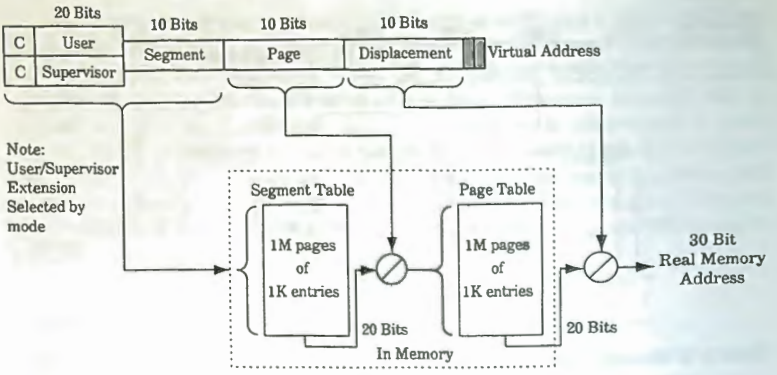


FIGURE 3.14 MC88200 table structure.

a pointer in the leaf tables that point not to the page in real memory but to another leaf table. This feature supports sharing of real pages but can significantly increase the number of references to memory that are required before a page fault is detected. I believe that there is a time-out feature to stop endless looping around indirect addresses.

MC88200. It is interesting to note that Motorola eliminated the programmability of the MC68851 in the MMU of the MC88200 [MOTO90a]. The table structure of the MC88200 is shown in Figure 3.14. This device supports a fixed page size of 4 Kbytes. The 32-bit effective address is extended by 20 bits using two registers found in the MMU that are loaded by the processor. Depending upon the mode of operation, the user of supervisor extension is concatenated to the effective address, providing a 52-bit virtual address. There are two levels of translation. The first level, called the segment table, has 2^{30} entries of 20 bits that are concatenated with the 10-bit page field that address the page table. Motorola calls their system a segmented system; segmentation is provided by software that determines the values to be used in the extension registers. The balance of the system is a hardware-paged system.

As with the i386, i486 page name translation systems, the MC88200 is a partial implementation of a direct map. There are 2^{30} entries in the page table rather than 2^{40} as required for a full implementation. The number of entries is greater than that of the i386, i486, thus requiring fewer allocations of translation information to the page table.

Control Bits

The use of control bits in a multilevel page table system is illustrated with the MC88200, which has an extensive control bit repertoire to

provide flexible support of differing operating systems. The first level of control bits is contained in the two *area descriptors* that are selected by mode to extend the processor effective address, as shown in Figure 3.14. These bits, noted as *C* in the figure, consist of

WT	Write Through: controls the write policy of the cache;
G	Global: controls the scope of snoopy coherency measures (Chapter 5);
CI	Cache Inhibit: inhibits caching of local instructions;
TE	Translation Enable: if not set, no page name translation.

The first-level table, known as the *segment table*, has entries called *segment descriptors*, with the following control bits:

WT	Write Through;
G	Global;
CI	Cache Inhibit;
SP	Supervisor Protection: controls translation of supervisor;
WP	Write Protect: controls write protection;
V	Valid: if not valid, translation fault (table not present).

The second level, known as the *page table*, has entries called *page descriptors*, with the following control bits:

WT	Write Through;
G	Global;
CI	Cache Inhibit;
SP	Supervisor Protection;
WP	Write Protect;
V	Valid (Page is valid);
M	Modified: indicates that the page has been modified;
U	Used: indicates that a page has been accessed.

Note that some of the levels have control bits for the same function. An example is the WT, Write Through bit. These bits are ORed together, and if any one is set the resulting action is taken based on this test of the control bits.

Inverted Page Table Translation

Direct multilevel table translation has served well for processors with virtual addresses limited to 32 bits. With virtual address extensions, partial page table implementations have been required. In the early 1980s designers at IBM contemplated very large virtual addresses and

determined that a different translation method would be required. For a system with a 1-Kbyte page and a 52-bit virtual address, the leaf nodes would require approximately 6×2^{42} bytes. Systems that followed this study are the IBM System 38 with a virtual address of 48 bits [HOUD80, HOUD81] and the IBM RS/6000 with a 52-bit virtual address [OEHL90]. A direct page table for a large virtual address would not only be very large but also sparsely populated and inefficient. The inverted page translation systems described in this section are both small and densely populated.

To reduce the size of the translation table, an inverted page table that contains *only the virtual pages that are currently resident in real memory* can be used [CHAN88] to translate long virtual addresses. An inverted page table can provide a substantial reduction in the size of the page tables and reduce the number of memory accesses required for a page name translation. The page name can be translated into the real address by either an associative search of the translation table, an n-wise set associative search, or by hashing into a linked list. Current design practice does not use n-wise set associative searching, thus associative search and hashing are discussed in the following paragraphs.

All known contemporary implementations of inverted page table virtual memory have been designed by either IBM, Hewlett-Packard, or the IBM/Motorola/Apple group (Sumerset). These systems will be described in the following sections.

Inverted Associative Page Table Search

An inverted *associative page table* (APT) has one entry for each of the pages in real memory. This page table is accessed by an associative search from the page name in the virtual address. As with all known virtual memory systems, an APT system is early select and congruent mapped.

The Atlas [KILB62, MATI77] is an example of an inverted APT. This memory system has a page size of 512 words, a virtual address of 2048 pages, and a real memory of 32 pages (16K words). A block diagram of the map is shown in Figure 3.15. The dotted arrow with *A* at its head signifies an associative search. The page table contains the page frame address and the page name of the pages in real memory, called *tags*. A page name is the key for an associative search of the tags. Note the one-to-one correspondence in the number of entries in the page table and the number of page frames in real memory.

A hit on the tags produces a 5-bit page frame address that is concatenated with the 9-bit displacement from the virtual address. The resulting 14-bit address is presented to the real memory. If the addressed page is not in real memory and therefore not in the page table, a miss occurs and the page is fetched from a drum memory in virtual address space.

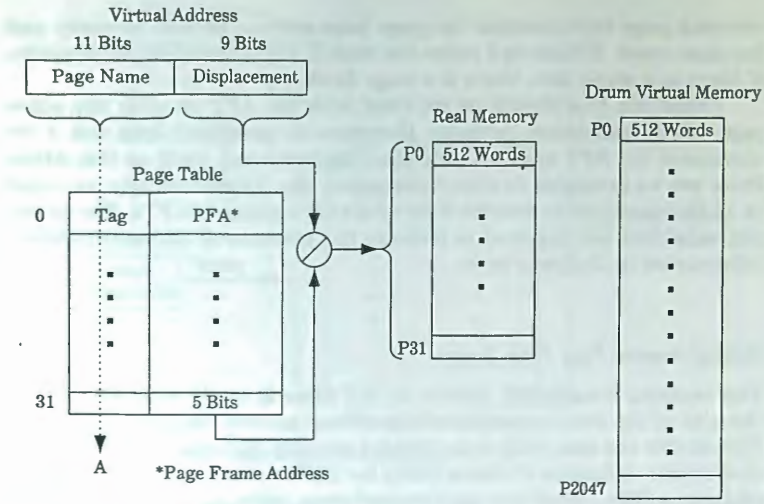


FIGURE 3.15 Atlas page name translation.

The drum address is found by sequentially searching a table holding the 11-bit virtual page number and the corresponding drum address (a process described in Section 3.5.9).

The designers of Atlas considered using a direct table method with the table in real memory. However, this form of translation requires that at least one real memory reference must be processed to see if there is a page fault and to generate the real memory address. This design would make every reference to real memory cost (at a minimum) two memory cycles. The APT method provides a faster associative search of a hardware table, adding only a small increment of time to the real memory access latency. Because it is faster, the associative method is superior and is used on the Atlas even though there was a significant circuit cost for the associative translation table.

For very large virtual addresses and large real memory, the size of an inverted APT is too large and costly for consideration. The small virtual address of the Atlas will never be used again. However, the APT is used in systems in the form of translation lookaside buffers (discussed in Section 3.4).

APT Control Bits

Systems using APT control the page name translation process by an associative search mechanism and require no control bits. Recall that there is one table entry for each real page frame. The entries in the

inverted page table contain the page base address in real memory and the page name. If there is a hit on the search, the translation is complete. If there is a miss, then there is a page fault.

Protection bits should be required with the APT as with the other page name translation systems. However, as protection was not a requirement for APT systems that were implemented, such as the Atlas, there are no examples to cite of protection bits. Protection bits are used in TLBs (described in Section 3.4), which are similar to APTs. For example, valid bits are required to indicate the presence of valid translation information in the page table.

Hashed Inverted Page Table Search

The inverted translation scheme of the Atlas is costly to implement if the size of the real memory and the virtual address become very large. This is now the case with many virtual memory systems. Hashing is the translation technique of choice today for IBM systems with large virtual addresses. The index into an inverted page table is found by hashing the page name (the displacement bits are not hashed). The hash number is then used as an index into the page table, which provides the page frame address. This technique is properly known as *key transformation* [PRIC71]. The key transformation technique reduces the large number of virtual page addresses into one of a number of noncongruent classes in which the real page addresses are linked together.

Hashed access into large address spaces or data structures was developed in the 1960s for access to symbol tables and the like [JOHN61, MORR68]. These techniques were then applied to the accessing of large files and have been, since the late 1970s, applied to virtual memory page name translation. A good hash transformation is one that "spreads the calculated address (sometimes known as hash addresses) uniformly across the available addresses" [MORR68]. The design of hashing functions is beyond the scope of this book, however, a tutorial on hashing can be found in [LEWI88].

It is possible to have a one level translation table that is accessed by hashing the virtual page name as illustrated in Figure 3.16. This method for translating virtual addresses into real memory addresses divides the virtual address into the page name field and the displacement field. The page name field is hashed for an index that accesses the page frame table. The table contains entries for all of the pages that are resident in real memory—an inverted translation system. The page frame table entry carries the page name *tag* and the page frame address of the page in real memory. When the page table is accessed the tag is compared to the page name and, if the comparison is true, the page frame address is gated out and concatenated with the displacement to

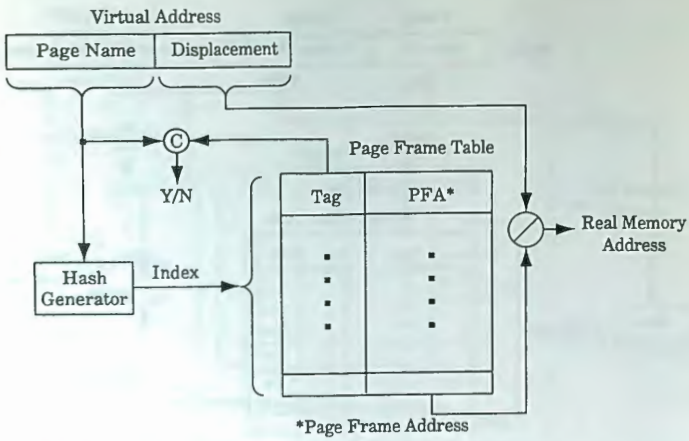


FIGURE 3.16 Hashed page name translation.

form the real memory address. If the comparison is false, the addressed page is not in real memory and a page fault is signaled.

A problem with this simple hashed translation is that two or more virtual addresses can hash into the same value, creating a collision into the same page frame table entry. These collisions increase the page name translation fault rate, significantly increasing the average page name translation time. A number of techniques for managing collisions and reducing their performance impact have been developed and implemented in real systems [MORR68]; a list of these techniques is given in Table 3.2 with examples noted. These examples will be described in the following paragraphs.

One-Level Large Page Frame Table

HP Precision Architecture. A system that uses a one-level page frame is the Hewlett-Packard Precision Architecture as shown in Figure 3.17

Technique	Example
One-level large page frame table	HP Precision Architecture
One-level linked list	None
Two-level linked list	IBM S/38–RS/6000
One-level sequentially indexed list	PowerPC 601
One-level disjoint collision table	Proposed [HUCK93]
n-way set associative	IBM S/360/168 TLB

TABLE 3.2 Collision handling techniques.

