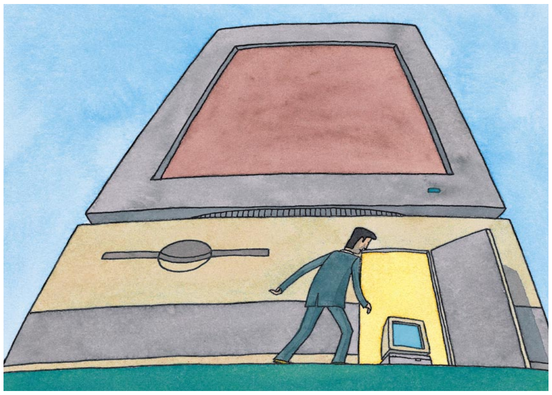
Virtual Memory: Issues of Implementation



The authors introduce basic virtual-memory technologies and then compare memory-management designs in three commercial microarchitectures. They show the diversity of virtual-memory support and, by implication, how this diversity can complicate and compromise system operations.

Bruce Jacob
University of Maryland

Trevor Mudge
University of Michigan

Virtual memory was developed to automate the movement of program code and data between main memory and secondary storage to give the appearance of a single large store.[1] This technique greatly simplified the programmer's job, particularly when program code and data exceeded the main memory's size. The basic technology proved readily adaptable to modern multiprogramming environments, which, in addition to a "virtual" single-level memory, also require support for large address spaces, process protection, address-space organization, and the execution of processes only partially residing in memory.[2] Consequently, virtual memory has become widely used, and most modern processors have hardware to support it.

Unfortunately, there has not been much agreement on the form that this support should take. The result of this lack of agreement is that hardware mechanisms are often completely incompatible. No serious attempts have been made to create a common memory-management support or a standard interface. Thus, designers and porters of system-level software have two somewhat unattractive choices: They can write software to fit many different architectures, which can compromise performance and reliability; or they can insert layers of software to emulate a particular hardware interface, which essentially forces one hardware design to look like another. Inserting this *hardware abstraction layer*[3,4] hides hardware particulars from the higher levels of software but can also compromise performance and compatibility; the higher levels of software often make unwitting assumptions about those hardware particulars, creating inconsistencies between expected and actual behavior.[5]

Here we present the software mechanisms of virtual memory from a hardware perspective and then describe several hardware examples and how they support virtual-memory software (see the architecture sidebars beginning on page 39 for hardware examples). Our focus is on the mechanisms and structures popular in today's OSs and microprocessors, which are geared toward demand-paged virtual memory. However, this focus in no way impedes our goal: to show the diversity of virtual-memory support and, by implication, how this diversity complicates the design and porting of OSs. Our companion article in the forthcoming July/August issue of *IEEE Micro* describes contemporary hardware support for memory management in more detail.[6]

BASIC CONCEPTS

In a well-designed virtual-memory system, the main memory holds only the most often used portions of a process's address space; other portions are stored on disk and retrieved as needed. This creates the illusion of a single-level store with the access time of random-access main memory rather than that of a disk. The OS and hardware support the illusion by translating virtual addresses to physical ones on the fly. This translation occurs at the granularity of *pages*, with support from hardware found in the memory-management unit.

As Figure 1 shows, the virtual-memory space is divided into uniform *virtual pages*, each of which is identified by a *virtual page number*. The physical memory is divided into uniform *page frames*, each identified

0018-9162/98/$10.00 © 1998 IEEE

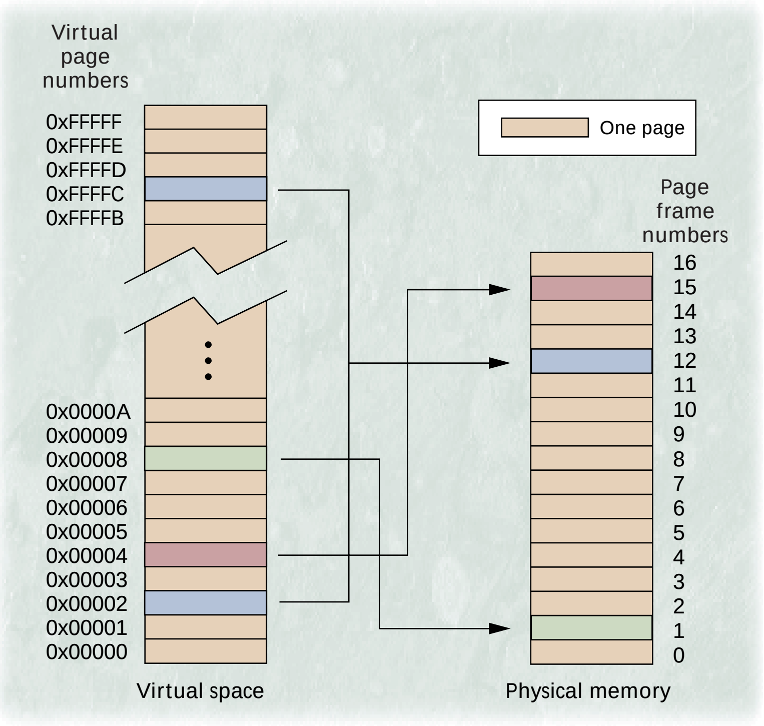


Figure 1. Virtual memory stores only the most often used portions of an address space in main memory and retrieves other portions from a disk as needed. The virtual-memory space is divided into pages identified by virtual page numbers, shown on the far left, which are mapped to page frames, shown on the right.

by a *page frame number*. The page frames are so named because they frame, or hold, a page's data. At its simplest, then, virtual memory is a mapping of virtual page numbers to page frame numbers. The mapping is a function: a given virtual page can have only one physical location. However, the inverse mapping—from page frame numbers to virtual page numbers—is not necessarily a function, and thus it is possible to have several virtual pages mapped to the same page frame. In Figure 1, for example, the OS has mapped two virtual pages (0x00002 and 0xFFFFC) to page frame 12. This type of mapping arrangement, called *virtual-address aliasing,* lets processes or threads share memory and supports different "views" of data with different protections or behaviors. In the latter case, for example, references to a location using one virtual address could cause a prefetch or nonfaulting data load, while references to the same location through a different virtual address could cause a normal load.

An OS *pages* when it moves pages in and out of memory. If space is needed and a particular page has not been used recently, it is *paged out* (stored to disk) and the space is freed up for more active pages. Pages that have been migrated to disk are *paged in* (returned to memory) when they are needed again. If an item can be paged, it implies that the item resides in virtual space: it is accessed using virtual addresses, and space is allocated for its mapping information. The OS allocates physical memory for the item itself only when the item is paged in. A virtual page is considered *mapped* when the OS has information on its location (in memory or on disk). An *unmapped* page is either not yet allocated or has been deallocated and its mapping information has been discarded. Finally, 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, thereby blocking the space for any other use.

