Memory Management Hardware, and its Support for Operating Systems

Bruce Jacob and Trevor Mudge

Advanced Computer Architecture Lab
EECS Department, University of Michigan
{blj,tnm}@eecs.umich.edu

Abstract. *This survey compares and contrasts the memory management designs of six commercial microarchitectures in the context of today's operating system requirements, which include such features as multiple processes with address space protection, shared memory, large virtual address spaces, and fine-grained protection.*

Keywords: *virtual memory, memory management units, translation lookaside buffers, address spaces, hardware support for operating system functions*

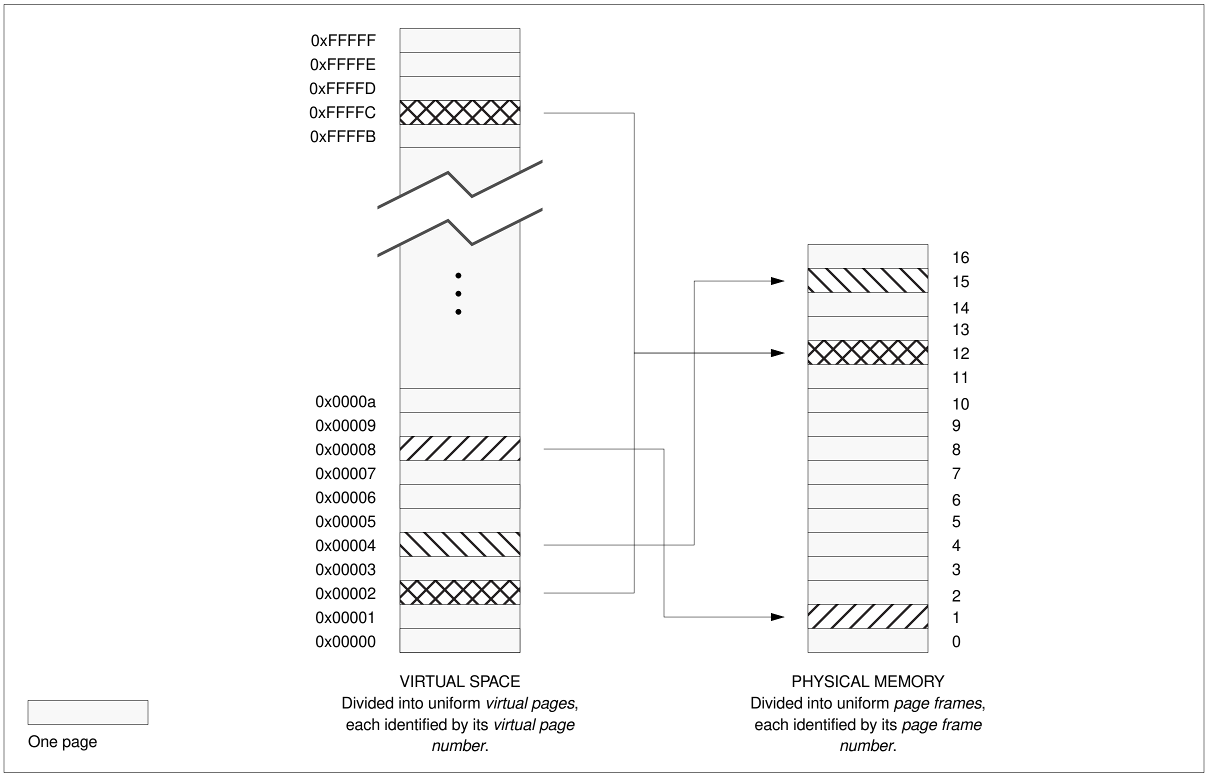

Figure 1. Mapping virtual pages into physical page frames.

PART ONE: A Virtual Memory Primer

Virtual memory supports the execution of processes only partially resident in memory; this allows operating systems to manage the physical memory effectively. In a virtual memory system, only the most-often used portions of a process's address space actually occupy physical memory; the rest of the address space is stored on disk until needed. Most processors support virtual memory through a hardware *memory management unit (MMU)* that translates virtual addresses to physical addresses.

The fundamentals of virtual memory are straightforward. Every process generates addresses for loads and stores as if it has the entire machine to itself—as if the computer offers an extremely large amount of memory. The operating system and hardware support this illusion by translating these virtual addresses to physical ones on the fly, as depicted in Fig. 1. Addresses are mapped at the granularity of *pages*; if any data in a virtual page are in physical memory, then all the data are. At its simplest, virtual memory is then a mapping of *virtual page numbers (VPNs)* to *page frame numbers (PFNs).* "Frame" in this context means "slot" — physical memory is divided into frames that hold pages. The mapping is a function; any virtual page can have only one location. However the inverse map is not necessarily a function; it is possible and sometimes advantageous to have several virtual pages mapped to the same page frame (to share memory between processes or threads, or to allow different views of data with different protections, for example). This is depicted in Fig. 1 by mapping two virtual pages (0x00002 and 0xFFFFC) to page frame number 12.

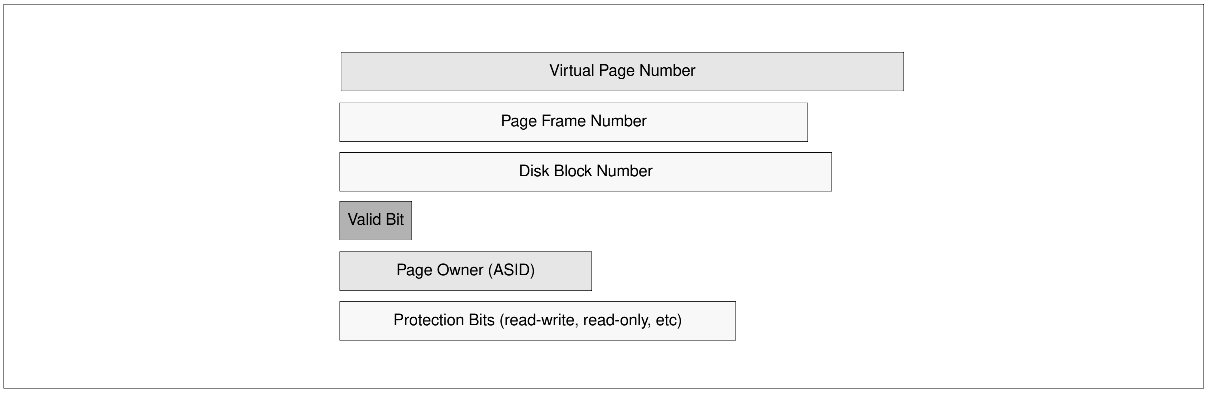

Figure 2. A logical Page Table Entry.

When the word *page* is used as a verb, it means to *map* a region of memory—to allocate mapping information that sets up a correspondence between a set of VPNs and PFNs, allowing the operating system to move the region between physical memory and disk depending on its usage pattern. The operating system moves regions in and out of memory at the granularity of pages. When a particular page has not been used recently it is stored to disk (*paged-out*) and the space is freed up for more active pages. Pages that have been migrated to disk are returned to memory (*paged-in*) once they are needed again.

Hardware architectures often divide the virtual address space into *mapped* and *unmapped* regions; the operating system translates virtual addresses in mapped regions but not those in unmapped regions. The hardware treats an unmapped virtual address as an absolute physical address—except for the top few bits which might be cleared to zero (the physical address is often smaller than the virtual address). Unfortunately the terms *mapped* and *unmapped* serve dual-purpose; they also indicate whether a virtual page in a mapped region currently has a valid mapping. In this context, a *mapped* virtual page is one for which the operating system currently maintains information on its location in memory or on disk; an *unmapped* page is either not allocated yet, or it has been de-allocated and its mapping information discarded.

Mapping information is organized into *page tables*, which are collections of *page table entries (PTEs)*. There is one PTE for every mapped virtual page; an individual PTE indicates whether its virtual page is in memory, on disk, or not allocated yet. The logical PTE therefore contains the VPN and either the page's location in memory (a PFN), or its location on disk (a disk block number). Depending on the organization, some of this information is redundant; actual implementations do not necessarily require both the VPN and the PFN. Later developments in virtual memory added such things as page-level protections. A modern PTE usually contains protection information as well, such as whether the page contains executable code, whether it can be modified, and if so by whom. Fig. 2 illustrates.

To speed translation, most hardware systems provide a cache for PTEs, called a *translation lookaside buffer (TLB)*; the TLB must contain the appropriate PTE for a load or store to complete successfully. Otherwise, the system takes a *TLB miss* to search the page table for the appropriate entry and place it into the TLB. If the mapping is not found in the page table, or if the mapping is found but it indicates that the desired page is

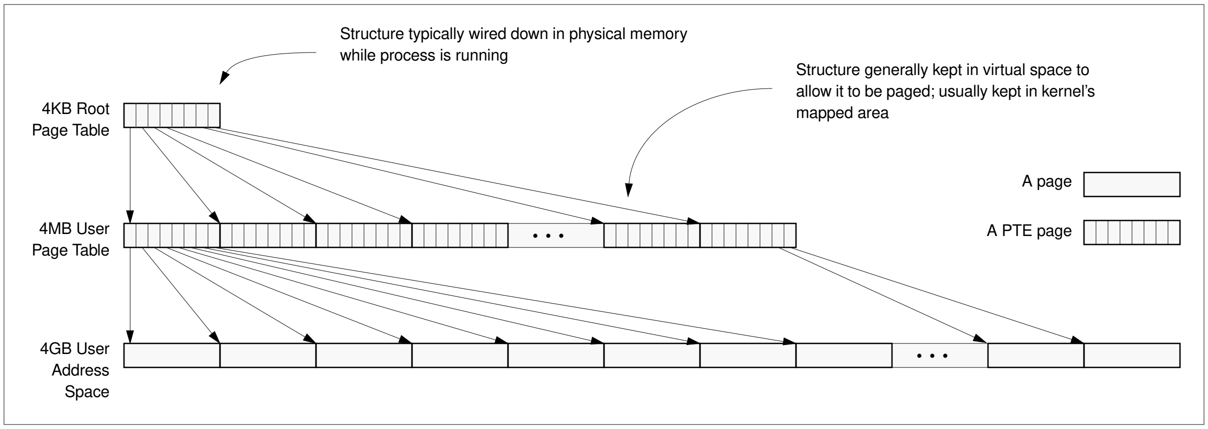

Figure 3. Classical two-level hierarchical page table.

not in memory but on disk, the system takes another exception to copy the page back into memory. This is known as a *page fault*. In this discussion we will refer to the virtual address causing a TLB miss as the *faulting* address, though this is not meant to imply that all TLB misses result in page faults. The page table is walked for every TLB miss, but not every TLB miss indicates a page fault.

