtual address space, the PA-RISC uses an inverted page table structure called the Physical Page Directory (PDIR). Each entry in the PDIR represents a physical page so the number of entries in the PDIR is proportional to the number of physical pages in memory. The advantage of an inverted page table structure over a standard page table structure is that the former grows with the size of physical memory while the later grows with the size of virtual memory.

On TLB misses, software must determine if there is a valid PDIR entry that satisfies the translation. To accomplish this, software must search the PDIR for an entry that maps the virtual page. One disadvantage of an inverted page table structure is that it is not straight forward to locate a page's PDIR entry given its virtual address. The PA-RISC suggests the use of a hashing algorithm. A hash function takes a virtual address as input and returns an index into a hash table. Each hash table entry represents a hash bin organized as a linked list. Software must traverse the list to determine whether the desired translation exists. If no suitable translation is found, a page fault handler is called. Figure 3 shows the format of a hash table entry (HTE) and figure 4 shows a PDIR entry (PDE). The linked list is composed of a HTE followed by a variable number of PDE's. The H-bit is used to determine when the end of the list has been reached. When the H-bit is 0, the

[1]Hewlett-Packard mentions that future implementations of the PA-RISC may provide hardware TLB refill.

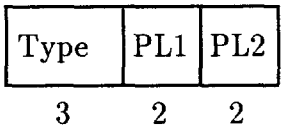

Figure 5: Access Rights Field

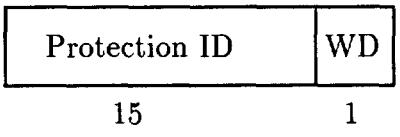

Figure 6: Protection ID

Next PDIR Index points to the next PDE in the list. (Otherwise the PDE is the last in the list.) The *Space Id* and *Page Frame* fields contain the space identifier and space offset (minus page offset) of the virtual page. (The physical page number is the index of the PDIR entry.) The protection information is stored in the *access rights* and *access ID* fields.

2.3 Memory Protection

In addition to supporting address translations, the TLB is also responsible for enforcing memory protection. Because the TLB is only used when the processor is in virtual addressing mode, protection checking is disabled when running in physical addressing mode. The TLB maintains protection information in two fields: the *access rights* and the *access ID*. The 7-bit *access rights* field encodes the allowed access types and privilege levels into three sub-fields: *type*, privilege level 1 (*PL1*), and privilege level 2 (*PL2*). Figure 5 shows the layout of the access rights field. Table 1 describes the interpretation of the field.

Because the PA-RISC uses a globally shared address space, it requires an additional form of protection mechanism that enforces protection between processes running at the same privilege level. This mechanism is the *access ID*. The *access ID* is a 15-bit field that can be thought of as a capability. This field must match one of the four protection ID's (PID) in the PA-RISC's control registers (CR8,CR9,CR12,CR13). An access ID of zero indicates the page is *public*. A public page always passes a protection ID check, i.e., only an access rights check is performed. Note that an access ID is assigned to a PDIR entry while a PID is assigned to a thread. A thread can have more than four protection ID's associated with it, but only four can be cached in the control registers at any point in time. Figure 6 shows the format of the protection ID field. In addition to the 15-bit protection ID, the field contains a *write-disable* (WD) bit. When this bit is set, writes that match the protection ID result in protection faults.[2]

[2]Unless PID checking is disabled by running in physical address mode or disabling the Processor Status Word's P-bit.

Table 1: Access Rights Interpretation

Type value (in binary)	Allowed access types and GATEWAY promotion	Privilege check
000	Read-only: data page	read: PL $\leq$ PL1 write: Not allowed execute: Not allowed
001	Read/Write: dynamic data page	read: PL $\leq$ PL1 write: PL $\leq$ PL2 execute: Not allowed
010	Read/Execute: normal code page	read: PL $\leq$ PL1 write: Not allowed execute: PL2 $\leq$ PL $\leq$ PL1
011	Read/Write/Execute: Dynamic code page	read: PL $\leq$ PL1 write: PL $\leq$ PL2 execute: PL2 $\leq$ PL $\leq$ PL1
100	Execute: promote to privilege level 0	read: Not allowed write: Not allowed execute: PL2 $\leq$ PL $\leq$ PL1
101	Execute: promote to privilege level 1	read: Not allowed write: Not allowed execute: PL2 $\leq$ PL $\leq$ PL1
110	Execute: promote to privilege level 2	read: Not allowed write: Not allowed execute: PL2 $\leq$ PL $\leq$ PL1
111	Execute: remain at privilege level 3	read: Not allowed write: Not allowed execute: PL2 $\leq$ PL $\leq$ PL1

The PDE also contains four bit flags which may be used to support memory management operations. The R-bit (reference) is set when a page has been referenced. The D-bit (dirty) is set when a page could have been modified. The T-bit and B-bit are used to generate traps which can be useful in program debugging. The use of these bit flags is explained in more detail in [11].

2.4 Virtually Addressed Cache

The PA-RISC uses a virtually addressed cache. This type of cache uses part of the virtual address (instead of the physical address) to index the cache directory. This allows the cache to be accessed without waiting for the TLB to generate the physical address. The PA-RISC's cache uses a physical index only when in physical addressing mode, i.e., the cache is still used in physical addressing mode. (This occurs either when address translation is disabled or the processor is executing instructions that load and store physical addresses.)

The major disadvantage of using a virtually addressed cache is dealing with address aliases. When address aliases are present it is possible for multiple copies of the same data to exist within the cache. This is undesirable for reasons of cache consistency. (Smith refers to this as the *synonym* problem [14].) The PA-RISC is rather unusual in that the cache is part of the architecture. Rather than implement alias detection within the cache, the PA-RISC requires that software maintain the consistency of the cache. The following quotation is from PA-RISC reference manual [7]:

> *Caches are not required to detect that the same physical memory location is accessed by different virtual addresses or by both an physical and a virtual address, except for equivalently-mapped addresses. Since this condition, loosely called aliasing, can be caused only by software running at the most privileged level, it is the responsibility of such software to avoid the ambiguities it may create. This requires flushing the affected address range from the caches prior to any of the following:*
>
> 1. *Changing the address mapping in the TLB's.*
> 2. *Making an absolute access to a location which might reside in the caches as a result of an access by a virtual address that was not equivalently-mapped.*
> 3. *Making a virtual access to a location which might reside in the caches as a result of an access by its absolute address that was not equivalently-mapped.*
> 4. *Making a virtual access to a location which might reside in the caches as a result of an access by a different virtual address.*

We separate types of address aliases into two categories:

virtual-virtual This type of aliasing occurs when two different virtual addresses map to the same physical address. Aliases of this type can be generated by performing memory mapping or remapping operations.

virtual-physical This type of aliasing occurs when non-equivalently mapped addresses are referenced using both virtual and physical addresses. The PA-RISC reference manual defines equivalently mapped addresses as those virtual addresses which meet two conditions: 1) The virtual address has a space identifier equal to 0, and 2) the virtual address has a virtual offset equal to the absolute address. These aliases are generated when physical addressing is used to access pages that are also virtually addressed.