Page table entries

Mapping information is organized into *page tables*, which are collections of *page table entries* (PTEs). Each PTE typically maintains information for one page at a time. At the minimum, a PTE indicates whether its virtual page is in memory, on disk, or unallocated. Over time, virtual memory evolved to handle additional functions including address-space protection and page-level protection, so a typical PTE now contains additional information such as whether the page holds executable code, whether it can be modified, and, if so, by whom. Most OSs today, including Windows NT, Linux, and other variations of Unix, support address-space and page-level protection in this way.

From a PTE, the OS must be able to determine

- the ID of the page's owner (the *address-space identifier*, sometimes called an *access key*);
- the virtual page number;
- the page's location in memory (page frame number) or location on disk (for example, an offset into a swap file);
- a *valid* bit, which indicates whether the PTE contains a valid translation;
- a *reference* bit, which indicates whether the page was recently accessed;
- a *modify* bit, which indicates whether the page was recently written; and
- *page-protection* bits, such as read-write, read-only, and so on.

The OS uses the reference and modify bits to implement an approximation to a least-recently-used page replacement policy. The OS periodically clears the reference bits of all mapped pages to measure page usage. The modify bit indicates whether a replaced page must be written back to disk, or can simply be discarded. Many OS texts offer detailed information on page replacement policies.[7]

For efficiency reasons, all of the information an OS needs is rarely stored explicitly in each PTE. Careful organization of the page table may allow some items to be implicit. Actual implementations do not need both the virtual page number and the page frame number; one or the other can often be deduced from the PTE's location in the table. The address-space identifier is unnecessary if every process has its own page table, or if there is another mechanism besides address-space identifiers that differentiates the virtual

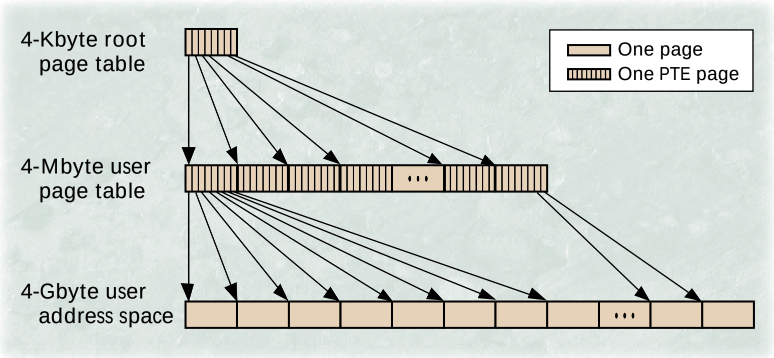


Figure 2. A two-level hierarchical page table. Typically, the root page table is wired down in the physical memory while the process is running, and the user page table is paged in and out of main memory.

addresses generated by unrelated processes. One such example is paged segmentation, in which virtual addresses are translated to physical addresses in two steps: the first is at a segment granularity, the second is at a page granularity. Other items like disk-block information are often placed in secondary tables. The net result is that a PTE can often be made to fit within a 32-bit word.

Translation lookaside buffers

To speed translation, most hardware systems provide a cache for PTEs called a *translation lookaside buffer* (TLB). The TLB takes as input a virtual page number, possibly extended by an address-space identifier, and returns the corresponding page frame number and protection information. The address-space identifier, if used, extends the virtual address to distinguish it from similar virtual addresses produced by other processes. For a load or store to complete successfully, the TLB must contain the PTE mapping that virtual location. If it does not, a TLB miss occurs and the system must search the page table for the appropriate entry and place it into the TLB.

Because the acting agent may vary over the range of different system implementations, when we discuss general mechanisms here we will use *system* to indicate either a hardware mechanism or a software mechanism (usually the OS). For example, in some implementations the OS searches the page table after a TLB miss; in others, a hardware state machine conducts the search.

If the system fails to find the mapping in the page table, or if it finds the mapping but the mapping indicates that the desired page is on disk, a *page fault* occurs. A page fault interrupts the OS, which must then do one of three things: retrieve the page from disk and place it into memory, create a new page if the page does not yet exist (as when a process allocates a new stack frame in virgin territory), or—if the access is to illegal space—send the process an error signal.

Here we 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. There is a form of inclusion between the TLB and main memory: if a page is in memory, its mapping may or may not be in the TLB, but if a page's mapping is in the TLB, the page must be in physical memory.

PAGE TABLE ORGANIZATION

A generation ago, when address spaces were much smaller, a single-level table—called a *direct table*[8]—mapped an entire address space and was small enough to be maintained entirely in hardware. As address spaces grew larger, the table size grew to the point that system designers were forced to move it into memory. They preserved the illusion of a large table held in hardware by caching portions of this page table in a hardware TLB and by automatically refilling the TLB from the page table on a TLB miss.

The search of the page table, called *page table walking*, is therefore a large part of handling a TLB miss. Accordingly, today's designers take great care to construct page table organizations that minimize the performance overhead of table walking. Searching the table can be simplified if PTEs are organized contiguously so that a virtual page number or a page frame number 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 virtual page number; and the *inverse-mapped* or *inverted page table*, indexed by the page frame number.

Hierarchical page tables

Figure 2 shows the classical hierarchical page table, which is based on the idea that a large data array can be mapped by a smaller array, which can in turn be mapped by an even smaller array. For example, if we assume 32-bit addresses, byte addressing, and 4-Kbyte pages, the 4-Gbyte address space is composed of 1,048,576 (2^{20}) pages. If each of these pages is mapped by a 4-byte PTE, we can organize the 2^{20} PTEs into a 4-Mbyte linear structure composed of 1,024 (2^{10}) pages, which can be mapped by 1,024 PTEs. Organized into a linear array, the 1,024 PTEs occupy 4 Kbytes. Since 4 Kbytes is a fairly small amount of memory, most OSs wire down this root-level table in memory while the process is running. As Figure 2 shows, the user page table maps the user space and the root page table maps the user page table. The two levels of the hierarchical table are often called the Level-1 and Level-2 tables, which is a useful naming convention when mapping larger spaces that require more than two levels. For example, the DEC Alpha supports a four-tiered hierarchical page table composed of Level-0, Level-1, Level-2, and Level-3 tables. The limiting case of a hierarchical page table is a single-level table, which, as we mentioned above, is the precursor to the hierarchical table.