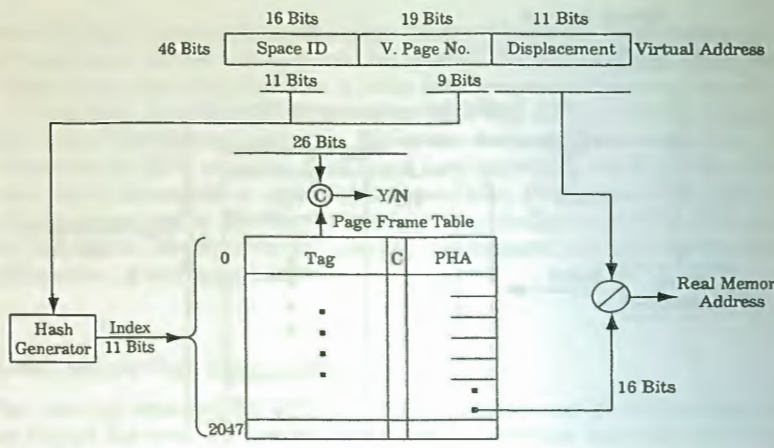


FIGURE 3.17 HP PA page name translation.

[MAHO86, LEE89]. This system depends on reducing the probability of a collision by implementing a large page frame table. Because the translation time through this system is quite fast, a translation lookaside buffer is not used. However, Hewlett-Packard describes this system as a translation lookaside buffer without another table structure for handling TLB misses.

The virtual address is composed of an 11-bit displacement, a virtual page number (either 19, 35, or 51 bits), and a 16-bit space ID. Nine bits of the virtual page number and 11 bits of the space ID are hashed into an 11-bit index. This index references a one-level, direct page frame table, called a translation lookaside buffer by HP. An entry contains a 26-bit tag that is compared to 26 bits in the virtual page name. The table produces a 16-bit page frame address that is concatenated with the 11-bit page displacement. A collision (no match on the tag but with a true valid bit) requires a full virtual software page name translation.

One-Level Linked List Page Frame Table

The approach to collision management discussed in this section is commonly used today because it provides a good balance between speed and cost. A one level system that mitigates the problem of hashing into the same page table entry by using a linked list is shown in Figure 3.18. Each entry of the *page frame table* (the name used in the IBM RS/6000) contains three fields: a page name tag, a page frame address, and a link field (*chain*). When a page is loaded into a page frame in real memory,

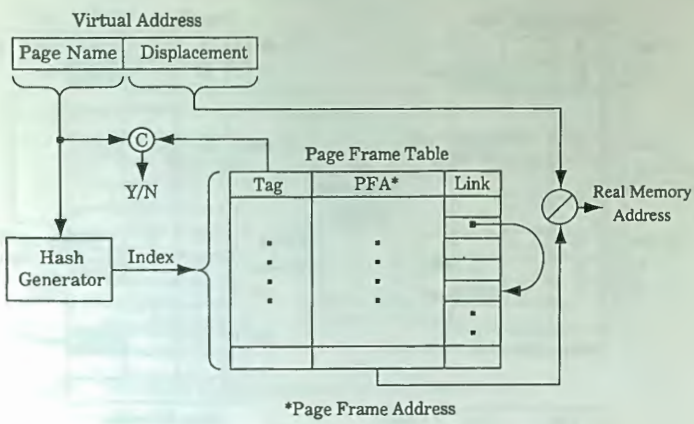


FIGURE 3.18 One-level hash accessed inverted page table.

its page name and the page frame address are placed into the page frame table.

A virtual memory page name translation is performed by hashing the page name to an index into the page frame table. The page name field of the addressed entry is compared to the page name of the virtual address. If they are the same, the correct page frame address is concatenated with the displacement to give the real memory address. If the page name is not the same, a hash collision has occurred and the link pointer accesses another entry, and so on, until there is a match on the tags or a terminator symbol is encountered in the last entry. If the last entry has been accessed without a match on the page names, the referenced page is not in real memory and a page fault results.

A one-level hash accessed translation suffers from a significant problem. For some update cases, the entries in the page frame table must be moved to make room for the allocation of a new entry in the page frame table. Moving entries in the page frame table consumes time and reduces the overall performance of the system. Allocation in a one-level system is illustrated in Figure 3.19, which shows the states of the page frame table before and after allocation. Before allocation we see three linked lists of page translation information. Note that the page names *A* and *B* both hash into "1", the page name *D* hashes into "8", and the page names *P*, *Q*, and *R* hash into "17". In this example, the page frame table entries have been allocated into consecutive locations following the hash address and are linked via the link fields. A valid bit is needed in each entry to identify the page frame table entries that are allocated or vacant.

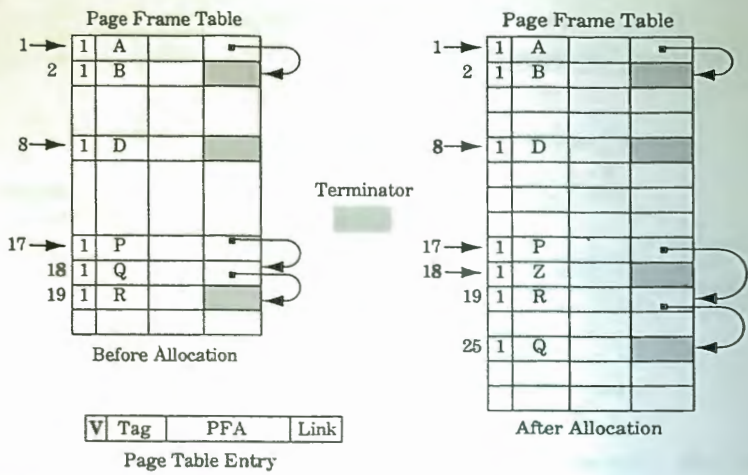


FIGURE 3.19 Allocation with a one-level hash accessed inverted page table.

Consider what happens if a new page is allocated and its page name "Z" hashes to the value "18". The entry "18" in the page frame table is already occupied by Q and must be moved in order to vacate location "18". This can be accomplished by moving the Q entry into location "25" and adjusting the link pointers. A vacant entry is found by scanning the PFT until a "0" is found, signifying a vacant table entry that can be allocated.

If there is a second allocation of a page with a page name that hashes into "18", its translation information is allocated to a vacant page frame table entry and appended to the linked list that starts with "18". Another situation exists for a new allocation hashed to an unused entry in the page frame table (the valid bit is 0). For this case, the tag and address information are allocated and the link field is loaded with the terminator symbol. This is a simple case, and no entry movement is required.

Two-Level Linked List Page Frame Table

A solution to the problem of page frame table movement is to introduce another level in the translation process that, in effect, provides an indirect pointer to the translation information in the page frame table. With a level of indirection, allocating new entries to the page frame table does not require that other entries be moved. This first-level table, named a *scatter index table* in the early literature, is known as a *hash index table* in the IBM System 38, a *hash table* in the IBM 801, and a

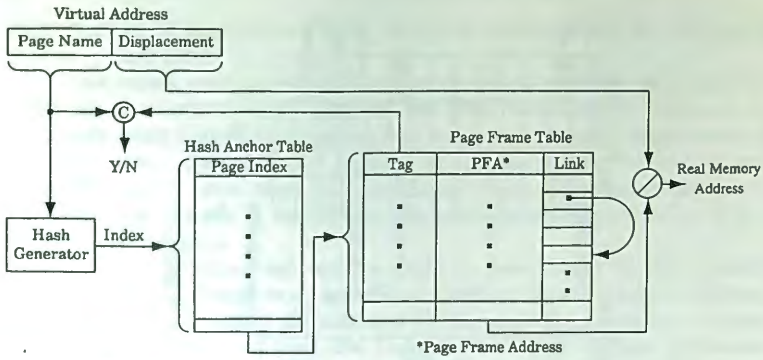


FIGURE 3.20 Two-level hash accessed inverted page table.

hash anchor table with the IBM RT PC and RS/6000, which is the term used in this text.

A two-level system is shown in Figure 3.20 [JOHN61, MORR68]. With this system, the page name is hashed for an index into the hash anchor table, which contains an index into the page frame table. The page frame table contains the page name (that is compared to the virtual address page name), the page frame address, and a link. The page frame address is concatenated with the displacement to form the real memory address.

The allocation example in Figure 3.18 used for the one-level system is illustrated in Figure 3.21 with the before and after cases shown. Here, the hash anchor table contains a pointer into the page frame table. For this example, the linked list *A*, *B* is located starting at location 100 of the page frame table, *D* is located at 200, and the linked list *P*, *Q*, *R* starts at 300. A memory reference to pages *P*, *Q*, or *R*, for example, hashes into address "17" and the pointer 300 is found that indexes to location 300 of the page frame table. The proper page frame address is found by comparing the tags to the virtual page name and traversing the linked list if necessary.

When page "*Z*" is allocated, location "18" in the hash anchor table is vacant and the page frame table entry is placed in a vacant location, 250 for example, of the page frame table. No movement of page frame table entries is required to effect the allocation of the new page. If a page is allocated with a page name that hashes into an already used hash anchor table address, the allocation can be to any vacant location in the page frame table. It is only necessary to append the entry to the linked list. The more successful the hashing function is in providing a uniform

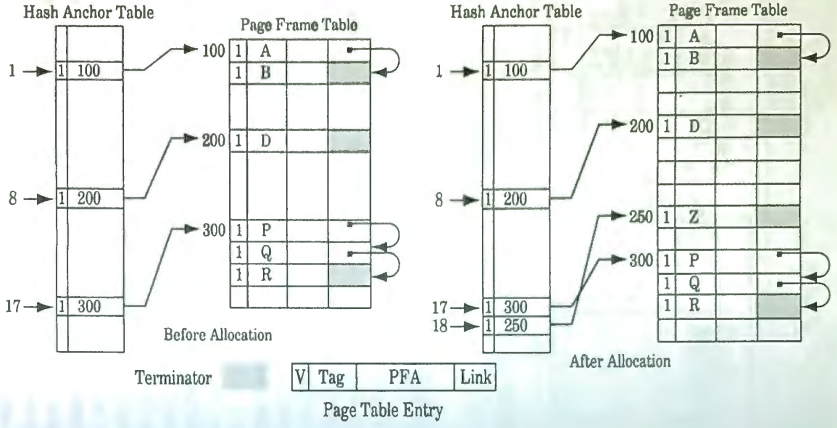


FIGURE 3.21 Allocation with a two-level hash accessed inverted page table.

distribution of transformed keys, the more balanced will be the length of the linked lists.

How many entries should there be in the page frame table and the hash anchor table? First consider the page frame table. Because an inverted page translation system has one entry for each page frame of real memory, the minimum number of entries is the number of real page frames. A large page frame table will have more vacant positions, reducing the length of the linked lists and the time required to find a page frame address.

The size of the hash anchor table is determined by the desired performance. A large hash anchor table has a larger hashed address, reducing the frequency of collisions in the page frame table. Johnson [JOHN61] determined that the average number of probes (reference attempts) into the page frame table is a function of the size of the page frame table and the hash anchor table:

$$P \approx 1 + \frac{\text{no. of entries in page frame table}}{2 \times \text{no. of entries in hash anchor table}}$$

where P is the average number of probes into the page frame table to translate a page name.

For example, if the hash anchor table has the same number of entries as the page frame table, 1.5 probes are required, on average, into the page frame table plus the access of the hash anchor table to translate a page name. If the hash anchor table has only 10% of the entries of the page frame table, six probes are required on average. An explanation of this model is found in the following consideration. If there is only one entry in the hash anchor table, every page name hashes into the same entry and all of the entries of the page frame table are linked together in one linked list. In this situation, the average number of probes approaches half the number of entries in the page frame table. This behavior is as expected from the time required to search a linked list sequentially. On the other hand, if the hash access table is very large, there are no collisions and each entry in the page frame table is unique (not linked). In this case only one probe is required.

Four IBM systems (S/38, 801, PC RT, and RS/6000) use a two-level hash access technique and are discussed in the following paragraphs.