Note that aliases become a problem only when the number of bits used for cache index include part of the page number. If the cache is indexed using only the bits representing the page offset, then aliases would fall within the same set in the cache. The PA-RISC can detect aliasing within the same set since it keeps the physical address as part of the tag.[3] By restricting the cache size to the logical page size multiplied by the cache set-associativity, aliases are forced to fall within the same set. Under this restriction virtually addressed caches would remain part of the *implementation* and not part of the *architecture*. Rather than limit cache sizes to fit this restriction the PA-RISC requires software to maintain cache consistency in the presence of address aliases.

[3] Most cache implementations are able to resolve aliases within the same set, but detecting aliases in different sets is a more difficult task.

3 Overview of the Chorus Memory Management Hierarchy

This section gives a very brief description of the Chorus virtual memory manager. The material in this section is covered in greater detail in two Chorus Systèmes technical reports [1, 2].[4]

The Chorus virtual memory management system is organized in a layered manner. Chorus defines a generic memory interface (GMI) which is supposed to separate the kernel dependent memory abstractions from any underlying hardware architecture. The kernel dependent layer above the GMI supports the Chorus virtual memory abstractions for system calls and interprocess communication. Underneath the the GMI lies a general memory manager for some type of memory management unit. For supporting architectures with demand-paged virtual memory management units, Chorus uses a layer called the paged virtual memory manager (PVM). Below the PVM lies the machine-dependent layer (MMU)[5] that is responsible for fully supporting a specific memory management unit.

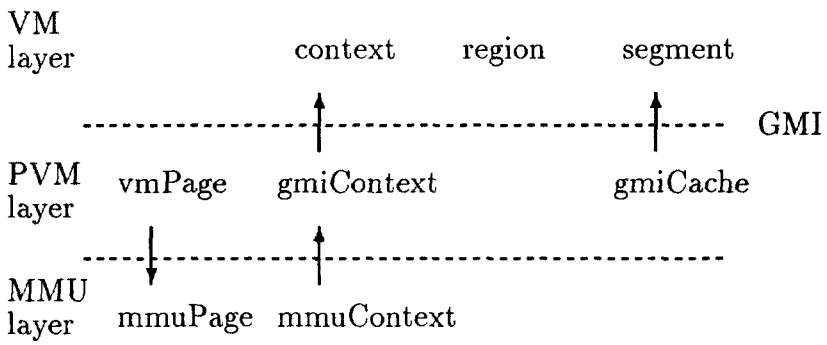

Figure 7: Chorus VM Class Hierarchy

Chorus is primarily implemented in C++, an object-oriented language. The virtual memory abstractions are implemented as a set of C++ classes. There are four major classes: **page**, **region**, **context**, and **object**. A **page** represents an instance of a physical memory page. There should be no more instances of **pages** than physical memory pages in the system. A **context** represents a virtual address space. A **region** represents a valid address range within a context. Regions are mapped to secondary storage objects called **segments**. Segments are managed outside the kernel by servers called **mappers**. Figure 7 gives a general overview of the Chorus v3.3 virtual memory class hierarchy and the layer in which each class resides. The arrows represent the direction of inheritance i.e., base-class → derived-class. Objects prefixed by *mmu* belong to the machine dependent layer (MMU) while objects prefixed by *gmi* tend to belong to the PVM layer. The Chorus kernel abstractions (context, segment, region) are kernel dependent classes. They exist in the virtual memory (VM) layer above the generic memory interface. The machine-dependent layer is responsible for implementing two classes: **mmuPage** and **mmuContext**.

[4]Chorus technical reports are available via anonymous ftp from cse.ogi.edu (129.95.40.2) in directory pub/chorus-reports

[5]The acronym MMU stands for the Memory Management Unit layer.

4 Porting Strategy

The port of the Chorus virtual memory management section to the Hewlett-Packard PA-RISC was performed in a series of phases:

1. Familiarization Phase
2. Task Specification Phase
3. Design Phase
4. Implementation Phase
5. Validation Phase
6. Analysis Phase

The Familiarization Phase dealt with getting to know the PA-RISC architecture and the Chorus virtual memory management section. The Task Specification Phase was responsible for determining which tasks were necessary to port the Chorus VM section to the PA-RISC. This included determining the correct specifications of the machine dependent interface. The design of the machine dependent layer was performed in the Design Phase. The code for the machine dependent layer was written during the Implementation Phase and validated during the Validation Phase. The Analysis Phase was responsible for performance analysis and optimization.

4.1 Familiarization Phase

This phase was used to study the PA-RISC and the Chorus virtual memory manager. There is an ample amount of well-written documentation on the PA-RISC memory management unit. We started out using the Precision Architecture and Instruction Set Reference Manual [7]. In addition to this document, we spent a great deal of time reading the reports out of the Tut project [5]. The Tut project involved porting Mach 2.0 to the PA-RISC and its reports were invaluable in assisting us with the Chorus virtual memory port. The Tut documentation also helped us understand the Tut source code (which was also provided to us).

Documentation on the Chorus virtual memory design was available in the form of two of technical reports [1, 2]. While detailing the design of the Chorus virtual memory management section, these reports did not help in determining the necessary steps involved in porting it. In particular, we had no information on the machine dependent interface for the Chorus MMU layer. In October 1990 we were given the Chorus Kernel v3.2 Implementation Guide [3], but it did not contain any useful information on the machine dependent virtual memory interface. The Chorus Kernel v3.3 Implementation Guide [4] contained more information on the machine dependent virtual memory interface but we didn't receive it until May 1991. We ended up having to determine the specifications of the machine dependent layer from reading the Chorus code of previous ports to other architectures.

