Segmented Addressing Solves the Virtual Cache Synonym Problem

Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland, College Park
blj@eng.umd.edu

Technical Report UMD-SCA-1997-01 December, 1997

ABSTRACT

If one is interested solely in processor speed, one must use virtually-indexed caches. The traditional purported weakness of virtual caches is their inability to support shared memory. Many implementations of shared memory are at odds with virtual caches—ASID aliasing and virtual-address aliasing (techniques used to provide shared memory) can cause false cache misses and/or give rise to data inconsistencies in a virtual cache, but are necessary features of many virtual memory implementations. By appropriately using a segmented architecture one can solve these problems. In this tech report we describe a virtual memory system developed for a segmented microarchitecture and present the following benefits derived from such an organization: (a) the need to flush virtual caches can be eliminated, (b) virtual cache consistency management can be eliminated, (c) page table space requirements can be cut in half by eliminating the need to replicate page table entries for shared pages, and (d) the virtual memory system can be made less complex because it does not have to deal with the virtual-cache synonym problem.

1 INTRODUCTION

Virtual caches allow faster processing in the common case because they do not require address translation when requested data is found in the caches. They are not used in many architectures despite their apparent simplicity because they have several potential pitfalls that need careful management [10, 16, 29]. In previous research on high clock-rate PowerPC designs, we discovered that the segmented memory-management architecture of the PowerPC works extremely well with a virtual cache organization and an appropriate virtual memory organization, eliminating the need for virtual-cache management and allowing the operating system to minimize the space requirements for the page table. Though it might seem obvious that segmentation can solve the problems of a virtual cache organization, we note that several contemporary microarchitectures use segmented addressing mechanisms—including PA-RISC [12], PowerPC [15], POWER2 [28], and x86 [17]—while only PA-RISC and POWER2 take advantage of a virtual cache.

Management of the virtual cache can be avoided entirely if sharing is implemented through the global segmented space. This gives the same benefits as single address-space operating systems (SASOS): if virtual-address aliasing (allowing processes to use different virtual addresses for the same physical data) is eliminated, then so is the virtual-cache *synonym problem* [10]. Thus, consistency management of the virtual cache can be eliminated by a simple operating-system organization. The advantage of a segmented approach (as opposed to a SASOS approach) is that by mapping virtual addresses to physical addresses in two steps, a segmented architecture divides virtual aliasing and the synonym problem into two orthogonal issues. Whereas they are linked in traditional architectures, they are unrelated in a segmented architecture; thus applications can map physical memory at multiple locations within their address spaces—they can use virtual address aliasing—without creating a synonym problem in the virtual cache.

In this tech report we describe a hardware/software organization that eliminates virtual-cache consistency problems, reduces the physical requirements of the page table, and eliminates contention in the TLB. Memory is shared at the granularity of segments and relocated at the granularity of pages. A single global page table maps the global virtual address space, and guarantees a one-to-one mapping between pages in the global space and pages in physical memory. Virtual-cache synonyms are thus eliminated, and the virtual memory system can be made less complicated. The global page table eliminates the need for multiple mappings to the same shared physical page, which reduces contention in the TLB. It also reduces the physical space requirements for the page table by a factor of two. We show that the segmented organization still supports the features expected of virtual memory, from sharing pages at different addresses and protections to complex operations such as copy-on-write.

2 BACKGROUND AND PERSPECTIVE

In this section we present the requirements of a memory-management design: it must support the functions of virtual memory as we have come to know them. We review the characteristics of segmented architectures, then the fundamental problems of virtual caches and shared memory.

2.1 Requirements

The basic functions of virtual memory are well-known [7]. One is to create a virtual-machine environment for every process, which (among other things) allows text, data, and stack regions to begin at statically known locations in all processes without fear of conflict. Another is demand-paging—setting a finer granularity for process residence than an entire address space, thereby allowing a process to execute as long as a single page is memory-resident. Today's expectations of virtual memory extend its original semantics and now include the following additional requirements:

Virtual-Address Aliasing: Processes must be able to map shared objects at (multiple) different virtual addresses.

Protection Aliasing: Processes must be able to map shared objects using different protections.

Virtual Caches: Fast systems require virtual caches. The operating system should not have to flush a virtual cache to ensure data consistency.

The traditional virtual memory mechanism is not well equipped to deal with these requirements. In order to distinguish between con-

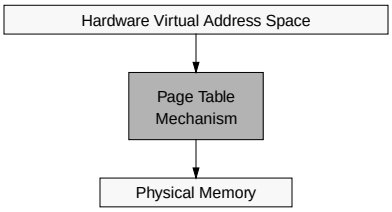


Figure 1: The single indirection of traditional memory-management organizations.

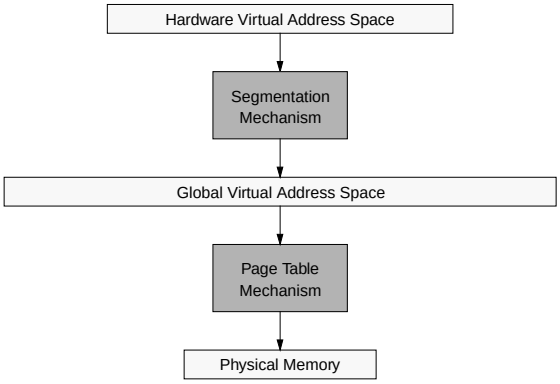


Figure 2: Multiple levels of indirection in a segmented memory-management organization.

texts, the traditional architecture uses *address space identifiers (ASIDs)*, which by definition keep processes from sharing pages easily. A typical ASID mechanism assigns one identifier to every page, and one identifier to every process; therefore multiple page-table and TLB entries may be required to allow multiple processes/ASIDs to map a given page—this can fill the TLB with duplicate entries mapping the same physical page, thereby reducing the hit rate of the TLB significantly [19].

2.2 Segmented Architectures

Traditional virtual memory systems provide a mapping between process address spaces and physical memory. SASOS designs place all processes in a single address space and map this large space onto physical memory. Both can be represented as a single level of mapping, as shown in Figure 1. These organizations manage a single level of indirection between virtual and physical memory; they combine into a single mechanism the two primary functions of virtual memory: that of providing a virtual operating environment and that of demand-paging on a small (page-sized) granularity. Segmentation allows one to provide these two distinct functions through two distinct mechanisms: two levels of indirection between the virtual address space and main memory. The first level of indirection supports the virtual operating environment and allows processes to locate objects at arbitrary segment-aligned addresses. The second level of indirection provides movement of data between physical memory and backing store at the granularity of pages.