There are two access methods for the hierarchical page table: *top down* or *bottom up*. A top-down tra-

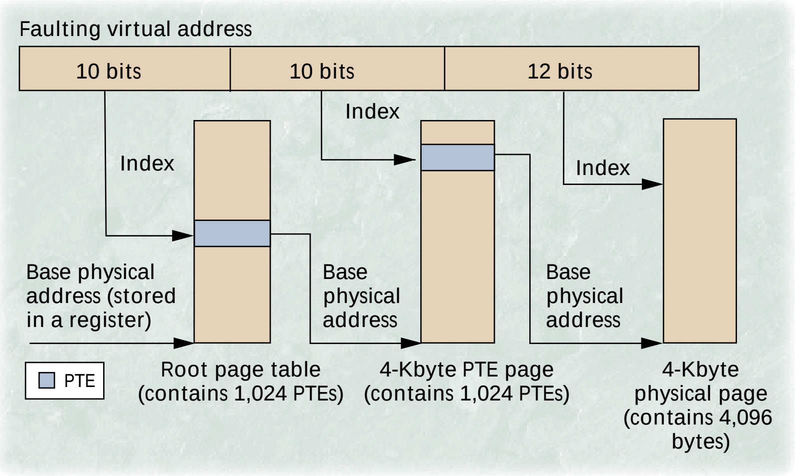

Faulting virtual address
10 bits
10 bits
12 bits
Index
Index
Index
Base physical address (stored in a register)
Base physical address
Base physical address
▨ PTE
Root page table (contains 1,024 PTEs)
4-Kbyte PTE page (contains 1,024 PTEs)
4-Kbyte physical page (contains 4,096 bytes)

Figure 3. The top-down access method splits the virtual address into three fields: the top 10 bits identify a PTE in the root page table that maps the PTE page; the middle 10 bits identify within that PTE page the single PTE that maps the data page; and the bottom 12 bits identify a byte within the 4-Kbyte data page.

versal uses physical addresses to reference the PTEs in the table; a bottom-up traversal uses virtual addresses.

Top-down traversal

Figure 3 shows the steps in the top-down hierarchical page table access. First, the top 10 bits index the 1,024-entry root page table, whose base address is typically stored in a hardware register. The referenced PTE gives the physical address of a 4-Kbyte PTE page or indicates that the PTE page is on disk or unallocated. Assuming the page is in memory, the next 10 bits of the virtual address index this PTE page. The selected PTE gives the page frame number of the 4-Kbyte virtual page referenced by the faulting virtual address. The bottom 12 bits of the virtual address index the physical data page to access the desired byte.

If at any point in the algorithm a PTE indicates that the desired page (which could be a PTE page) is paged out or does not yet exist, the hardware raises a page-fault exception. The OS must then retrieve the page from disk (or create a new page or signal the process) and place its mapping into the page table and possibly the TLB.

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. Due to its simplicity, this algorithm is often used in hardware table-walking schemes, such as the one in Intel's IA-32 architecture.

Bottom-up traversal

The top-down access method requires as many memory references as there are table levels. A bottom-up traversal lowers this overhead and typically accesses memory only once to translate a virtual address. It resorts to a top-down traversal if the initial attempt fails.

In the bottom-up method, the top 20 bits of the virtual address are offset into the 4-Mbyte user page table, which is continuous in virtual space and can be aligned on a 4-Mbyte boundary to simplify the pointer arithmetic. As shown in Figure 2, the index of a PTE in the 4-Mbyte virtually addressed user page table is the same as the virtual page number of the page it maps. On a TLB miss, the virtual address for this user PTE is used to load the PTE from the user page table. If this load is successful, the system inserts the user PTE into the TLB. If this load instead causes another TLB miss, the system must search the 4-Kbyte root page table for the appropriate root PTE. The top 10 bits of the faulting virtual address index this root PTE in the root table. Once the root PTE is in the TLB, a retry of the virtual reference to the user PTE will succeed. The following pseudocode briefly illustrates these steps:

```
load user data /* load misses TLB*/

  /* invokes TLB-miss handler: */
  construct virtual address for user PTE
  load user PTE /* load misses TLB*/

     /* invoke root TLB-miss handler: */
     construct physical address for
        root PTE
     load root PTE /* cannot cause
             * TLB miss, because it uses
             * a physical address */
     put root PTE into TLB
     jump to faulting instruction

  /* return to user TLB-miss handler: */
  load user PTE /* this time, load
     succeeds */
  put user PTE into TLB
  jump to faulting instruction

/* return to user mode */
load user data /* this time, load
   succeeds */
```

Figure 4 shows the bottom-up method. In step 1, the top 20 bits of a faulting virtual address are concatenated with the virtual offset of the user page table. The bottom two bits of the address are zero, because a PTE is four bytes long. The virtual page number of the faulting address is equal to the PTE index in the user page table. Therefore this virtual address points to the appropriate user PTE. If a load using this address succeeds, the user PTE is placed into the TLB and can translate the faulting virtual address.

The user PTE load can, however, cause a TLB miss of its own. In step 2, the system generates a second address when the user PTE load fails. The mapping PTE for this load is an entry in the root page table, and the index is the top 10 bits of the faulting address's virtual page number, just as in the top-down method. These 10 bits are concatenated with the root page table's base address to form a physical address for the appropriate root PTE.

Unlike a virtual address, using this physical address cannot cause another TLB miss. The system loads the root PTE and inserts it into the TLB to map the page containing the user PTE. When the root PTE is loaded into the TLB, the root-table handler ends and the user PTE load is retried. Once this user PTE is loaded into the TLB, the user-table handler ends, and the faulting user-level load or store is retried. Usually, however, the first PTE lookup—the user PTE lookup—succeeds and then a TLB miss requires only one memory reference to translate the faulting user address.

Architectures that use the bottom-up approach include MIPS and Alpha.

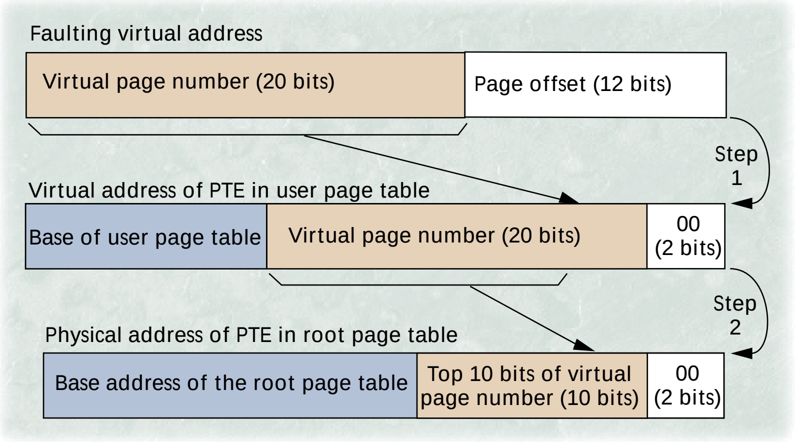

Faulting virtual address
Virtual page number (20 bits)
Page offset (12 bits)
Step 1
Virtual address of PTE in user page table
Base of user page table
Virtual page number (20 bits)
00 (2 bits)
Step 2
Physical address of PTE in root page table
Base address of the root page table
Top 10 bits of virtual page number (10 bits)
00 (2 bits)

Figure 4. The bottom-up method for accessing the hierarchical page table typically accesses memory only once to translate a virtual address. It resorts to a top-down traversal if the initial attempt fails.