For the most part, both the Tut and Chorus kernel code was well documented and very readable. There was a section of the virtual memory code implementing the Chorus PVM layer that could have been better document. Initially, we did have a problem reading the Tut code because of the proliferation of `#ifdef`'s. Once we were able to filter this out, the code was much easier to read.

We also used this phase to become more familiar with the PA-RISC instruction set and its assembly language. We wrote several assembly language programs and experimented with the debugger (adb). Unfortunately, many of the more interesting instructions could only be executed at the highest privilege level. It would have been nice to have some form of simulator or development platform on which we could have executed code at the highest privilege level.

4.2 Task Specification Phase

The Task Specification Phase was used to define tasks necessary to port the Chorus virtual memory manager to the PA-RISC. To help facilitate this, three members of the group visited Chorus Systèmes in late October 1990. Three general tasks were outlined at this stage:

- Virtual memory initialization
- Machine dependent layer
- Page Fault Interface

The virtual memory initialization is the machine dependent section of the Chorus boot program that enables virtual addressing. The machine dependent layer is a set of C++ classes used by the Chorus portable layers. These classes form the interface between the portable layer and the machine dependent layer in a manner similar to Mach's **pmap**. The page fault interface is a bridge between the machine dependent exceptions (page faults and protection) and the Chorus portable layer. It allows higher level Chorus mechanisms to use page and protection faults to implement various optimization features e.g., copy-on-write memory and lazy page allocation. Each of these tasks is further documented in other reports [8, 11].

4.3 Design Phase

The design phase involved mapping the Chorus virtual memory abstractions on to the PA-RISC memory management features. This section details some of the major design decisions we made and our rationale for each decision. In some cases the design is described in other documents. We followed a few basic principles in developing our designs:

- Reuse Tut code (modified HP-UX 2.0). By reusing this code (which we knew worked) we hoped to save time in getting our initial implementation running. It was especially important for us to attempt to reuse the low level assembly language code since we were not very familiar with the PA-RISC assembly language.

- Simplicity. Originally, we started to develop some rather ambitious plans and ended up finding out that it was extremely hard to get *anything* to work. We then revised our policy to keep things simple, get something working, and then get ambitious!
- Avoid changes in the Chorus portable layer. Where possible, we wanted to avoid making any modifications to the portable code. This would make integrating our code with future kernel releases much simpler.
- Keep HP-UX compatibility. When we had to make choices, many of these were influenced by how HP-UX did things. Initially, one of our goals was to port Chorus/MiX in a way that allowed us to run HP-UX binaries.

4.3.1 Address Space Partitioning

An early problem in using a shared address space is deciding how to partition it. We decided to take a similar approach to HP-UX. HP-UX uses the four space registers (SR4–SR7) to partition a process' address space into four quadrants (see figure 8). HP-UX assigns quadrants for a process as follows:

- First quadrant (SR4): Used for code. When a process enters kernel mode SR4 is set to the kernel's space number (SID=0).
- Second quadrant (SR5): Used for a process' private data and stack.
- Third quadrant (SR6): Used for shared libraries
- Fourth quadrant (SR7): Used to access the shared gateway page[6] and shared memory segments.

One interesting decision we faced was whether to extend a process address space beyond 32-bits. Since Chorus assumes 32-bit addressing in both its machine-independent and machine-dependent interfaces we decided to limit each process to a 32-bit address space, i.e., use only short pointers. Extending the Chorus virtual memory manager to incorporate 64-bit addresses was an enormous task that we didn't want to perform. Once we limited a process to a 32-bit address space we needed to decide how to use our four space registers (SR4-SR7). Unlike HP-UX, which shares text by sharing the same text segment, Chorus uses memory mapping. For this reason we assigned a single space identifier for a process' text and data segments. The initial implementation doesn't support shared libraries so SR6 is not used. Figure 9 shows the space register usage used by PA-Chorus.

4.3.2 Address Translation

The major design decision here was the format of the page tables i.e., whether or not we would use the PDIR format. Since TLB refill is software controlled we had a great deal of flexibility in our

[6]The gateway page is used to promote a process privilege level during a system call.

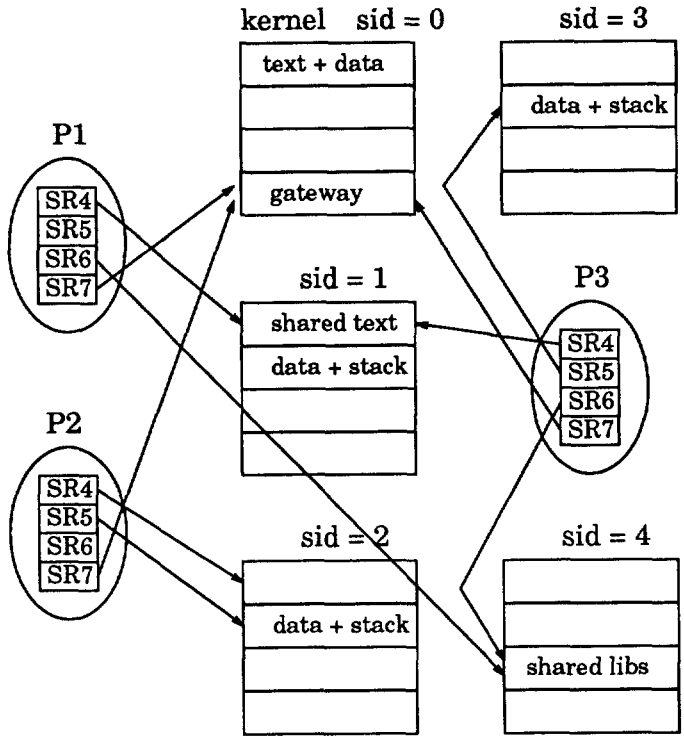

Figure 8: HP-UX Process Model

page table design. We first weighed the advantages and disadvantages of using the PDIR structure. We considered the following advantages of using the PDIR structure:

1. Number of entries is proportional to the amount of physical memory.
2. We could reuse a large portion of the HP-UX code for PDIR management.
3. We could take advantage of hardware TLB refill in later implementations.