This organization is shown in Figure 2. Processes operate in the top layer. A process sees a contiguous address space that stretches from 0x00000000 to 0xFFFFFFFF, inclusive (we will restrict ourselves to using 32-bit examples in this report for the purposes of brevity and clarity). The process address space is transparently mapped onto the middle layer at the granularity of hardware segments, identified by the top bits of the user address. The segments that make up a user-level process may in actuality be scattered throughout the global space and may very well not be contiguous. However, the addresses generated by the process do not reach the cache; they are mapped onto the global space first. The cache and TLB see global addresses only. There is therefore no critical path between address generation and a virtual cache lookup except for the segmentation mechanism—and if the segment size is larger than the L1 cache size the segment bits are not used in the cache lookup, thus the segmentation mechanism can run in parallel with the cache access.

Segmented systems have a long history. Multics, one of the earliest segmented operating systems, used a segmented/paged architecture, the GE 645 [23]. This architecture was similar to the Intel Pentium memory management organization [17] in that both the GE 645 and the Intel Pentium support segments of variable size. An important point is that the Pentium's global space is no larger than an individual user-level address space; processes generate 32-bit addresses that are extended by 16-bit segment selectors. The selectors are used by hardware to index into one of two descriptor tables that produce a base address for the segment corresponding to the segment selector. This base address is added to the 32-bit virtual address produced by the application to form a global 32-bit virtual address. The segments can range from a single byte to 4GB in size. There is no mechanism to prevent different segments from overlapping one another in the global 4GB space. The segment selectors are produced indirectly. At any given time a process can reference six of its segments; selectors for these six segments are stored in six segment registers that are referenced by context. One segment register is referenced implicitly by executing instructions; it contains a segment selector for the current code segment. Another segment register holds the segment selector for the stack. The other four are used for data segments, and a process can specify which of the segment registers to use for different loads and stores. One of the data segment registers is implicitly referenced for all string instructions, unless explicitly overridden.

In contrast, the IBM 801 [2] introduced a fixed-size segmented architecture that continued through to the POWER and PowerPC architectures [15, 21, 28]. The PowerPC memory management design maps user addresses onto a global flat address space much larger than each per-process address space. Segments are 256MB contiguous regions of virtual space, and (in a 32-bit implementation) 16 segments make up an application's address space. Programs generate 32-bit "effective addresses." The top four bits of the effective address select a segment identifier from a set of 16 hardware segment registers. The segment identifier is concatenated with the bottom 28 bits of the effective address to form an extended virtual address. It is this extended virtual address space that is mapped by the TLBs and page table. For brevity and clarity, we will restrict ourselves to using fixed-size segmentation for examples throughout this report.

Segmented architectures need not use address space identifiers; address space protection is guaranteed by the segmentation mechanism.[1] If two processes have the same segment identifier, they share that virtual segment by definition. Similarly, if a process has a given segment identifier in several of its segment registers, it has mapped the segment into its address space at multiple locations. The operating system can enforce inter-process protection by disallowing shared segments identifiers, or it can share memory between processes by overlapping segment identifiers.

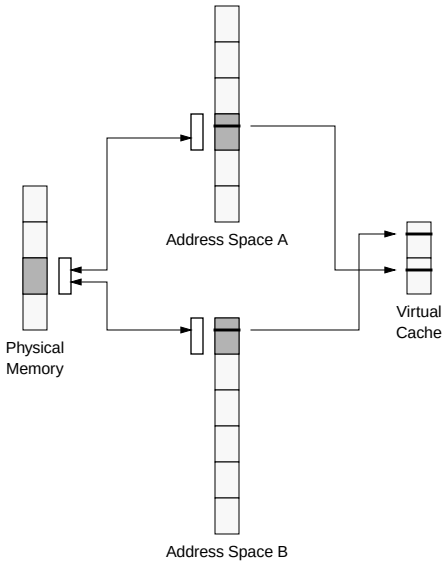


Figure 3: The synonym problem of virtual caches. If two processes are allowed to map physical pages at arbitrary locations in their virtual address spaces, inconsistencies can occur in a virtually indexed cache.

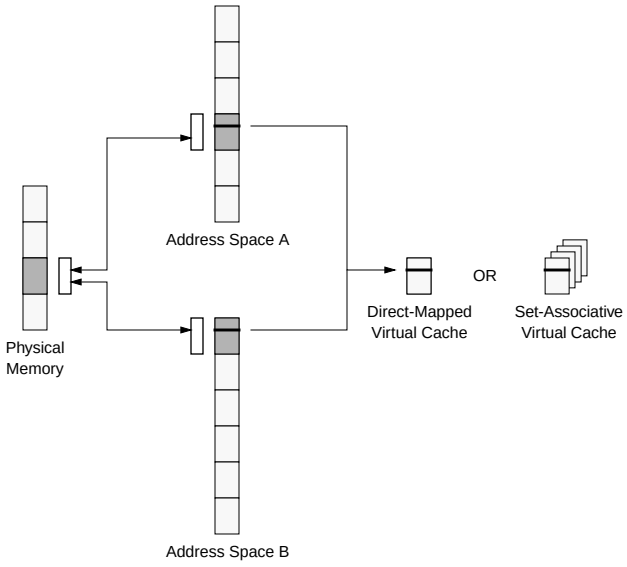


Figure 4: Simple hardware solution to page aliasing. If the cache is no larger than the page size and direct-mapped, then no aliasing can occur. Set-associative caches can be used, provided they have physical tags.

2.3 The Consistency Problem of Virtual Caches

A virtually indexed cache allows the processor to use the untranslated virtual address as an index. This removes the TLB from the critical path, allowing shorter cycle times and/or a reduced number of pipeline stages. However, it introduces the possibility of data integrity problems occurring when two processes write to the same physical location through different virtual addresses; if the pages align differently in the cache, erroneous results can occur. This is called the virtual cache *synonym problem* [10]. The problem is illustrated in Figure 3; a shared physical page maps to different locations in two different process address spaces. The virtual cache is larger than a page, so the pages map to different locations in the virtual cache. As far as the cache is concerned, these are two different pages, not two different views of the same page. Thus, if the processes write to the page at the same time, two different values will be found in the cache.