Inverted page tables

Figure 5 shows the classical inverted page table, which offers several advantages over the hierarchical table. Instead of one entry for every virtual page belonging to a process, the inverted page table has one entry for every page frame in main memory. The index of the PTE in the inverted table is equal to the page frame number of the page it maps. 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 a designer is dealing with 64-bit address spaces. Because there are rarely unused entries wasting space, it is compact in size, making it a good candidate for hardware-managed mechanisms that need the table to be wired down in memory. Finally, depending on the implementation, inverted page tables can also have fewer memory references to service a TLB miss; however, they have a longer worst-case access time than hierarchical page tables because there is no fixed maximum number of memory references required to find a PTE.

The page table's structure is called *inverted* because it indexes PTEs by page frame numbers rather than virtual page numbers. However, the system typically searches the page table to find the page frame number for a given virtual page number, so the page frame number is not usually available. Therefore the inverted table also uses a hashed index based on the virtual page number. Since different virtual page numbers might produce identical hash values, a collision-chain mechanism is used to let these mappings exist in the table simultaneously. In the classical inverted table, the collision chain resides within the table itself. When a collision occurs, the system chooses a different slot in the table and adds the new entry to the end of the chain. It is thus 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 chain length short, a designer can increase the range of hash values produced and thus the size of the hash table. However, if the inverted page table's size were changed, the page frame number could no longer be deduced from the PTE's location within the table. It would then be necessary to explicitly include the page frame number in the PTE, thereby increasing the size of every PTE. Note that in classical inverted-table implementations, the PTE is already large compared to that of the hierarchical table, as each entry contains a pointer to the next PTE in the collision chain.

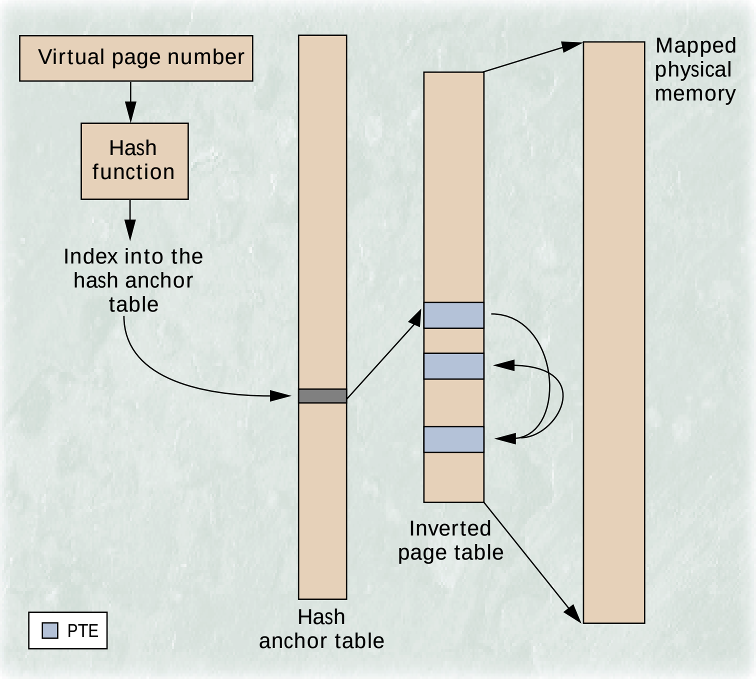


Figure 5. The inverted page table contains one PTE for every page frame in memory, making it densely packed compared to the hierarchical page table. It is indexed by a hash of the virtual page number.

As a trade-off to keep the table small, the designers of early systems increased the number of memory accesses per lookup: they added a level of indirection, the *hash anchor table* (HAT). The hash anchor table is indexed by the hash value and points to the chain head in the inverted table corresponding to each value. Doubling the size of the hash anchor table reduces the

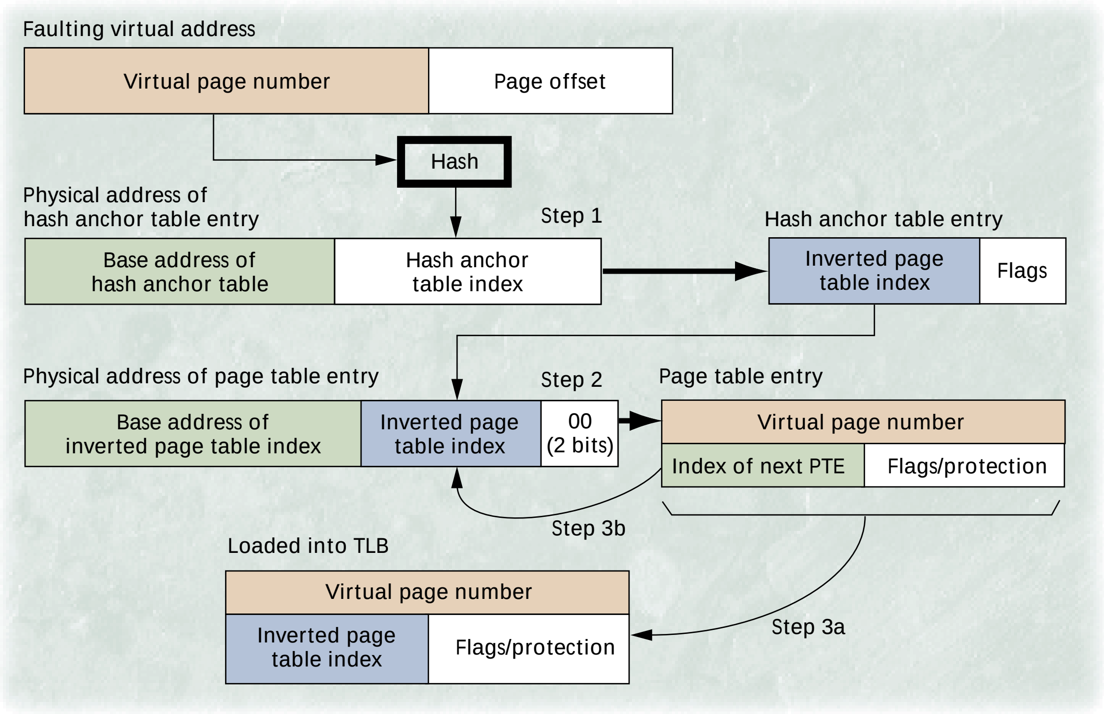


Figure 6. The lookup algorithm for the inverted page table. If the page is mapped and in main memory, the mapping will be found before the algorithm terminates without having to access the disk.

average collision-chain length by half, without having to change the size of the inverted page table. Since the entries in the hash anchor table are smaller than the entries in the inverted table, it is more memory efficient to increase the size of the hash anchor table to reduce the average collision-chain length.