IBM S/38. The virtual memory of the S/38 resulted from research at IBM in the late 1960s and early 1970s on the so-called Future System, which was abandoned due to lack of compatibility with S/370 [LEVY84]. The S/38 virtualized a segmentation system on top of the hardware-paged system [HOUD79, HOUD81]. The S/38 has a 48-bit virtual

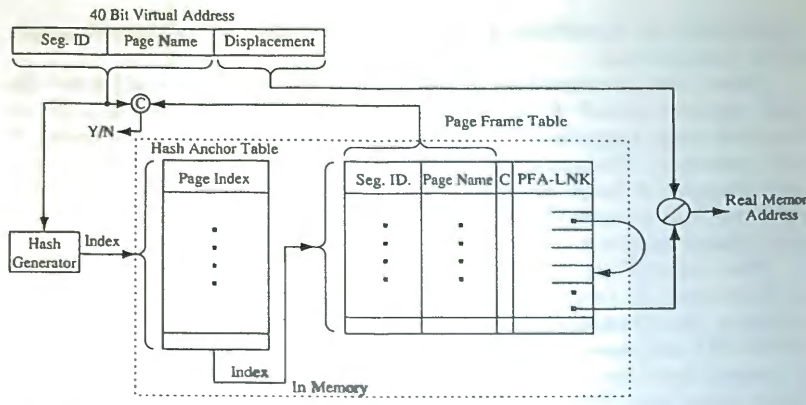


FIGURE 3.22 IBM 801 page name translation.

address, with a 512-byte page. The 39-bit virtual page name is hashed by logical XORing three fields (two of which are bit reversed) in the virtual page name in order to generate a 7-bit index into the hash anchor table. A page frame table index that indexes into the 128-entry page frame table is produced, and the entries in the page frame table are linked together with pointers. Approximately 2.5 memory accesses are required to translate a page name in the page frame table; the total translation time is 3.5 accesses.

IBM 801. The inverted page table system of the S/38 evolved into the virtual memory system of the experimental IBM 801; its page name translation tables are shown in Figure 3.22 [CHAN88]. The IBM 801 generates a 32-bit effective address that is expanded to a 40-bit virtual address using a map that is similar to that adopted by the RS/6000 shown in Figure 3.5. Page name translation is via a hash anchor table and a page frame table stored in real main memory. Unfortunately the published literature does not give the sizes of the three fields of the virtual address.

The IBM 801 design combined the page frame address and link into one field of the page frame table. This field is noted in Figure 3.22 as the PFA-LNK field and is large enough to address any page frame in real memory (at the page level) or to address the next entry in the page frame table [CHAN88]. The operation of this combined field is described below.

A page name extended with a segment ID is hashed for an index into the hash anchor table, which then provides an index into the page frame table. If there is not a match on the page name, the link field

indexes into another entry. If there is a match, the link field is the page frame address in real memory and is concatenated with the displacement field from the virtual address. In other words, the link field is multiplied by 2 to the power of the displacement bits (left shifted the number of bits of the displacement). When there is not a match, the link field has its MSBs set to zero and the resulting value becomes the index into the page frame table. This process assumes that the page tables are allocated in the low address page frames of real memory.

Both the hash anchor table and the page frame table are stored in real memory. Therefore, a translation requires a minimum of two memory accesses to translate a page name. The size of the hash anchor table and page frame table are set when the system is generated depending upon the amount of installed real memory in the system. The hash anchor table can be set to be either equal to or twice the size of the page frame table.

IBM PC RT. The IBM PC RT also stores both the hash anchor table and the page frame table in memory [SIMP86, HEST86], as shown in Figure 3.23. The 32-bit effective address is expanded to a 40-bit virtual address by means of a 16-entry segment register. The page size can be set at either 2 Kbytes or 4 Kbytes, leaving either 28 or 29 bits that are hashed by an XOR operation for an index into the hash anchor table.

Experience with the IBM RT PC [CHAN88] shows that 2.5 storage accesses, or 1.5 probes to the page frame table, are required per page name translation. From this result we would assume that the hash anchor table has the same number of entries as the page frame table—a conclusion based on the Johnson [JOHN61] model. As with the IBM 801, setting the size of these tables is done when the system is generated.

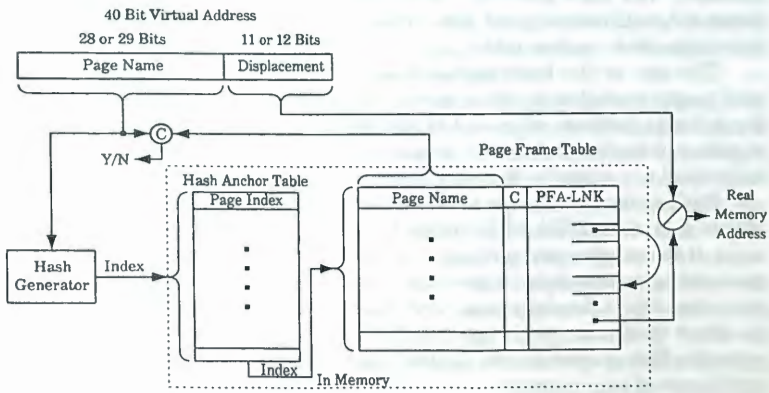


FIGURE 3.23 IBM PC RT page name translation.

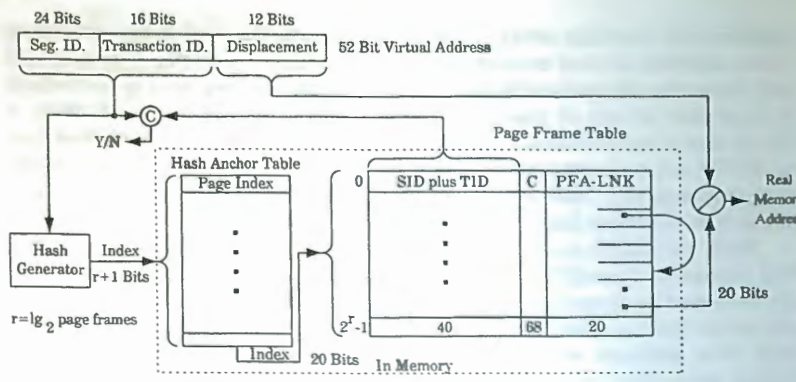


FIGURE 3.24 IBM RS/6000 page name translation.

IBM RS/6000. The IBM RS/6000 continues the approach started with the IBM Future System development [IBM90]. Because the RS/6000 is a recently introduced computer, its virtual memory system is described in more detail than the other IBM systems described above. The RS/6000 can be viewed as a paged-segmented system with the segments defined by the address map of Figure 3.5; the balance of the system is pure paged. A block diagram of the RS/6000 page name translation is shown in Figure 3.24.

The 52-bit virtual address is divided into three fields: a 24-bit segment ID, a 16-bit transaction page ID, and a 12-bit byte displacement [IBM90]. The hash generator XORs the 24-bit SID and the 16-bit TID extended with zeros to 24 bits. The low-order $r + 1$ bits of the result index the hash anchor table.

The size of the hash anchor table is set to be twice the number of real pages installed on the system, while the number of entries in the page frame table is set equal to the number of pages in installed real memory, which can reach a maximum of 2^{20} pages. These table size selections are made by the operating system.

Each entry in the page frame table consists of 16 bytes (4 words). A 20-bit link field (PFA-LNK) links to the next entry if the comparison fails. If the comparison succeeds, the link value of the entry point to the succeeding comparison is the page frame address and is concatenated with the 12-bit displacement, providing a 32-bit real memory address. In other words, on a hit the link field is multiplied by 2^{12}, while on a miss the link is used as the address into the page frame table in the low addresses of real memory.

		S/38	801	PC-RT	RS/6000
Names	Level 1	Hash Index Table	Hash Table	Hash Anchor Table	Hash Anchor Table
	Level 2	Page Directory Table	Page Table	Inverted Page Table	Page Frame Table
Sizes	Level 1	128		r	2^{r+1} up to 2^{21}
	Level 2	64		r	2^r up to 2^{20}
No. of Real Memory Page Frames					up to 2^{20}
Page Size		512 bytes		2 K or 4 Kbytes	4 Kbytes
Virtual Address		48 bits		40 bits	52 bits
Max Real Memory Size				2^{22} bytes	2^{32} bytes

Note. $r = \lg_2$ (Number of page frames in real memory).

TABLE 3.3 IBM hash accessed inverted page tables.

A linked list presents the danger of an infinite translation loop due to a programming error. That is, the links can create a circular list and, if there is no hit on tag comparisons, the translation can run forever. To prevent this from happening, a maximum of 127 probes are permitted before a search is declared a failure. The number of probes for the other systems described above is not known.

The IBM systems have evolved over a period of time. In order to illustrate this evolution, Table 3.3 contains a summary of the relevant parameters of the IBM systems discussed above.

As noted previously, the hash anchor tables and page frame tables are too large to be stored in fast hardware registers. Consequently, it is necessary to enhance the performance of these systems by using a translation lookaside buffer [HOUD81, CHAN88, OEHL90]. After a page miss, the requested page is loaded into the real memory. In addition, the TLB, the hash anchor table, and page frame tables are updated. A subsequent reference to this page will find the page name translation information in the TLB. However, later references may find that an address cannot be translated in the TLB and recourse to the translation tables is required. After this translation, the TLB is updated. TLBs are described in Section 3.4 and memory allocation is discussed in Section 3.5.2.

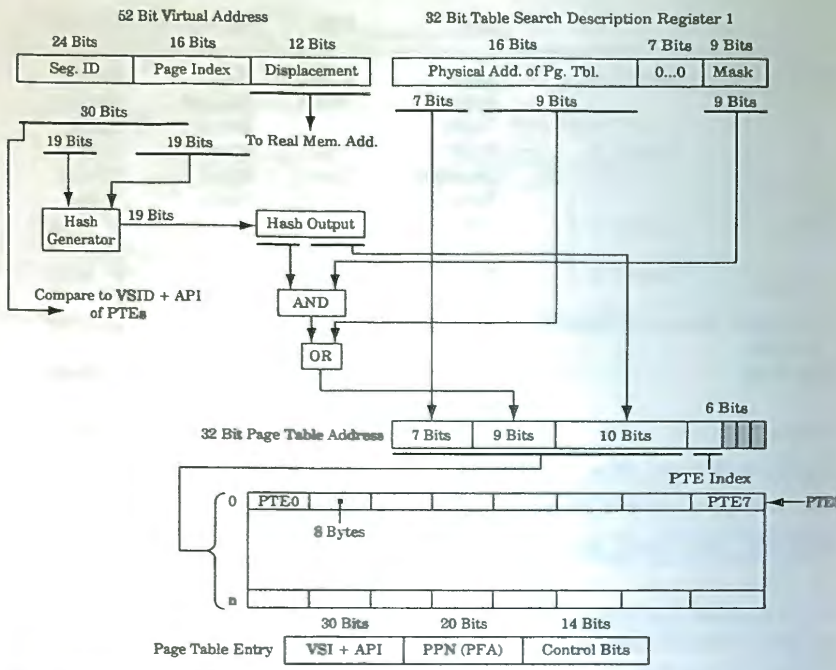


FIGURE 3.25 PowerPC 601 page name translation.

One-Level Sequential Indexed Page Frame Table

PowerPC 601. The PowerPC 601 uses a hash-indexed inverted page name translation that resolves collisions by sequentially searching the page table entries. A simplified description of this system is given and illustrated in Figure 3.25; the terminology of [MOTO93] is used.

The virtual address is formed by a map that is similar to that of the RS/6000 shown in Figure 3.5. This 52-bit address is used in conjunction with a 32-bit register called the table search description register 1. A 19-bit hash index is formed by taking the XOR of the 19 LSBs of the segment ID and the 16-bit page index, zero extended to 19 bits. The 9-bit mask of the TSDR1 is ANDed with the 9 MSBs of the hash output; the result is ORed with 9 LSBs of the physical address field of the TSDR1. These fields are concatenated to form the 32-bit page table address.

The page table is organized as a collection of the 64-bit (8 bytes) page table entries. The page table address indexes into a *page table entry*

Total Main Memory	Memory for Page Tables	Number of PTEGs	Number of PTEs
8 Mbytes	64 Kbytes	1 K	8 K
128 Mbytes	1 Mbytes	16 K	128 K
4 Gbytes	32 Mbytes	512 K	4 M

TABLE 3.4 PowerPC 601 page table size recommendation.

group (PTEG) consisting of eight PTEs. The VSI + API bits of PTE0 are compared to 30 bits of the virtual address. If there is a miss, the low-order bits of the page table address are incremented by 8, to give a new PTE, and the tag of PTE1 is compared. This process continues until all eight PTEs have been tested. If there is still no hit, a second hash function is applied and the page table is again accessed. An excerpt of the recommended size of the page table is given in Table 3.4.

As with other table translation systems, the PowerPC 601 uses a translation lookaside buffer (described in Section 3.4).

One-Level Linked List Into a Disjoint Collision Table

J. Huck et al. [HUCK93] describes an inverted name translation system that combines a one-level system with the combined page frame address and link field of the IBM systems. This system is shown in Figure 3.26 and is called by the author a *hashed page translation table*. The first allocated entry to a hashed index is placed in the page frame table, and subsequent allocations are placed in the disjoint collision resolution table.