The disadvantages of the PDIR structure were as follows:

1. Address translation on a TLB miss fault is more expensive because it requires a hash on the virtual address followed by a linear search through a portion of the PDIR.
2. The PDIR only contains the protection information for pages currently in memory.
3. The PDIR entry does not support aliasing since, in its current format, it only has space for one virtual page entry.

The final choice reduced to two separate decisions. The first was whether or not to use an inverted page table. The second was whether or not to keep the PDIR format used by the HP-UX code. We

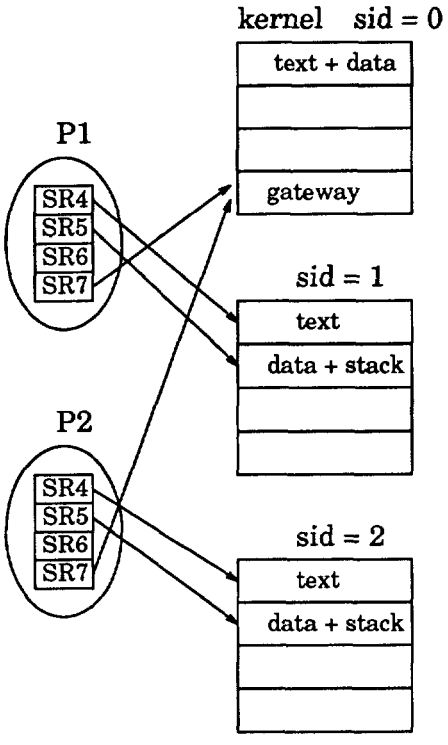

Figure 9: PA-Chorus Process Model

decided to use both an inverted page table and the PDIR entry format for our initial implementation. The biggest factor involved in this decision was the reuse of HP-UX code, however there were other reasons to support the decision. One was the fact that the inverted page table format was well suited to Chorus' approach to memory protection. Chorus maintains memory protection information at the *region* level which allows protection to be maintained at a much coarser granularity for pages currently not in memory.

4.3.3 Protection

One easy task was to translate the Chorus protection rights into PA-RISC access rights. Table 2 shows the mapping used between the Chorus protection rights and the PA-RISC access rights. While this let us determine how to use the access rights, we still needed to decide on how we would use the protection identifiers (PID's). The PA-RISC uses four control registers (CR8,CR9,CR12,CR13) to check protection rights. HP-UX assigns one PID to each of the text (CR8) and data (CR9) sections. CR12 and CR13 are used to cache shared memory PID's. Since the PA-RISC uses a globally shared virtual address space, PID's are used to protect a process's address space. Since Chorus shares text by memory mapping, we didn't have a machine-dependent interface where we could have used to PID's to implement copy-on-write or shared text. We ended up assigning one PID to each Chorus context.

Table 2: Protection Translation (Numeric values in binary)

Chorus Protections		PA-RISC Protections			
Value	Symbolic Name	Type	PL1	PL2	Symbolic Name
000	mmuReadU	000	11	11	PDE_AR_URKR
001	mmuExecU	010	11	11	PDE_AR_URXKR
010	mmuWriteU	001	11	11	PDE_AR_URW
011	mmuWriteExecU	011	11	11	PDE_AR_URWX
100	mmuReadS	000	00	00	PDE_AR_KR
101	mmuExecS	010	00	00	PDE_AR_KRX
110	mmuWriteS	001	00	00	PDE_AR_KRW
111	mmuWriteExecS	011	00	00	PDE_AR_KRWX

4.3.4 Kernel Memory Map

The PA-Chorus kernel memory map and virtual memory initialization are described in [8].

4.3.5 MMU Classes

The interface between the Chorus portable layers and the machine dependent layer is specified by two C++ classes. The description of the interface is described in section 5 and the implementation is presented in section 6.

4.3.6 Traps and Page Faults

The design and implementation of the PA-RISC trap handlers for PA-Chorus is described in [11].

4.3.7 Handling Aliasing

The majority of the hard problems in the virtual memory port were related to dealing with address aliases. As pointed out in section 2.4 the PA-RISC requires software to maintain cache consistency in the presence of address aliases. We decided to deal with aliasing in three stages.

Stage 1: Never allow aliases to exist within the cache.

Stage 2: Under certain safe conditions allow aliases to exists within the cache.

Stage 3: Modify the interfaces which generate aliases.

Stage 1 followed our principle of keeping things simple. In this phase we prevented aliasing from occurring within the cache by flushing the cache whenever aliases could have been generated.

This was implemented by synchronizing access to physical pages by allowing only a single address translation to exists for each physical page at any point in time. For example, if a physical page was mapped into two contexts only one of those contexts would be allowed to access the page. If a thread operating in the other context attempted to address the page it would result in a pseudo-page-fault. This fault would not result in any I/O activity, instead the address translation for the first context would be invalidated and the cache would be flushed. (Since the PA-RISC allows you to selectively flush the cache, only a single page need be flushed rather than the entire cache). A new mapping to the second context would be established. We call this technique *pseudo-aliasing* since the portable layer believes that multiple mappings exists while the machine dependent layer allows at most one mapping to exists at any point in time.

The objective of phase 2 is to eliminate unnecessary cache flushes. While pseudo-aliasing is rather simple to implement, it is not very efficient. There is a performance cost in the time spent flushing the cache plus the time spent refilling it. Stage 2 involves recognizing when aliases can safely exist within the cache and avoiding cache flushes under those conditions.

Stage 3 is even more ambitious. The objective of this phase is to modify the interfaces in the machine dependent layer to avoid generating aliases in the first place. This phase requires that modifications be made to code in the portable layer.

Stage 1 was used for the initial implementation. Stage 2 and 3 are scheduled from late 1991 till March 1992. The techniques used in this port and their performance implications will be published in another document [9].

4.4 Implementation Phase

Once the design reached a stable state, the implementation was relatively easy. The hardest part was writing the assembly language code! The implementation of the three major tasks is documented in other project technical reports. In addition to the implementation of the design there were other tasks that needed to be performed. The Tut code needed to be integrated into the Chorus source environment. There were several cases where conflicts (or duplications) between the Chorus and Tut header files needed to be resolved. There was also a slight delay in attempting to compile some of the Chorus sources because we didn't have a C++ 1.2 compiler. Luckily, Chorus Systèmes was very responsive in getting us the newer v3.3 sources that could be compiled under C++ 2.0.