The inverted table has a simple access method that is often used in hardware-walked page table designs, such as the PowerPC. Figure 6 shows the inverted page table's lookup algorithm, which either the OS or hardware can perform. In step 1, the faulting virtual page number is hashed, indexing the hash anchor table. The corresponding anchor-table entry is loaded and points to the chain head for that hash value. In step 2, the indicated PTE is loaded, and its virtual page number is compared with the faulting virtual page number. If the two match, the algorithm terminates. The mapping, composed of the virtual page number and the page frame number (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 there is a next entry, it is loaded and compared. If the last entry fails to match, the algorithm terminates and causes a page fault.

As described in the next section, variations of the inverted table are used in the PA-RISC and PowerPC architectures (as well as in some Sparc-based OSs, notably Solaris).

DETAILS (AND THEIR DEVILS)

When a designer implements a virtual-memory system on real-world hardware, subtle problems surface. The choice between hierarchical and inverted page tables is not an obvious one: there are many trade-offs between performance and memory usage. Implementations of shared memory width vary widely in performance, especially with different hardware support. For example, shared memory with virtual caches requires more consistency management than shared memory with physical caches,[5] and shared memory's interactions with different page table organizations can yield significant variations in TLB performance. The address-space protection scheme is also heavily dependent on hardware support and has great impact on the shared-memory implementation. Also, the TLB can be managed by the OS or managed in hardware, presenting a trade-off between flexibility and performance. Virtual memory is a complex system that integrates several disparate hardware and software mechanisms. It is not surprising that the interactions are often subtle and nonintuitive.

Up, down, or inverted?

The first hierarchical tables were accessed top-down. The most common complaint against this design is that it provides inefficient support for large or sparse address spaces. The top-down access variant wastes time because it requires more than two tiers to cover a large address space, and each tier requires a memory reference during table-walking. The bottom-up variant, which first appeared commercially on the MIPS processor, is more efficient; although it also may require more than two tiers to map a large address space, the bottommost tier is probed first, using an easily constructed virtual address. Because the user PTEs needed to map the user address space are likely to be in the cache, it often requires just a single memory reference to cover a TLB miss.

Both access variants can waste memory, because space in the table is allocated by the OS 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 virtual pages that are contiguous with the first page, they might also be mapped by the existing PTE page. The reason is that the PTEs that map the new pages will be contiguous with the first PTE and will likely fit within the first PTE page. If instead the process adds to its address space another page that is distant from the first page, its mapping PTE will also be distant from the first PTE and is thus unlikely to lie within the first PTE 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—composed of many individual virtual pages spaced far apart—most entries in a given PTE page will be unused, but will consume memory nonetheless. Thus, the organization can degrade to using as many PTE pages as there are mapped virtual pages.

The inverted table was developed in part to address the hierarchical table's potential space problems. No matter how sparsely populated the virtual address space, the inverted page table wastes no memory as the OS allocates space in the table one PTE at a time. Because table size is proportional to the number of physical pages available in the system, a large virtual address does not affect the number of table entries.

However, the inverted organization does have a few drawbacks. First, the table only contains entries for virtual pages actively occupying physical memory. An alternate structure is thus required to maintain information for pages on disk in case they are needed again. The organization of this backup page table could potentially negate the space-saving benefit the inverted organization offers. Second, because the inverted table contains only one entry for each page frame in the system, it cannot simultaneously hold mappings for different virtual pages mapped to the same physical location. For example, suppose that two processes, A and B, share a page. If we map virtual page 1 in process A's address space and virtual page 2 in process B's address space to the same page frame, both mappings cannot reside in the table at the same time. If the processes use memory for communication, the OS could potentially service two page faults for every message exchange.

Like the bottom-up table, inverted tables can be accessed quickly. With a large hash anchor table, each lookup averages just over two memory references. To improve access time further, many of today's inverted page tables—such as in the PowerPC and PA-RISC—eliminate the hash anchor table and hash the inverted table directly. This technique has mixed results. It reduces the minimum number of memory references by one, but it increases the page table's size, requiring the PTE to contain both the virtual page number and the page frame number. This scheme lifts the restriction that the table contain only as many entries as there are page frames in physical memory and allows a designer or the OS to increase the number of table entries to decrease the average collision-chain length. This solves the earlier problem of not allowing multiple mappings to the same page frame to coexist in the page table simultaneously.

MIPS Architecture

MIPS defines a very simple memory-management architecture in which the OS handles TLB misses entirely in software. The OS walks the page table, fills the TLB, and can implement virtually any TLB replacement policy. Figure A shows the MIPS architecture R10000.

The hardware supports a bottom-up hierarchical page table through the *TLB context register*, which holds a virtual address partitioned into a software-loaded segment and a hardware-loaded segment. The software-loaded segment comprises the topmost bits and holds the base virtual address of a user page table. The hardware-loaded segment comprises the bottommost bits and holds the virtual page number of a faulting address. The hardware-loaded segment is filled whenever a user-level reference misses the TLB, and it indexes a single PTE within the linear user page table (this corresponds to the action shown in step 1 of Figure 4). On a TLB miss, the context register contains a virtual address for the very PTE that maps the faulting address. A TLB miss handler can simply perform a load using this address to obtain the mapping PTE.

MIPS uses address-space identifiers to provide address-space protection. To access a page, the address-space identifier of the currently active process must match the address-space identifier in the page's TLB entry. Periodic cache and TLB flushes are unavoidable, as there are only 64 unique context identifiers in the R2000/R3000 and 256 in the R10000. Many systems have more active processes than this, requiring sharing of address-space identifiers and periodic remapping.

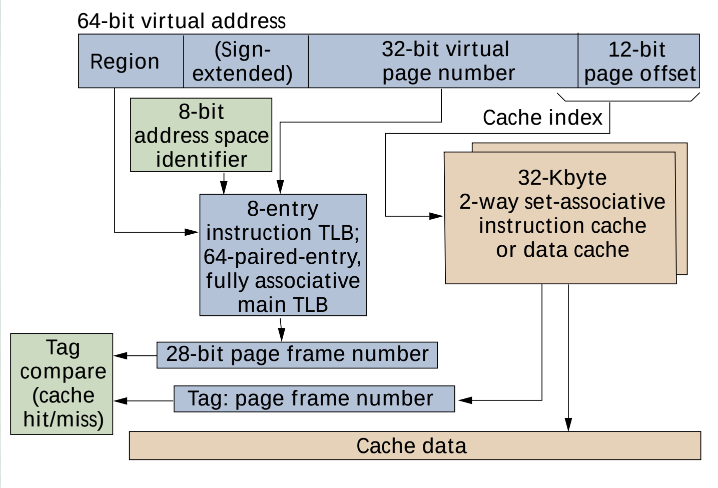

64-bit virtual address
Region
(Sign-extended)
32-bit virtual page number
12-bit page offset
8-bit address space identifier
Cache index
8-entry instruction TLB; 64-paired-entry, fully associative main TLB
32-Kbyte 2-way set-associative instruction cache or data cache
Tag compare (cache hit/miss)
28-bit page frame number
Tag: page frame number
Cache data

Figure A. The MIPS R10000 architecture is a simple software-based memory-management architecture that supports a bottom-up hierarchical page table in hardware. The TLB is accessed in parallel with the virtual cache. The earliest MIPS designs had physically indexed caches, and, if larger than the page size, the cache was accessed in series with the TLB.