Hardware Solutions

The synonym problem has been solved in hardware using schemes such as dual tag sets [10] or back-pointers [27], but these require complex hardware and control logic that can impede high clock rates. One can also restrict the size of the cache to the page size, or, in the case of set-associative caches, similarly restrict the size of each *cache column* (the size of the cache divided by its associativity, for lack of a better term) to the size of one page. This is illustrated in Figure 4; it is the solution used in the PowerPC and Pentium processors. The disadvantages are the limitation in cache size and the increased access time of a set-associative cache. For example, the Pentium and PowerPC architectures must increase associativity in order to increase the size of their on-chip caches and both architectures have reached 8-way set-associative cache designs. Physically-tagged caches guarantee consistency within a single cache set, but this only applies when the virtual synonyms map to the same set.

Software Solutions

Wheeler & Bershad describe a state-machine approach to reduce the number of cache flushes required to guarantee consistency [29]. The mechanism allows a page to be mapped anywhere in an address space, at some cost in implementation complexity. The aliasing problem can also be solved through operating system policy, as shown in Figure 5. For example, the SPUR project disallowed virtual aliases altogether [13]. Similarly, OS/2 locates all shared segments at the same address in all processes [6]. This reduces the amount of virtual memory available to each process, whether the process uses the shared segments or not; however, it eliminates the aliasing problem entirely and allows pointers to be shared between address spaces. SunOS requires shared pages to be aligned on cache-size boundaries [11], allowing physical pages to be mapped into address spaces at almost any location but ensuring that virtual aliases align in the cache. Note that the SunOS scheme only solves the problem for direct-mapped virtual caches or set-associative virtual caches with physical tags; shared data can still exist in two different blocks of the same set in an associative, virtually-indexed, virtually-tagged cache. Single address space operating systems such as Opal [4, 5] or Psyche [24] solve the problem by eliminating the concept of individual per-process address spaces entirely. Like OS/2, they define a one-to-one correspondence of virtual to physical addresses and in doing so allow pointers to be freely shared across process boundaries.

1. Page-level protection is a different thing entirely; whereas address-space protection is intended to keep processes from accessing each other's data, page-level protection is intended to protect pages from misuse. For instance, page-level protection keeps processes from writing to text pages by marking them read-only, etc. Page-level protection is typically supported through a TLB but could be supported on a larger granularity through the segmentation mechanism. However there is nothing intrinsic to segments that provides page-level protection, whereas address-space protection *is* intrinsic to their nature.

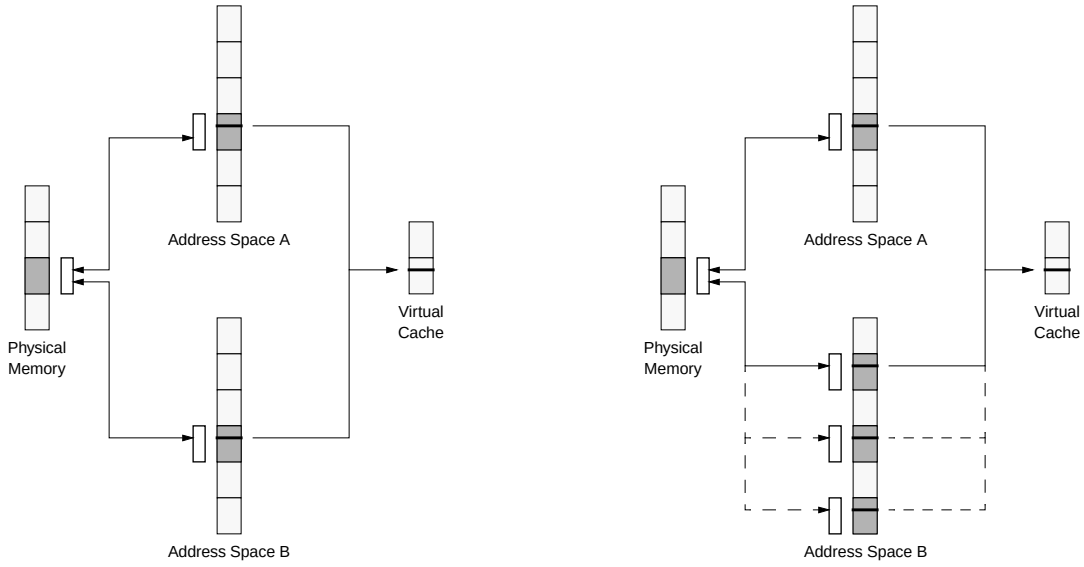


Figure 5: Synonym problem solved by operating system policy. OS/2 and the operating system for the SPUR processor guarantee the consistency of shared data by mandating that shared segments map into every process at the same virtual location. SunOS guarantees data consistency by aligning shared pages on cache-size boundaries. The bottom few bits of all virtual page numbers mapped to any given physical page will be identical, and the pages will map to the same location in the cache. Note this works best with a direct-mapped cache.

3 SHARED MEMORY VS. THE VIRTUAL CACHE

This section describes some of the higher-level problems that arise when implementing shared memory on virtual cache organizations. As described above, virtual caches have an inherent consistency problem; this problem tends to conflict with the mechanisms used to implement shared memory.

3.1 The Problems with Virtual-Address Aliasing

Virtual-address aliasing is a necessary evil; it is useful, yet it breaks many simple models. Its usefulness outweighs its problems, therefore future memory management systems must continue to support it.

Virtual-Address Aliasing is Necessary

Most of the software solutions for the virtual-cache synonym problem address the consistency problem by limiting the choices where a process can map a physical page in its virtual space. In some cases, the number of choices is reduced to one; the page is mapped at one globally unique location or it is not mapped at all. While disallowing virtual aliases would seem to be a simple and elegant way to solve the virtual cache consistency problem, it creates another headache for operating systems—virtual fragmentation.

When a global shared region is garbage-collected, the region cannot help but become fragmented. This is a problem: whereas de-fragmentation (compaction) of disk space or physically-addressed memory is as simple as relocating pages or blocks, virtually addressed regions cannot be easily relocated. They are location-dependent; all pointers referencing the locations must also be changed. This is not a trivial task and it is not clear that it can be done at all. Thus, a system that forces all processes to use the same virtual address for the same physical data will have a fragmented shared region that cannot be de-fragmented without enormous effort. Depending on the amount of sharing this could mean a monotonically increasing shared region, which would be inimical to a 24x7 environment, i.e. one that is intended to be operative 24 hours a day, seven days a week. 64-bit SASOS implementations avoid this problem by using a global shared region that is so enormous it would take a very long time to become overrun by fragmentation. Other systems [8, 9] avoid the problem by dividing a fixed-size shared region into uniform sections and/or turning down requests for more shared memory if all sections are in use.