The page name is hashed, providing an index into the page frame table. A collision is followed by the link accessing another entry in a disjoint address space called the *collision resolution table* that cannot be accessed by a hashed index. Thus when new page information is allocated, its entry can go into (1) the page frame table if vacant or (2) any vacant slot in the collision reservation table. The entries in the collision resolution table are organized as a linked list. The advantages of this system are that noncolliding translations require only one memory reference and that the need to move entries in the page frame table is eliminated with disjoint link entry storage. As with all hashed access systems, increasing the size of the table will reduce the probability of a collision.

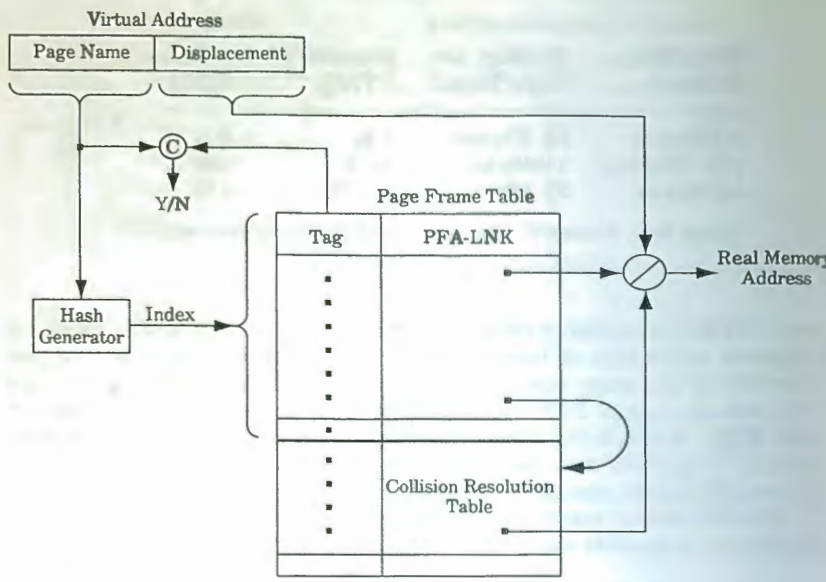


FIGURE 3.26 Hashed page translation tables.

Control Bits

This section uses the IBM RS/6000 design to illustrate the control bits used in these systems. The hash anchor table has only one control bit, a valid bit that is used to indicate whether or not there is a valid pointer into the page frame table. If valid, the translation proceeds; if not, then a page fault results because an inverted page table has entries for all resident pages. In addition, there are various control and protection bits.

i	Invalid	l	Lock Type
f	Reference	w	Grant Write Lock
c	Change	r	Grant Read Lock
p	Protection keys (4 bits)	a	Allow Read
b	Lock Bit for cache lines		

Other bits provide a transaction ID, and there are a number of reserved bits. Note that the RS/6000 treats valid bits nonuniformly across the hash anchor table and the page frame table. The valid bit of the hash anchor table signifies that the entry is vacant and can be allocated to a new translation. The valid bit in the page frame table signifies the presence of the requested page in real memory, assuming a hit on the page name comparison.

3.3.2 *Segmented Systems*

Segmented systems have a variable-length allocation unit known as a *segment* that should not be confused with the fixed-length virtual address extensions shown in Table 3.1. Segment name translation can be accomplished with both one-level and multilevel systems. A one-level system is described first. The operating system sets the length of a segment under the constraint that the sum of all active segments must be no greater than the available real memory.

This discussion on segmented systems is brief because of the many problems associated with these systems. No segmented systems have been designed since the first early machines; paged-segmented systems have replaced them. Interactive programming systems such as Lotus 1-2-3 virtualize a rudimentary form of segmentation to provide interactive allocation and de-allocation of segments.

The motivation for segmentation is that the size of each allocation unit can be set to reduce internal fragmentation. With fixed size pages, some fraction of a page frame (on average, 1/2 page) will be wasted when allocating variable-length records, an issue extensively studied by Wolman [WOLM65]. Because real memory was very expensive, wasted memory was a cost that was difficult to justify and segmentation minimized this undesirable cost. However, as will be discussed, segmentation introduces a number of undesirable problems that must be solved.

The address translation process for a segmented system consists of the steps [DENN76]

$$\text{virtual address} \Rightarrow (n, d) \Rightarrow (f(b), d) \Rightarrow f(b) + d = \text{real address}.$$

That is, the virtual address is first de-allocated into the pair (segment name (n), displacement (d)). The segment name portion is translated by a function f, and the result is added to the displacement. Recall that for paged systems, the translated page name and displacement are concatenated because the allocation unit is fixed at some binary multiple.

The design of the processor, by selecting a displacement field length, sets the maximum length of a segment. Any allocated segment must be equal to or less than this length. Thus, the *segment length* (SL) field of the segment table will be the same length as the virtual address displacement field. When a segment is accessed, the contents of the displacement and segment length fields are compared and the access is valid if $(d) \leq (\text{SL})$.

Figure 3.27 shows a one-level segment translation table that is recognized as a direct map with an entry for every virtual segment. Because $f(n)$ is not constrained, segmented systems are not congruence mapped. That is, segments can be of variable length with the maximum length

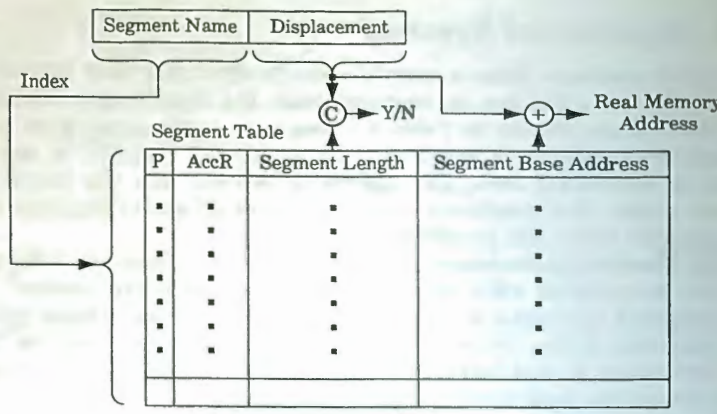


FIGURE 3.27 One-level segment table.

set by the displacement d. Variable-length segments lead to the requirement that there must be a way of specifying the length in the segment table.

Segment table entries have at least the following fields:

1. present bit, similar to the present bit of a one-level direct page table,
2. access rights or control bits,
3. segment length,
4. segment base address.

The segment name indexes into the segment table. If the present bit is set, the access rights are proper, and the segment length is equal to or greater than the displacement value, then the segment base address is added to the displacement providing the real memory address. Note that because the segment base address is added to the displacement, segments can overlap, unlike paged systems. If the present bit is not set, a missing segment fault is indicated that requires a segment fetch from virtual memory. When a segment is allocated to real memory, its length is placed in the segment length field of the segment table. If the segment length is less than the displacement value or if the access rights do not check, an error signal is generated and the name translation process will not continue.

There are a number of serious problems with segmentation. For example, it can be more difficult to find a place in real memory to allocate a requested segment. This allocation problem exists because of the need to find a contiguous space in real memory that is large enough for

the new segment, a problem similar to Relocatable Partitioned Memory described in Figure 3.2.

Another problem with segmented systems is *external fragmentation*, previously defined. As segments of various sizes are moved in and out of real memory, open or unused spaces appear in real memory and the available space becomes fragmented. It becomes difficult, if not impossible, to find a contiguous space for a new segment even though there is ample, but fragmented, space in real memory. From time to time the operating system must compress or compact the allocated spaces to open up contiguous space for new segments. Note that as the allocated space is exactly the space needed, there is no internal fragmentation as with a paged system, one of the benefits of a segmented system. Section 3.5.2 discusses the allocation of segments.

I am only aware of one system that is implemented with a one-level segment table, the Rice University Computer [LEVY84]. This processor, implemented in the early 1950s, had a virtual address that specified a maximum 32-K word segment (a 15-bit displacement field). The experimental purpose of this computer was to evaluate capabilities addressing objects in conjunction with a segmented virtual memory.

Control Bits

The use of the present bit was noted above. The access rights bits provide for various levels of protection. For example, read-only data segments and read/write segments can be indicated in the access rights field. Other information such as lexical level and the data type of the segment can also be indicated.

Multilevel Segmentation Systems

A multilevel segmented system uses the first-level table to select subsequent translations for each of the active processes as shown in Figure 3.28. Note that the first level is indexed by the MSBs of the segment name and organized like a paged system with the first level similar to the map shown in Figure 3.5. The second level is identical to that shown in Figure 3.27.

The LSBs of the segment name and the table address are concatenated to form the segment table address, thus segment overlap is prevented. Said another way, there are 2^B nonoverlapping segment tables of $2^{\text{Table Address}}$ entries. I know of no actual system that is implemented with multilevel segment tables. Concurrent with the realization of the problems with pure segmentation, paged segmentation became the implementation method of choice and is discussed in the following section.

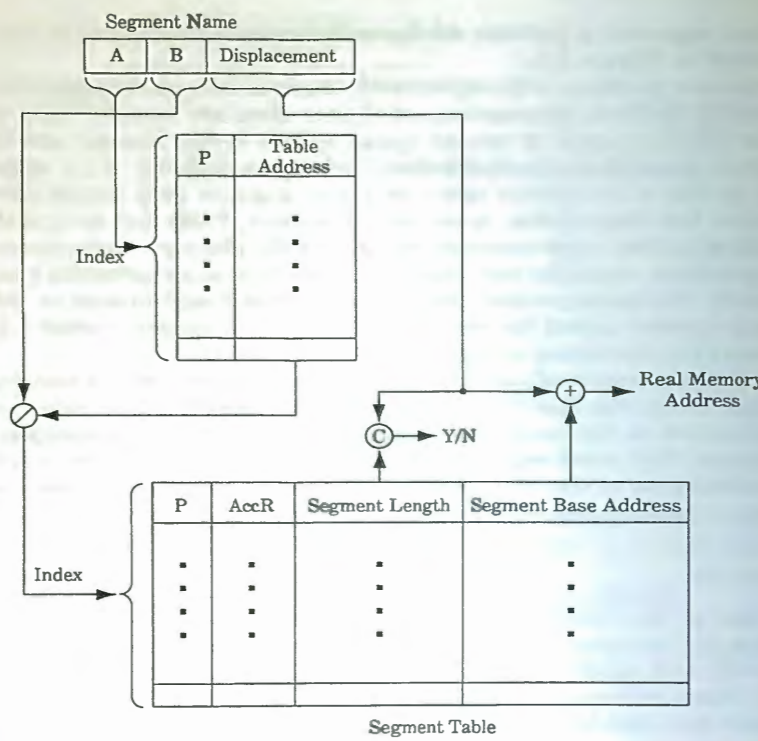


FIGURE 3.28 Two-level segment table.

3.3.3 *Paged Segmentation*

As noted previously, there are a number of problems with segmentation systems that have hampered their use. One significant problem involves finding space for a new segment in real memory. This allocation problem is difficult, and external fragmentation can significantly reduce the effectiveness of real memory. If compaction is used, there can be a serious reduction in performance. Another significant problem is that a TLB, described in Section 3.5, has to be allocated at the word or byte level; this process makes the TLB quite large, slow, and expensive. Researchers have recognized that if the allocation unit is a page of, say, 1 Kbyte, a compromise system combining the characteristics of a segment system

with a paged system could be designed. In other words, a segment is not a variable number of AUs but is a variable number of pages.

The tradeoffs that favor a paged-segmented system over a pure segmented system concern accepting internal fragmentation for no external fragmentation, the simplicity of allocating a page as compared to a variable-length segment, and the performance improvement due to simpler allocation.

The organization of a paged-segmentation system is shown in Figure 3.29. This system has two levels; the first level establishes the segments of pages while the second level is a one-level page system. Note that the second level could translate page names by any one of the techniques discussed above. This translation can be associatively searched, one-level or multilevel page tables or translation information could be hash addressed. The example shown, however, is a one-level page translation system.