When a TLB miss occurs, it is necessary to look up the appropriate PTE to find where the page resides. This lookup can be simplified if PTEs are organized contiguously, so that a VPN or PFN can be used as an offset to find the appropriate PTE. This leads to two primary types of page table organization: the *forward-mapped* or *hierarchical page table*, indexed by the VPN; and the *inverse-mapped* or *inverted page table*, indexed by the PFN. Each design has its strengths and weaknesses. The hierarchical table supports a simple lookup algorithm and simple sharing mechanisms, but can require a significant fraction of physical memory. The inverted table supports efficient hardware table-walking mechanisms but inhibits sharing by not allowing the mappings for multiple virtual pages to exist in the table simultaneously if those pages map to the same page frame.

Hierarchical Page Tables

The classical hierarchical page table, depicted in Fig. 3, comes from the self-similar idea that a large space can be mapped by a smaller space, which can in turn be mapped by an even smaller space. If we assume 32-bit addresses and 4KB pages, the 4GB address space is comprised of 1,048,576 (2^{20}) pages. If each of these pages is mapped by a 4-byte PTE, we can organize the PTEs into a 4MB linear structure. This is rather large, and since it is likely that not all the user's pages will be mapped (much of the 4GB virtual address space might never be touched), why *wire down*[1] the entire 4MB array of PTEs if most will be empty? Why not map the page table itself—place the array into virtual space and allow *it* to be paged just like "normal" virtual

1. To "wire down" a region of virtual memory is to reserve space for it in physical memory and not allow it to be paged to disk. Thus, the space cannot be used for any other purpose.

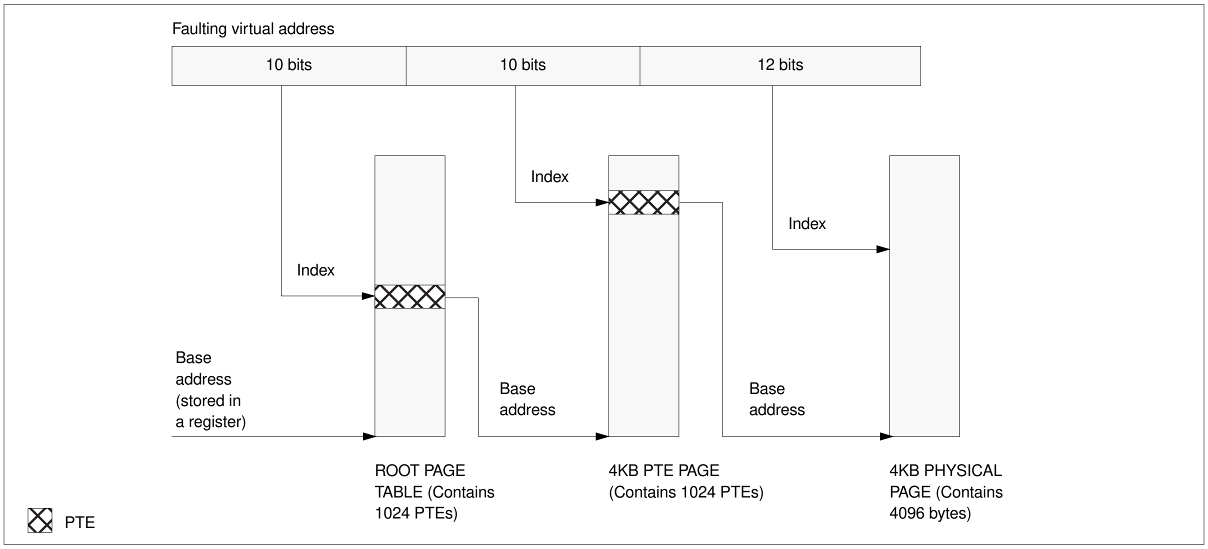

Figure 4. Top-down access method for hierarchical page table.

memory? A 4MB linear structure occupies 1024 (2^{10}) pages, which can be mapped by 1024 PTEs. Organized into a linear array, they occupy 4KB—a reasonable amount of memory to wire down for a running process.

There are two access methods for the hierarchical page table: *top-down* or *bottom-up*. A top-down traversal uses physical addresses to reference the PTEs in the table, while a bottom-up traversal uses virtual addresses.

Top-Down Traversal

Fig. 4 shows how the classical hierarchical page table can be accessed in a top-down fashion. Many of the early hierarchical page tables were traversed this way, so the term *forward-mapped page table* is often used to mean a hierarchical page table accessed top-down. A virtual address is broken into three fields. The bottom 12 bits together identify a byte within a 4KB page. If a PTE is four bytes long then a 4KB page can contain 1024 PTEs. The next ten bits of the virtual address together identify a PTE within a page of PTEs; this PTE maps the data page. The top ten bits of the virtual address identify a PTE within the root page table that maps the PTE page.

A lookup proceeds as follows. First, the top ten bits index the 1024-entry root page table, whose base address is typically stored in a hardware register. The referenced PTE gives the physical address of a 4KB PTE page. The next ten bits of the virtual address index this structure. The indicated PTE gives the PFN of the 4KB virtual page referenced by the faulting virtual address. The bottom 12 bits of the virtual address index the physical page to access the desired byte.

The mapping information, if in the table, will be found by the end of the process. Due to its simplicity, this algorithm is often used in hardware table-walking schemes such as in IA, the Intel Architecture (Intel's name for the x86 architecture). If at the end of the algorithm the hardware does not have the PTE, it raises a page fault exception, indicating that the requested page is not in physical memory. At this point, the operat-

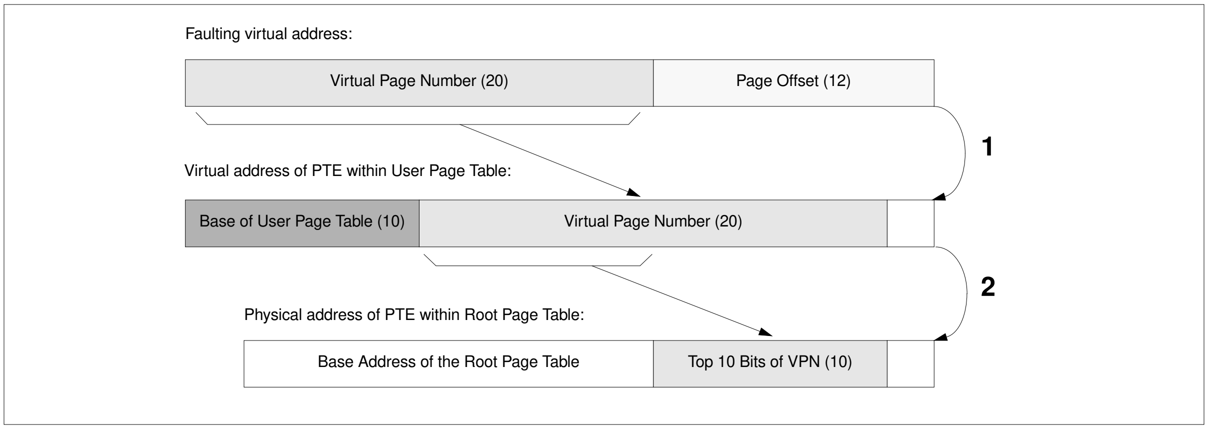

Figure 5. ***Bottom-up*** **access method for hierarchical page table.**

ing system must retrieve the page from disk and place the mapping into the page table (and possibly the TLB) before processing can continue.

Bottom-Up Traversal

The top-down access method requires three memory references to satisfy one memory request—two to translate the virtual address and one to load the data. Alternatively, one can use a bottom-up traversal and often incur but a single memory access to translate the virtual address. The top 20 bits of the virtual address are a *virtual* offset into the 4MB user page table, which is continuous in virtual space. Referring to Fig. 3, we see that the VPN indexes the 4MB user page table. If this virtual address causes a TLB miss, the operating system then searches the 4KB root page table for the appropriate root PTE. The top ten bits of the virtual address index this root PTE within the root table.

Fig. 5 depicts the algorithm. In step 1, the top twenty bits of a faulting virtual address are concatenated with the virtual offset of the user page table. The bottom bits of the address are zero, as a PTE is several bytes long. Since the user page table is a linear array of PTEs that mirrors the user's linear array of virtual pages, the VPN of the faulting address is equal to the index of the PTE within the page table, therefore the virtual address points to the mapping PTE. If this reference succeeds, the PTE is loaded to translate the original faulting virtual address.

However, this virtual address may cause an overlapping TLB miss. In step 2, the operating system generates a second address when the virtual PTE access fails. The mapping PTE is an entry in the root page table, and the index is the top 10 bits of the VPN, just as in the top-down method. These ten bits are concatenated with the base address of the root page table to form a physical address for the appropriate PTE. This PTE is loaded to obtain the physical address of the user PTE (the PTE that the virtual address in step 1 failed to load). Once this PTE is loaded, the user data can be loaded.

