Architectural Support for Single Address Space Operating Systems

Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 92-03-10

March 1992
Revised July 1992

Architectural Support for Single Address Space Operating Systems

Eric J. Koldinger, Jeffrey S. Chase and Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 92-03-10
March 1992
Revised July 1992

Abstract

Recent microprocessor announcements show a trend toward wide-address computers: architectures that support 64 bits of virtual address space. Such architectures facilitate fundamentally new operating system organizations that promote efficient data sharing and cooperation, both between complex applications and between parts of the operating system itself. One such organization is the *single address space operating system*, in which all processes run within a single global virtual address space; protection is provided not through conventional address space boundaries, but through *protection domains* that dictate which pages of the global address space a process can reference.

This paper focuses on the architectural implications of single address space operating systems, specifically the interaction between the memory system architecture and the operating system's use of addressing and protection. Our purpose is to explore certain architectural opportunities created by single address space systems by evaluating two protection models that support them. The first provides protection on a per-page, per-domain basis; we define the *protection lookaside buffer*, a hardware structure that implements this model. The second provides protection on a page-group basis; this model is implemented in the Hewlett-Packard PA-RISC architecture.

This paper will appear in identical form in the Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems, October 1992. This work was supported in part by by the National Science Foundation under Grants No. CCR-8619663, CCR-8907666, CDA-9123308, CCR-9200832 and MIP-9058-439, by the Washington Technology Center and by the Digital Equipment Corporation Systems Research Center and External Research Program.

1 Introduction

The rapidly increasing demands of software (both applications and operating systems), together with increasing VLSI densities, have brought us to the verge of a major architectural change: the move from a 32-bit to a 64-bit virtual address space. This trend can be seen, for example, in the segmented architectures of the HP PA-RISC [Lee 89] and the IBM RS/6000 [Groves & Oehler 90] and ESA/370 [Scalzi et al. 89], and the flat architectures of the MIPS R4000 [MIP 91], SPARC Version 9 [SPA 92], and Digital's Alpha family [Dig 92].

Unlike the move from 16- to 32-bit addressing, a 64-bit address space will be revolutionary instead of evolutionary with respect to the way operating systems and applications can use virtual memory. Consider that 40 bits can address a terabyte, two orders of magnitude beyond the primary and secondary storage capacity of all but the largest systems today, and that a 64-bit address space, consumed at a rate of 100 megabytes per second, would last five thousand years.

Large address spaces remove a basic assumption that has driven operating systems since the 1960s, namely, that virtual addresses are a scarce resource that must be multiply allocated in order to supply each executing program with sufficient name space. The small address spaces of current architectures cause most operating systems (e.g., Unix[1]) to execute each program in a private virtual address space. While private address spaces facilitate protection, they interfere with sharing and cooperation between applications, because virtual addresses in shared data are ambiguous; that is, their interpretation depends on the process using the address.

Next-generation operating systems can utilize the

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

wider virtual addresses of emerging 64-bit architectures by exploiting the concept of a *single virtual address space* [Carter et al. 92, Chase et al. 92a, Chase et al. 92b, Okamoto et al. 92, Scott et al. 90]. In such a system, addresses are unique and context-independent: an address has the same meaning and translation, independent of the process that issues it. Hardware-based memory protection still exists within this single address space, however; programs execute in *protection domains* that control their access to virtual memory. A protection domain defines the private data, code and stacks that an application can access, along with any data shared with other domains. In many ways it is the analog of the address space of a Unix process or Mach task: the primary difference is that it defines a private set of access privileges to globally addressable pages, rather than a private virtual naming environment. Thus, the single virtual address space separates the concepts of translation and protection. We call a system with this addressing model a *single address space operating system*. We are currently building one, called Opal [Chase et al. 92a, Chase et al. 92b]. Examples in this paper use Opal terminology, although many of the concepts (such as protection domains) will be common to other single address space systems.

This paper focuses on the architectural implications of single address space operating systems, specifically the interaction between the memory system architecture and the operating system's use of addressing and protection. A key issue is how to represent the protection information in the cache structures. Protection domains are supported in the architecture by tagging cache structures with protection information and matching the tags with protected register state in the processor on each memory reference. The choice of where the tags are stored and what they represent determines the actions that the operating system must take to implement common operations, especially those that change a protection domain's access to data. Since protection operations are becoming more frequent and expensive (i.e., they involve trapping to the kernel and possibly invalidating useful cache entries), these choices can have a substantial effect on performance [Anderson et al. 91, Appel & Li 91].

The paper is organized as follows. The next section discusses the advantages of single address space operating systems and their architectural implications. Section 3 describes the problems with current memory system architectures with respect to single address space operating systems, and presents two alternative protection models, each with a representative implementation, that support these systems better. First, we present a new memory system mechanism called the *Protection Lookaside Buffer*, which separates protection from translation and caches protection mappings on a per-domain, per-page basis. The PLB is a natural counterpart to a virtually indexed, virtually tagged cache in single address space systems, and permits relocation of the translation buffer to a second level, off the critical path. The second alternative is a slight variation of the Hewlett-Packard PA-RISC architecture [Hew 90], in which each executing protection domain has access to one or more *page-groups*. (A page-group is a set of pages treated as a unit for the purpose of access control.) Section 4 evaluates the two model implementations with respect to the needs of single address space operating systems, focusing on how the operating systems' use of protection interacts with architectural design choices in the virtual memory system. Section 5 looks at similar work, and Section 6 summarizes our presentation.

2 Implications of Single Address Space Operating Systems

Single address space operating systems have two broad architectural implications. First, they encourage sharing of memory between protection domains; existing architectures designed to run private address systems do not always support this sharing efficiently. Second, since they define a unique mapping between virtual and physical addresses, they introduce a new constraint that can be exploited in the architecture. This simplifies the use of high-performance virtually indexed data caches, and allows address translation to be removed from the critical path of processor memory references.

2.1 Support for Sharing

The principal advantage of single address space systems is that virtual addresses have a globally unique interpretation – a given piece of data appears at the same virtual address regardless of where it is stored or which domains access it. This context-independence simplifies sharing in two ways. First, virtual addresses (pointers) can be passed between domains, and linked data structures stored in the global address space are meaningful to any protection domain that can access them. Second, a domain can address any piece of shared data without risk of name conflicts with other data. This property is essential to supporting dynamic sharing patterns, in which shared data and procedures can be passed efficiently by reference.