The virtual address is divided into three fields: segment name, page name, and displacement. The segment name may be concatenated with a user ID number, forming an index into the segment table. The register holding the user ID is also known as *segment table origin register*. A length field in the segment table is compared with the virtual page name

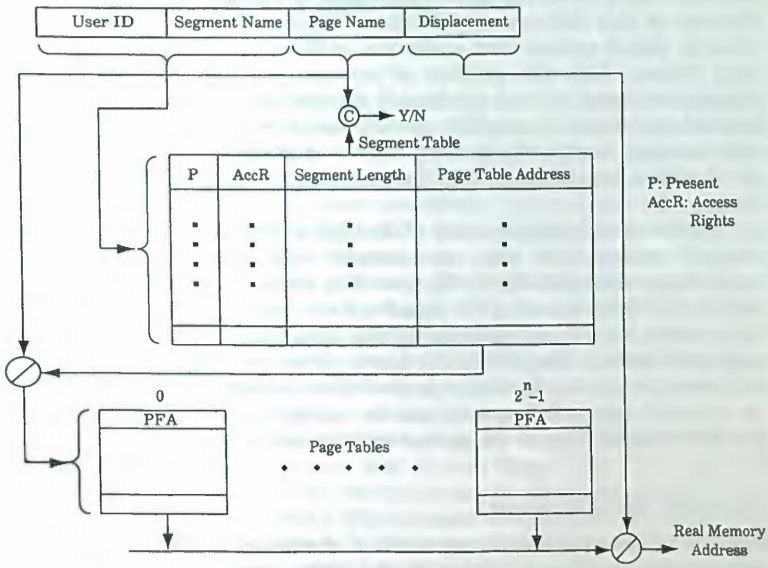


FIGURE 3.29 Paged segmentation.

field to verify that the length of the segment, in increments of pages, is proper. In other words, the page name is a displacement into a segment. The page table address, from the segment table, is concatenated to the page name field to index into the page tables. The selected page frame address is concatenated with the displacement field to give the real memory address. A present bit indicates whether or not an entry in the segment table has its corresponding page table in real memory. Access rights bits provide for protection and control.

Only multilevel paged-segmentation systems are possible; one-level systems cannot be implemented. This is because, as shown in Figure 3.29, at least one level is required for the segmentation table and another for the page tables. As with the other systems discussed earlier, all of the page tables do not need to be resident in real memory at once, only the active tables. If an access is made to a segment or page table that is not in memory, it can be assumed that the page is not in memory and the operating system must fetch the page(s) and update the segment/age tables. The Multics system paged the segment table, leading to complications discussed below.

The paged-segmented system overcomes many of the problems of a pure segmented system. The unit of allocation of a segmented system is a word or byte, while the unit of allocation of a paged system is a page. Because of this difference, the segment system can now operate as a demand paged system that shifts the problem of finding space to the page system. Also, the problem of external fragmentation and space compaction found in pure segmented systems are eliminated. However, because the pages of a segment are not placed in contiguous locations of real memory, locality for caching may be degraded; the same effect is found with a paged system that does not consider locality during a page swap.

For the above reasons, many of the high-performance microprocessor memory management units now support both paged and paged-segmented operation [MILE90]. The operating system can manage a virtual segment table on top of a pure paged system. The page tables are stored in memory, loaded and modified by the operating system, and may be paged themselves. See [MOTO87] for the details of a system. With such a system, no special hardware is needed to support the segment table as a paged-segmented system can be implemented in software on a hardware-paged system. Thus the two systems are isomorphic.

Segment Table Support

Because of the complicated organization of segment tables, interpretive segmentation has been found to be out of the question for performance reasons. Therefore, hardware support is mandatory if a reasonable level

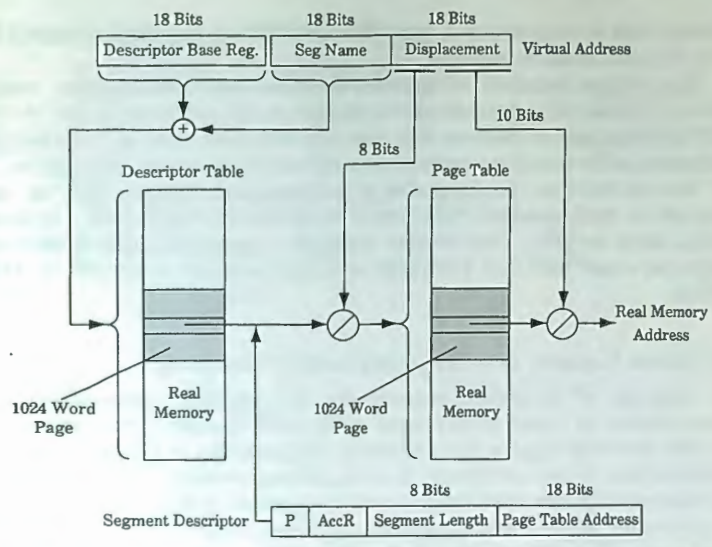


FIGURE 3.30 Multics name translation tables.

of performance is to be achieved. One of the earliest segmentation systems is Multics, which was developed at MIT in conjunction with General Electric. A GE 635 computer was modified to support paged segmentation becoming the GE 645. Multics system [SCHR71, ORGA72] used a multilevel paged-segmentation system completely implemented in software, augmented with a TLB. Figure 3.30 shows a simplified view of the Multics segment name translation system.

The processor produces a 36-bit effective address, the page size is 1K Words, and a segment can have up to 256 pages or 256K Words. The effective address is augmented with an 18-bit program loaded *descriptor base register* that points to the beginning of a segment in the descriptor table allocation of real memory that contains *segment descriptors*. Note that the segment descriptors are themselves paged.

A segment descriptor contains a segment length field of 8 bits, a page table address of 18 bits, and various control bits. The segment length field is compared, by the processor, to the eight MSBs of the displacement to verify that a displacement is within the allocated segment. The page table address is concatenated with the eight MSBs of the displacement to form the address into the page table, also in real memory and paged. The entries in the page table contain the page frame

address that is concatenated with the ten LSBs of the displacement to form the real memory address.

This system required two memory accesses to perform a name translation, a rather slow process as the processor did not have a cache. As will be described in Section 3.4, the GE 645 also used a translation lookaside buffer to cache the most recently used segment descriptors.

Due to the poor performance of a segmented system that is virtualized in real memory, two forms of hardware supported segment tables have evolved. One system supports segment length checking, while the other does not. Examples of these two types are given in this section.

Hardware Support with Segment Length Checking

An example of hardware support for the segment table of paged segmentation is found in the Intel i386, i486, and Pentium microprocessors shown in Figure 3.31, which is an extension of Figure 3.6. The i386 member of the x86 family had significant changes to the memory management system that have carried over into all subsequent members of the family. While there are differences between these later processors, they are sufficiently similar that only one description is given. The terminology used is that of the i486 and Pentium, which is different from the terminology of the i386. The segmentation system evolved from steps taken to extend the address space of the ix86 family of processors. Evolution of the mapping structure design was constrained by the requirement for upward compatibility for MS-DOS and applications software. A break in this capability occurred with the i386 with its effective address extended to 32 bits compared to the 16 bits of previous members of the family [MORS80].

There are two paths for obtaining a segment base address value; one path uses active descriptors stored in segment registers, the other path uses descriptors found in descriptor tables in memory. The six active descriptors, found in the *invisible part* of the segment registers, are indexed by implication with each memory reference in parallel with the selectors in the *visible part* of the segment registers; this is essentially a zero time access. The descriptors (both in the segment registers and descriptor tables) have a 32-bit segment base address field, a 20-bit segment limit field, and 12 control bits. One of the control bits is a valid bit, indicating that the descriptor information contained is valid. If the entry is valid, segment base address value selection is performed via the descriptor table and the memory resident translation tables are bypassed. The linear address is formed by adding the segment base address to the displacement field from the instruction.

In addition to various access checks based on the control bits, the

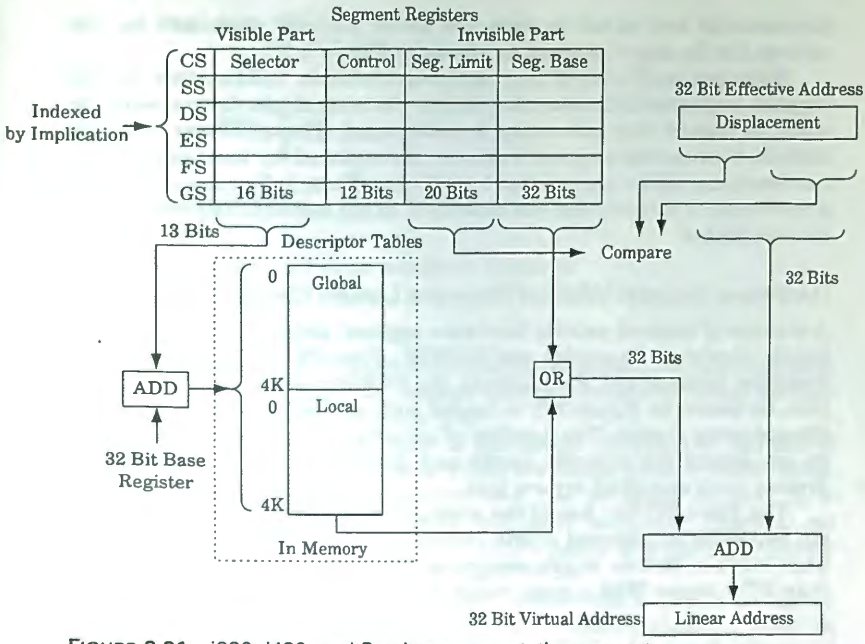


FIGURE 3.31 i386, i486, and Pentium segmentation support.

segment length is also checked. A bit in the control field determines if the comparison is between the 20 MSBs or the 20 LSBs of the displacement: if the MSBs, the segment length is measured in pages as a page is 4 Kbytes; if the LSBs, the segment length is measured in bytes. Thus the segment size is either $2^{20} - 1$ pages or bytes.

If the segmentation information in the segment registers is not valid, segment base address values are obtained via memory-based descriptor tables indexed by one of six, 16-bit selector registers, called *visible selectors*. These selectors are indexed by implication in parallel with the hidden descriptors. The 13 MSBs of the indexed selector register are used, after being added to a 32-bit base register value, as an address into memory to access a descriptor table entry; each descriptor is eight bytes. These tables are in two halves: the *global* table and the *local* table. The field allocation to the entries of the descriptor tables is identical to the hidden descriptors. The same access checks and segment limit checks are made before a memory access can proceed. The segment base and

displacement are added to form the linear address that will be the address for the paged system, as shown in Figure 3.12.

With two instances of the same segmentation information in the segment registers and descriptor tables, the normal provisions must be made to ensure that coherency is maintained. This problem is made difficult because the segment registers are updated by hardware while the descriptor tables are updated by software. Thus, software can change a descriptor in memory and the descriptor in the segment registers must be invalidated.

Hardware Support Without Segment Length Checking

A number of systems provide hardware segment tables without segment length checking. Examples are RS/6000, PowerPC 601, and the HP Precision Architecture. For example, the RS/6000 map of segment registers, as shown in Figure 3.5, is loaded with values that are created by the operating system. The function of assuring that valid displacements do not exceed the segment length and generating the page table addresses is an operating system task.

The PowerPC 601, one of the examples of paged segmentation without hardware for segment length checking, is shown in Figure 3.25. The page index is 16 bits, which constrains the segment size to be no more than 2^{16}-1 pages. With a page size of 4 Kbytes, a segment is 26 Mbytes.

3.4 Translation Lookaside Buffers

The translation of a virtual page name into the page frame address via memory resident tables can take a significant number of memory cycles. To hide the translation time, a page table system (direct or inverted) is usually augmented with a cache of active page frame addresses. This type of cache is usually known as a *translation lookaside buffer* (TLB), a name used by IBM. Motorola uses the term *address translation cache* (ATC); DEC uses the term *translation buffer* (TB); and Intel uses the term *page translation cache* (PTC). The Multics system employed the first known instance of a TLB [SCHR71].

Translating page names and block names, as shown in Figure 3.32, is usually performed in a hierarchy of translation systems, much as a memory reference is processed in a hierarchical memory. This figure illustrates the number of translation maps possible, not the tables for any particular processor. Usually, all of the TLBs are accessed in parallel and the first valid translation is used to form the real memory address. If translation fails in the TLBs, the page tables, either direct or inverted, are then accessed before a page fault is declared. As will be discussed when the MC88200 TLBs are described, one level of this hierarchy may

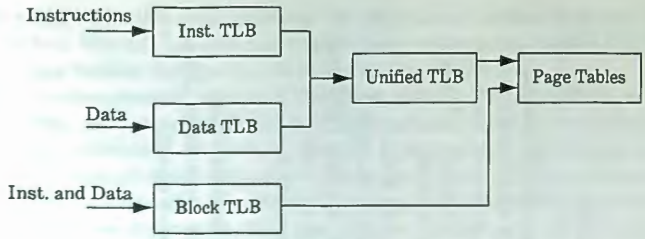


FIGURE 3.32 Canonical name translation hierarchy.

be implemented for both page and block name translation. Also, portions of the page tables may be resident in the processor's cache, further improving the name translation time.