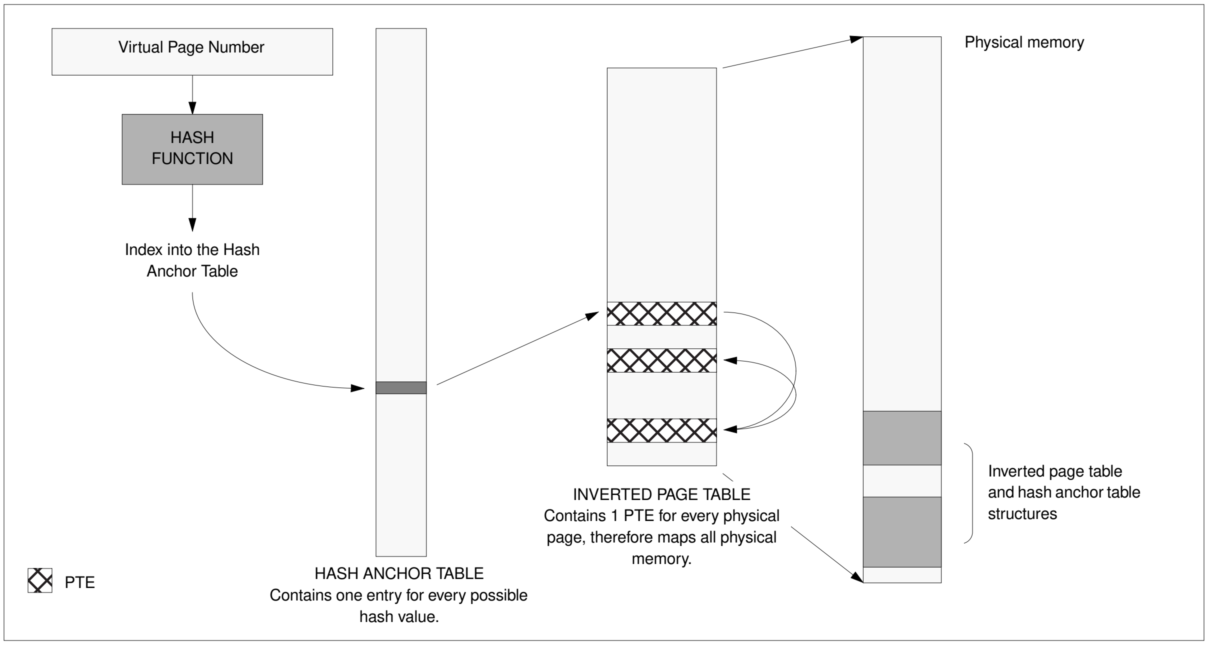

Figure 6. Classical inverted page table structure.

Though the bottom-up scheme can get complicated, usually the first virtual reference succeeds and only two memory references are required to complete a user-level load: one to obtain the PTE, one to obtain the user data.

Inverted Page Tables

The classical inverted page table, pictured in Fig. 6, has several advantages over the hierarchical table. Instead of one entry for every virtual page belonging to a process, it contains one entry for every physical page in main memory. Thus, rather than scaling with the size of the virtual space, it scales with the size of physical memory. This is a distinct advantage over the hierarchical table when one is concerned with 64-bit address spaces. Depending on the implementation, it can also have a lower average number of memory references to service a TLB miss than a typical hierarchical page table. Its compact size (there are usually no unused entries wasting space), makes it a good candidate for a hardware-managed mechanism that requires the table to be wired down in memory.

The structure is said to be *inverted* because the index of the PTE in the page table is the PFN, not the VPN. However, one typically uses the page table to *find* the PFN for a given VPN, so the PFN is not readily available. Therefore a hashing scheme is used; to locate a PTE the VPN is hashed to index the table. Since different VPNs might produce identical hash values, a collision-chain mechanism is used to allow different virtual mappings to exist in the table simultaneously. When a collision occurs a different slot in the table is chosen, and the new entry is added to the end of the chain; thus it is possible to chase a long list of pointers while servicing a single TLB miss. Collision chains in hash tables are well-researched; to keep the average

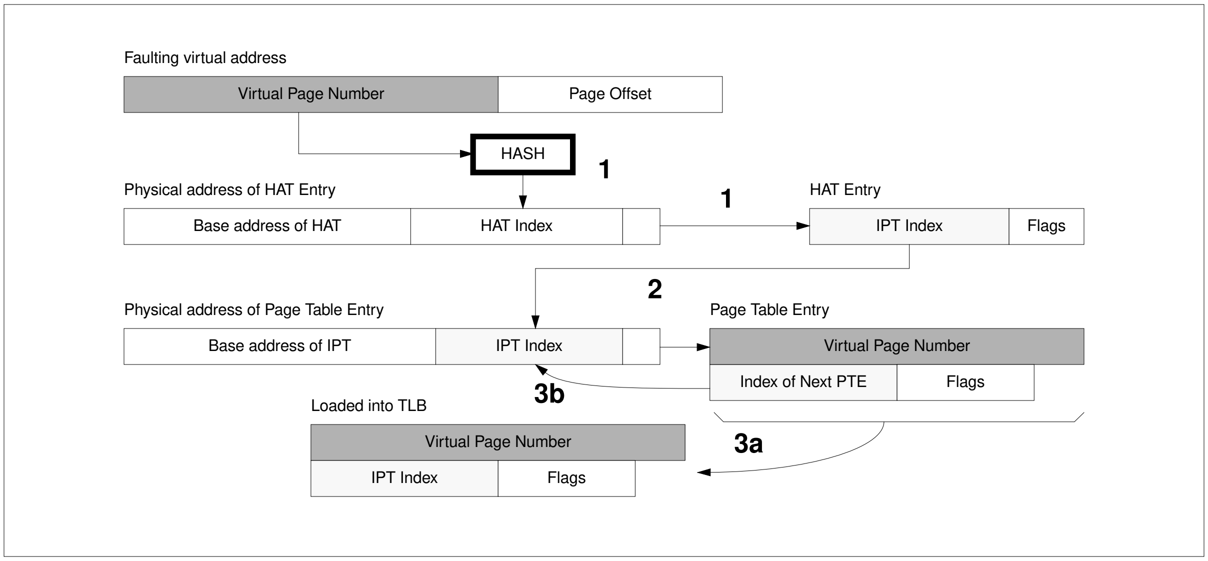

Figure 7. Lookup algorithm for inverted page table.

chain length short one can increase the size of the hash table. However, by changing the inverted page table's size, one loses the ability to index the table by the PFN. Therefore a level of indirection is used; the *hash anchor table (HAT)*, another in-memory structure, points to the chain head for every hash value. Every doubling of the HAT size reduces the average chain length by half, so the HAT is generally several times larger than the inverted page table.

The inverted table has a simple access method, appropriate for use in a hardware-managed mechanism. The mapping, if in the table, will be found by the end of the algorithm. Fig. 7 illustrates. In step 1, the faulting VPN is hashed, indexing the HAT. The corresponding entry is loaded and points to the chain head for that hash value. In step 2, the indicated PTE is loaded, and its VPN is compared with the faulting VPN. If the two match the algorithm terminates, and the mapping, comprised of a VPN (that of the faulting address and the PTE) and a PFN (the PTE's index in the inverted page table), is placed into the TLB (step 3a). Otherwise, the PTE references the next entry in the chain (step 3b), or indicates that it is the last in the chain. If possible, the next entry is loaded and compared. If the last entry fails to match, the algorithm terminates unsuccessfully and causes a page fault.

The Translation Lookaside Buffer

Most architectures provide a TLB to support memory management; the TLB is a special-purpose cache that holds only virtual-physical mappings. When a process attempts to load from or store to a virtual address, the hardware searches the TLB for the virtual address's mapping. If the mapping exists in the TLB, the hardware can translate the reference to a physical address without the aid of the page table. If the mapping does not exist in the TLB, the process cannot continue until the correct mapping information is loaded into the TLB. The loading can be performed by the operating system or by the hardware directly; a system is said to have a *software-managed TLB* if the OS is responsible for the loading, or a *hardware-managed TLB* if the hardware is responsible.

The performance difference between the two is due to the page table lookup and the method of operation. In a hardware-managed TLB design a hardware state machine walks the page table; there is no interaction with the instruction cache. By contrast, the software-managed design uses the general interrupt mechanism to invoke a software TLB miss-handler—a primitive in the operating system usually 10-100 instructions long. If this miss-handler is not in the instruction cache at the time of the TLB miss exception, the time to handle the miss can be much longer than in the hardware-walked scheme. In addition, the use of the general-purpose interrupt mechanism adds a number of cycles to the cost by draining the pipeline and flushing a possibly large number of instructions from the reorder buffer; this can add up to something on the order of 100 cycles. However, the software-managed TLB design allows the operating system to choose any organization for the page table, while the hardware-managed scheme defines an organization for the operating system. If TLB misses are infrequent, the flexibility afforded by the software-managed scheme can outweigh the potentially higher per-miss cost of the design.

Page Table Perspective

The first hierarchical tables were accessed top-down. The most common complaints against this design were that it was time-inefficient for supporting large address spaces and space-inefficient for supporting sparse address spaces. The hierarchical table is said to waste time because it requires more than two tiers to cover a large (e.g. 64-bit) address space, and for every tier in the page table the lookup scheme requires one memory reference to locate the PTE. The hierarchical table is said to waste memory because space in the table is allocated an entire page at a time. A process address space with a single page in it will require one full PTE page at the level of the user page table. If the process adds to its address space more pages that are contiguous with the first page, they can also be mapped by the single PTE page, since the PTEs that map them will be contiguous with the first PTE and will likely fit within the initial PTE page. If instead the process adds to its address space another page that is distant from the first page, a second PTE page will be added to the user page table, doubling the amount of memory required to map the address space. Clearly, if the address space is very sparsely populated (e.g. an address space composed of many individual virtual pages spaced far apart), most of the entries in a given PTE page will be invalid but will consume memory nonetheless, and the organization can degrade to using as many PTE pages as there are mapped virtual pages.