Shared memory

Shared memory is often implemented through virtual memory to increase the efficiency of memory usage and decrease execution time. Shared memory lets multiple processes reference the same physical code and data through (potentially) different virtual addresses. When multiple instances of a program run,

PowerPC Architecture

Figure B shows the PowerPC 604, which maps an application's "effective" addresses onto a global flat virtual address space much larger than each per-application address space. Segments are 256-Mbyte contiguous regions of virtual space, and 16 segments make up an application's 4-Gbyte address space. The top 4 bits of the 32-bit 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 that indexes the caches and is mapped by the TLBs and page table.

The PowerPC defines a hashed page table for the OS: a variation on the inverted page table that acts as an eight-way set-associative software cache for PTEs. Its design does not guarantee it will hold all mappings, so it is merely a cache and, like the classical inverted table, requires a backup page table. On TLB misses, hardware walks the hashed page table; software can only insert PTEs into the TLB indirectly by placing them into the page table and retrying the load, which causes a hardware walk of the table.

The architecture does not provide explicit address-space identifiers; address-space protection is supported through the segment registers, which can only be modified by the OS. If two processes have the same segment identifier in one of their segment registers, they share that virtual segment. The OS enforces protection by controlling the degree to which segment identifiers are overlapped. The segment identifiers are 24 bits wide and can uniquely identify over a million processes. If shared memory is implemented through the segment registers, the OS will rarely need to remap identifiers.

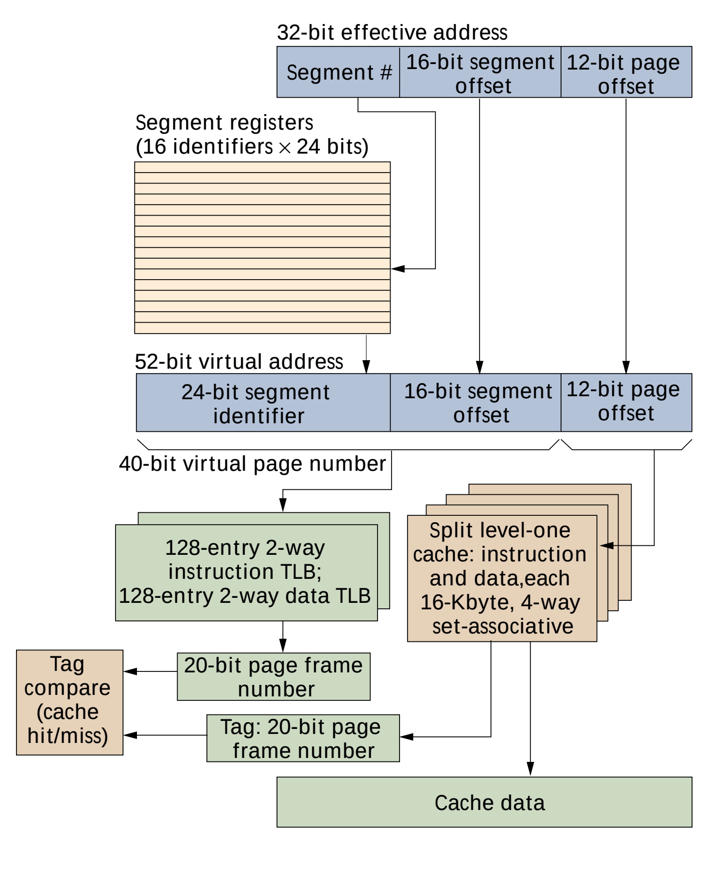

32-bit effective address
Segment #
16-bit segment offset
12-bit page offset
Segment registers
(16 identifiers × 24 bits)
52-bit virtual address
24-bit segment identifier
16-bit segment offset
12-bit page offset
40-bit virtual page number
128-entry 2-way instruction TLB; 128-entry 2-way data TLB
Split level-one cache: instruction and data,each 16-Kbyte, 4-way set-associative
Tag compare (cache hit/miss)
20-bit page frame number
Tag: 20-bit page frame number
Cache data

Figure B. The PowerPC 604 uses a hardware-managed TLB and a variant on the inverted page table—an eight-way software cache for PTEs. The cache index is the same size as the page offset, effectively making the caches physically indexed.

sharing the program code and libraries reduces physical memory requirements. When processes communicate, shared memory can avoid the data copying of traditional message-passing mechanisms.

There are several implementation possibilities for shared memory. One of the most common is virtual address aliasing, or simply *aliasing*. In this scheme, each mapping to the same physical page requires its own logical PTE. Maintaining multiple PTEs makes it possible for different processes to use different virtual addresses for the same physical data, for different processes to map the same physical data with different protections, or for different virtual references to the same physical data to cause different behaviors. For example, two processes can map the same data at different locations within their address spaces. One can map the data read-only while the other maps the data read-write. Accesses through the first virtual address can cause nonfaulting loads, while accesses through the second virtual address can cause normal (faulting) loads and stores.

The primary disadvantage of aliasing is the increased overhead of managing multiple mappings to the same physical page. Every time the OS changes a page's location, for example, it must update every single PTE that maps the page. The multiple mappings also compete for space in the TLB as if every process had its own copy of the shared page. In addition, using several different virtual addresses for the same physical data can lead to confusion as pointers in the shared region may point to other items in that region. For example, a linked list in the shared region causes problems because each process uses different pointer values to reference data within the region, and so each process interprets the pointers in the linked list differently.

A variation on aliasing is to share portions of page tables whenever data is shared. This minimizes the duplication of PTEs. For example, using a top-down hierarchical page table, two processes could share a 4-Mbyte region in each of their address spaces by duplicating a single PTE at the root level of their page tables. Being identical, the two root-level PTEs would map the same physical PTE page, so the two user-level tables would share a page that maps the shared 4-Mbyte region. Note that there is nothing preventing the OS from duplicating the PTEs at different offsets within the root tables, or at multiple offsets within a single root table; this would allow two processes or even a single process to map a physical region at several different virtual addresses. Compared to the normal aliasing mechanism, this variation reduces the overhead of managing PTEs and reduces the impact of multiple PTEs on the TLB. The disadvantage of the scheme is that it can require sharing at large granularities—in this case, 4 Mbytes—and suggests that all mappings to the same physical region should have the same protection, though the latter can be overcome with a little ingenuity, especially if the hardware translation mechanism is segmented and supports protection at the segment level.

Another alternative to aliasing is to have all processes share a global virtual address space. This space is often called a “flat” address space because it is not divided into disjunct per-process spaces by address-space identifiers. Typically, these single address-space OSs map the entire space of all processes with a single page table, which considerably reduces the management overhead. For machines with 64-bit address spaces, this is an attractive choice, as it offers very low overhead and is as flexible as other schemes.