Multilevel TLBs have many of the design problems of multilevel caches and hierarchical memories. There are design issues of information allocation, coherency, particularly when the processor is used in a multiprocessor configuration, and multilevel inclusion. There is no need for read and write policies as with a cache, but there are issues dealing with invalidating the contents of TLBs.

The reduction in name translation latency with a TLB is similar to that of a data or instruction cache. There is some probability that the page frame address will not be found in the TLB; as a result, the name translation time is the weighted average of the time to translate in the TLBs or in the page tables. On a TLB miss, the delay in access time is the two or more memory cycles for accessing the page tables plus the table walking time (discussed in Section 3.5.5). Note that there can be two reasons for a TLB miss. First, if the referenced page is not in real memory, there will be a miss in the translation tables. Second, a miss occurs when the page is in real memory and its translation tables are correct but the translation information is not in the TLB. It is this second form of TLB miss that is most often noted in the literature.

A TLB holds the translated addresses of pages that are also resident in the page tables. Most access checks have been made and do not require verification when translating a virtual page or segment name in a TLB. Because the name translation process can be hierarchical, the treatment of multilevel inclusion is a design issue.

The PowerPC 601 has three levels of translation: the Instruction TLB (ITLB), Unified TLB (UTLB), and the hash-accessed inverted page tables. For this system, multilevel inclusion is maintained with the following policy. Allocation of translation information to the page tables is a software process that follows a page fault. Allocation to the ITLB and UTLB is performed by hardware following a page fault that brings

Operation	ITLB	UTLB	Page Table
Add entry	Add on most recent translation (MRT), set valid bit.	Add on most recent translation (MRT), set valid bit.	The page table is locked, the entry added with a move instruction, then the table is unlocked, all under program control.
Replace entry	Replace if a new entry is translated and there is no vacant slot. MRT bits select entry to be replaced.	Replace if a new entry is translated and there is no vacant slot. MRT bits select the slot within the sector if there is a set conflict.	An entry is: locked, invalidated, flushed, updated, marked valid and unlocked, all under program control.
Delete entry	Valid bit is reset explicitly by the execution of tibie instruction or as a side effect of some other instructions.	Valid bit is reset explicitly by the execution of the tibie instruction or as a side effect of some other instructions.	All entries have their valid bits reset when there is a context switch.

TABLE 3.5 PowerPC 601 maintaining translation multilevel inclusion.

a new page into the real memory. At any time, entries that are in the ITLB are in the UTLB and the page tables and entries in the UTLB are in the page tables but not necessarily in the ITLB.

The management policy of the three tables, ITLB, UTLB, and the page tables, is shown in Table 3.5. Three cases must be comprehended to assure multilevel inclusion: adding a new entry, modifying an old entry, and deleting an old entry. Note that the two TLBs are managed by hardware, while the page table is managed by software. Also note the synchronization operations that are required to lock and unlock the page tables.

3.4.1 *TLB Organization*

The organization of a canonical TLB is shown in Figure 3.33. Note that this figure is similar to Figure 2.4 and differs only in that page table entries (PTEs) with page frame addresses are stored rather than data and only a valid bit is required. If a page is evicted to the disk, the entry in the TLB can become invalid and is modified by the operating system or microcode. A TLB, being a cache of translation information, can be organized for access as: (1) associative search, (2) n-way set associative search, and (3) hash accessed.

Note that the size of a TLB is sometimes described by its *length* or *size*, not the number of sets. With associative TLBs, the size is described in *entries*, not sectors as with caches. A TLB operates on the principle of temporal page locality; when a page is loaded into real memory, the page tables and TLB are updated. Because of page locality, caching addresses can be quite effective in reducing the latency of name translation and the access of real memory. Simulations and measurements show that TLBs have a hit ratio in the range of 90% [SATY81]. The hit ratio

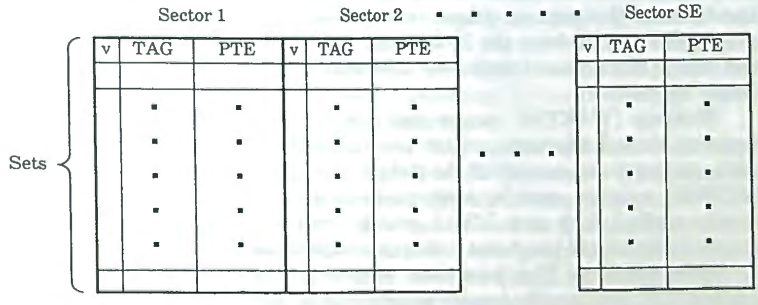


FIGURE 3.33 Canonical TLB.

of a TLB is, as with other caches, a function of its size. D. W. Clark et al. [CLAR85] and M. D. Hill [HILL87] provide data on TLB hit ratios.

In general, TLBs are invisible to the program as they are loaded under hardware control and serve to speedup the name translation process. However, TLBs must usually be invalidated when there are changes to the page tables such as will occur when a page table change occurs that will violate coherency. There must be a replacement policy for the n-way set associative TLB, and the same policies described in the section on caches apply here. For example, the policies may be MRU, LRU, Random, Clock, and others [BAER80].

Associative Searched TLBs

Associative searched TLBs have a number of positive features. One feature is that, for small TLBs, the latency of translating a name is quite small. Another feature is the ease of expandability. As there is only one set, the number of sectors in the TLB can be increased without architectural change. And, the number of sectors need not be an even power of two. This feature eases implementation of TLBs on a chip. For these reasons, most contemporary microprocessors use associative searched TLBs. Examples are discussed in the following paragraphs.

MC68451. An example of an associatively searched TLB, called an *address register translation table*, is the Motorola MC68451, shown in part in Figure 3.34 [MOTO83]. This rather strange device was designed for use with the MC68000 to support paged segmentation. Providing virtual page and segmentation support for functions such as segment length checking and table management is a pure software function.

The virtual 23-bit word address is divided into a segment name and displacement fields. The segment name is the key for an associative access of the page directory, reading out a page frame address, mask, and control bits. The mask selects the MSBs of the page address and the LSBs of the segment name concatenating them with the 7-bit displacement field to form the 23-bit real address. By adjusting the mask, the size of the segment varies for 128 words (mask all 1s) to 65K words (mask all 0s).

Wakerly [WAKE89] states that the MC68451 was not successful because of its limitations, which are believed to be the limited virtual address space and too much flexibility. For this and other reasons, the MC68451 was not used by early customers for the MC68000. Sun and Apollo crafted their own MMU, which are modeled after the memory mapping system of the Atlas Computer (described in Figure 3.15).

IBM RS/6000. This processor architecture specification calls for a unified TLB, but specific implementations may have separate ITLBs and DTLBs [IBM90]. One such implementation has a split TLB. The DTLB

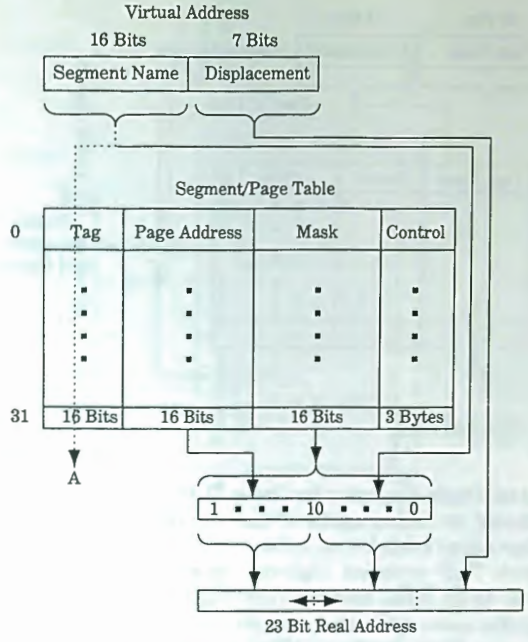


FIGURE 3.34 Associative register name translation.

is organized as (64, 2, 1, PTE) and is late select. However, the ITLB of this processor is believed to be associatively searched as shown in Figure 3.35. The 40-bit page name is used as the key to search the ITLB associatively; the number of sectors is not revealed in the literature. A hit on a translation provides a 20-bit page frame address that is concatenated with the 12-bit displacement of the virtual address. If the translation is successful, the reference to real memory is initiated. If not successful, the reference address is translated in the translation tables resident in either the cache or real memory.

Note the similarity between this TLB design and the Atlas name translation system depicted in Figure 3.15. The major differences between the two are the design parameters, not the organization.

MC88200. The MC88200 that supports several Motorola processors (88100, 68030) has two UTLBs, one for user programs with 4-Kbyte pages and one for system programs with 512-Kbyte blocks [MOTO90a]. Note that Motorola calls TLBs *address translation caches*. Both of the TLBs are unified, translating the page and block names of both instruc-

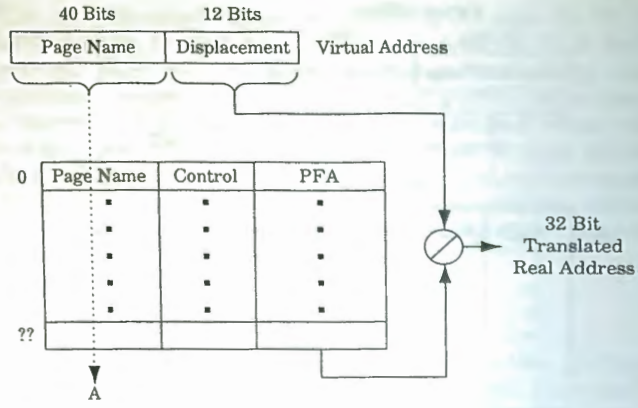


FIGURE 3.35 RS/6000 ITLB.

tions and data. Block diagrams for these TLBs are shown in Figure 3.31, and note should be taken again of the similarity between these TLBs and the Atlas name translation system.

The block TLB contains high-use translation information that is cached at the block level, thereby reducing TLB misses. Operating systems occupy the same address space, for instructions and data, over long periods of time. Thus, having a large page, called a *block,* can significantly reduce the number of block name translation misses and page misses. Note that these blocks are not the same as cache blocks.

The block TLB has 10 sectors that are associatively searched on the 13-bit block name and is organized as (1, 10, 1, PFA). On a hit, the 13-bit page frame address is concatenated with the 17-bit block displacement to give a 30-bit real memory address. Two of the 10 sectors are hardwired to provide an identity mapping in the upper 1M bytes of real memory for supervisor space.

User programs have higher paging activity, and the page TLB reduces external and internal fragmentation that would occur if the page is the same size as the block. The page TLB, also shown in Figure 3.36, has 56 sectors that are associatively searched on the 20-bit page name from the virtual address. The organization is (1, 56, 1, PFA). The 20-bit page name is associatively compared to the 56 page name tags of the resident pages. Upon a hit, the 20-bit page frame address is concatenated with the 10-bit page displacement to give a 30-bit real memory address. The page TLB is managed by the operating system with entries being evicted on a FIFO basis; a new entry pushes out the oldest entry.

A page name translation is conditionally and concurrently attempted

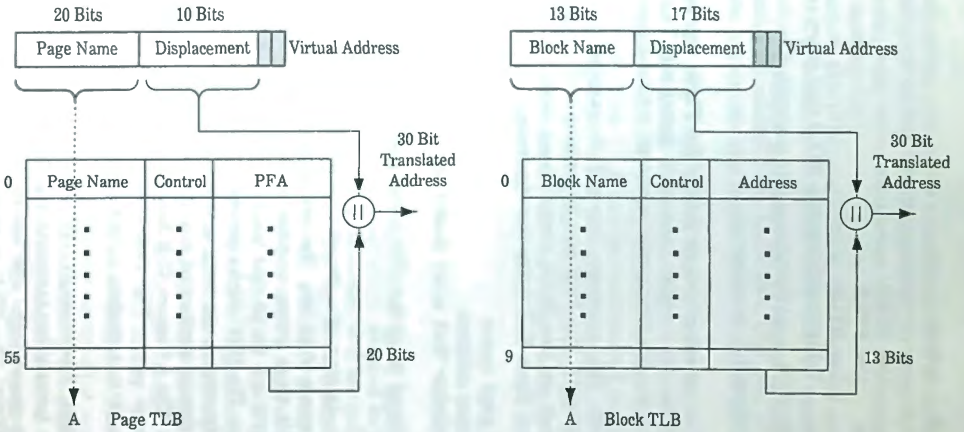


FIGURE 3.36 MC88200 TLBs.