Virtual-Address Aliasing is Detrimental

There are two issues associated with global addresses: one is that they eliminate virtual synonyms, the other is that they allow shared pointers. If a system requires global addressing, then shared regions run the risk of fragmentation, but applications are allowed to place self-referential pointers in the shared regions without having to *swizzle* [22] between address spaces. However, as suggested above, this requirement is too rigid; shared memory should be linked into address spaces at any (page-aligned) address, even though allowing virtual aliasing can reduce the ability to store pointers in the shared regions.

Figure 6 illustrates the problem: processes A and Z use different names for the shared data, and using each other's pointers will lead to confusion. This problem arises because the operating system was allowed or even instructed by the processes to place the shared region at different virtual addresses within each of the two address spaces. Using different addresses is not problematic until processes attempt to share pointers that reference data within the shared region. In this example, the shared region contains a binary tree that uses self-referential pointers that are not consistent because the shared region is located at different virtual addresses in each address space.

It is clear that unless processes use the same virtual address for the same data, there is little the operating system can do besides swizzle the pointers or force apps to use *base+offset* addressing schemes in shared regions. Nonetheless, we have come to expect support for virtual aliasing, therefore it is a requirement that a system support it.

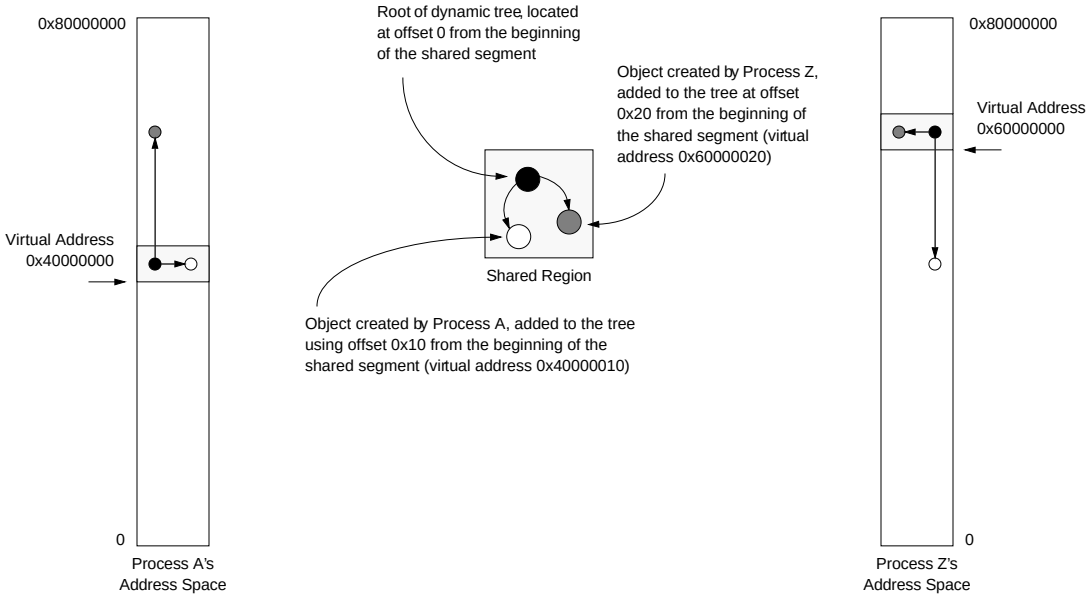


Figure 6: The problem with allowing processes to map shared data at different virtual addresses.

3.2 The Problems with Address-Space Identifiers

Sharing memory also causes performance problems at the same time that it reduces the need for physical memory. The problem has been mentioned earlier—the use of ASIDs for address space protection makes sharing difficult, requiring multiple page table and TLB entries for different aliases to the same physical page. Khalidi and Talluri describe the problem:

> Each alias traditionally requires separate page table and translation lookaside buffer (TLB) entries that contain identical translation information. In systems with many aliases, this results in significant memory demand for storing page tables and unnecessary TLB misses on context switches. [Addressing these problems] reduces the number of user TLB misses by up to 50% in a 256-entry fully-associative TLB and a 4096-entry level-two TLB. The memory used to store hashed page tables is dramatically reduced by requiring a single page table entry instead of separate page table entries for hundreds of aliases to a physical page, [using] 97% less memory. [19]

Since ASIDs identify virtual pages with the processes that own them, mapping information necessarily includes an ASID. However, this ensures that for every shared page there are multiple entries in the page tables, since each differs by at least the ASID. This redundant mapping information requires more space in the page tables, and it floods the TLB with superfluous entries; for instance, if the average number of mappings per page were two, the effective size of the TLB would be cut in half. In fact, Khalidi & Talluri report the average number of mappings per page on an idle system to be 2.3, and they report a decrease by 50% of TLB misses when the superfluous-PTE problem is eliminated. A scheme that addresses this problem can reduce TLB contention as well as physical memory requirements.

The problem can be solved by a global bit in the TLB entry, which identifies a virtual page as belonging to no ASID in particular; therefore, every ASID will successfully match. This reduces the number of TLB entries required to map a shared page to exactly one, however the scheme introduces additional problems. The use of a global bit essentially circumvents the protection mechanism and thus requires flushing the TLB of shared entries on context switch, as the shared regions are unprotected. Moreover, it does not allow a shared page to be mapped at different virtual addresses, or with different protections. Using a global-bit mechanism is clearly unsuitable for supporting sharing if shared memory is to be used often.

If we eliminate the TLB, then the ASID, or something equivalent to distinguish between different contexts, will be required in the cache line. The use of ASIDs for protection causes the same problem but in a new setting. Now, if two processes share the same region of data, the data will be tagged by one ASID and if the wrong process tries to access the data that is in the cache, it will see an apparent cache miss simply because the data is tagged by the ASID of the other process. Again, using a global-bit to marked shared cache lines makes them unprotected against other processes and so the cache lines must be flushed on context switch. This is potentially much more expensive than flushing mappings from the TLB because the granularity for flushing the cache is usually a cache line, requiring many operations to flush an entire page.

4 THE "VIRTUE" OF SEGMENTATION

One obvious solution to the synonym and shared memory problems is to use global naming, as in a SASOS implementation, so that every physical address corresponds to exactly one virtual location. This eliminates redundancy of page table entries for any given physical page, with significant performance and space savings. However, it does not allow processes to map objects at multiple locations within their address spaces—all processes must use the same name for the same data, which conflicts with our stated requirement of allowing processes to map objects at different virtual addresses, and at multiple locations within their address space.