The enhanced support for sharing can be used to improve the structure and performance of both applications and the operating system by reducing the need for explicit communication between programs executing in separate protection domains. This is important, because the communication mechanisms popular today (e.g., RPC) are based on data copying and protection domain switches, both of which have increased in cost relative to integer performance on recent processors [Anderson et al. 91, Ousterhout 90]. At the same time, software systems are growing in complexity, encouraging decomposition into multiple protection domains. The recent move toward server-structured, kernelized operating systems (e.g., Mach [Golub et al. 90], Chorus [Rozier et al. 92], Amoeba [Mullender & Tanenbaum 86] and Windows NT) is one example of this trend.[2]

2.2 Virtually Indexed Caches

Virtually indexed caches typically provide faster effective access than physically indexed caches, because they do not require address translation before the access [Knapp 85, Smith 82, Wood 90].[3] Furthermore, compilers can predict and control data placement in the cache, which is difficult or impossible in physically indexed caches in which placement depends on dynamically determined virtual-to-physical mappings [Lam et al. 91].

There are two principal problems with using virtually indexed caches in systems with multiple address spaces: *synonyms* (also called *aliases*) and *homonyms*. The synonym problem occurs when a physical page is mapped into two or more different virtual pages. Reference to an item through different virtual addresses causes that item to appear simultaneously in multiple cache lines, creating a coherency problem on writes. Many solutions to this problem have been devised, such as allocation restrictions (as in Sun OS [Cheng 87]), sharing on segment boundaries only (SPUR [Wood et al. 87] and the IBM RISC processors [Chang & Mergen 88, Groves & Oehler 90]), flushing the cache on process switches (as required by the i860 [Int 89]), lazily flushing aliased pages from the cache [Wheeler & Bershad 92], or hardware support for synonyms [Goodman 87, Wang et al. 89]. The first two solutions restrict the generality of the system, the third discards potentially useful cache state and adds to the cost of process switches, the fourth complicates the operating system, and the fifth complicates the hardware.

The homonym problem occurs when each address space has a different translation of the same virtual address; for example, every address space can have its own address 100, but each instance of that address refers to a different physical location. Homonyms can be eliminated by flushing the cache on process switches, extending the virtual address with an address space identifier or using physical (rather than virtual) tags in the cache. All techniques have drawbacks: flushing exacerbates the cold-start problem when returning from a process switch; address extension requires additional bits in each cache line, and introduces the synonym problem when different address spaces use the same virtual address to refer to the same location [Smith 82]; virtually indexed, physically tagged caches are also vulnerable to the synonym problem, because the same physical address can still reside in multiple locations in the cache [Wheeler & Bershad 92].

Neither synonyms nor homonyms need exist on a single address space system. Synonyms, typically used to facilitate sharing between address spaces or by the operating system, do not cause problems, because write-shared data always appears at the same virtual address in each process (protection domain) that uses it.[4] Homonyms cannot occur on a single address space system, because by definition only one translation exists for any address. Thus, by alleviating these problems, a single address space system removes several impediments to the use of a virtually indexed cache, which is the most attractive caching organization for performance reasons. Furthermore, the virtually indexed cache can be supported without flushing on process switches and without the need for additional address space identifier bits.

3 Protection Architectures

This section presents two alternative models of protection for single address space systems and representative implementations for their caching structures that contain protection information. First, however, we begin by examining conventional, multiple address space architectures and their relationship to single address space operating systems. In particular, we discuss TLB contents and their handling.

3.1 Conventional Architectures and Single Address Spaces

Most current architectures are designed to support multiple virtual address spaces, which are required by common operating systems like Unix and VMS. [5] These ar-

[2] The software implications of single address space systems, including incompatibilities for programs written under the multiple address space model, are discussed in more detail in Chase, et al. [Chase et al. 92a].

[3] Organizations in which a physically indexed cache is accessed in parallel with address translation are comparable in speed, but suffer from restrictions in cache size or memory mapping flexibility.

[4] Note that this does not prevent the use of copy-on-write optimizations. Copy-on-write uses read-only synonyms which do not have to be kept coherent. As soon as a write occurs to one copy of an address, the page is copied, and the synonym no longer exists.

[5] VMS is a trademark of Digital Equipment Corporation.

chitectures could support a single address space operating system, but in doing so they may incur unnecessary performance costs. For example, the VAX [Dig 81] and SPARC [Cyp 90] store virtual-to-physical mappings in linear page tables maintained separately for each protection domain. This organization causes two problems for single address space systems. First, protection domains in such a system have a sparse view of the global address space; that is, each domain would typically reference small and widely scattered pieces of the address space. Linear page tables cannot represent such sparse sets of mappings compactly. Second, translation mappings for shared pages must be duplicated in the page tables for each domain; this wastes space and forces the kernel to keep the duplicated mappings consistent.

Architectures with software-loaded TLBs, such as the MIPS processors [Kane & Heinrich 92] or the DEC Alpha [Dig 92], are better suited to supporting single address space systems. Without a hardware-enforced page table structure, the operating system can choose the most convenient representation for its virtual memory data structures, such as a single table of translations that is shared by all domains and a separate protection table for each domain, (similar to the inverted page table on the IBM 801).

If the TLB is tagged with an address space identifier [Kane & Heinrich 92], it can function as a protection domain identifier. Sharing of a page by multiple domains causes replication of TLB protection entries, even though each replicated entry has the same translation information. The duplication reduces the effectiveness of the TLB as sharing increases. The system also needs to ensure that all TLB entries for a page are coherent when mapping changes take place. Inconsistencies can be avoided by purging the TLB on protection domain switches (process switches) or mapping changes; however, this is undesirable, because purging removes not only the protection information, which is different for the newly scheduled domain, but also the translation information, which is the same for all domains.

3.2 Protection Organizations for Single Address Spaces

Memory protection can be implemented differently in single address space architectures. Since all protection domains reside in the same virtual address space, any program can attempt to reference any virtual page. Virtual-to-physical translations are global, but the hardware must represent page access permissions separately for each protection domain.