The inverted table was developed to address these problems; no matter how sparsely populated the address space, memory is not wasted because space in the table is allocated one PTE at a time. Since the size of the table is proportional to the number of physical pages available in the system, a large virtual address does not affect the number of entries in the table. The inverted organization does have a few drawbacks. Since the table only contains entries for virtual pages actively occupying physical memory, one needs an

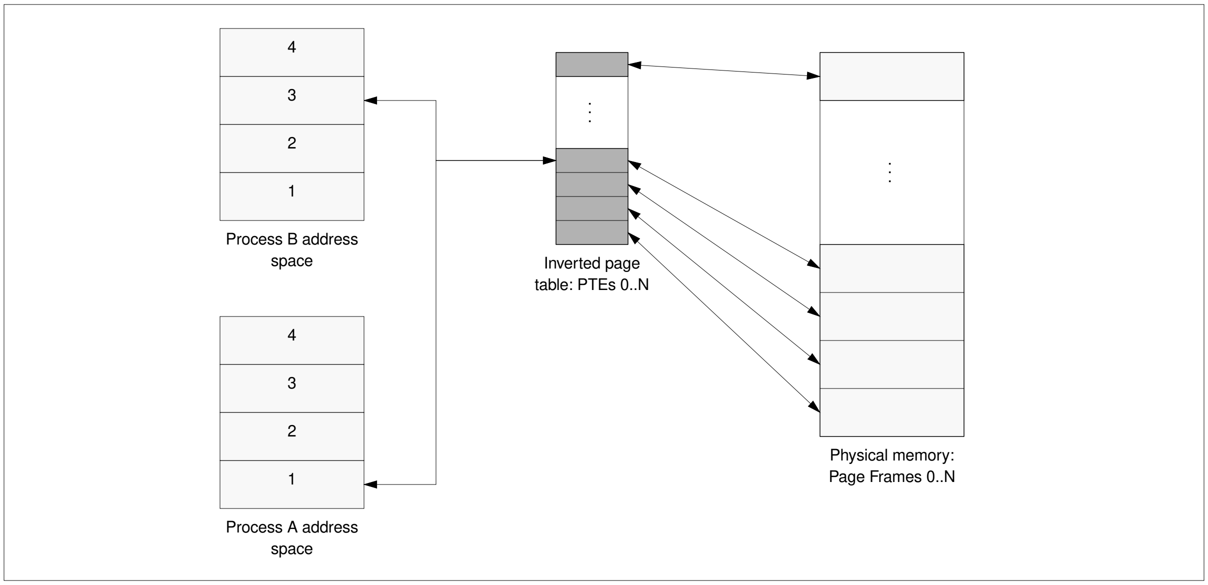

Figure 8. Inverted page table and shared memory.

alternate structure to locate pages on disk once they are needed again. Depending on the choice of organization, this backup page table can potentially prevent a system from obtaining the space-saving benefit the inverted organization offers. Also, since the inverted table can only hold one entry for each page frame in the system, the classical inverted table cannot simultaneously hold mappings for virtual pages mapped to the same physical location. This is pictured in Fig. 8. If virtual page 1 in process A's address space and virtual page 3 in process B's address space are mapped to the same page frame, thereby allowing A and B to share memory, both mappings cannot reside in the table at the same time. Thus, if the processes are using the memory for communication, the operating system could potentially service two page faults for every message exchange.

It must also be pointed out that the bottom-up variant of the hierarchical table, first commercially appearing on the MIPS processor, does not have the same problems that hamper the top-down variant. Despite the fact that it requires more than two tiers to map a 64-bit address space, the scheme is not usually time-inefficient since the PTEs needed for mapping will likely be found in the cache, requiring but a single memory lookup to find the mapping information. The claim that a hierarchical page table wastes space for a sparsely populated address space is still valid.

PART TWO: Memory Management Hardware, and its Support for Operating Systems

The virtual memory interface is defined differently for every microarchitecture. More often than not, the operating system running on a microprocessor was not initially designed for that particular hardware interface. Sometimes there is a substantial performance penalty to be paid, sometimes not. Generally, hardware is designed with little regard to the system software that will control it. Generally, operating systems outlast the hardware design they were designed and built on. Hardware Abstraction Layers exacerbate the problem by hiding hardware particulars from most of the operating system, allowing system designers to write an operating system for no hardware in particular. Though it makes the job of writing an operating system easier, this mismatch between operating system and microarchitecture has been shown many times over to cause significant performance problems [12]. Often, simple reorganizations of the system software that are intended to better take advantage of hardware features result in order-of-magnitude speedups [11]. Thus an operating system that is not tuned to the hardware on which it operates is not likely to demonstrate excellent performance.

This is a survey of hardware interfaces existent for operating systems. The memory management systems of several processors are described in terms of today's software requirements. We have chosen current examples within the most visible commercial architecture families: MIPS R10000, Alpha 21164, PowerPC 604, PA-8000, UltraSPARC, and Pentium.

Operating System Requirements

There is a core set of functional mechanisms associated with memory management that computer users have come to expect. These are found in nearly every modern microarchitecture and operating system and include the following:

Address Space Protection. User-level applications should not have direct access to the data of other applications or the operating system. A common hardware assist uses *address space identifiers (ASIDs)*, which extend virtual addresses and distinguish them from those generated by different processes. Alternatively, protection can be provided by software means [17].

Shared Memory. Shared memory allows multiple processes to reference the same physical data through (potentially) different virtual addresses. When multiple instances of a program run, sharing the program code reduces physical memory requirements. When processes are communicating, shared memory avoids the data copying of traditional message-passing mechanisms. Since page-size copies are typically an order of magnitude slower than system calls, zero-copy shared-memory

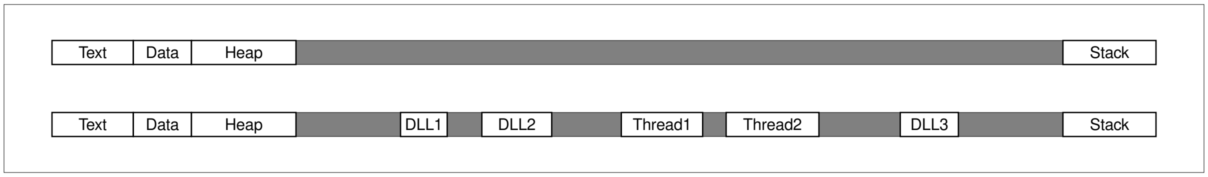

Figure 9. Sparse address spaces. The top address space is that of a traditional 4.3BSD process, with contiguous text, data, and heap segments and a continuous stack segment. The bottom address space contains modern features like dynamically loaded libraries and multiple threads of control, which leave holes within the address space, and thus would leave holes within a linear page table. A wired-down linear page table (as in 4.3BSD) would not be practical.

schemes are possible, in which the operating system unmaps pages from the sender's address space and re-maps them into the receiver's address space [16]. Shared memory acts in opposition to most address space protection schemes. If every page is tagged with a single identifier and every process is tagged with a single identifier, the only way to allow more than one process to access a given page is to circumvent the protection mechanism. This is often done by explicitly duplicating mapping information across page tables, or by marking a shared page as globally visible and flushing its mapping whenever a process runs that should not have access to the page.

Large Address Spaces. Virtual address spaces continue to grow and the industry is moving toward 64-bit machines. However, a large address space is not the same thing as a large address; large addresses are simply one way to implement large address spaces. Another is to provide each process a window into a larger global virtual address space, the approach used by the PA-RISC and PowerPC architectures [8, 13]. Here, an application generates addresses that are mapped by the hardware onto a larger virtual space—for example: the PowerPC supports a 32-bit window onto a 52-bit space, the PA-RISC 2.0 supports a 64-bit window onto a 96-bit space. The application has access to the larger space, but can only access a fixed amount at a time.

Fine-Grained Protection. Fine-grained protection marks objects as *read-only*, *read-write*, *execute-only*, etc. The granularity is usually a page, though a larger or smaller granularity is sometimes desirable. Many systems have used protection to implement various memory-system support functions, from copy-on-write to garbage collection to distributed shared virtual memory [1]. Protection is implemented by setting bits in the page table entry, which correspond to bits in the TLB entry. If a process attempts to use a page incorrectly (e.g. write to a page marked *read-only*), the hardware raises an exception.

Sparse Address Spaces. Dynamically loaded shared libraries and multithreaded processes are becoming commonplace, but these designs require support for sparse address spaces. This simply means that holes are left in the address space between different objects to leave room for dynamic growth. In contrast, 4.3BSD Unix [10] had an address space composed of two continuous regions, depicted in Fig. 9. This design allowed the user page tables to occupy as little space as possible. Saving space was important, given that the original BSD virtual memory implementation did not allow page tables to be paged. The numerous holes in a sparse address space leave numerous holes within a page table mapping that space; thus the wired-down, linearly-indexed page table of 4.3BSD would not be practical, as it would require 4MB of physical memory to map a 32-bit address space regardless of how much or little of the virtual space is actually used.

Superpages. Reflecting a trend toward larger data sets, many objects can be found that need to be mapped for virtual access and yet are larger than a single virtual page. These include commonly used structures such as large arrays, memory-mapped files, and buffers holding graphical images (or portions thereof) for processing. The numerous PTEs required to map these structures flood the TLB and crowd out other entries. Systems have addressed this problem by supporting "blocks" or "superpages" that are multiples of the hardware page size mapped by a single entry in either the TLB or a specialized Block TLB. For example, the Pentium and the MIPS R4000 allow mappings for 4MB pages to reside in the TLB alongside normal mappings, and the PowerPC and PA-RISC 1.0 define Block TLBs which are accessed in parallel with the normal TLB. Studies have shown that large performance gains are made when the number of TLB entries to cover the current working set is reduced through the support of superpages [15].

In the following sections we describe examples of commercial memory-management designs and summarize each with their support for these requirements.

(INSERT SIDEBAR: Table 2—comparison of designs)

MIPS [4, 9]

MIPS (Fig. 10) defines one of the simplest memory-management architectures among commercial microprocessors. The only restriction on the operating system is the format of the TLB entry. The operating system handles TLB misses entirely in software; the TLB is filled by software, and the TLB replacement policy is up to the operating system.

The TLB has 64 fully associative entries, partitioned between *wired* and *random* entries. The R2000/R3000 has eight wired entries, and the R10000 has a flexible partition defined by software. Software can overwrite any entry in the TLB but the wired entries must be over-written intentionally. The hardware provides a mechanism to choose one of the *random* slots randomly; the hardware provides a random number on request, between index values of 8 and 63, inclusive (the R10000 gives values between *N* and 63, inclusive, where *N* is set by software). This random number indexes only the *random* entries of the TLB; if it is used on TLB refill, the operating system can only overwrite TLB entries in the *wired* slots (slots 0 through 8, or 0 through *N*) if done so explicitly. Most operating systems use this feature to store root-level page-table entries and kernel mappings in the protected slots, keeping easily replaced user mappings in the *random* slots.