4.5 Validation Phase

In order to validate the kernel we used a set of kernel tests provided by Chorus Systèmes. These tests are described in other documents [6, 10]. We did need to write an additional test to validate the virtual memory section. This test attempted to verify that cache consistency was being maintained by the virtual memory manager.

4.6 Analysis Phase

In progress (i.e., this document was published before this phase was completed. Performance results may be published in future documents.)

5 MMU Interface

The MMU interface is similar in nature to Mach's **pmap**. It specifies the interface between the portable and machine-dependent layer. The Chorus MMU interface is specified by the methods of two C++ classes: **mmuPage** and **mmuContext**.

5.1 MMU Interface: Class mmuPage

The class **mmuPage** is derived from PVM class **vmPage**. An mmuPage represents the interface between the PVM abstract page and the machine-dependent MMU page. Each instance of an mmuPage represents a physical page of memory. This class must implement the following methods:

- **operators** – In C++ 2.0, the operators **new** and **delete** are used to allocate and deallocate the storage used by a class. Constructors are used to initialize the instance of the class once storage has been allocated.
- **mmuPage(u_long)** – This constructor initializes a mmuPage descriptor but doesn't touch the contents of the page. The parameter is a historical artifact from earlier versions of the Chorus nucleus and can be ignored. It was used to represent the memory protection rights the page should be assigned.
- **mmuPage(u_long,mmuPage*)** – This constructor also initializes the contents of the page descriptor, but the contents of the page are copied from a given source page. As with the first mmuPage constructor, the u_long parameter can be ignored.
- **~mmuPage()** – This destructor should perform all the necessary tasks of cleaning up a mmuPage descriptor.
- **void unload()** – Removes the page descriptor (for this page) from all page tables mapping the page.
- **void readOnly()** – Resets the protection rights for this page so that writes are disallowed.
- **void fillZero(unsigned long, unsigned long)** – Fills a section of the page with zeros.
- **mmuPageStatus getStatus()** – Returns the most significant modification status for this physical page, either *mmuPageWritten* or *mmuPageUntouched.*
- **void setStatus()** – Marks this page dirty (*mmuPageWritten*) or unmodified (*mmuPageUntouched*).

- **PhAddr getPhysAddr()** – Returns the physical address of the page. This method is only called by **vmObject::getPhysAddr()** which is called from the system call **ScVmGetPhysAddr()**. We were informed that this system call is only used for the Compaq's floppy disk driver.

- **void* getAddr(u_long)** – Returns a pointer that can be used to access the page. Rather than using actual physical addresses, Chorus uses a *global map* [7] to emulate the use of physical addresses. The global map is a section of the kernel's virtual address space that maps the entire physical address space. Rather than return a physical address, **getAddr()** should return the virtual address within the global map that corresponds to page offset specified by the parameter.

5.2 MMU Interface: Class mmuContext

The class **mmuContext** represents a virtual address space. It is the base class of derived classes **vmContext** and **context**. There is one instance of this class for the kernel and system actors (referenced by the pointer **KernelContext**) and one instance for each user actor (with the currently executing actor's context being referenced by the pointer **CurrentContext**). In the Chorus ports to the Motorola 88000 and Intel 80386 this class was used as a process's address translation tree.[8] The PA-RISC uses a single globally shared inverted page table and has no requirements for any particular page table format. (This is a feature of having software TLB miss handling.) It uses a physical page directory (PDIR) for both physical-to-virtual and virtual-to-physical address translations for all pages in memory. This simplifies the implementation of this class since the PA-RISC does not require separate address translation trees for individual processes. The following methods must be supported by this class:

- **mmuContext()** – Creates a new context. This involves allocating the physical pages needed to store the address translation tree and initializing them.

- **~mmuContext()** – Destroys a context. This involves deallocating the physical pages used to store the address translation tree.

- **void unload(VmAddr,VmAddr)** – Unmaps all virtual pages within the address range, i.e., their address translations are invalidated.

- **void load(VmAddr,mmuPage*, mmuProtection)** – This method maps a physical page to a virtual address and assigns it certain protections rights.

- **void schedule()** – Schedule a context. Normally this method is used to adjust the virtual memory translation tables from one context to another. This usually involves modifying some processor registers so the hardware TLB miss handling routines can access the correct page tables.

[7] This should not be confused with the class GlobalMap.

[8] An address translation tree is a hierarchical structured page table.

6 MMU Implementation

This section describes the implementation of the two machine dependent classes: **mmuPage** and **mmuContext**. The implementation was complicated by the PA-RISC's virtually indexed cache and the use of address aliasing by the Chorus MMU interface.

6.1 Class: mmuPage

The class **mmuPage** represents a physical page of memory. Memory locations for memory mapped I/O are not represented by instances of this class.

6.1.1 Local Variables

There seem to be no restrictions on the names used for instance variables, as long as this class can execute the methods specified by its interface. The class mmuPage maintains four instance variables:

```
pde* ref;
PGstatus pageStatus;
mmuPageStatus status;
dlist pgCtx;
```

A reference to the page's PDE is kept in **ref**. Since the PDE's are ordered in a one-to-one relationship with the mmuPage descriptors (this was a design decision, not a Chorus restriction), it is not necessary to keep a reference to a mmuPage's PDE. This was a space vs. performance decision that trades off keeping the reference rather than recalculating it each time it is needed. The **pageStatus** flag indicates whether a page is UNALLOCATED (unused), ALLOCATED (used but unmapped), LOADED (contains one valid virtual address translation), or SHARED (contains more than one valid virtual address translation). The **status** flag indicates whether a page is dirty or clean. This flag may be unnecessary as the information is also kept in the PDE D-bit. Later versions will most likely eliminate the **status** flag. Its original purpose was as a double check to make sure the integrity of the D-bit was maintained when changing the mapping from one virtual page to another during address aliasing. The **pgCtx** link is used by the class **mmuContext** to keep track of the pages currently mapped into its address space.

6.1.2 Operators: new(size_t) & new(size_t,mmuPage*)

As stated previously, the operators **new** and **delete** are used to allocate and free the storage used by an instance of this class. There are two **new** operators: one that is normally used and one that is only used during kernel initialization. The operator used during normal operation allocates a page descriptor from the pool of free page descriptors. The second operator returns the same page

descriptor given to it as an argument i.e. the specific page. This method is only used during kernel initialization when it is necessary to initialize the pages of the boot actors, i.e., there is a need to allocate specific pages rather than arbitrary ones.