In this section we present two models for supporting protection in single address space systems. Both separate protection from translation. At a high level, the two alternatives are:

- The *domain-page model*, which specifies access rights explicitly for each *(domain,page)* pair, independent of the other domains and pages. This is the model used by most current systems; it can be implemented in single address space architectures by removing protection domain tags from the TLB and placing them in a separate *protection lookaside buffer* that contains no translation information.
- The *page-group model*, which defines logical groupings of pages called *page-groups*. Each page is a member of a single page-group, and a protection domain is defined by the set of page-groups that it can access. Each page within a group has access rights that are used by all domains with access to the group. Page-groups are used in the Hewlett-Packard PA-RISC.

The next two subsections discuss implementations of these two models, the protection lookaside buffer (PLB) for the domain-page model, and a variant of the Hewlett-Packard PA-RISC for the page-group model.

3.2.1 The Protection Lookaside Buffer

The domain-page model can be implemented by a new memory system protection cache, called the *protection lookaside buffer*, or PLB. The PLB caches protection mappings on a per-domain, per-page basis; that is, each PLB entry contains the protection information (the access rights) granted to one protection domain for one specific virtual page. If two domains each have access rights to the same virtual page, and both domains have recently accessed that page, then the PLB is likely to have two entries for that page, one for each domain.

Figure 1 shows the high-level organization of the PLB and a virtually indexed, virtually tagged cache. On each memory reference the PLB is accessed by the virtual page number (VPN), which is extracted from the virtual address, and by the current protection domain identifier (PD-ID). A processor control register, changed on domain switches, provides the current PD-ID used in a PLB lookup, just as some processors provide an address space identifier to virtually indexed caches or TLBs. If a match occurs on both the VPN and the PD-ID, the protection information is extracted directly from the entry in the PLB. If no match occurs, a PLB miss is signaled, and the PLB must be loaded with the appropriate protection mapping.

Note that in this scheme, VPN bits are used for both the cache and the PLB lookups. Thus, the cache and PLB searches can occur completely in parallel, because the cache lookup is not dependent on information provided by the PLB. If the cache lookup succeeds, the PLB-provided information indicates whether the memory reference has permission to proceed. If the cache

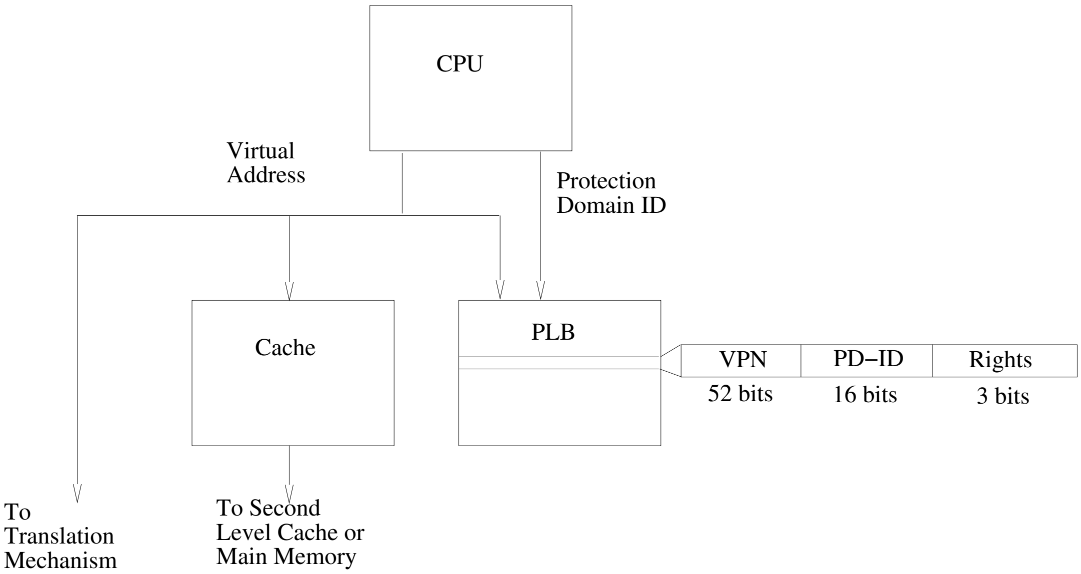

Figure 1: System with a Protection Lookaside Buffer and a virtually indexed, virtually tagged cache. Numbers shown indicate field widths, assuming 64 bit addresses and 4Kbyte pages. The VPN bits assume a fully associative PLB; fewer would be needed with a direct-mapped or associative organization.

lookup fails, there are two possible situations. If the PLB indicates that the reference is illegal, an exception will be generated. Otherwise, a cache miss occurs, and a translation must be provided by the TLB, so that data can be loaded into the cache.

As mentioned above, when pages are shared across domains, the PLB contains multiple entries for the page, one per domain. If both protection and translation data were stored in the PLB (comparable to a traditional TLB), each entry for a page could contain different access rights, but identical translation information. Separating the two avoids this duplication and allows the PLB to be used in conjunction with a virtually indexed, virtually tagged cache. (One drawback of this cache organization should be noted: that because of the wide addressing, more bits are needed for the tags, relative to a virtually indexed, physically tagged cache. For example, in a system with 64-bit virtual addresses, 36-bit physical addresses and 32 byte cache lines, a virtually tagged cache would be about 10% larger.)

As with all virtually indexed, virtually tagged caches, address translation is required only on the small percentage of accesses that either miss in the cache or require a writeback. The TLB can therefore be moved out of the critical path of the processor, and even off the processor chip; an obvious organization would place the TLB along with the cache controller for the second-level cache (similar to the design in Wang et al. [Wang et al. 89]). An advantage of moving the TLB off-chip is that it permits a larger TLB than that typically found in microprocessors. The PLB would still reside on the microprocessor chip, but with different configuration constraints than the TLB it replaces. Although it has fewer fields, more entries are required when pages are shared.

Note that since protection information for each domain is stored in the PLB, the TLB need contain only the VPN/PFN mapping and the dirty and reference bits.[6] Furthermore, the TLB requires only one entry for each virtual-to-physical page mapping; therefore, a purge is required only on the change of a virtual-to-physical translation. Protection domain switches do not require a purge of either the PLB or the TLB.