Unix-based OSs tend to implement shared memory through aliasing; PA-RISC systems use a global address space.

Address-space identifiers versus segmentation

If the OS is to provide address-space protection, user-level applications should not have direct access to the code or data of the OS or other applications. Virtual-memory support often ensures this protection. Two common hardware assists for providing address-space protection are *address-space identifiers* and *paged segmentation*.

Address-space identifiers, found in MIPS, Alpha, and Sparc architectures, extend virtual addresses and distinguish them from those generated by different processes. The OS places a process’s address-space identifier in a protected register, and every virtual address the process generates is concatenated with the address-space identifier. The address-space identifier and the virtual page number together make up an extended virtual page number translated by the TLB. In this sense, a 32-bit architecture with 8-bit address-space identifiers has a 40-bit virtual address space. This space is segregated by the address-space identifier: multiple processes can coexist, each thinking it owns the full extent of a 32-bit address space, provided there are enough address-space identifiers to go around. Each process is unable to produce addresses that mimic those of other processes, because to do so it must control the contents of the protected register holding the address-space identifier.

In paged segmentation (as implemented in the PowerPC, PA-RISC, and x86 architectures), virtual–physical translation occurs in two steps. In the first step, user addresses are mapped onto a global address space at the granularity of segments, which are typically (but not necessarily) larger than pages. In the

x86 Architecture

Figure C shows the Pentium Pro, another segmented architecture with no explicit address-space identifiers. Its segmentation mechanism is much more general than the PowerPC’s, but it can require extra memory references while executing instructions to load segment-mapping information. For performance reasons, the x86’s segmentation mechanism often goes unused by today’s OSs, which instead flush the TLBs on context switch to guarantee protection. The per-process hierarchical page tables are hardware-defined and hardware-walked. The OS provides to the hardware a physical address for the root page table in one of a set of control registers, CR3. Hardware uses this address to walk the two-tiered table in a top-down fashion on every TLB miss. If each process has its own page table, the TLBs are guaranteed to contain only entries belonging to the current process—that is, those from the current page table—provided that on context switch the TLBs are flushed and the value in CR3 is changed.

Like the PowerPC, the x86 uses segmentation 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 4 Gbytes in size, and the global virtual space is the same size as an individual user-level address space (4 Gbytes). User-level applications generate 32-bit addresses that are extended by 16-bit segment selectors. Hardware uses the 16-bit selector to index one of two software 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.

For performance, the hardware caches six of a process’s selectors in a set of on-chip segment registers that are referenced by context. One selector is referenced implicitly by executing instructions; its corresponding segment holds code. Another selector maps the stack. The other four map data segments, and a programmer can specify which of the segment registers to reference for different loads and stores.

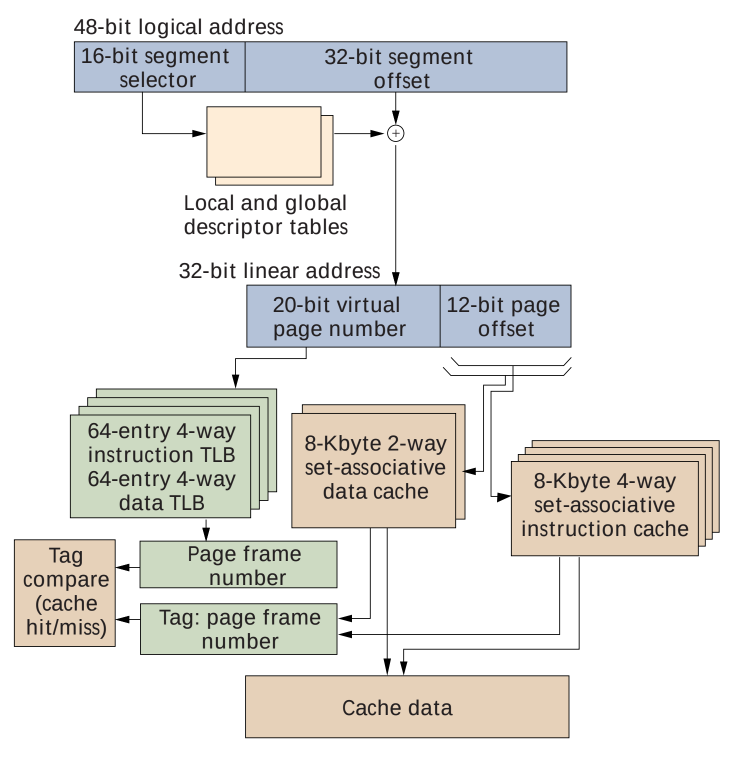

48-bit logical address
16-bit segment selector
32-bit segment offset
Local and global descriptor tables
32-bit linear address
20-bit virtual page number
12-bit page offset
64-entry 4-way instruction TLB 64-entry 4-way data TLB
8-Kbyte 2-way set-associative data cache
8-Kbyte 4-way set-associative instruction cache
Tag compare (cache hit/miss)
Page frame number
Tag: page frame number
Cache data

Figure C. The x86 is a segmented architecture that uses a hardware-managed TLB with a hierarchical table walked top-down. The instruction-cache index is smaller than the page offset, and, like the PowerPC, the data cache index is the same size as the page offset. This effectively makes the caches physically indexed.

Software-managed TLBs are found in MIPS, Sparc, Alpha, and PA-RISC architectures. PowerPC and x86 architectures use hardware-managed TLBs.

second step, virtual addresses from the global space are mapped onto physical memory at the granularity of pages. The top bits of the user's virtual address identify the segment; the bottom bits of the user's virtual address identify an offset within the segment. A process address space is usually composed of many segments, so the OS maintains a set of segment identifiers for each process. Like the hardware scheme for address-space identifiers, the hardware can provide registers to hold the process's segment identifiers, and if those registers can be modified only by the OS, the segmentation mechanism also provides address-space protection.

Segment translation from the process address space to the global address space is like the hardware scheme for address-space identifiers: a segment identifier is combined with the user address to form an extended global address. Instead of concatenating a segment identifier with the user address, the top bits of the user's address are usually replaced by a segment identifier found in a hardware register. In some schemes (PowerPC and PA-RISC), these replaced bits determine which hardware register to choose from. The TLB translates the new address—which may be longer than the original, since the segment identifier may be longer than the bit field it replaces—just like the extended virtual address in the address-space identifier scheme. Segmentation is therefore analogous to having multiple address-space identifiers per process—one for each segment in the user address space. The advantage to this is that sharing at the segment level is as simple as duplicating segment identifiers between two processes. If two processes share a segment identifier, they share a segment; if they do not have any identical segment identifiers, they are completely protected from each other (provided the segment registers are protected from user-level access).