6.1.3 Operator: delete()

This operator returns an mmuPage descriptor to the pool of free page descriptors.

6.1.4 Method: mmuPage(u_long)

This constructor sets *status* to ALLOCATED. As mentioned previously, the single parameter has no value and can be ignored.

6.1.5 Method: mmuPage(u_long,PGstatus)

The second constructor performs the exact same tasks as the constructor mentioned above but sets the page status as LOADED rather than ALLOCATED. This constructor is used during kernel initialization when the pages of the boot actors are being initialized. These pages have been mapped during boot and their status should show this.

6.1.6 Method: mmuPage(u_long,mmuPage*)

The last constructor operates in a similar manner to the first, but it also initializes the contents of the page from a source page. The first parameter (supposedly specifying the protection rights) can be ignored. The second parameter is a reference to the mmuPage descriptor representing the source page.

The initial implementation flushes the virtual addresses (if the pages are mapped), copies data using physical addressing, and flushes the physical addresses after the copy completes. Since flushing the cache is an expensive operation, future implementations will attempt to eliminate these cache flushes by copying using virtual addresses [12]. Since this method is normally used to initialize an unmapped page from a mapped page, we can reduce the amount of cache flushes to one page by using using virtual addresses for the loads and physical addresses for the stores [9].

6.1.7 Method: ~mmuPage()

This method resets the page's *status* to UNALLOCATED.

6.1.8 Method: void map(sid_t,soffset_t,mmuProtection,u_int)

This method (called **load()** in other ports) is used to establish a mapping between this page and a virtual page. The first two parameters contain the space identifier and space offset of the virtual page. The third parameter specifies the (Chorus) protections to be associated with this page. The final parameter contains the access identifier to be used by this page.

This method becomes complicated if the page is already mapped. We have two things to worry about: address aliasing and PDIR entries. As pointed out in a previous section, the operating system is responsible for maintaining cache consistency in the presence of address aliases. In addition, the PDE structure isn't suited to provide more than one mapping per physical page. We would need to augment this structure with additional information so the TLB miss handlers could resolve aliased pages without consulting the portable layers. One such approach is described in [13]. Because of these problems, we took a simple approach for the initial implementation. If the page is already mapped, it is first unloaded using the **unload()** method and then the new mapping is installed. We call this technique *pseudo-aliasing* because it simulates address aliasing but allows no more than one mapping to exist within the PDIR (and TLB) at a given point in time.

The new mapping is established by adding the PDE to the hash bin for that virtual address and initializing the PDE. The page's PDE fields are filled using the information passed in the parameters. Note that the protection identifier is passed as a 15-bit value so it should be shifted over before being assigned to the *pde_protid* field.[9] All the bit flags in the PDE are cleared with the exception of the D-bit.

6.1.9 Method: void unload()

The method must flush the page's contents from the cache. Even though the page is clean it must still be flushed to avoid leaving stale data in the cache. The page's PDIR entry is the removed from the hash bin it belongs to and flushed from the TLB. In the future, this method may take a flag indicating whether a cache flush is really necessary (see mmuContext method **unload()**).

6.1.10 Method: void readOnly()

This method sets a page's protection rights to disallow writes. The page's PDIR entry is examined to determine its current access rights. These access rights are set by modifying the 3-bit *type* field of the access rights to indicate a read-only page. If the page is not already marked read-only, the entry should be flushed from the TLB.

[9]The header files for HP-UX include the write-disable bit in the *pde_protid field.*

6.1.11 Method: void setProtection(mmuProtection)

This is a more general version of the method above in that it resets the PDIR entry's protection to any possible Chorus protection. Note that the protection passed in is a Chorus protection. It must be translated into the appropriate access rights and stored in the PDIR entry. Table 2 shows the translation from various Chorus protections to PA-RISC protections. If the new protection differs from the old one, the TLB entry for the page must be flushed.

6.1.12 Method: void fillZero(u_long offset,u_long size)

This procedure fills a section of a page with zeros. The first parameter specifies the offset within the page and the second parameter specifies how many bytes to zero.

If the page is mapped, then this routine zeros the area using 64-bit virtual addresses. Note that protection id (PID) checking should be disabled while this routine is running. This avoids protection faults when the kernel doesn't have the appropriate PID for the page. If the page is not mapped then this routine zeros the area using physical addresses and then flushes those addresses from the cache.

6.1.13 Method: mmuPageStatus getStatus()

This method must return *mmuPageWritten* if the page is dirty or *mmuPageUntouched* otherwise. Since the TLB (dirty) D-bit trap handler was not modified to update the *status* flag, this method must examine the D-bit in it's PDE. If the D-bit is set then the local flag *status* should be updated and the value *mmuPageWritten* should be returned. Otherwise *mmuPageUntouched* is returned.

6.1.14 Method: setStatus()

We restrict this operation to mapped pages. Should an unmapped page execute this method, it will not have any effect on the MMU data structures. In marking a page *mmuPageWritten* the D-bit should be set in the page's PDE. In marking a page *mmuPageUntouched* the PDE's D-bit should be examined first. If the page is already clean then nothing more needs to be done. If the page was dirty, the D-bit should be reset and the TLB should be flushed for that PDE. This will enable future writes to that page to have a chance to set the D-bit.

6.1.15 Method: vmAddr getPhysAddr()

This is a dangerous interface for the PA-RISC. Any use of physical addresses on non-equivalently mapped pages can result in cache inconsistency. This method can be easily implemented by calculating the page number from its index in the pagePool. The page number would then be shifted over by the correct number of bits to obtain the physical address.

We considered taking an exception if this method was called. We could then determine what programs were attempting to use this method. The compromise solution was to return the physical address but print a message stating that a dangerous method was called that could possible corrupt cache consistency.

6.1.16 Method: void* getAddr(u_long)

This method is usually implemented using some form of aliasing such as the **global map**. Rather than actually implement a global map in the kernel address space, we set up a temporary mapping in the PDIR and TLB. A portion of the kernel address space (starting at 0x3F000000) was reserved for the global map.