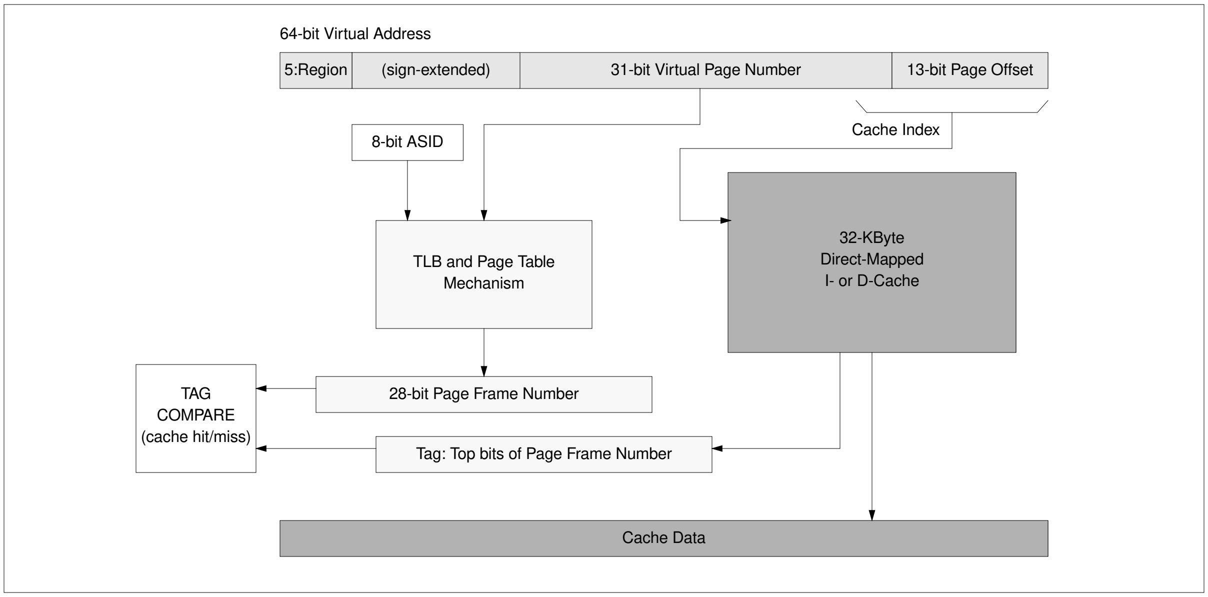

Figure 10. The MIPS R10000 address translation mechanism. The split instruction and data caches are larger than the page size and virtually indexed. The TLB lookup proceeds in parallel with the cache lookup. The earlier MIPS designs had physically indexed caches, and the cache was accessed in serial with the TLB, if the cache was larger than the page size. For mapped addresses (those that are translated by the R10000 TLB), bits [58:44] are sign-extended from bit 43.

The R2000/R3000 virtual address is 32 bits wide, the R10000 virtual address is 64 bits wide. While all 32 bits of the address are used in the R2000/R3000, only the bottom 44 of 64 bits are translated in the R10000. The top bits of the R10000 virtual address are called *region* bits and divide the virtual space into areas of different behavior (cached/uncached, mapped/unmapped). The top two bits distinguish between user, supervisor, and kernel spaces; the top five bits divide the kernel space into different regions of behavior, and one of the unmapped kernel regions is further subdivided by the top seven bits of the address. Virtual addresses are extended with an address space identifier (R2000/R3000: 6 bits, R10000: 8 bits) to distinguish between contexts. The TLB translates virtual page numbers to physical page numbers, the high-end bits of which are used to match against the cache tags. The top two bits of the region are kept in the TLB entry to distinguish between VPNs from user, supervisor, and kernel spaces. In the R2000/R3000, the top bit divides the 4GB address space into user and kernel regions, and the next two bits further divide the kernel's space into regions of cached/uncached and mapped/unmapped behavior.

Pages are shared by setting the *global* bit in the TLB entry; this is a per-entry bit that indicates the ASID match should be ignored for that page. When sharing user pages, entries with the *global* bit set must be flushed on context switches else address space protection can become compromised. Kernel pages are off-limits to user-level processes through hardware enforcement—the top half of the virtual address space is accessible in kernel-mode only.

Periodic cache and TLB flushes are unavoidable in the MIPS architecture, as there are 64 unique context identifiers in the R2000/R3000 and 256 in the R10000. Many systems have more active processes than this, requiring ASID sharing and periodic remapping. When a new process is mapped to an old ASID, it is neces-

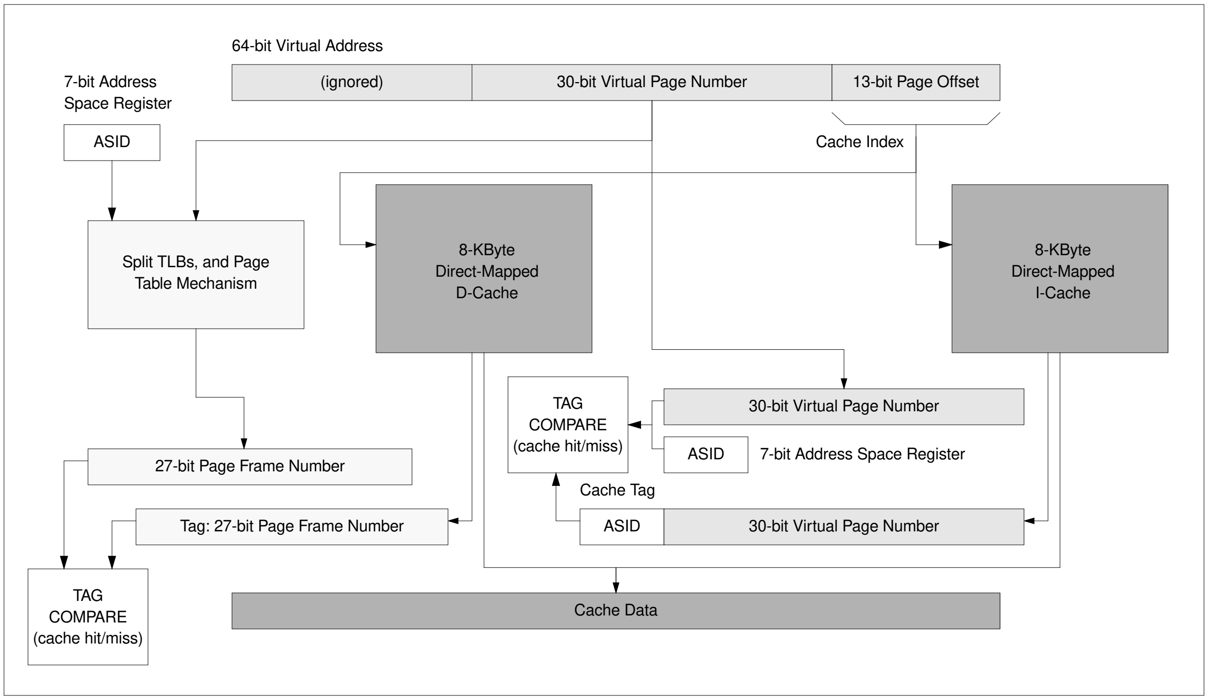

Figure 11. The Alpha 21164 address translation mechanism. The split caches are 8KB each and so are effectively virtually indexed by the virtual bits of the page offset. The data cache is physically tagged and the instruction cache is virtually tagged. The I-side TLB is accessed on every reference to provide page protection.

sary to flush TLB entries with that ASID. It is possible to avoid flushing the cache by flushing the TLB; since the caches are physically tagged the new process will not be able to read or overwrite the old process's data.

Summary

The MIPS architecture implements address space protection with 6-bit or 8-bit ASIDs. As noted earlier, the ASIDs protect contexts from one another, but their small numbers necessitate periodic remapping and thus TLB flushing. The software-managed TLB helps solve the sparse address space problem by allowing the operating system to choose an appropriate page table organization. Shared memory is supported through the *global* bit in each TLB entry, but this mechanism provides no direct support of sharing memory between a small number contexts. The R10000 architecture provides 64-bit addressing for support of large address spaces. The R2000 and R3000 designs provide 32-bit addressing which was a large address space for their time. The R10000 supports a variable page size from 4KB to 16MB by multiples of four; each TLB entry contains a value that identifies its page size.

Alpha [3, 14]

The Alpha architecture (Fig. 11) has split TLBs and split caches. Like MIPS, the TLBs are software-managed. However, the operating system does not have direct access to the TLB but indirect access through the *PAL-*

code—the Privileged Access Library. TLB replacement policy is defined by PALcode, not by the operating system. Similarly, the process control block is defined and managed by the PALcode.

An application generates a virtual address 64 bits wide. Alpha hardware recognizes a virtual address 43-55 bits wide, depending on the software-defined page size. The choices of page size and corresponding maximum virtual address size are depicted in Table 1. The virtual address sizes are chosen to give the largest

Table 1: Address size as a function of page size

Page Size	Page Offset	Virtual Page Number	Max Virtual Address
8 KBytes	13 bits	30 bits	43 bits
16 KBytes	14 bits	33 bits	47 bits
32 KBytes	15 bits	36 bits	51 bits
64 KBytes	16 bits	39 bits	55 bits

address space that can be mapped by a three-tiered hierarchical page table, given the chosen page size. Using a *granularity-hint* bit, a TLB entry can be configured to group together 8, 64, or 512 virtual pages to support superpages of 64-512KB, 512-4096KB, or 4-32MB, respectively. Note that only the operating system has permissions to modify TLB entries.

The Alpha 21164 supports a 43-bit virtual address and a 40-bit physical address only; the page size is fixed at 8KB. The 21164 uses 7-bit address space identifiers; this scheme has the same need for flushing as the MIPS architecture due to a small number of contexts. The 21164 split cache sizes are 8KB each, which happens to be the page size, making the caches effectively indexed by the physical address (since the *page offset* bits of the virtual address are identical to the page offset bits of the translated physical address).

Page-level sharing is implemented by an *address-space-match* bit set in the TLB entry, similar to the MIPS *global* bit. The TLB also distinguishes between user-mode access and kernel-mode access, which enables the kernel to protect itself from user programs on a page-by-page basis. This allows the operating system to make certain virtual pages read-only to user-level processes and writable only by itself. This is not possible in the MIPS architecture.

Summary

The Alpha implements address space protection via address space identifiers. The 7-bit ASID of the 21164 will need to be remapped frequently if the system is running more than 128 concurrent processes. Support for sparse address spaces is provided by a software-managed TLB that allows the operating system to choose a page-table organization. Shared memory is supported by an *address-space-match* bit associated with each virtual page that, when set, instructs the processor to allow all address spaces to access that page. Superpages are supported within the TLB entry via the *granularity-hint*, providing maximum superpage sizes of 4-32MB, depending on the choice of page size.