A segmented architecture avoids this conflict; segmentation divides virtual aliasing and the synonym problem into two orthogonal issues. A one-to-one mapping from global space to physical space can

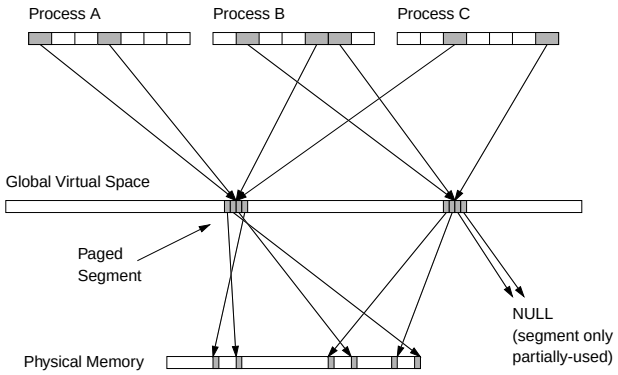


Figure 7: The use of segments to provide virtual address aliasing.

be maintained—thereby eliminating the synonym problem—while supporting virtual aliases by independently mapping segments in process address spaces onto segments in the global space. Such an organization is illustrated in Figure 7. In the figure, three processes share two different segments, and have mapped the segments into arbitrary segment slots. Two of the processes have mapped the same segment at multiple locations in their address spaces. The page table maps the segments onto physical memory at the granularity of pages. If the mapping of global pages to physical pages is one-to-one, there are no virtual-cache synonym problems.

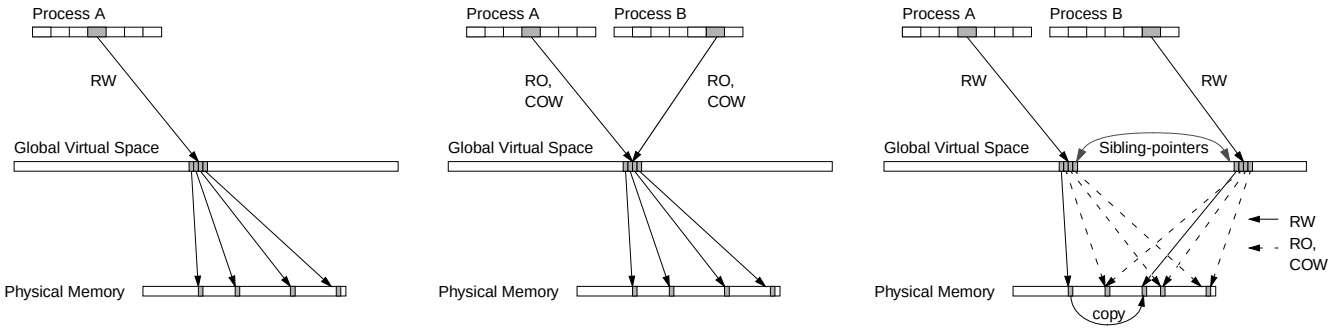
When the synonym problem is eliminated, there is no longer a need to flush a virtual cache or a TLB for consistency reasons. The only time flushing is required is when virtual segments are re-mapped to new physical pages, such as when the operating system runs out of unused segment identifiers and needs to re-use old ones; if there is any data left in the caches or TLB tagged by the old virtual address, data inconsistencies can occur. Direct Memory Access (DMA) also requires flushing of the affected region before a transaction, as an I/O controller does not know whether the data it overwrites is currently in a virtual cache.

Applications may map objects using different protections; the same object can be mapped into different address spaces with different segment-level protections, or mapped into the same address space at different locations with different protections. To illustrate, Figure 8 shows an example of one possible copy-on-write implementation. It assumes hardware support for protection in both the segmentation mechanism (segment granularity) and the TLB (page granularity), as in the Pentium [17]. In the first step, a process maps an object with read-write permissions. The object is located in a single segment, and the permissions are associated with the segment. In the second step, the process grants access to another process, which maps the object into its address space at another virtual address. The two share the object read-only, copy-on-write. In the third step, Process B has written to the object and the written page has been copied. At this point there are two choices. One is to copy the entire object, which could be many pages, into a new set of page frames. This would allow both processes to map their copies of the object read-write. Alternately, one could stop after the first page (the written page) to delay copying the rest until absolutely necessary, maintaining reference pointers until the entire object is copied; this scenario is shown in Figure 8(c). At this point, both processes have read-write access to the object at the segment level, but this could fail at the page-access level. If either process writes to the read-only pages they will be copied. The disadvantage of this scheme is that it requires sibling-pointers to the original mappings so that if a copy is performed, access to the original page can be changed to read-write. An alternate organization is shown in Figure 9, in which there is no hardware support for page-level protection. Here, we need sibling-pointers at the segment level. As in the previous example, we can avoid chains of sibling-pointers by simply copying the entire object when it is first written.

The issue becomes one of segment granularity. If segments represent the granularity of sharing and data placement within an address space (but not the granularity of data movement between memory and disk), they must be numerous and small. They should still be larger than the L1 cache, to keep the critical path between address generation and cache access clear. Therefore the address space should be divided into a large number of small segments, for instance 1024 4MB segments, 4096 1MB segments, 16,384 256KB segments, etc.

5 DISCUSSION

In this section, we discuss a page-table mechanism that supports the required virtual memory features. We compare its space requirements against a more traditional organization, and we briefly describe how the page table would work on several commercial architectures.



(a) Single process owns an object, placed in a segment

(b) Process shares object with another process via shared segments

(c) Second process writes to first page in segment, gets own copy

Figure 8: Copy-on-write in a segmented architecture.

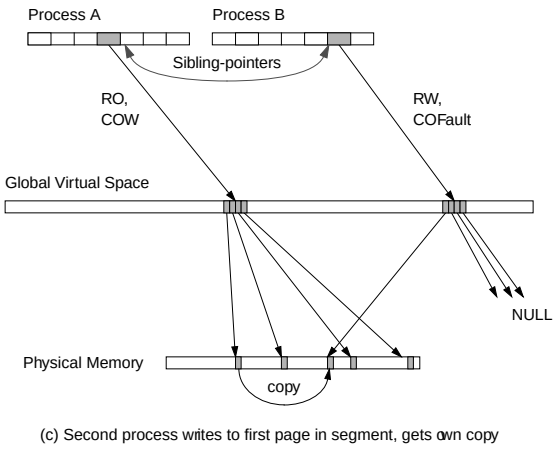


Figure 9: An alternate implementation of copy-on-write.