When **getAddr()** is called, a mapping to that page's **global map** address is set up in the PDIR. If the page is currently mapped, the page is first unloaded in a manner similar to that of pseudo-aliasing. If some other thread requires the use of the original mapping, it takes a TLB miss trap into the Chorus page fault handler. During this process, the original mapping is restored and the mapping to the global map is removed. Though the mapping is normally used right after the call, there is a chance that it can be lost before it is used. For this reason, we added additional code to the TLB miss fault handling routines to reestablish a mapping if the fault lies in the region reserved for the global map (only for processes executing at the most privileged level). This is a hack and a quick solution for some fundamental incompatibility between the MMU interface and virtually addressed caches.

6.2 Class: mmuContext

With the PA-RISC's use of an inverted page table, there is no need to use a multi-level page table for each context. On this architecture, mmuContext represents a protection domain within a global address space instead of a private process address space. We assign each context two unique identifiers. One represents the context's space identifier (SID) and the other represents the context's protection identifier (PID). For simplicity, it was decided to constrain each context to a single SID for our initial implementation. Because Chorus uses memory mapping, as opposed to sharing global address segments, to shared memory we couldn't make much use of the PA-RISC's global address space without modifying portable code. (This is discussed in more detail in section 7.3.)

6.2.1 Local Variables

This class contains four instance variables:

```
sid_t spaceId;
u_int protId;
dlist pgList;
vmAddr myDataPointer;
```

The *spaceId* and *protId* represent the space identifier and access identifier for all pages in this context. The *pgList* is a list of mmuPage descriptors that are currently mapped to virtual pages in this context. The content of *myDataPointer* is the PA-RISC's data pointer value. When switching between threads belonging to different system actors the data pointer must be reset.[10] The value is stored in the mmuContext structure.

6.2.2 Method: mmuContext()

All this constructor does is initialize the *spaceId* and *protId* fields to unique identifiers obtained from special pools for space and access identifiers.[11] In other ports of Chorus, when a new context is created it is allocated some physical memory resource which it uses for the root page of its memory tree structure. The PA-RISC doesn't require any per-process page tables since it uses the PDIR for translations.

6.2.3 Method: ~mmuContext()

This method returns the values of it's *spaceId* and *protId* to their respective pools.

6.2.4 Method: void unload(vmAddr,vmAddr)

In order to unmap the physical pages within the address range this method must determine which virtual pages within the address range are currently in memory (mapped to physical pages). Since the PA-RISC uses an inverted page table (it keeps physical pages in sequential order) it was more difficult to sequentially invalidate virtual pages. We came up with several choices:

- Search the entire PDIR for addresses in the virtual range.
- Maintain a bitstring for each context. Each bit would represent a single PDIR entry, so searching the PDIR would be reduced to searching a bitstring for 1's.
- Maintain an unsorted list of pointers to PDIR entries used by the context.

The first method is undesirable since it is obviously slower than the second method. The third method has advantages over the second method in that you can search only PDIR entries mapped to the context. We chose to implement the third method. The head of the page list is represented by *pgList*. When this method is called, a linear search over the pgList is made and all pages in the desired range are unmapped. Since this method is normally called before context destruction, a linear search over the pgList is not as expensive as it may seem. One inefficiency in the initial implementation is that this method calls the mmuPage method **unload()** which flushes the page from the cache. If this context has many physical pages in memory it may be more efficient to flush

[10]This needs to be done only within the kernel's address space since user actor's data pointers are automatically assumed to be placed at 0x40000000, i.e., the second quadrant.

[11]These unique identifiers are NOT Chorus UID's.

the entire cache once and avoid the individual page flushes for each physical page loaded into this context.

6.2.5 Method: void load(vmAddr,mmuPage*,mmuProtection)

This method calls the mmuPage method map and adds the page to its list of pages. A check is made to determine if the page is already loaded into the context before adding it to the list. The copy-on-write process uses page faults (which can result in loads) to perform page protection modifications.

6.2.6 Method: schedule()

The purpose of this method is to switch the page table structure from one context's mapping to another, i.e., from one context's address translation tree to another's. Since we used the PA-RISC PDIR table (which is shared by all contexts), this method doesn't need to perform any task.

7 Evaluation

The hardest part of the virtual memory port was determining what the correct interfaces were. The lack of documentation on the Chorus MMU interface caused numerous bugs in the machine dependent design which slowed us considerably. Establishing the interfaces from reading source code was not very efficient and should have been avoided. This was evident when we received the newer v3.3 kernel sources in April 1991. Several of the interfaces taken from older v3.2 sources were obsolete and needed to be updated to the v3.3 interface. Most of these changes were simple (required for C++ 2.0 compatibility) but required that the new source code be read.

We found that the features of the PA-RISC's memory management unit support the Chorus mmuContext interface very well but has several conflicts with the mmuPage interface.

7.1 Evaluation: mmuContext

The class context represent's a process' address space. We found that it could be implemented quite easily as a process' protection domain. This class seems to be an artifact of older private per-process address space architectures and was easily implemented on a global address space architecture.

Using the PDIR simplified the construction and destruction of instances of class mmuContext. There was no need to allocated physical pages for per-context page tables. This made context creation much faster and required less physical memory space for page tables.

The only difficult tasks in implementing mmuContext was the deallocation of the pages for a given address range. This method is usually used to unload all physical pages within a context

prior to context deletion. Since an inverted page table isn't well suited to traverse a virtual address range (you would need to hash and search for each virtual page) we needed to maintain a list of physical pages loaded in each context. When a range of pages are deleted, this list is searched and all physical pages within the range are deallocated. Though we haven't had a chance to profile the system yet, we don't believe this will result in a significant performance penalty.

7.2 Evaluation: mmuPage

The major problem with the implementation of class **mmuPage** centers on the operating system's responsibility to maintain cache consistency. While maintaining cache consistency is not necessarily difficult, it can grow in complexity as performance becomes more of an issue. The complexity of the implementation depends on the amount of performance expected. The loss in performance is a result of the ability of the mmuPage interface to generate address aliases. This requires the operating system to flush parts of the cache for certain operations. Pseudo-aliasing is a very simple solution but suffers a large loss of performance due to cache flushes. (We plan to quantify this performance loss in the next evaluation of the system [9].)