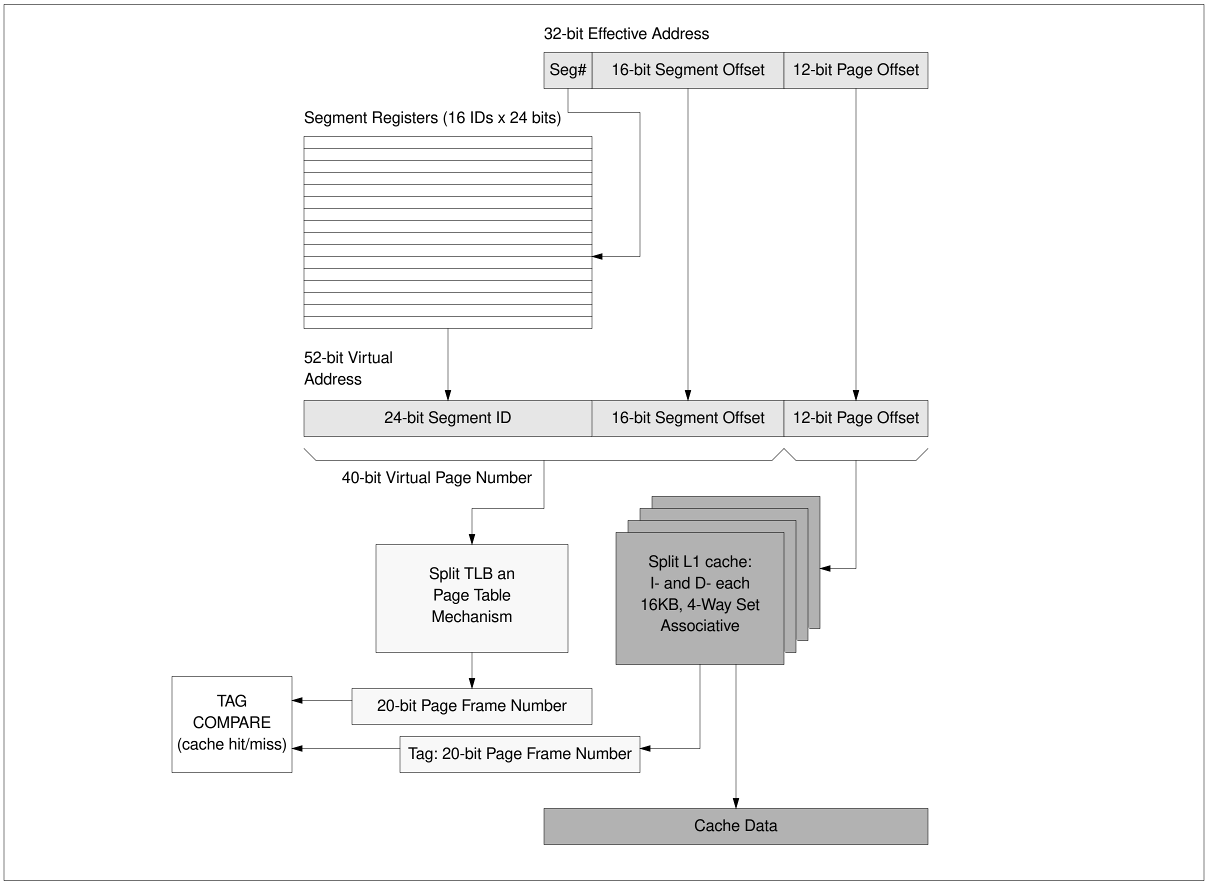

Figure 12. The PowerPC 604 address translation mechanism. Processes generate 32-bit effective addresses that are mapped onto a larger address space via sixteen segment registers, using the top four bits of the effective address as an index. It is this extended virtual address that is mapped by the TLB and page table. Each cache is 16KB, four-way set-associative, which means that the page offset can be used as a cache index, effectively making the cache physically indexed. A Block Address Translation mechanism occurs in parallel with the TLB access, but is not shown.

PowerPC [5, 13]

The PowerPC (Fig. 12) uses IBM 801-style segments [2] to map user “effective” addresses onto a global flat address space much larger than each per-process address space. Segments are 256MB contiguous regions of virtual space, and 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.

The architecture does not provide explicit address space identifiers; address space protection is provided through the segment registers, which can only be modified by the operating system. If two processes have the same segment identifier in one of their segment registers, they share that virtual segment by definition. The operating system enforces protection by disallowing shared segments, and can share memory between processes by overlapping segment numbers. Like the Alpha, the PowerPC allows the kernel to make individual virtual pages accessible through user-mode that are readable or writable only in kernel mode. Therefore, the operating system can map portions of itself into the address spaces of user processes without compromising its integrity. The segment identifiers are 24 bits wide, which guarantees over a million unique processes on a system (the lower bound, which assumes no sharing between processes). Therefore, flushing of the caches or TLBs will be required very infrequently, assuming shared memory is implemented through the segment registers.

The PowerPC defines for the operating system a hashed page table that is partially hardware-managed. A variation on the inverted page table, it is used as an eight-way set-associative software cache for PTEs. On TLB misses, the page table is walked by hardware. The advantage is that it scales well to a 64-bit implementation. However, since the PTE cache is not guaranteed to hold all active mappings, the operating system will have to create and manage a backup page table as well.

The PowerPC defines a *block address translation (BAT)* mechanism that operates in parallel with the TLB and takes precedence over the TLB if the BAT registers signal a "hit" on any given translation. The BAT registers translate virtual addresses at the granularity of *blocks*, which are from 128KB to 256MB in size, by multiples of two.

Summary

The PowerPC architecture implements address space protection through segment IDs, which map the 16 segments of a 4GB address space onto the 4-petabyte (4096TB) global space. An address space is uniquely defined by a set of 16 segment IDs, and sets with null intersections are protected from each other. No *global* bit is necessary in the TLB, as sharing segment IDs implies shared memory. Flushing of the TLB and caches is not necessary on context switches if memory is shared through global addresses. Sparse address spaces are supported by a hashed page table, a variant of the inverted page table. The architecture supports large address spaces; a process could extend its address space by having the operating system move new values into its segment registers, giving every process access to the entire 52-bit virtual space. Superpages are supported through the block address translation registers (BAT registers), which map contiguous regions of memory of size 128KB to 256MB. The drawback is that since the architecture does not use ASIDs, there is no protection for these superpages; the BAT registers must be reloaded on context switches, else they become visible to all processes.

PA-RISC 2.0 [8]

The PA-RISC 2.0 (Fig. 13) is a 64-bit design that uses a "space register" mechanism very similar to PowerPC-style segments. User-level applications generate 64-bit addresses which are extended by 64-bit space identifiers to form 96-bit virtual addresses mapped by the TLB and page table. An application identifies a space register in one of two ways: explicitly or implicitly. One can select any of the eight registers explicitly, either

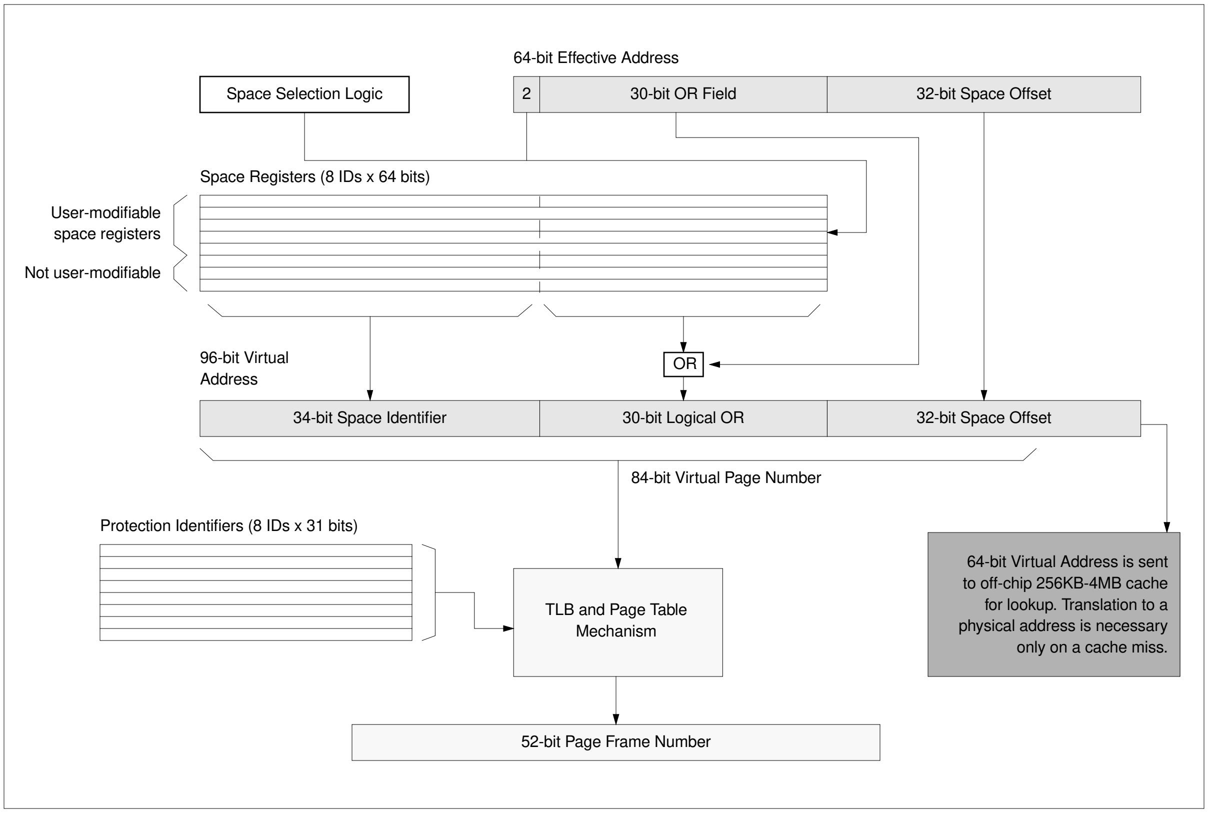

Figure 13. The PA-8000 address translation mechanism. The PA-RISC uses eight Space Registers, which are similar to PowerPC-style segments. The architecture is unique in its use of multiple protection IDs per process. These IDs allow hardware to enforce finite-group sharing—the ability to share a page between a small, well-defined set of processes. The page size is a variable set per TLB entry and can range from 4KB to 64MB, by multiples of four.

directly by a 3-bit field in the instruction, or indirectly through the bottom three bits of a specified register in the general-purpose register file. If no explicit method is chosen, the top two bits of the user's address select one of the top four space registers.