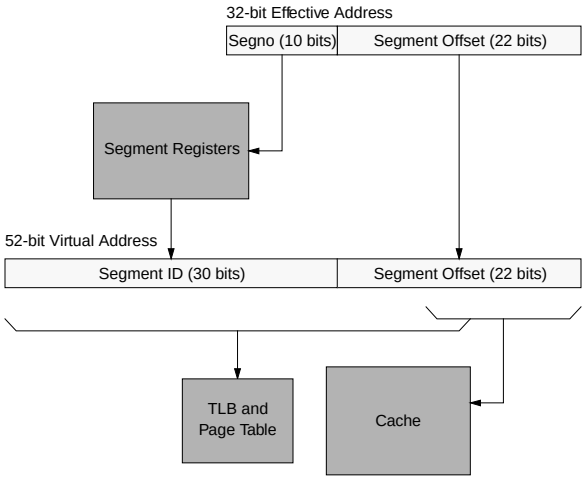


Figure 10: Segmentation mechanism used in Discussion.

5.1 Global Page Table

The segmentation mechanism suggests a two-tiered page table, one table mapping global pages to physical page frames, and a per-process table mapping segments onto the global space. For the purposes of this discussion, we assume PowerPC-like segmentation based on the top bits of the address space, a 32-bit effective address space, a 52-bit virtual address space, and a 4KB page size. Figure 10 illustrates the mechanism. We assume that the segmentation granularity is 4MB; the 4GB address space is divided into 1024 segments. This simplifies the design and should make the discussion clear; a four-byte page-table entry (PTE) can map a 4KB page, which can in turn map an entire 4MB segment.

Our page table organization uses a single global table to map the entire 52-bit segmented virtual address space. Any single process is mapped onto at most 2GB of this global space, and so it requires at most 2MB of the global table at any given moment. The page-table organization is pictured in Figure 11; it shows the global table as a 4TB linear structure at the top of the global virtual address space, composed of 2^{30} 4KB PTE pages that each map a 4MB segment. Each user process has a 2MB address space (as in MIPS [18]), which can be mapped by 512 PTE pages in the global page table. These 512 PTE pages make up a *user page table*, a disjunct set of virtual pages at the top of the global address space. These 512 pages can be mapped by 512 PTEs—a collective structure small enough to wire down in physical memory for every running process (2KB). This structure is termed the *user root page table* in the figure. In addition, there must be a table of 512 segment IDs for every process, a 4KB structure, since each segment ID is 30 bits plus protection and "mode" bits such as copy-on-write. The size of this structure can be cut in half if we can encode the protection and mode information in two bits.

This hardware/software organization satisfies our requirements. Each process has a virtual-machine environment in which to operate; the segment mechanism maps the process address space onto the global space transparently. Demand-paging is handled by the global page table. Processes map objects at arbitrary segment-aligned addresses in their address spaces, and can map objects at multiple locations if they wish. Processes can also map objects with different protections, as long as the segmentation mechanism supports protection bits for each segment. And, as we have described, the global page table maintains a one-to-one mapping between global pages and physical page frames, therefore the virtual-cache synonym problem disappears. Though we have not stated it as a requirement, the virtual-memory fragmentation problem is also solved by this organization; there is no restriction on where an object is placed in the global space, and there is no restriction on where an object is placed in a process address space.

5.2 Page Table Efficiency

The theoretical minimum page table size is 0.1% of working set size, assuming 4KB pages, 4B page table entries, and fully-populated page table pages. However, most virtual memory organizations do not share PTEs when pages are shared; for every shared page there is more than one PTE in the page table. Khalidi & Talluri show that these extra PTEs can increase the page table size by an order of magnitude or more [19].

We compare the size of the global page table to the theoretical minimum size of a traditional page table. Khalidi & Talluri report that the average number of mappings per page on an idle system is 2.3, and the average number of mappings to *shared* pages is 27. This implies that the ratio of private to shared pages in an average system is 19:1 or that 5% of a typical system's pages are shared pages.[2] These are the figures we use in our calculations. The overhead of a traditional page table (one in which there must be multiple PTEs for multiple mappings to the same page) can be calculated as

$$\frac{(\text{number of PTEs})(\text{size of PTE})}{(\text{number of pages})(\text{size of page})} = \frac{(p + 27s)4}{(p + s)4096} = \frac{(p + 27s)}{(p + s)1024}$$

where p is the number of private (non-shared) pages in the system, and s is the number of shared pages in the system. We assume a ratio of 1024:1 between page size and PTE size. This represents the theoretical minimum overhead since it does not take into account partially-filled PTE pages. For every shared page there is on average 27

2. The average number of mappings per page is the total number of mappings in the system divided by the total number of pages, or $\frac{p + 27s}{p + s} = 2.3$, which yields a p:s ratio of 19:1.

processes mapping it, therefore the page table requires 27 PTEs for every shared page. The overhead is in terms of the physical-page working set size; the fraction of physical memory required to map a certain number of physical pages. As the percentage of sharing increases, the number of physical pages does not increase, but the number of PTEs in the page table does increase.

The global page table overhead is calculated the same way, except that PTEs are not duplicated when pages are shared. Thus, the overhead of the table is a constant:

$$\frac{(p+s)4}{(p+s)4096} = \frac{1}{1024}$$

Clearly, the global page table is smaller than a traditional page table, and it approaches the minimum size necessary to map a given amount of physical memory. Figure 12 shows the overhead of each page table organization as the level of sharing in a system changes. The x-axis represents the degree of sharing in a system, as the number of pages that are shared ($s/(p+s)$). The y-axis represents the overhead of the page table, as the size of the page table divided by the total size of the data pages. In an average system, where 5% of the pages are shared, we should expect to use less than half the space required by a traditional page table organization.

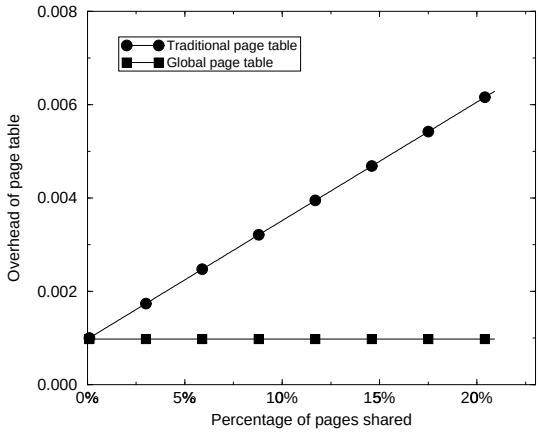


Figure 12: Comparison of page table space requirements.