PTLB	BTLB	Action
Hit	Hit	Use BTLB
Hit	Miss	Use PTLB
Miss	Hit	Use BTLB
Miss	Miss	Table search, update PTLB and retry translation

Note. The BTLB is loaded by the operating system.

TABLE 3.6 MC88200 TLB management.

in both TLBs. If an access misses on *both* of the TLBs, the name translation is made via the page tables previously described. Table 3.6 shows the action taken for the four possible events when accessing the two TLBs.

MIPS R2000. This processor has a TLB integrated onto the processor chip [MIPS87]. The TLB has 64 entries and is accessed by an associative search on the 20-bit virtual page name. The page size is fixed at 4 Kbytes. Each TLB entry has the fields:

20 bits: virtual page name;
20 bits: page frame address;
6 bits: PID number that must match the PID value in a processor register;
4 bits: Control bits: Read Only, Non-Cacheable, Valid and Global;
14 bits: unused.

The page name from the effective address plus the PID value gives a virtual name space of 26 bits. The addressable unit is a 4-byte (32-bit) word. The 20-bit page frame number is concatenated with the 10-bit displacement, giving a 30-bit real memory address or 4 Gbytes.

DEC Alpha. The DEC Alpha processor has split, associative TLBs: one for the instruction page names and one for the data page names [DIGI92]. The virtual memory of the Alpha system is paged with variable page sizes. The I-Stream TLB has 12 sectors, 8 for 8-Kbyte pages and 4 for 4-Mbyte pages. The D-Stream TLB has 32 sectors, each of which can be used to translate to 8-Kbyte, 64-Kbyte, 512-Kbyte, or 4-Mbyte pages.

PowerPC 601. The PowerPC 601 has a small first-level instruction TLB (ITLB) for translating instruction page names and a unified TLB (UTLB) for data page names and instruction page names that miss in the ITLB [MOTO93]. The ITLB is searched associatively and has four sectors. The 20-bit page name of the 32-bit effective address is the search key. A hit provides a 20-bit page frame address that is concatenated with the 12-bit displacement to give the real memory address. Note that the extension to the virtual address is not part of the search key.

If the instruction page name translation fails in the ITLB, the translation is attempted in the UTLB organized as (256, 2, 1, PFA). If this translation fails, page name translation is then performed via the tables described in Section 3.3.1.

In addition to the ITLB and UTLB, the processor can also translate block names in a small TLB, called *block-address translation registers* (BATR). These registers are organized as (1, 4, 1, PFA). The BATR holds four block name translated base addresses. The size of the blocks is a programmable parameter and can be in the range of 18 Kbytes doubling to 8 Mbytes.

VAX 11/780. The VAX 11/780 has a small first-level instruction TLB, called an *instruction translation buffer.* This ITLB has only one sector organized as (1, 1, 1, PFA). In other words, the last translated page name is held in the ITLB and is concatenated to the displacement for every instruction read. Thus instruction page name translation is a zero time process. If an instruction fetch crosses the boundary of the page (the page name changes), the new page name is translated in a second-level unified page table.

n-Way Set Associative TLBs

A number of TLBs are implemented with *n*-way set associative organizations in which the page name is congruence mapped for translation. To accomplish this, the page name is allocated to a set index field and an unnamed field that is compared to the tags of an accessed set. Some of the TLBs that use this organization are discussed below.

i386, i486, and Pentium. These processors support paged segmentation, and virtual page names are translated via a multilevel page tables system as shown in Figure 3.12. A TLB organized as (8, 4, 1, PFA) is implemented as a unified TLB on the i386, as shown in Figure 3.37. This same design is also used as a unified TLB on the i486. With the Pentium, two copies of the same TLB are used as split TLB without a unified TLB.

The page name is allocated into two fields: an 8-bit index into the TLB and a 12-bit field that is compared to the TLB tags. If there is a true comparison on one of the four sectors, the 20-bit page frame address is concatenated with the 12-bit page displacement to form the real memory address. If there is a miss in the TLB, the address is translated via the multilevel page tables.

Note that the TLB provides name translation of the 20-bit page name and not the extended virtual address; the 20-bit extension is not translated. The consequence of this restriction is that only translated page names for one root directory at a time are in the TLB. As a result,

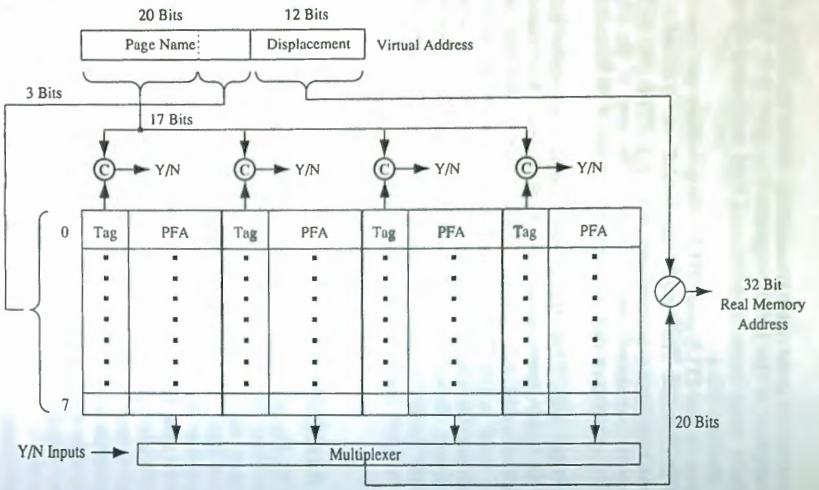


FIGURE 3.37 i386, i486, and Pentium TLBs.

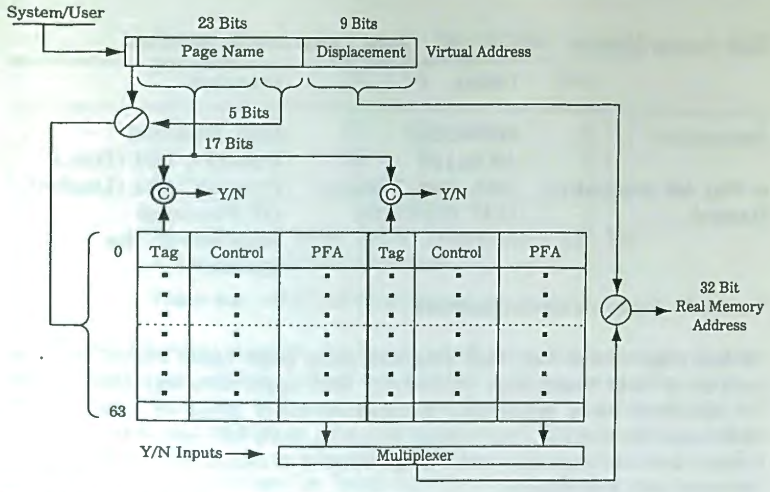


FIGURE 3.38 VAX 11/780 unified TLB.

aliases are possible. To prevent aliases, any change of the root directory by the executing program requires a reload of the TLB.

VAX 11/780. The VAX 11/780 has a unified TLB that translates the page names of data references and misses in the ITLB [SATY81] and [HAMA90]. A block diagram of the unified TLB is shown in Figure 3.38. The 32-bit virtual address is divided into a 9-bit page displacement field and a 23-bit page name field. The five LSBs of the page name are concatenated with the MSB to form a 6-bit index into the 2-way set associative TLB organized as (64, 2, 1, PFA). An interesting feature of this TLB is that the MSB of the page name (system/user bit) is used as the MSB of the index to select between two halves of the TLB. This guarantees that the system and user both have full access to 1/2 of the TLB and 1/2 of virtual memory space.

PowerPC 601. The PowerPC 601 has a first-level TLB and a unified (UTLB) that translates instruction page names that failed to translate in the ITLB and data page names. This TLB is organized as (256, 2, 1, PFA).

Hash-Accessed TLBs

IBM S/370/168. This system has a unified TLB translating both instruction and data references. The 14-bit page name is hashed to create a 6-bit index into the TLB that is organized as (64, 2, 1, PFA), an unusual organization [MATI77]. By making the TLB 2-way set associative, two

TLB Access Method	Page Table Access Method	
	Direct	Inverted
Associative	MC88200, MC88110	IBM RS/6000 PowerPC 601 (Inst.)
n-Way set associative	i386, i486, Pentium	PowerPC 601 (Unified)
Hashed	IBM S/370/168	HP Precision (only one in the system)

TABLE 3.7 Name translation methods.

virtual page names can hash into the same page table entry and yet produce a valid translation. Remember that approximately two probes are required for a successful translation. This level of redundancy eliminates the need to link entries (common with full name translation in hash accessed systems), and a large fraction of collisions can be accommodated with this design.

HP Precision Architecture. This processor does not have a TLB to speedup the name translation process. The hashed access page translation (Figure 3.24) is as fast as a TLB, thus one is not used.

Summary of TLB Organizations

The selection of the TLB organization by a designer is sometimes constrained by the design of the translation tables. Table 3.7 shows the TLB design and the table translation method for a number of processors.

3.4.2 *TLB Miss Ratio Data*

Satyanarayanan and Bhandarker [SATY81] performed a miss ratio evaluation of TLB size, degree of associativity, sensitivity to context switching, and replacement algorithms. The tests are conducted by means of a simulator and FORTRAN programs for the VAX product family. The evaluation was performed to set the design parameters prior to the construction of a processor. A clear interval is used as a surrogate for context switching. By periodically clearing the TLB, the effect of clearing based on a context switch can be evaluated. The authors of this evaluation characterize TLB performance as

$$\text{TLB miss ratio} = \frac{\text{no. of main memory references to page table}}{\text{explicit virtual memory reference}}$$

No. of Sectors per Set	No. of Sets			
	32	64	128	256
1	97	47	22	8
2	25	9	3	2
4	7	2	1.5	1

Number of page table references $\times 10^3$ for 5.5×10^6 explicit references.

TABLE 3.8 VAX TLB miss ratios.

Explicit virtual memory references are those references generated by the program. In addition, there are *implicit* references to memory that are generated by the translation process itself. The number of implicit references is influenced by the organization of the translation tables themselves, such as the number of levels. One of the performance results of the VAX study is shown in Table 3.8. The data indicates the reduction in page table references as the size and organization of the TLB are changed.

As with other figures that show P_{miss} data for cache, the diagonals of this figure are for TLBs of equal size. The total number of explicit references to the memory system is 5.5×10^6, and the number of references that miss the TLB are shown. Thus, for this table, small numbers represent an improvement in the number of page names translated in the fast TLB. This data indicates a clear preference for a large TLB and for higher degrees of associativity and a smaller number of sets. Higher degrees of associativity reduce the occurrence of set conflicts. Thus the preference is for associative TLBs in many processors that are implemented in VLSI technology where the increase in tag bits is not a significant cost problem.

Another source of TLB miss data is [WOOD86]. Five benchmarks are executed on six processors. Four of the processors are of the VAX family (512-byte page) and two are IBM S/370 or Amdahl compatibles (4-Kbyte page); excerpted results of the tests for TLB miss ratios, expressed in percentages, are shown in Table 3.9.

This information is quite interesting in that while the size and organization of the TLB are important, the size of the page is even more important. A larger page will result in a lower miss ratio for TLBs of equal size. This result confirms the use of a page TLB, as with the MC88200 and the PowerPC 601, for the operating system. This data also confirms the advantage of associative TLBs.

Saavedra and Smith [SAAV93] measured TLB miss ratios on a number of commercial processors for six SPEC benchmarks. Table 3.10

No. of Sectors per Set	No. of Sets		
	128	256	512
1 (VAX, 512 Kbytes)	3.68		0.639
2 (VAX, 512 Kbytes)	1.78		0.324
2 (IBM S/370, 4 Kbyte)	0.097	0.023	0.014

LISZT Benchmark.

TABLE 3.9 TLB miss ratios (%).