3.2.2 The Hewlett-Packard PA-RISC

The page-group model has been implemented in the Hewlett-Packard PA-RISC architecture. In the PA-RISC, the TLB entry for a page includes a set of access rights (the Rights field in Figure 2), and a field called an *access identifier*, or AID, that contains a page-group number, in addition to translation information. On a memory reference, the TLB is indexed with the virtual page number, returning the physical address translation and the AID for the page.

The processor must then determine if access to the page-group specified in the AID is permitted to the

[6] See Koldinger et al. [Koldinger et al. 91] for a discussion of their management.

currently executing protection domain (process). The set of page-groups accessible to the current domain is stored in a set of four *page-group registers* (called PIDs). In addition, there is a page-group that is global to all domains (group 0). If the page-group number in the AID matches one of the PIDs or the AID field is zero, then the exact access rights allowed are determined by a combination of: (1) access rights specified for the page in the Rights field of the TLB entry, (2) the current processor privilege level (PL in Figure 2) and (3) a write-disable bit in the PID register (D in Figure 3). The last allows disabling writes from a protection domain to an entire page-group, regardless of the value in the Rights field in the TLB. The Rights field may specify different access rights for different processor privilege levels (e.g., user, kernel, etc.), but we ignore this detail.

If the page-group number in the AID does not match any of the PIDs or the access allowed is insufficient to complete the memory reference, then an access violation is signaled and the operating system kernel is invoked. The kernel may respond by modifying the TLB or page-group registers and restarting the instruction, or it may deliver an exception to the protection domain that issued the instruction.

The current PA-RISC architecture, with only four page-group registers, limits the number of page-groups that a domain can efficiently access at a time; in addition, it provides no information (such as LRU information) to help the operating system manage the loading of the page-group registers. Thus, for the purposes of our discussion in the next section, we replace the PA-RISC's page-group registers with a cache of permitted page-groups, with support for LRU replacement (suggested by Wilkes and Sears [Wilkes & Sears 92]).

Because the page-group model requires only one set of access rights per page (and therefore one entry in the protection cache), protection and translation information can be combined in a TLB without duplicating translation data. Since the TLB must be accessed on each memory reference to obtain the protection information, the implementation includes an on-chip TLB. Note that the configuration used by the domain-page model implementation, i.e., separate caches for protection and translation information, coupled with a virtually indexed, virtually tagged cache and an off-chip TLB, could have been used here as well. Although the choice of protection model and the configurations of the protection, translation and data caches are orthogonal issues, separating the protection and translation structures and coupling them with a virtually indexed, virtually tagged cache and an off-chip TLB is the more natural configuration for the domain-page model. In this respect, the page-group model is the more flexible of the two, in that it works well with either.

4 Evaluation of the Protection Models

In this section we compare the PLB-style protection system (the domain-page model) and the PA-RISC-style system (the page-group model) with respect to tasks that we believe will be commonly performed by single address space operating systems. The comparisons will focus on how the operating system manipulates the primary hardware structures in each model, and the performance implications of these structures. A summary of this discussion appears in Table 1.

When we refer to the PA-RISC-style system, we assume a slightly modified architecture containing an LRU cache of page-groups, as described in Section 3.2.2, rather than the four page-group registers. We will refer to this configuration as the page-group implementation. In addition, to allow a fair comparison, we will assume that the PLB and the page-group TLB (both on-chip) have the same number of entries. In fact, there may be several reasons for differences in the number of entries. For example, the PLB requires multiple entries for shared pages where the page-group TLB would have only one. However, recall that PLB entries are smaller than page-group TLB entries (about 25%, assuming the field sizes in Figure 1 and a physical address of 36 bits), since they don't contain virtual-to-physical translations, allowing more entries in the same amount of space.

4.1 Operating System Tasks

4.1.1 Attaching and Detaching Segments

Single address space operating systems need a method for allocating space to the various protection domains. Our underlying model is based on *virtual segments*, the mechanism used in Opal. Virtual segments are sequences of one or more contiguous virtual pages, occupying a fixed contiguous range of virtual addresses, assigned when the segment is created and disjoint from the address ranges occupied by all other segments. Virtual segments are *logical* groupings of pages providing access control and storage management; their boundaries are unknown to the hardware, and addressing is independent. Virtual segments are the basic unit of sharing; they are used to represent many types of objects, such as code, shared libraries, private and shared data regions (heaps and stacks), mapped files and RPC communication channels [Bershad et al. 89, Bershad et al. 90].

Before a program can access the pages that make up a virtual segment, it must *attach* the segment to its domain. We believe that once mechanisms exist to facilitate sharing and cooperation, domains will typi-

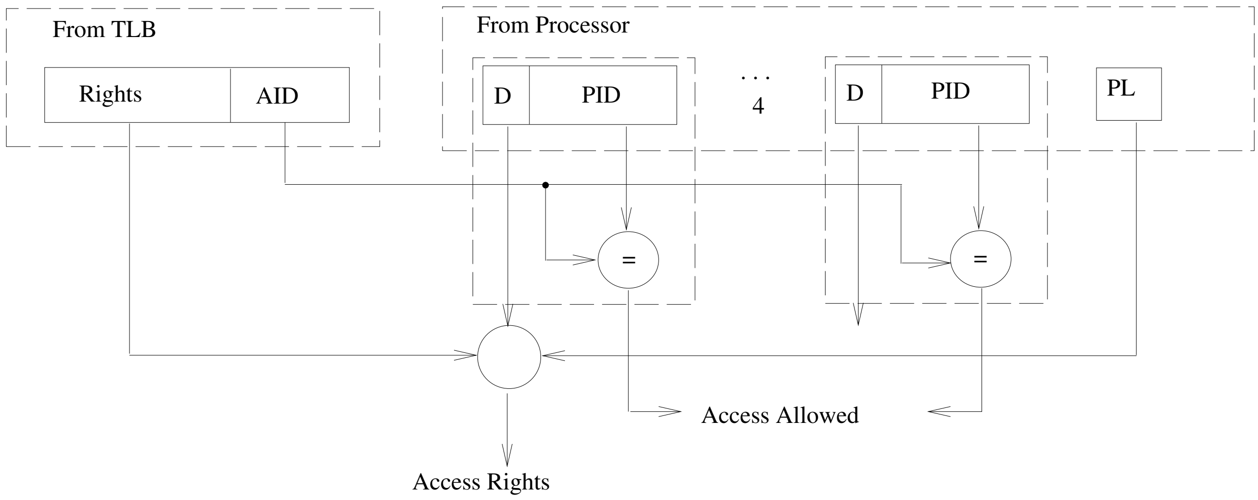

Figure 2: PA-RISC Protection Architecture

cally attach to multiple virtual segments; therefore, the architecture should efficiently support large numbers of active segments. Segment attachment should also be efficient, since they will be attached whenever a new file is accessed, a code library is first touched or communication is first established between a client and a server.