5.3 Portability

Since sharing is on a segment basis, we would like fine-grained segmentation, which is unavailable in most commercial processors. Therefore any segmented architecture could benefit from this organization, but a granularity of large segments might make the system less useful. Also, the lack of protection bits associated with segments in most architectures (including PA-RISC and PowerPC) means that processes will not be able to share segments with different protections. In architectures lacking segment protection bits, all mappings to an object will be protected through the page table, not through segments. Since the page table does not distinguish between different mappings to the same virtual segment, all mappings to a given virtual segment will have the same protections.

Design for PowerPC

The PowerPC organization would differ in two additional ways. As described above, since there are no protection bits associated with segments the system would not allow different mappings to the same object with different protections. A copy-on-write mechanism could still be implemented, however, through the global page table—by marking individual pages as read-only, copy-on-write. This scheme would require back-pointers to determine the multiple aliases to a physical page, so that they could be re-mapped when the page is copied. Second, since there are only sixteen segments available, only 16 entries would be needed in the segment table—it could therefore fit in

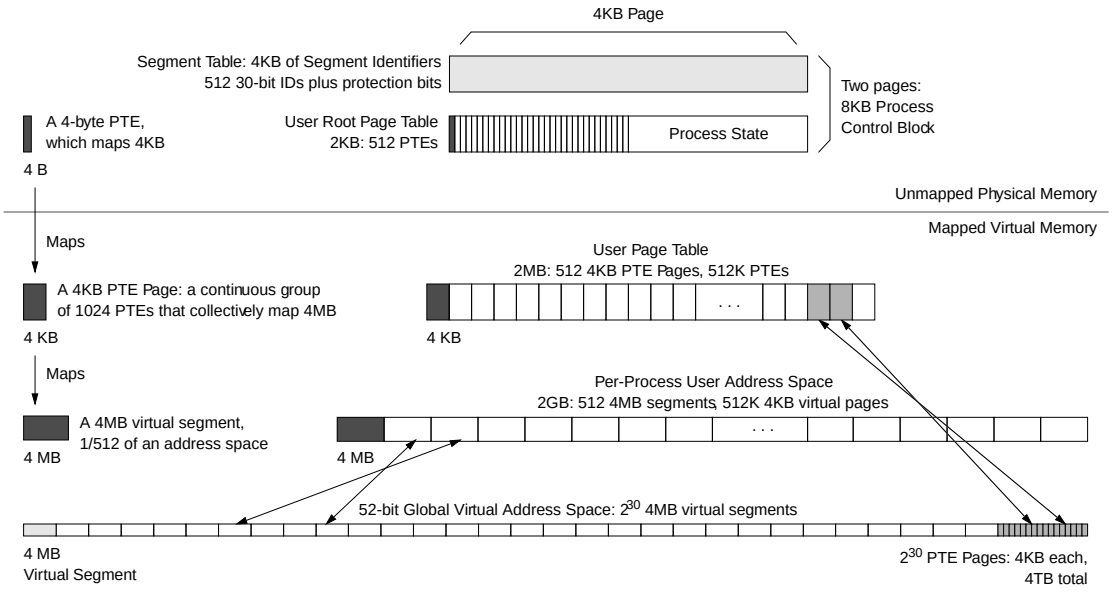


Figure 11: A global page table organization.

with process state, and so the process control block would be half as large.

The PowerPC hardware-defined inverted page table is not a true page table (not all mappings are guaranteed to reside in the page table), but rather a software TLB [1]. Therefore we can use any page table organization we want.

Design for PA-RISC

The PA-RISC architecture [12] has a facility similar to PowerPC segments, a set of 8 *space registers*, that maps a 32-bit address onto a larger (64-bit) virtual address space. User applications may load values into the space registers at will. Protection is guaranteed by restricting access to some of the registers and by the use of protection IDs, similar to ASIDs. The processor maintains four protection IDs per process and if any protection ID matches the access ID of a page, access is granted. Note that a process may not alter the contents of the protection-ID registers.

PA-RISC supports the concept of a global address space and a global page table though the space registers. In fact, researchers at Hewlett-Packard have stressed that this is the best way to share memory on PA-RISC:

> [One can] take advantage of the global virtual address space provided by the PA-RISC to eliminate all remapping of text pages shared between tasks. The idea is to map the text object to a single range of virtual addresses in the global virtual address space, and to have all tasks sharing the text object to access it using that same range of virtual addresses. This not only eliminates virtual address aliasing at the hardware level, it eliminates it at all levels. This is the "right" way to share memory on PA-RISC. [3]

Unlike PowerPC, PA-RISC does not specify a page table organization for the operating system, though HP-UX has traditionally used an inverted page table [14]. One can therefore use a global page table organization. The difference is that the user root page tables would not be as simple as in our generic design, in which a process only has access to a 2GB window at any time and so the maximum size of the user root page table is 2KB. PA-RISC allows processes to extend their address space at will without operating system intervention, by placing space IDs into the space registers—subject to the access-ID constraints. This allows the process to swap global *spaces* in and out of its address space at will, implying that the size of the wired-down user root page table can grow without bounds. This can be solved by another level of indirection, where the user root page table is a dynamic data structure; the disadvantage is that user root page table access becomes slower.

Design for Pentium

The Pentium [17] memory management architecture corresponds very closely to the needs of our generic design. It maps 32-bit addresses onto a global 4GB "linear" address space. Besides the limitation of a small global address space, the architecture is very nearly identical to the hardware described in this section. The mapping from user address space to global linear space is made before cache access. The segments have associated protection independent of the underlying page protection. Every feature of our addressing scheme is supported.

However, the architecture does not take full advantage of its own design. The cache is effectively virtually indexed, but only by constraining the cache index to be identical to the 4KB page size. There are six segment registers and they are addressed according to the context in which they are used—there is only one register for code segments, one register for stack segments, etc. The segment registers are therefore much less flexible than PowerPC segments, and they could have a lower hit rate. The segmentation mechanism is not used by many systems because a process requiring numerous segments will frequently reference memory to reload segment registers. Pentium performance and flexibility could improve dramatically if the caches were virtual and larger (allow the cache index to be larger than the page size) and if the segment registers were less context-oriented and more numerous.

The Pentium segment registers include one for the stack, one for code, and four for data—one of which is used by string instructions. An application can reference 8192 local (per-process) segments, and 8191 global segments. Segment sizes are software-defined and can range from 1 byte to 4 gigabytes. Each segment has a four-bit protection ID associated with it encoding *read-only*, *read/write*, *execute-only*, etc. The protection ID also encodes information such as whether the segment size is allowed to change.