Address-space identifiers and segmentation interact with shared memory differently. Conceptually, shared memory acts in opposition to address-space protection schemes: if each page is tagged with a single protection identifier, and each process is tagged with a single protection identifier, then the only way that multiple processes can access the same page is by circumventing the protection mechanism. With address-space identifiers, this is often done by explicitly duplicating mapping information across page tables (through virtual-address aliasing) or by marking a shared page as visible to all processes—explicitly turning off protection for that page. In the latter case, the page's mapping must be flushed from the TLB whenever a process runs that should not access the page. Segments, in contrast, allow both protection and fine-grained sharing. Two processes can safely share a segment, and can do so without making the segment visible to other processes.

TLBs revisited

When a process attempts to load from or store to a virtual address, the hardware searches the TLB for the address's mapping. If the mapping exists in the TLB, the hardware can translate the reference without using the page table. This translation gives the page's protection information, which is often used to control access to the on-chip instruction and data caches, as opposed to maintaining protection information in the cache for each resident block. If the mapping does not exist in the TLB, the hardware does not have immediate access to the protection information, and it denies the process access to the cache. Even if the data is present in the cache, the process is blocked until the TLB is filled with the correct mapping information.

Either the OS or the hardware can refill the TLB when a TLB miss occurs. With a hardware-managed TLB, a hardware state machine walks the page table; there is no interrupt or interaction with the instruction cache. With a software-managed TLB, the general interrupt mechanism invokes a software TLB-miss handler—a primitive in the OS that is usually 10-100 instructions long. If the handler code is not in the instruction cache at the time of the TLB miss exception, it can take much longer to handle the miss than in the hardware-walked scheme. In addition, the use of the interrupt mechanism adds to the cost by flushing the pipeline, possibly removing many instructions from the reorder buffer. This can result in hundreds of cycles. However, the software-managed TLB design allows the OS to choose any page table organization, whereas the hardware-managed scheme defines a page table organization for the OS. This flexibility in the software-managed scheme can outweigh its potentially higher per-miss cost.[9]

Software-managed TLBs are found in MIPS, Sparc, Alpha, and PA-RISC architectures. PowerPC and x86 architectures use hardware-managed TLBs. The PA-7200 uses a hybrid approach, implementing the initial probe of its hashed page table[10] in hardware, and—if this initial probe fails—walking the rest of the page table in software.

If the hardware provides an address-space protection mechanism such as address-space identifiers or segmentation, the TLB does not need to be flushed on process context switch unless there are globally shared pages. Flushing is typically required only whenever the OS reassigns an address-space identifier or segment identifier to a new process (such as at process creation), or when there are fewer address-space identifiers than currently active processes, which necessitates a temporary ID remapping. Flushing is also required if the protection mechanism goes unused, as is often the case for the segmentation mechanism provided by the x86 architecture. Generally, the TLB need only be flushed with respect to the address-space identifier in question. In most cases, only those entries tagged with that address-space identifier need to be flushed. However, whereas most instruction sets provide an instruction to invalidate a single TLB entry with a specified virtual page number and address-space identifier, most do not provide an instruction that invalidates all TLB entries matching an address-space identifier. As a result, the OS must often invalidate the entire TLB contents or individually invalidate each entry that matches the address-space identifier. Typically, it is cheaper to invalidate all TLB contents than to maintain a list of entries to be flushed on context switch, as this list will typically be large and expensive to maintain.

There is wide diversity in how today's commercial processors support memory management. However, the specific details of one processor's memory-management architecture do not seem to confer a clear performance advantage over another's. Why, then, the incompatible designs? Incompatibility makes porting applications and OSs more difficult. Indeed, system developers often use only a small subset of the complete functionality of memory-management units to make porting more manageable, which results in a suboptimal port. Poor performance stems as much from this fact as from any disadvantage caused by inadequate hardware support.

The hardware–software mismatch in virtual memory is unlikely to change soon, particularly given the industry's penchant for proprietary designs. Although a de facto standard exists in the Intel x86 virtual-memory architecture, competing microarchitectures have yet to adopt it and are unlikely to do so any time soon. Because the x86 offers most of the functionality modern systems require, an emerging architecture may adopt the x86 standard, either as is or in simplified form.

With increasing cache sizes (especially the multimegabyte high-speed Level-2 caches in today's workstations), it is also possible to eliminate memory-management hardware altogether.[11] If systems used large, virtually indexed, virtually tagged cache hierarchies, hardware address translation would be unnecessary. Virtual caches require no address translation when the data is found in the cache, and, if the caches are large enough, there is rarely a need to go to main memory. Address translation would be performed only on the rare cache miss and could therefore afford to be expensive. Instead of using hardware, the OS itself could perform virtual-memory functions—including address translation and protection checks—resulting in increased flexibility and simplifying the job of porting system software. ❖

References

1. T. Kilburn et al., "One-Level Storage System," *IRE Trans.*, Apr. 1962, pp. 223-235.
2. E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass., 1972.
3. R. Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Trans. Computers*, Aug. 1988, pp. 896-908.
4. H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, Wash., 1993.
5. C. Chao, M. Mackey, and B. Sears, "Mach on a Virtually Addressed Cache Architecture," *Proc. Mach Workshop*, Usenix Assoc., Berkeley, Calif., Oct. 1990, pp. 31-51.
6. B.L. Jacob and T.N. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, Aug. 1998, to appear.
7. A. Silberschatz and P.B. Galvin, *Operating Systems Concepts*, 5th ed., Addison-Wesley, Reading, Mass., 1998.
8. H.G. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett, Sudbury, Mass., 1996.
9. D. Nagle et al., "Design Trade-Offs for Software-Managed TLBs," *ACM Trans. Computer Systems*, Aug. 1994, pp. 175-205.
10. J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Int'l Symp. Computer Architecture (ISCA-20)*, IEEE CS Press, Los Alamitos, Calif., May 1993, pp. 39-50.
11. B.L. Jacob and T.N. Mudge, "Software-Managed Address Translation," *Proc. Third Int'l Symp. High Performance Computer Architecture (HPCA-3)*, IEEE CS Press, Los Alamitos, Calif., Feb. 1997, pp. 156-167; http://www.computer.org/conferen/hpca97/77640156.pdf.

Bruce Jacob is an assistant professor of electrical engineering at the University of Maryland. His research interests include the design of hardware architectures for real-time and embedded systems. Jacob received an AB in mathematics from Harvard and an MS and a PhD in computer science and engineering from the University of Michigan, Ann Arbor.

Trevor Mudge is a professor of electrical engineering and computer science and director of the Advanced Computer Architecture Laboratory at the University of Michigan, Ann Arbor. His research interests include computer architecture, computer-aided design, and compilers. Mudge received a BSc in cybernetics from the University of Reading, England, and an MS and a PhD in computer science from the University of Illinois, Urbana-Champaign.

Contact Jacob at Department of Electrical Engineering, University of Maryland, College Park, MD 20742; blj@eng.umd.edu; (301) 405-0432. Contact Mudge at Advanced Computer Architecture Lab, EECS Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122; tnm@eecs.umich.edu; (734) 764-0203