If the system represents virtual segments as PA-RISC style page-groups, attaching a new virtual segment is trivial; the operating system merely adds the page-group representing the segment to the set of groups accessible to the current domain, possibly adding an entry for it in the page-group cache. Similarly, detaching a segment simply requires removing the appropriate page-group identifier from the set of page-groups accessible to the current domain, and purging it from the page-group cache. All domains with access to a particular page group use the same TLB entries for pages in that group, and these entries are not affected by segment attach and detach operations.

In the domain-page model of the PLB, attaching segments is comparable in complexity to the page-group implementation. The operating system simply marks the segment as accessible by the protection domain; no hardware structures need to be manipulated. The individual PLB entries for each domain-page pair are lazily faulted into the PLB as the pages are accessed. Detaching, however, is more intricate. The operating system must purge the PLB of any entries that map the detaching segment from the current protection domain. In the worst case, this could require inspecting all the entries in the PLB and eliminating those that match.

4.1.2 Manipulating Page Permissions

Applications that use the protection mechanism to enforce particular memory semantics on segments will often need to manipulate a domain's access rights to individual pages. For example, virtual page protection may be used to support recoverable virtual memory (e.g., Camelot [Eppinger 89]), distributed shared memory [Li 86, Carter et al. 91], concurrent checkpointing [Li et al. 90] or transactional shared memory systems (as on the IBM 801) [Chang & Mergen 88], among other applications [Appel & Li 91].

Some of these applications involve changing the page permissions for all protection domains, whereas others require that page protections be maintained on a per-domain, per-page basis. For example, transactional shared memory on the IBM 801 runs each transaction in a separate protection domain and initially denies that domain access to all pages in the shared memory segment. As the transaction references pages, it generates protection faults on those pages; the system responds by granting locks and access rights for pages to the transaction. Each transaction locks pages independently of the others, so the system must be able to represent separate access rights for each domain and each page.

Such access right manipulation is straightforward on a PLB-based system. Since access rights are stored on a per-domain, per-page basis, changing a domain's access rights to a page simply requires updating a PLB entry.

Changing individual rights in this manner on the page-group implementation requires different operations, depending on the number of domains involved. If the rights are being changed for all domains that

Application Type	Action and Description	Frequency per Domain	Implementation	
			Domain-Page	Page-Group
Any	*Attach Segment*	Once per segment	Allow access rights to be faulted into the PLB, one page at a time.	Add the page-group identifier for the segment to the page-group cache.
	Detach Segment	Once per segment	Purge the PLB or inspect each entry and eliminate those for the segment-domain pair affected.	Remove the appropriate page-group identifier from the page-group cache.
Concurrent Garbage Collection [Appel et al. 88]	*Flip Spaces:* Change *to-space* to *from-space*. Create new segment for new *to-space*. Make both spaces read-write for the collector only.	Once per garbage collection	Inspect each entry in the PLB, marking those for *from-space* as no access for the application.	Remove the page-group identifier of *from-space* from the page-group cache for the application domain. Add separate *to-space* identifiers to the page-group cache for the application and the collector.
	Access unscanned to space: Trap the access, garbage collect the page and move it to "scanned" to-space, making it read-write for the application.	Once per page touched	Garbage collect the page. Mark it as read-write for the application domain.	Garbage collect the page. Place in to-space page-group for the application domain.
Distributed VM [Carter et al. 91, Li 86]	*Get Readable:* Trap the access, get a readable copy of the page and make it read-only.	Once per access of the remote page	Check to see if the copy in memory is valid, and retrieve it from the remote host if it's not. Set read-only rights in the PLB.	Check to see if the copy in memory is valid; retrieve it from the remote host if it's not. Put in accessible page-group and set read-only rights in the TLB.
	Get Writable: Trap the access, get an exclusive copy of the page and make it read-write.	Once per access of the remote page	Check to see if the copy in memory is valid; retrieve it from the remote host if it's not. Invalidate any other remote copies. Set read-write access in the PLB.	Check to see if the copy in memory is valid; retrieve it from the remote host if it's not. Invalidate any other remote copies. Put in accessible page-group and set read-write rights in the TLB.
	Invalidate: A remote machine invalidates the page. Make it inaccessible on this node.	Once per remote access to the local page	Set access rights to none in the PLB.	Set access rights to none in the TLB.
Transactional VM [Chang & Mergen 88, Eppinger 89]	*Lock (read):* Allow shared, read-only access	Once per page touched, per transaction	Determine if the page can be locked; if so, mark it as locked and set the read bit in the PLB entry for transaction's domain.	Determine if the page can be locked; if so, mark it as locked. Determine the correct page-group for the pages locked by the current domain, and move this page to that page group. Set the access rights to allow reading.
	Lock (write): Allow private, read-write access	Once per page touched, per transaction	As above, except set both the read and write bits in the PLB entry for the transaction's domain.	As above, except set the access rights to allow both reading and writing.
	Commit: Unlock all locked pages and return them to the inaccessible state.	Once per page touched, per transaction	Remove all the locks held by this transaction. For each locked page, look up the page in the PLB, and change the access rights to inaccessible; or change the PD-ID to represent the new transaction.	Remove all the locks held by this transaction. For each page that was locked, look up in the TLB and move to inaccessible page-group; or remove lock groups from the page-group cache and allocate new groups for the next transaction's locks.
Concurrent Checkpoint [Li et al. 90]	*Restrict Access:* Remove clients' access rights to all pages in the segment.	Once per checkpoint taken	Inspect each entry in the PLB and mark the pages as read-only for the application.	Mark the page-group for this segment as read-only to the application. Allocate a different group as read-write for this segment for both the application and server.
	Checkpoint Page: Trap the access and write the page to disk. Make the page read-write for the application.	Once per page, during the checkpoint operation	Write the page to disk. In the PLB mark it as read-write for the application.	Write the page to disk. Move it to a new read-write group in the TLB.
Compression Paging [Appel & Li 91]	*Page-out:* Make the page inaccessible to the application, compress the data on the page, write it to disk and unmap the page	Every time a page is deallocated	Mark the page inaccessible to the client in the PLB. Compress the data on the page and write it to disk. Remove the page entry from the TLB and allow the page to be reallocated.	Move the page to the page-group private to the server in the TLB. Compress the data on the page and write it to disk. Remove the page entry from the TLB and allow the page to be reallocated.
	Page-in: Allocate the physical page, read data from the disk and decompress. Make the page accessible to the client.	Every time a page is brought in from secondary storage.	Allocate the physical page, map it in the TLB, and mark it accessible to the server in the PLB. Read the page and decompress the data. Make the page accessible to the client in the PLB.	Allocate the physical page, map it in the TLB, and put it in the server's private page-group with read-write access. Read the page and decompress the data. Move to page-group representing this segment for the client.