The system supports a global 4MB page table that maps the 4GB shared linear address space. The main problem is the relatively small global address space. Four gigabytes is not much room in which to work, which could cause the memory allocation logic to become complicated. On the other hand, a benefit is that the segmentation mechanism would become an address space protection mechanism. This is similar to the use of segments in the PowerPC architecture. A set of segments uniquely identifies a process address space; full protection is guaranteed by not overlapping segments. Any segment that is not shared by another process is protected from all other processes. The advantage is that one classical argument against the Intel architecture—that its lack of ASIDs is a performance drag by requiring TLB flushes on context switch—disappears. Since the TLB maps addresses from the global linear space, no flushing would be necessary.

Design for 64-bit Architectures

The virtual-cache synonym problem does not automatically go away with 64-bit addressing, unless one uses the 64-bit address space for a SASOS organization. As described earlier, this has some disadvantages and does not support all the required virtual memory features. A segmentation mechanism is still necessary in a 64-bit architecture in order to satisfy all of our stated requirements.

Note that the 64-bit PowerPC implementation has the same fixed segment size as the 32-bit implementation: 256 MB. The architecture maps a 64-bit user address space onto an 80-bit global virtual space, at a 256MB granularity, then maps the global space onto physical memory at a page granularity. This satisfies one of the requirements discussed earlier: that the architecture offer numerous segments for sharing. The 64-bit PowerPC offers 2^{36} segments per address space, which should be enough for even an operating system like Windows, in which there are thousands of shared libraries. The segment mapping in this implementation is through a *segment lookaside buffer*, an associative cache managed like a TLB, as opposed to the lookup table found in the 32-bit implementation.

The primary difference when moving to a 64-bit machine is the structure of the page table. The page table need not be linear, or even hierarchical; it simply must map the global space and provide a guarantee that global pages map one-to-one onto the physical memory. Therefore several organizations are possible, including the hierarchical table of OSF/1 on Alpha [25], a guarded page table [20], or an inverted page table [14], including the variant described by Talluri, et al. [26].

6 CONCLUSIONS

One can employ a virtually indexed cache in order to meet the memory requirements of a high-speed processor and avoid the potential slowdown of address translation. However, virtual caches do not appear in the majority of today's processors. Virtual caches help achieve fast clock speeds but have traditionally been left out of microprocessor architectures because the naming dichotomy between the cache and main memory creates the potential for data inconsistencies, requiring significant management overhead. A segmented architecture adds another level of naming and allows a system to use a virtual cache organization without explicit consistency management, as long as the operating system ensures a one-to-one mapping of pages between the segmented address space and physical memory.

7 REFERENCES

[1] K. Bala, M. F. Kaashoek, and W. E. Weihl. "Software prefetching and caching for translation lookaside buffers." In *Proc. First USENIX Symposium on Operating Systems Design and Implementation (OSDI-1)*, November 1994, pp. 243–253.

[2] A. Chang and M. F. Mergen. "801 storage: Architecture and programming." *ACM Transactions on Computer Systems*, vol. 6, no. 1, February 1988.

[3] C. Chao, M. Mackey, and B. Sears. "Mach on a virtually addressed cache architecture." In *USENIX Mach Workshop*, October 1990.

[4] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. "How to use a 64-bit virtual address space." Tech. Rep. 92-03-02, University of Washington, March 1992.

[5] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. "Lightweight shared objects in a 64-bit operating system." Tech. Rep. 92-03-09, University of Washington, March 1992.

[6] H. Deitel. *Inside OS/2*. Addison-Wesley, Reading MA, 1990.

[7] P. J. Denning. "Virtual memory." *Computing Surveys*, vol. 2, no. 3, pp. 153–189, September 1970.

[8] P. Druschel and L. L. Peterson. "Fbufs: A high-bandwidth cross-domain transfer facility." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993, pp. 189–202.

[9] W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallumm, J. A. Thomas, R. Wisniewski, and S. Luk. "Linking shared segments." In *USENIX Technical Conference Proceedings*, January 1993, pp. 13–27.

[10] J. R. Goodman. "Coherency for multiprocessor virtual address caches." In *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-2)*, October 1987, pp. 72–81.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[12] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard Company, 1990.

[13] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. K. Ousterhout, and D. A. Patterson. "Design decisions in SPUR." *IEEE Computer*, vol. 19, no. 11, November 1986.

[14] J. Huck and J. Hays. "Architectural support for translation table management in large address space machines." In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA-20)*, May 1993, pp. 39–50.

[15] IBM and Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola, 1993.

[16] J. Inouye, R. Konuru, J. Walpole, and B. Sears. "The effects of virtually addressed caches on virtual memory design and performance." Tech. Rep. CS/E 92-010, Oregon Graduate Institute, 1992.

[17] Intel. *Pentium Processor User's Manual*. Intel Corporation, Mt. Prospect IL, 1993.

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

[19] Y. A. Khalidi and M. Talluri. "Improving the address translation performance of widely shared pages." Tech. Rep. SMLI TR-95-38, Sun Microsystems, February 1995.

[20] J. Liedtke. "Address space sparsity and fine granularity." *ACM Operating Systems Review*, vol. 29, no. 1, pp. 87–90, January 1995.

[21] C. May, E. Silha, R. Simpson, and H. Warren, Eds. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, San Francisco CA, 1994.

[22] J. E. B. Moss. "Working with persistent objects: To swizzle or not to swizzle." *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 657–673, August 1992.

[23] E. I. Organick. *The Multics System: An Examination of its Structure*. The MIT Press, Cambridge MA, 1972.

[24] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. "Design rationale for Psyche, a general-purpose multiprocessor operating system." In *Proc. 1988 International Conference on Parallel Processing*, August 1988.

[25] R. L. Sites, Ed. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, Maynard MA, 1992.

[26] M. Talluri, M. D. Hill, and Y. A. Khalidi. "A new page table for 64-bit address spaces." In *Proc. Fifteenth ACM Symposium on Operating Systems Principles (SOSP-15)*, December 1995.

[27] W.-H. Wang, J.-L. Baer, and H. M. Levy. "Organization and performance of a two-level virtual-real cache hierarchy." In *Proc. 16th Annual International Symposium on Computer Architecture (ISCA-16)*, June 1989, pp. 140–148.

[28] S. Weiss and J. E. Smith. *POWER and PowerPC*. Morgan Kaufmann Publishers, San Francisco CA, 1994.

[29] B. Wheeler and B. N. Bershad. "Consistency management for virtually indexed caches." In *Proc. Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, October 1992, pp. 124–136.