The following four methods can generate address aliases: mmuPage (third constructor), getPhysAddr, getAddr, map. The third mmuPage constructor initializes a target (this) page from a source page. Aliases can be created if the target page is not yet mapped i.e., need to use physical addresses. If the mapping of the target page has been determined by the time this method is called, then the interface could be modified to allow the portable layer to pass the virtual address of the target page down to the machine-dependent layer. By using the virtual address (instead of the physical address) in the copying routine, it would be possible to avoid flushing the cache.

The method **getPhysAddr()** returns a physical address. Use of that physical address on non-equivalently mapped pages can result in the generation of virtual-physical aliases. Use of this call by code in the portable layer may result in cache consistency problems for architectures with virtually addressed caches.

The method **getAddr()** is used to generate a mapping to a physical page which can then be used to address that page. Should a mapping already exists, this can result in virtual-virtual aliases being generated. Since this method is heavily used by the IPC mechanisms (to obtain addresses which are then used to copy data), it can result in degraded IPC performance. This interface could be improved by using a method of the form **int getAddr(Address*)** where *Address* would be a structure representing an address. This *Address* could also be passed to a kernel machine-dependent memory copy routine. For the PA-RISC, this would allow the MMU layer to return a global 64-bit address instead of a local 32-bit address. By using the suggested interface, a machine-dependent copy routine could use global addresses and avoid generating a new mapping. There would still be a problem if the page was not mapped, i.e., no valid virtual address. A mapping could then be created and returned, but future use of that mapping could still create address aliases.

The method **map()** is used to load pages into contexts. When a page is loaded into more than one context, address aliases are created. Chorus uses page sharing to implement copy-on-write memory and shared text. It could more efficiently implement shared text at a higher level of abstraction.

7.3 Supporting a mmuRegion class

Chorus maintains the abstractions of a segment, context, region, and page in the PVM layer while only allowing the MMU layer to support the abstractions for the context and page. In certain cases, it might be beneficial to provide a mmuRegion class. We weren't able to take advantage of certain PA-RISC features because Chorus implements certain operations on regions by performing many operations on pages. In certain cases, it would be more efficient to operate on regions rather than pages. As an example let's look at protection modifications in the case of copy-on-write data and shared code.

Chorus maintains protection at *region* granularity in the portable layer and at *page* granularity in the MMU layer. When the protection of a Chorus region changes, the protection for each physical page in that region must be changed. (This is normally done to support copy-on-write.) The PA-RISC provides a write-disable bit for each protection ID. When this bit is set, writes to pages with the same access ID result in protection traps. This port did not use the write-disable bit at all for two reasons. The first deals with the concept of threads. Protection ID's are associated with each thread context. To mark pages with a certain access ID read-only, all threads with matching protection ID's must be found and the write-disable bit set on all of them. The other problem concerns the assignment of the protection ID's. Ideally, protection ID's would be assigned to regions since Chorus maintains protection at the region level. Unfortunately, the region level resides in the portable layer and not in the machine-dependent layer.

One disadvantage of the mmuRegion approach is that implementations on architectures without large granularity protection may become more complicated, i.e., they may now have to do the work that was performed in the portable layers (mapping region operations to operations on pages). The protection handlers may also be more of a problem, e.g., the first write to a copy-on-write region would be more complicated to handle.

In the case of shared text, Chorus establishes multiple regions mapping to the same segment. For the PA-RISC, this results in the generation of address aliases. It would be more efficient to inform the machine-dependent layer that the entire segment is being shared. This would allow us to share segment identifiers and remove aliasing created by sharing code. (Some of these suggestions will be addressed in more detail in another document [9].)

8 Conclusions

We presented the strategy used in porting the Chorus virtual memory manager to the HP PA-RISC. Key design decisions were discussed and explained. We described the interface for the Chorus machine-dependent layer and the implementation for the Hewlett-Packard PA-RISC. We noted that the implemention of the class mmuContext was very simple while the implementation of the class mmuPage was more difficult. The difficulty was not necessarily in correctness, but in performance. In this implementation, performance was not a goal. Because of this, we expect the performance of the operating system to be rather poor compared to commercial operating systems running on the same platform.

Several changes to the Chorus machine-dependent interface were suggested to take advantage of a globally shared virtual address space and to improve performance on virtually addressed caches.

9 Future Work

The majority of the future work involves improving the performance of the MMU layer. This work is aimed at generalizing the MMU interface for virtually addressed caches and reducing the amount of cache flushes needed to support address aliasing. A variety of performance related experiments are currently being planned. These experiments and their performance benefits are described in another document [9].

10 Acknowledgements

Many people participated in the discussions about the virtual memory port. I would like to extend special thanks to Vadim Abrossimov, for his time and patience in explaining the inner details of the Chorus machine dependent interface; Bart Sears, for the excellent tutorials on the Tut project's virtual memory implementation; Mendel Rosenblum, who suggested the enhancement to the mmuContext method unload(); and Jean-Jacques Germond, whose enthusiasm and energy was definitely contagious.

References

[1] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual Memory Management in Chorus. In *Proceedings of Progress in Distributed Operating Systems and Distributed Systems Management.* Springer Verlag, April 1989. Also published as technical report CS/TR-89-30.

[2] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 3-6 1989. Also published as technical report CS/TR-89-18.

[3] CHORUS Kernel v3.2 Implementation Guide. Technical Report CS/TR-90-5, Chorus Systèmes, 1990.

[4] CHORUS Kernel v3.3 Implementation Guide. Technical Report CS/TR-90-71, Chorus Systèmes, 1991.

[5] Ahmed Ezzat, Chia Chao, Milon Mackey, and Bart Sears. Tut VM Book. Technical Report HPL-DSD-89-32, Hewlett-Packard Laboratories, 1989.

[6] Jean-Jacques Germond. Specifications of the CHORUS/MiX Kernel v3.2 Test Suites. Technical Report CS/TR-90-27, Chorus Systèmes, 1990.

[7] Hewlett-Packard. *Precision Architecture and Instruction Set Reference Manual*, third edition, April 1989.

[8] 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.

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

[10] 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.

[11] 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.

[12] Milon Mackey. The cost of a page copy on HP-PA. Technical Report HPL-DSD-89-24, Hewlett-Packard Laboratories, 1989.

[13] Bart Sears. Read-Only memory aliasing in Tut. Technical Report HPL-DSD-90-9, Hewlett-Packard Laboratories, 1990.

[14] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.