Table 1: Some Common Functions that Manipulate Protection

can access a shared page-group, the change is easily made in a single TLB entry. However, changing the access rights for a *subset* of all domains that can access a shared page might require moving pages between page-groups. For example, suppose that two domains require read-write access to different pages within an otherwise read-only segment. Allowing these domains write access to their respective pages requires two additional page-groups, each containing the read-write pages for its respective writing domain. The original page-group for the segment is modified to contain only the remaining read-only pages.[7] This is necessary because of the global nature of page-group protection, which permits a domain that can write to one page in a page-group to write to any writable page in that page-group.

Similarly, representing read-locks in a transactional locking system can be done in one of two ways on the page-group implementation: putting all locks held by a given domain into a page-group private to that domain, or putting each locked page into a separate page-group shared by all domains that have a read-lock on that page. The first solution allows a domain to lock a large number of pages without penalty, but requires changing the page-group of a locked page if the lock becomes shared. This can cause a page to alternate between page-groups on each context switch. The second solution allows concurrent access to the locked page without changing its page-group, but can fill the cache of active page-groups if a domain holds many locks.

Each system has its advantages and disadvantages. A PLB system will take fewer faults in situations where there is active sharing and frequent protection changes. However, it does this at the cost of redundant entries in the PLB. The page-group implementation, on the other hand, will incur fewer TLB misses than the PLB in situations where sharing is static or protection changes are infrequent.

4.1.3 Paging Operations

Pages must be protected from access by applications while page-in and page-out operations (i.e., moving pages to and from secondary store) are in progress. If these operations are implemented by user-level paging servers [Young et al. 87] that read and write the pages directly, then the paging server's protection domain must have exclusive access to the page during the operation. In a PLB system access rights are simply updated in the PLB; the number of entries changed depends on the number of domains that have access to the page. In a page-group system a special page-group is used to represent the paging server's access rights. Pages are moved to the paging server's group prior to paging operations, by adding or updating the TLB entry for the page.

When virtual pages are unmapped, it is necessary to modify the caching and protection structures. This typically requires two steps: the page must be unmapped in the page tables and TLBs, and also must be flushed from the data caches. Unmapping pages is similar in the two systems; the page needs to be removed from the TLB, which is done with a small number of instructions on each processor. On the PLB-based system, no maintenance of the PLB is required. The entries will eventually be purged from the PLB through normal block replacement. Attempts to access a page before its entry leaves the PLB may generate a legal access (no protection violation), but the lack of a TLB entry will generate a fault, since the page will have been flushed and unmapped. Flushing the page from the data caches is approximately the same on both the PLB and page-group systems; one cache access is required for each cache line in the page (assuming cache flush is implemented as a series of individual *flush cache line* instructions, as it is on most modern processors).

4.1.4 Domain Switches

The cost of protection domain switches can have a large effect on performance. Domain switches are becoming more frequent, since many operating system services are now provided by application-level servers accessed with RPC calls [Bershad et al. 89].

A protection domain switch on a PLB-based system requires changing only a single register, the PD-ID register in the processor. Protection rights for the individual pages do not have to be loaded or unloaded from the PLB; rights for the old domain are tagged with the domain identifier, and rights for the new domain can be faulted in lazily.

Domain switching on the page-group implementation involves purging the active page-group cache and loading in the page-groups for the new domain. The page-group cache can be reloaded lazily via protection faults, but for performance reasons it may be advantageous to explicitly reload it on domain switches.

4.2 Implementation Considerations

Protection checking in the page-group implementation requires two steps performed in sequence during the memory reference cycle: first, the TLB is indexed to obtain the page-group and access rights for the page; then the page-group cache is checked to determine if the current protection domain has access to the page-group. These cannot be performed in parallel, since the

[7] It is possible to allow only one domain to have read-write access to some subset of the pages using a combination of the access rights attached to the page and the write-disable bit attached to the domains' PIDs.

second lookup is dependent on the result of the first. The sequentiality may result in higher cycle times for processors using the page-group model, especially if the page-group cache is large.

The PLB requires only a single cache lookup, which provides the access rights. However, the tags being compared in the PLB are wider than either of the comparisons in the page-group implementation (the VPN and PD-ID are used in the lookup).

4.3 Granularity of Protection

As mentioned in Section 3, the PLB explicitly separates protection and translation.[8] One advantage of this separation is that the granularity of protection and translation can easily be different. There are various reasons why this might be desirable. Larger physical pages are attractive, because they improve TLB performance; with a larger page size each TLB entry covers more data. (This benefit must be balanced against an increase in memory fragmentation.) The units of protection, on the other hand, should be optimized to benefit the operating system and applications that are using the protection mechanism.