Like the PowerPC, the space identifiers are effectively concatenated with the user's effective address, but the mechanism is designed to allow a variable-sized segment. Whereas in PowerPC the top four bits are replaced by the segment identifier, in PA-RISC anything from the top two bits to the top 32 bits are replaced by the space identifier. The top 34 bits of the 96-bit virtual address come from the space identifier, the bottom 32 bits come from the user's effective address, and the middle 30 bits are the logical OR of the 30-bit overlap of the two. This allows the operating system to define a segmentation granularity. The middle 30 bits of the 96-bit global virtual address are a logical OR, which suggests a simple and effective organization. One can choose a partition (say, for example halfway: at the 15-bit mark), and allow processes to generate only virtual addresses with the top 15 bits of this 30-bit field set to 0. Similarly, all space IDs would have the bottom 15 bits set to 0. Therefore the logical OR would effectively yield a concatenation. This example gives us a PowerPC segment-like concatenation of a (34+15 = 49-bit) space ID with the (32+15 = 47-bit) user virtual address.

Each running process has eight 31-bit *protection identifiers* associated with it. Each virtual page has a single associated *access identifier*, and a process has permissions to access a page if any of its *protection IDs* match the page's *access ID*. As in the PowerPC, this allows hardware to enforce finite-group sharing, the ability to allow a small set of user-level processes to access a given page. This scheme differs from the all-or-nothing approach found in most processors, in which both processes and pages have single IDs and the only way to support hardware access to shared pages with different IDs is to mark the pages globally sharable.

Summary

The PA-RISC implements address space protection through its use of *access IDs* and *protection IDs*. This is also the mechanism (combined with the space registers) used to implement shared memory. Since the scheme uses multiple address space identifiers per process, it is effective at sharing memory among small groups of processes, as opposed to the all-or-nothing schemes found in the MIPS and Alpha architectures. Sparse address spaces are supported by a software-managed TLB. Large address space support is provided by the space registers, giving each process a 64-bit window to a 96-bit global virtual space. The space registers can be modified directly by the application, which makes it relatively simple to add support for extended address spaces, for example through library support. Superpages are supported through the TLB by setting a field within the TLB entry. This allows variable page sizes from 4KB to 64MB by multiples of four.

SPARC V9 [18]

SPARC V9 (Fig. 14) is another 64-bit design; processes generate 64-bit virtual addresses which are translated by the MMU to physical addresses. Like the MIPS and Alpha, not all of the 64 bits are recognized by each implementation; in UltraSPARC the top twenty bits must be sign-extended from the 44th bit. The size of the physical address is up to the implementation. The processor adds an eight-bit *address space identifier (ASI)* when sending a 64-bit virtual address to the MMU, though 'ASI' is a misnomer. The ASI is not used to directly identify different contexts, but to identify data formats and privileges, and to indirectly reference one of a set of context identifiers. ASIs are therefore more like MMU opcodes. User instructions can specify ASIs directly, or use the default ASI provided by the integer unit. They cannot, however, directly specify context identifiers. The following are the ASIs reserved and defined by the architecture; individual implementations of the architecture are allowed to add more.

```
Non-restricted:
ASI_PRIMARY              {_LITTLE}
ASI_PRIMARY_NOFAULT      {_LITTLE}
ASI_SECONDARY            {_LITTLE}
ASI_SECONDARY_NOFAULT    {_LITTLE}
```

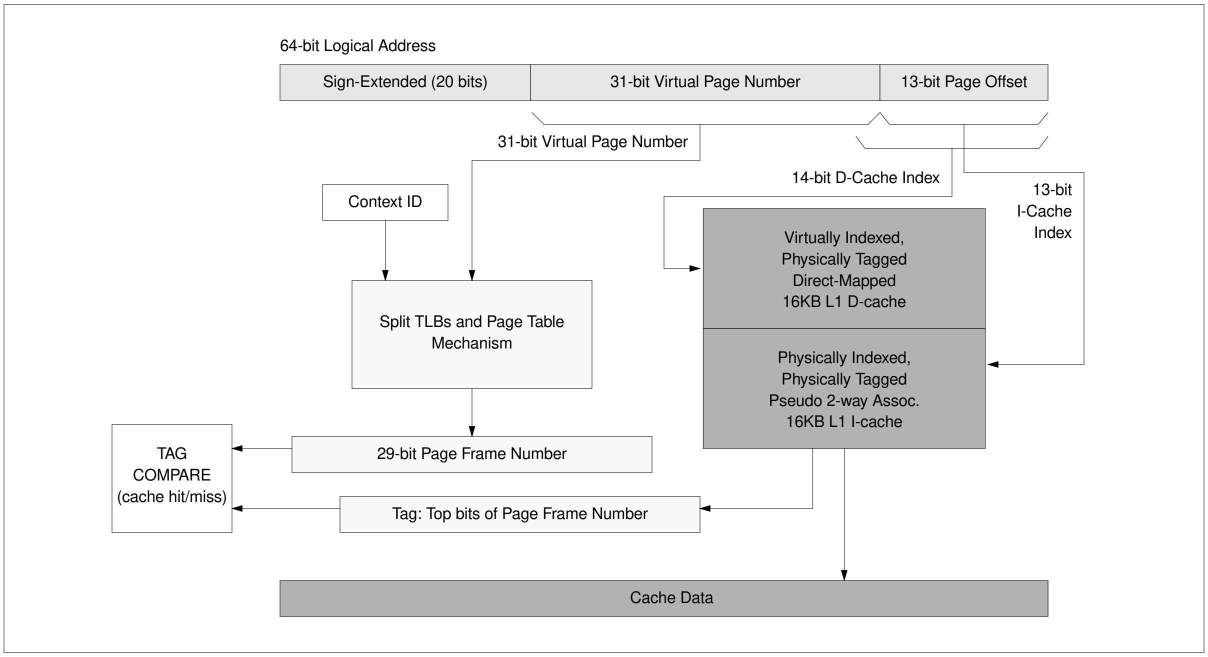

Figure 14. The UltraSPARC address translation mechanism. The SPARC uses address space identifiers as directives to the MMU, encoding such information as byte-ordering (little- or big-endian), and whether or not a given load or store should cause a fault. The ASI is eight bits wide, and the context ID is 13 bits wide. The I-cache is pseudo 2-way set associative, so the cache index is the same size as the page offset; therefore the cache is effectively indexed by the physical address and there can be no cache synonym problems. The data cache is twice the size of the page size and is direct-mapped; therefore the cache is indexed by the virtual address and synonym problems can occur. It is up to the operating system to ensure that such problems do not occur, by restricting where processes can map shared pages. When two pages are shared by two processes they must align to the same half of the data cache.

```
Restricted:
ASI_NUCLEUS                {_LITTLE}
ASI_AS_IF_USER_PRIMARY     {_LITTLE}
ASI_AS_IF_USER_SECONDARY   {_LITTLE}
```

Any ASI with PRIMARY in it refers to the ID held in the PRIMARY context register, any ASI with SECONDARY in it refers to the ID held in the SECONDARY context register, and any ASI with NUCLEUS in it refers to the ID held in the NUCLEUS context register. The {_LITTLE} suffixes, if used, indicate that the target of the load or store is to be interpreted as a little-endian object, otherwise the MMU is to treat referenced data types as big-endian. The NOFAULT directives tell the MMU to load the data to cache without generating a page fault if the data is not actually in memory. This function is used for prefetching.

The default ASI is ASI_PRIMARY; this identifier indicates to the MMU that the current user-level process is executing. Only processes operating in privileged mode are allowed to generate the restricted ASIs; user-level processes can generate non-restricted ASIs. When the operating system executes, it runs under ASI_NUCLEUS, and can peek into the user's address space by generating PRIMARY or SECONDARY ASIs (the AS_IF_USER ASIs). The ASI_SECONDARY identifier can be used for user-level operating system servers that sometimes need to peek into the address spaces of normal processes. For example, a server implementing a particular operating system API, as in Mach or Windows NT, would need to set up and manage the address space of processes running under its environment. It would run as PRIMARY and move the context ID of the child process into the SECONDARY context register. It could explicitly use SECONDARY ASIs to load and store values to the child's address space, and use PRIMARY to execute its own instructions and reference data in its own address space.

Summary

The SPARC architecture implements address space protection through context identifiers. Frequent cache flushing is avoided, as the context ID is thirteen bits wide. The scheme does have the same problems as in the MIPS; ID mapping will eventually occur, necessitating flushing of memory structures—however, this only needs to happen after 8192 processes execute. A SPARC implementation with software-managed TLBs will support sparse address spaces, as the operating system is free to choose its page table organization. Shared memory is implemented by indirectly specifying an alternate context identifier while executing, therefore referencing data in an alternate address space. There is direct support for large address spaces, as processes have free run of a 64-bit space. Superpages are supported in the UltraSPARC by a variable page size of 8KB, 64KB, 512KB, or 4MB, set in the TLB entry.