As noted above, many applications are utilizing hardware-based protection and page faults to detect when sharing or writing are taking place. Such detection would be more effective with a small granularity. The large page sizes on current architectures are too coarse-grained for many VM uses, causing, for example, an increase in false sharing for distributed virtual memory systems. Large page sizes would cause a similar problem with transactional locking; for this reason, the IBM 801 processor supports lock bits for every 128 bytes of memory to support database concurrency control [Chang & Mergen 88]. In a PLB-oriented system, protection and translation utilize different hardware structures; the PLB could be organized to provide protection control on sub-page units.

Protection pages that are larger than a single translation page are also useful. Many segments, such as stacks, temporary heaps and code segments, span many pages, yet have a constant protection value for the entire segment. For these segments, a single PLB entry could map the entire region, regardless of the number of physical pages it spans.[9] This is particularly useful in alleviating the duplication problem for shared segments. Duplicate PLB entries would still be required; however, since each entry maps a larger page, fewer PLB entries would be used.

Supporting multiple protection page sizes requires changes to the PLB. Currently, several commercial processors support multiple sized pages [Cyp 90, Dig 92, Hew 90, MIP 91]; the issues in designing a PLB that supports multiple protection page sizes are similar [Talluri et al. 92].

5 Related Work

The two models presented in this paper are representative of protection organizations for single address space systems, but they are by no means exclusive. Other protection architectures for single address space systems have ranged from no protection at all (as on the Xerox Dorado processors [Lampson & Pier 81]) to capability-based architectures [Fabry 74, Levy 84, Pinnow et al. 82].

Much of the groundwork for single address space operating systems was explored in earlier single address space systems, such as Pilot and Cedar [Redell et al. 80, Swinehart et al. 86]. These systems benefited from the use of a common address space, but they had a single protection domain as well as a single name space. These were dedicated-application or single-user systems that relied completely on language protection for safety. New hardware allows us to generalize this model to multiple protection domains, as in Psyche [Scott et al. 90].

Segmented systems (e.g., Multics [Bensoussan et al. 72] and the IBM 801 [Chang & Mergen 88]) support uniform sharing to some degree. The first phase of address translation on segmented architectures concatenates a global segment identifier with a segment offset, yielding a *long-form* address from a global virtual address space.[10] The segment identifier is retrieved from a segment table or a vector of segment registers associated with the current domain. The segment table is typically accessed with the high order bits of the domain specific address. Domains define a local view of portions of the global address space by overlaying global segments into their private segment registers. Uniform addressing in these systems is subject to some or all of the following restrictions: (1) cross-segment pointers are not supported, (2) multiple pointer forms must be treated differently by applications and (3) software must coordinate segment register usage to create an illusion of a single address space. The HP PA-RISC differs from other segmented architectures in that it allows applications to use long-form virtual addresses directly; thus the PA-RISC could be viewed as a single address space architecture. However, most software on the PA-RISC uses short-form addresses, because they are more compact and more efficient to dereference, and

[8] The page-group implementation can separate protection and translation as well. See section 3.2.2.

[9] The segment would have to be aligned to a power of two sized page, but this is a relatively minor problem.

[10] Multics did not generate a true long form address, instead referencing directly into the page table for the segment.

they permit backward compatibility with operating systems that rely on multiple virtual address spaces.

Okamoto et al. [Okamoto et al. 92] extend the domain-page model by mapping access to a page either by protection domain[11] or by the address where the program is currently executing; that is, page A can be marked so that it has read-only access by any thread that is currently executing code from page B.

A single address space system also bears some resemblance to earlier capability-based architectures, which relied on special hardware support to provide fine-grained protection within a global address space. In contrast, our approach uses traditional page-level protection structures and a 64-bit address space to provide global addressing. We believe that this approach can obtain many of the expected benefits of capability systems but without the costs and complexities of fine-grained hardware-based protection.

6 Summary

It is clear from the newest RISC architectures that the move to 64-bit virtual addressing is already well underway. A 64-bit address space can be used for more than running large programs: it increases our present addressability by nine (decimal) orders of magnitude, certainly more than will be needed by any single application for the foreseeable future. Instead it facilitates a fundamentally new operating system organization, the single address space system, which promotes efficient data sharing among protection domains.

Single address space operating systems also allow a reexamination of the conventional memory system architecture, and in particular, the caching and protection structures. In this paper, we have presented two protection models: the domain-page model and the page-group model. The domain-page model is implemented by a new memory system mechanism, the protection lookaside buffer. The PLB permits each protection domain to have its own set of access rights to any page, and permits the rights of any one domain to be changed without affecting the rights of other domains in the system. The PLB also completely separates protection and translation at the hardware level, allowing optimizations in both the granularity of protection and translation. The page-group model, implemented in the HP PA-RISC, clusters pages into groups, and allows domains to access entire groups, simplifying operations that take place on the group level.

We have looked at the tradeoffs between the two model implementations with respect to several operating system tasks. Until both systems are built, it will be hard to tell which model can take best advantage of single address space characteristics and which implementation provides better performance. Many of the answers will depend on how the systems will be used, i.e., which operations are most common.

Our current research explores these issues. We are building an operating system (Opal) that uses a single address space for all programs. Work is underway in compiling programs for a single address space and support for user-level segment servers which control the semantics and the protection for each segment. We will also evaluate the protection models presented here and determine appropriate configurations for their protection structures.

[11] They use the term thread, but the ideas are interchangeable in this context.

7 Acknowledgments

Hank Levy was a substantial contributor to the ideas in this paper and its initial presentation. John Wilkes provided detailed comments on an earlier draft and helped improve our understanding of the HP PA-RISC architecture. In addition, we would like to thank Miche Baker-Harvey, Robert Bedichek, Brian Bershad, Alex Klaiber, Ed Lazowska, Dylan McNamee, Sape Mullender, Bart Sears and the reviewers for their useful comments.

References

[Anderson et al. 91] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Proc. of the 4th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, Apr. 1991.

[Appel & Li 91] A. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of the 4th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–108, Apr. 1991.

[Appel et al. 88] A. Appel, J. Ellis, and K. Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proc. of the 1988 Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[Bensoussan et al. 72] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[Bershad et al. 89] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proc. of the 12th ACM*

Symposium on Operating System Principles, pages 102–113, Dec. 1989.

[Bershad et al. 90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-level interprocess communication for shared memory multiprocessors. Technical Report 90-05-07, Univ. of Washington, Department of Computer Science and Engineering, July 1990.