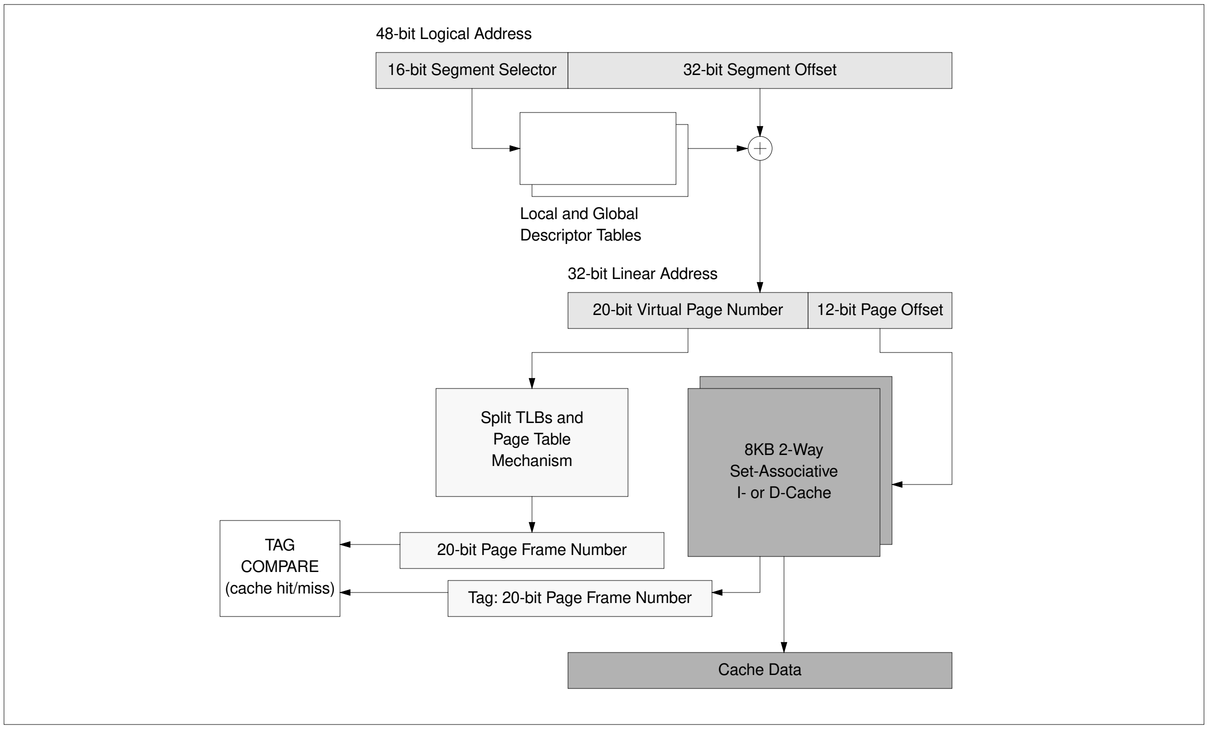

Figure 15. The Pentium address translation mechanism. The Pentium is unique in its use of several different orthogonal address translation mechanisms. It uses a segmentation mechanism much more general than the PowerPC's but which is generally not used since it requires extra memory references on loads and stores. When an access misses in the TLB, the hardware performs a top-down walk of a two-level hierarchical page table. While this is expensive (because every page table lookup takes two memory references), it is creates a very flexible sharing mechanism, and provides simple support for 4MB superpages.

Pentium [6]

The Pentium (Fig. 15) does not provide address space protection directly; there are no address space identifiers and the TLBs are typically flushed on context switch. The caches are physically tagged, and therefore do not need flushing. Protection is usually provided indirectly through hierarchical page tables, which are hardware-defined and hardware-walked in a top-down fashion. The hardware is given the location of the root level page table and walks the two-tiered table to find the appropriate mapping whenever a TLB miss occurs. If every process has its own page table, then the TLB is guaranteed to contain only entries belonging to the current process—i.e. those from the current page table—provided that on context switch the TLBs are flushed and the pointer to the current page table pointer is changed.

Like the PowerPC, the Pentium solves cache aliasing problems by restricting the size of the cache index to the page size of 4KB. Larger virtually-addressed caches can only be obtained by increasing set associativity. Since the page table is walked in a top-down manner, the processor provides easy support for shared 4KB pages or shared 4MB segments; all that is required is shared page table entries. If two page tables have identical entries in two of their root-level PTEs, they share a PTE page at the user page-table level. Therefore, they share a 4MB region of their address spaces. The processor provides additional support for superpages by allowing a root-level PTE to directly map a 4MB region by setting a bit in the TLB entry.

Also like the PowerPC, the Pentium uses a segmentation mechanism to first map user-level addresses onto a global linear address space. Unlike the PowerPC, the segmentation mechanism supports variable-sized segments from 1 byte to 4GB in size, a value that is set by software and can be different for every segment. The Pentium's global space is the same size as an individual user-level address space; both are 4GB. Processes generate 32-bit addresses that are extended by 16-bit segment selectors. Hardware uses the 16-bit selector to index one of two descriptor tables, producing a base address for the segment corresponding to the selector. This base address is added to the 32-bit virtual address generated by the application to form a global 32-bit "linear" address. There is no hardware mechanism to prevent different segments from overlapping one another in the global 4GB space.

The segment selectors are produced both indirectly and according to the context of the operation. At any given time a process can reference six of its segments; these six values 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 a 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.

NOTE: CRAWFORD PATENT?

Summary

The x86 memory management architecture does not provide address space protection; operating systems must implement protection on their own. This is typically done by flushing the TLBs on every context switch and providing a distinct page table for every process, though one could use the segmentation mechanism just as in PowerPC to provide address-space protection and thus avoid having to flush the TLBs. However, this would require all processes to fit into a global 4GB flat virtual address space, which is not much room in which to work. The hardware-defined, hardware-walked hierarchical page table is considered to be an inefficient design for sparse spaces, and extremely inefficient for mapping large (e.g. 64-bit) user address spaces. Shared memory can be implemented at a 4KB page granularity by duplicating mapping information between individual page table entries. Shared memory is also provided at the granularity of 4MB regions by allowing shared page table structures (implemented by duplicating page table entries in the root tables). Both schemes are supported by the hardware. There is no support for address spaces larger than four gigabytes, though there is support for superpages; by toggling a bit a TLB entry can map either a 4KB page or a 4MB superpage.

Conclusion

There is tremendous diversity among today's commercial processors in their support for memory management. This incompatibility contributes significantly to the problems in porting system software between processors; the issues in porting an operating system simply from one processor to another within the same architecture has been shown to be challenging [12]. Porting *between* architecture families is much more difficult because choices available within one family are not often available in another. Software-managed TLBs, as seen in the MIPS, Alpha, SPARC, and PA-RISC architectures, allow an operating system to choose its organization for a page table. Hardware-managed TLBs, as in the Intel and PowerPC, force the operating system to use a certain organization. However, some software-managed designs are less flexible than others; for instance, the Alpha's TLB placement algorithm is performed in PALcode, not by the operating system. The Intel architecture is likely limited to a 32-bit virtual-address unless its page-table organization is to be redefined; the linear address can be expanded beyond 32 bits transparently to user-level code. The Alpha's virtual address size is restricted so that the virtual address space can be mapped by a three-tiered hierarchical page table; should the architecture support larger virtual addresses, a different page table organization could be necessary. The PowerPC's hardware-defined hashed page table may limit the flexibility of the operating system, but it should scale to 64-bit implementations. Moreover, the segmentation mechanism of the PowerPC, like the x86 segmentation mechanism and the multiple address-space identifiers of the PA-RISC, is well-suited to supporting finite-group shared memory—the ability to safely share a page between a small number of processes.

When faced with such diversity in high-performance architectures, it seems that the design of one's memory-management architecture is not likely to lend one a clear performance advantage over one's competitors. Why then the wildly incompatible designs? They only serve to make the porting of applications and operating systems more difficult. With increasing cache sizes (especially the multi-megabyte L2 caches of today's workstations), a simple solution may be to eliminate memory-management hardware completely [7]. If systems used large virtually-indexed, virtually-tagged cache hierarchies, hardware address translation would be unnecessary; virtual caches require no address translation, and if the caches are large enough one will rarely need to go to main memory. Address translation would be performed only on the rare cache miss, and it therefore could afford to be expensive. Instead of hardware, the operating system could perform address translation, resulting in increased flexibility and making the job of porting system software easier.

References

[1] A. W. Appel and K. Li. "Virtual memory primitives for user programs." In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4)*, April 1991, pp. 96–107.

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

[3] Digital. *Digital Semiconductor 21164 (366 MHz Through 433 MHz) Alpha Microprocessor Hardware Reference Manual*. Digital Equipment Corporation, Maynard MA, 1996.

[4] J. Heinrich, Ed. *MIPS R10000 Microprocessor User's Manual, version 1.0*. MIPS Technologies, Inc., Mountain View CA, June 1995.

[5] IBM and Motorola. *PowerPC 604 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola, 1994.

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

[7] B. L. Jacob and T. N. Mudge. "Software-managed address translation." In *Proc. Third International Symposium on High Performance Computer Architecture (HPCA-3)*, San Antonio TX, February 1997.

[8] G. Kane. *PA-RISC 2.0 Architecture*. Prentice-Hall PTR, Upper Saddle River NJ, 1996.

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

[10] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[11] J. Liedtke. "Improving IPC by kernel design." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993, pp. 175–187.

[12] J. Liedtke. "On micro-kernel construction." In *Proc. Fifteenth ACM Symposium on Operating Systems Principles (SOSP-15)*, December 1995.

[13] 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.

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

[15] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. "Tradeoffs in supporting two page sizes." In *Proc. 19th Annual International Symposium on Computer Architecture (ISCA-19)*, May 1992.

[16] S.-Y. Tzou and D. P. Anderson. "The performance of message-passing using restricted virtual memory remapping." *Software—Practice and Experience*, vol. 21, no. 3, pp. 251–267, March 1991.

[17] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient software-based fault isolation." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993, pp. 203–216.

[18] D. L. Weaver and T. Germand, Eds. *The SPARC Architecture Manual version 9*. PTR Prentice Hall, Englewood Cliffs NJ, 1994.

SIDEBAR: Comparison at a Glance

Table 2: Comparison of architectural features

	MIPS	Alpha	PowerPC	PA-RISC	SPARC	x86
Address Space Protection	ASIDs	ASIDs	Segmentation	Multiple ASIDs & Segmentation	ASIDs	Segmentation
Shared Memory	GLOBAL bit in TLB entry	GLOBAL bit in TLB entry	Segmentation	Multiple ASIDs & Segmentation	Indirect specification of ASIDs	Segmentation & sharing PTE pages
Large Address Spaces	64-bit addressing	64-bit addressing	52-bit segmented addressing	96-bit segmented addressing	64-bit addressing	None
Fine-Grained Protection	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry, also per-segment
Sparse Address Spaces	Software-managed TLB	Software-managed TLB	Inverted software TLB cache	Software-managed TLB	Software-managed TLB	None
Superpages	Variable page size set in TLB entry: 4KB-16MB, by 4	Groupings of 8, 64, or 512 pages (set by *granularity hint* in TLB entry)	Block Address Translation: 128KB-256MB, by 2	Variable page size set in TLB entry: 4KB-64MB, by 4	Variable page size set in TLB entry: 8KB, 64KB, 512KB, 4MB	Variable page size set in TLB entry: 4KB or 4MB

Table 2 compares memory-management support, as exhibited in the six commercial architectures discussed, for today's most salient operating-systems features.