[Carter et al. 91] J. Carter, J. Bennett, and W.Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symposium on Operating System Principles*, pages 152–164, Oct. 1991.

[Carter et al. 92] J. Carter, A. Cox, D. Johnson, and W. Zwaenepoel. Distributed operating systems based on a protected global virtual address space. Technical Report Rice COMP TR92–186, Rice Univ., Department of Computer Science, Apr. 1992. Also appeared in the 3rd IEEE Workshop on Workstation Operating Systems, Apr. 1992.

[Chang & Mergen 88] A. Chang and M. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, Feb. 1988.

[Chase et al. 92a] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Univ. of Washington, Department of Computer Science and Engineering, Mar. 1992. Shorter version appeared as *Opal: A Single Address Space System for 64-Bit Architectures*, 3rd IEEE Workshop on Workstation Operating Systems, Apr. 1992.

[Chase et al. 92b] J. Chase, H. Levy, E. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1992.

[Cheng 87] R. Cheng. Virtual address caches in UNIX. In *Proc. of the Summer 1987 USENIX Technical Conference and Exhibition*, pages 217–224, 1987.

[Cyp 90] Cypress Semiconductor, San Jose, CA. *SPARC RISC User's Guide*, 2nd edition, Feb. 1990.

[Dig 81] Digital Equipment Corporation, Maynard, MA. *VAX Architecture Handbook*, 1981.

[Dig 92] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1992.

[Eppinger 89] J. Eppinger. *Virtual Memory Management for Transaction Processing Systems.* PhD dissertation, Carnegie Mellon Univ., Feb. 1989.

[Fabry 74] R. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Golub et al. 90] D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an application program. In *Proc. of the Summer USENIX*, pages 87–96, 1990.

[Goodman 87] J. Goodman. Coherency for multiprocessor virtual address caches. In *Proc. of the 2nd Conference on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, Apr. 1987.

[Groves & Oehler 90] R. Groves and R. Oehler. RISC system/6000 processor architecture. In M. Misra, editor, *IBM RISC System/6000 Technology*, pages 16–23. International Business Machines, 1990.

[Hew 90] Hewlett-Packard, Cupertino, CA. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1st edition, Nov. 1990.

[Int 89] Intel Corp., Santa Clara, CA. *i860 Microprocessor Programmer's Reference Manual*, 1989.

[Kane & Heinrich 92] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Knapp 85] V. Knapp. *Virtually Addressed Caches for Multiprogramming and Multiprocessing Environments.* PhD dissertation, Univ. of Washington, Department of Computer Science, June 1985.

[Koldinger et al. 91] E. Koldinger, H. Levy, J. Chase, and S. Eggers. The protection lookaside buffer: Efficient protection for single-address space computers. Technical Report 91-11-05, Univ. of Washington, Department of Computer Science and Engineering, Nov. 1991.

[Lam et al. 91] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. of the 4th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[Lampson & Pier 81] B. Lampson and K. Pier. A processor for a high-performance personal computer. Technical report, Xerox Palo Alto Research Center, Jan. 1981.

[Lee 89] R. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, Jan. 1989.

[Levy 84] H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[Li 86] K. Li. *Shared Virtual memory on Loosely Coupled Multiprocessors*. PhD dissertation, Yale Univ., Sept. 1986.

[Li et al. 90] K. Li, J. Naughton, and J. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proc. of the 2nd Conference on the Principles and Practice of Parallel Programming*, pages 79–88, Mar. 1990.

[MIP 91] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, 1st edition, 1991.

[Mullender & Tanenbaum 86] S. Mullender and A. Tanenbaum. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–299, 1986.

[Okamoto et al. 92] T. Okamoto, H. Segawa, S. Shin, H. Nozue, K. Maeda, and M. Saito. A microkernel architecture for next generation processors. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 83–94, Apr. 1992.

[Ousterhout 90] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *Proc. of the Summer 1990 USENIX*, pages 247–256, June 1990.

[Pinnow et al. 82] K. Pinnow, J. Ranweiler, and J. Miller. The IBM System/38: Object-oriented architecture. In D. Siewiorek, C. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, pages 537–540. McGraw-Hill, New York, 1982.

[Redell et al. 80] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.

[Rozier et al. 92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 39–69, Apr. 1992.

[Scalzi et al. 89] C. Scalzi, A. Ganex, and R. Schmalz. Enterprise Systems Architecture/370: An architecture for multiple virtual space access and authorization. *IBM Systems Journal*, 28(1):15–37, 1989.

[Scott et al. 90] M. Scott, T. LeBlanc, and B. Marsh. Multi-model parallel programming in Psyche. In *Proc. of the 2nd Conference on the Principles and Practice of Parallel Programming*, pages 70–78, Mar. 1990.

[Smith 82] A. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1982.

[SPA 92] SPARC International, Mountain View, CA. *SPARC Architecture Manual, Version 9*, 1992.

[Swinehart et al. 86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.

[Talluri et al. 92] M. Talluri, S. Kong, M. Hill, and D. Patterson. Tradeoffs in supporting two page sizes. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 415–424, May 1992.

[Wang et al. 89] W.-H. Wang, J.-L. Baer, and H. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proc. of the 16th International Symposium on Computer Architecture*, pages 140–148, May 1989.

[Wheeler & Bershad 92] B. Wheeler and B. Bershad. Consistency management for virtually indexed caches. In *Proc. of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992. these proceedings.

[Wilkes & Sears 92] J. Wilkes and B. Sears. A comparison of *protection lookaside buffers* and the PA-RISC protection architecture. Technical Report HPL–92–55, Hewlett-Packard Laboratories, Palo Alto, CA, Mar. 1992.

[Wood 90] D. Wood. *The Design and Evaluation of In-Cache Address Translation*. PhD dissertation, Univ. of CA, Berkeley, Mar. 1990.

[Wood et al. 87] D. Wood, S. Eggers, and G. Gibson. SPUR memory system architecture. Technical Report UCB/CSD 87/394, Univ. of CA, Berkeley, Computer Science Division, Dec. 1987.

[Young et al. 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. of the 11th ACM Symposium on Operating System Principles*, pages 63–76, Nov. 1987.