Processor	No. of Sets	Sectors per Set	Page Size	Miss Ratio (%)
DECstation 3100	64	64	4096 Bytes	2.42
DECstation 5400	64	64	4096 Bytes	2.42
DECstation 5500	64	64	4096 Bytes	2.42
MIPS R/2000	64	64	4096 Bytes	2.42
VAX9000	1024	2	8192 Bytes	0
RS/6000 530	128	2	4096 Bytes	1.31
HP 9000/720	64	64	8192 Bytes	1.30

TABLE 3.10 TLB percentage miss ratios for SPEC Benchmarks.

shows an excerpt of their results; the average TLB miss ratio is given for the six benchmarks.

This data also indicates the primacy of page size in determining TLB miss ratios. A large page, or the granularity of the TLB, significantly reduces the miss ratio.

We can speculate that the arguments advanced by Hill and others on the advantage of direct caches will apply to TLBs as well. The arguments (discussed in Chapter 2) suggest that a direct cache, with its shorter logic path but with a higher miss ratio, may be the most effective design for reducing TLB translation time. [HILL86] also provides TLB miss ratio data for a number of processors and three benchmarks.

As discussed in Chapter 5, TLB translation time is of significant importance for caches that are addressed with translated real addresses. However, for some cache organizations, the TLB must be only fast enough to match the access time of the cache. Thus, higher degrees of set associativity, with lower miss ratios, may be quite satisfactory.

TLB Size, Sectors	Normalized No. of Instructions Executed	TLB Miss Ratio
0	1.0	1.0
4	2.3	0.106
8	2.6	0.029
16	2.8	0.0125

Note. LRU algorithm.

TABLE 3.11 GE-645 Multics performance.

3.4.3 *TLB Impact on Virtual Memory Performance*

There is little published information on the performance of a virtual memory system as the various parameters (for example, page size, TLB size and organization, and main memory size) are varied. The little data that does exist is some material on performance: the papers on Multics [SHEM66, SCHR71] and the VAX 11/780 [CLAR85].

Schroeder [SCHR71] reports on the performance of a virtual memory system as the TLB size is varied. The system is the GE-645; the operating system is Multics, which is a paged-segmented system. The page size is 1K words or 4.5 Kbytes. The system was evaluated with associative TLBs having 0, 4, 8, and 16 entries. The time for a TLB search is between 200 and 600 ns, while the main memory is 1.2 us—a ratio of 2:6 to one. This ratio is probably valid today under the assumption the tables are cached. Table 3.11 shows the normalized number of instructions executed per unit of time. This metric is a relatively good measure of the TLB's effectiveness; the larger the number, the more effective is the TLB. Table 3.11 also shows the TLB miss ratios as the size is varied.

These data indicate that an associative TLB with 8 sectors is probably large enough, nor do these data support the large TLBs found in modern memory systems; the reason being that the GE-645 is a microprogrammed machine that takes many clocks to execute an instruction, unlike modern pipelined processors with CPIs of 1–2. Thus, with modern pipelined processors, name translation delays have a greater impact on overall performance. Additional information on the impact of TLBs on performance is found in [SHEM66].

3.4.4 *Instruction Support for TLBs*

TLBs are generally considered to be invisible to the program, as with early caches, and serve only to reduce the time of name translation.

Unfortunately, this simple view is not completely correct. Recall the use of processor instructions for enforcing MLI, described in Table 3.5. Two classes of instructions are required. First, some of the TLB registers must be loaded explicitly under program control; second, the TLB must be invalidated in order to maintain coherency. The complexity of invalidation is illustrated by the PowerPC 601 *translation lookaside buffer invalidate entry* (*tibie*) instruction, which is used for MLI control.

A name translation of the page name of the *tibie* instruction (a supervisor-level instruction) is attempted. If there is a hit on the TLB, the TLB entry is invalidated by setting the valid bit to zero. A TLB invalidate is also broadcast on the system bus so that coherency can be maintained across the system.

With a multiprocessor system, the broadcast *tibie* must be in a critical section controlled by software locking so that only one *tibie* can be issued at a time. Resynchronization is established by issuing a *sync* instruction after every *tibie* at the end of the critical section. When a processor receives a broadcast *tibie* instruction, it halts the execution of new load, store, cache control, and *tibie* instructions; waits for the completion of all memory operations; and then invalidates both sectors in the user TLB.

3.5 Virtual Memory Accessing Rules

This section discusses the rules that are followed when a virtual memory system is accessed for reads and writes. The effect of a cache on this process is ignored; however, the combination of a cache with virtual memory is addressed in Chapter 4. The reader should note that all of the design ideas and much of the published performance data on virtual memory systems are from an earlier time when small memory was the rule. Today, with significantly larger memories, these designs may no longer be appropriate. Caution should therefore be exercised in the use of specific design data.

3.5.1 *Read Accesses*

The various policies that govern the actions when a paged virtual memory is accessed are discussed in this section. The various design issues involved are described here with illustrations taken from an actual virtual memory system. The basic steps in performing a read access into a virtual memory system consist of the following.

1. Determine if a referenced page is in real memory; detect for a page fault.
2. On a hit (the referenced page is determined to be in real memory)

translate the virtual page name to the page frame address and fetch the AU into the processor.

3. On a miss (the referenced page is determined to be absent from real memory):

(a) Determine if a page must be evicted because of a capacity miss; identify the page to move; evict the page to the disk. This policy is known as the *replacement policy*.
(b) Depending upon the organization of the virtual memory, a determination of the page frame to receive the loaded page may need to be made. Fetch the page from the disk into the real memory and adjust the tags and valid bits in the name translation tables. This policy is known as the *placement rule*. Translating the virtual address to the disk address is discussed in Section 3.5.9.
(c) Complete the read of the AU into the real memory.

Note that if the system is multitasking, a new task may be swapped in during this process. There are commonly three successful name translation paths through a virtual memory with a TLB and a cache; the fourth path is the page miss path. These paths are shown in Table 3.12.

Path 1. The page frame address is found in the TLB.

Path 2. The page frame address is not in the TLB but is found in the page tables that have been placed in the cache. After the reference, the TLB is updated with page name translation information for use by a future reference.

Path 3. The page frame address is found by table walking in real memory. After the reference, the cache and the TLB are updated with page name translation information.

Path 4. The translation misses on the TLB, cache, and tables in memory. This is a page fault path and requires the allocation of the missed page into real memory. All translation tables are updated.

The three successful cases above are for the name translation only. The assumption is made that a name translation miss on a page that is resident in real memory is not permitted. For Path 4, the total failure

Path	Page Name Translation Path Hit/Miss		
	TLB	Cache	Memory
1	Hit	X	X
2	Miss	Hit	X
3	Miss	Miss	Hit
4	Miss	Miss	Miss

TABLE 3.12 Name translation paths.

of name translation indicates that the page is not in real memory and results in a page fault.

3.5.2 *Allocation*

This section discusses the issues of allocating new pages or segments into real memory following a page fault—a problem that can be different from late select cache allocation. Recall that congruence-mapped, n-way set associative (including direct) late select caches dictate the sector into which a new allocation must be placed. Only the block within the sector is open to choice. Associative caches, on the other hand, can allocate into any segment with the choice based on one of the replacement policies discussed in Chapter 2.

Virtual memory page or segmentation organization is early select, which is an associative accessing process. Therefore, allocation of pages to real memory is a matter of (1) finding a vacant location and (2) if a vacant location cannot be found, creating a vacant space using one of the policies discussed in Section 2.1.7. In addition to the allocation problems associated with finding space for a page frame or a segment, the name translation information must be allocated to the name translation tables as well. The problem here is quite similar to the allocation problems of caches, noted above. The organization of the translation tables dictates the allocation of name translation information.

The techniques for performing allocation in virtual memory systems have their historical roots in the problems of allocating files to disks, discussed in many texts on operating systems. Also, because allocation is highly dynamic and interactive in LISP systems, research in these systems has made significant contributions. Even interactive programming environments such as Lotus 1-2-3 must have solutions to the requirement for dynamic allocation and de-allocation of segments.

Page Allocation

The procedure for allocating pages to page frames is the same regardless of the name translation method. That is, either direct or inverted mapping systems use the same procedure. The operating system maintains a *free page frame list* (a linked list of vacant page frames in real memory) that contains the starting addresses of all unallocated pages. The list can be maintained as a linked list or a LIFO stack. Upon a page fault, the free page frame list is searched; if there is a free page frame, it is allocated. Recall that because a paged virtual memory is early select, congruence mapped, and associative, any free page frame can be allocated. If a page frame is not available, a replacement policy is then invoked.

There can be additional policies imposed by the operating system on page allocation from the free list. For example, should there be reserved page frames for the operating system and the user? If the system is multiprogrammed, should there be page frames reserved for each of the resident programs? These issues are beyond the scope of this book but are treated in books on operating systems.

The hardware must support reading and writing the free page frame list. Thus linked list or stack manipulation is needed. Most processors designed today provide for stack manipulation, which is the implementation of choice. Processors that have been optimized for list or LISP processing use linked lists for the free page frame list. The size of the free page frame list is not excessive, as there must be, at most, one entry for each real memory page. For a system with a 4-Kbyte page and 16 Mbytes of real memory the free page frame list has only 4K entries. If an entry is 4 bytes, 4 pages are occupied by the free page frame list, approximately 0.1% of the total real memory space.

The question of whether or not the free page frame list itself is paged or locked into an unpaged region of real memory is addressed in Section 3.5.4. Recall that most virtual memory systems today provide for an identity page name translation for operating system access to memory. In addition, some systems provide block name translation facilities to enhance the management of the operating system in virtual memory space. A block composed of a number of pages eliminates much of the paging that could occur if only pages are implemented.

Segment Allocation

For the discussions to follow, it is helpful to remember that space in memory, either page frames or space for segments, can be in one of three states.

1. *Free*. The space is vacant and available for allocation.
2. *Allocated*. The space is occupied by a valid segment, either clean or dirty.
3. *Garbage*. The space is occupied by a segment that has been deallocated.

Segmented systems pose additional, and significant, allocation problems to those of paged systems. There must be a way to locate a contiguous region of free space in real memory that is equal to or larger than the segment to be allocated. This requirement means that a *free space list* (called spaces because they are not yet allocated to segments) must contain not only the starting address of free space but the length of each space as well. Note that the size of a free space list is not bounded, as is a free page frame list, because in the limit, every space can be one

addressable unit and the free space list would occupy all of real memory! Segments can overlap for sharing, thereby creating further allocation problems. Allocation for nonoverlapping segments follows three steps.

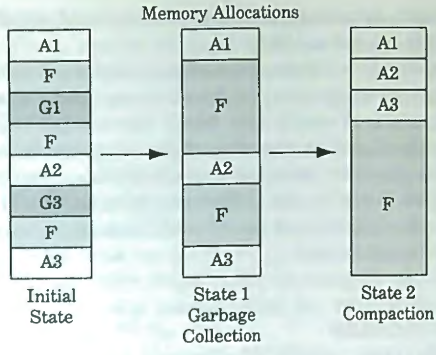
1. Find a free space that is large enough to receive the allocated segment.
2. If step 1 fails, determine if there is enough total free space in real memory. If there is, compact enough available free space to create a space for the segment to be allocated.
3. If steps 1 and 2 fail, find an allocated segment that is large enough and can be evicted based on criteria discussed in Chapter 2.

Step 1. This step is similar to page allocation. A *free space list* is maintained that lists the starting addresses and length of all free space. A number of algorithms have been proposed [KNEE68, DENN70] for segment allocation from the free space list. A detailed discussion of these algorithms is outside the scope of this book; nevertheless, four algorithms are

1. *First-fit.* Search the free space list, and find the first space into which the segment will fit regardless of the efficiency of space use. Place any unallocated portion of this space on the free space list.
2. *Best-fit.* Search the complete free space list, and find the free space that gives the most efficient use of space. Place any unallocated portion of this space on the free space list.
3. *Worst-fit.* Search the complete free space list, and find the free space segment that gives the least efficient use of space. Place any unallocated portion of this space on the free space list.
4. *Buddy.* Free space is initially divided into groups having even power of two AUs as with pages. When a page is allocated one of three allocations is made: (i) half the group with the other half placed on the free space list; (ii) the full group; or (iii) two groups, four groups, etc. This requires that all free space groups and allocated segments must be even power of two in length.

Knuth [KNUT68] reports on simulations that show that the first-fit policy gives the best results, defined to be that allocations continue longer before Step 1 fails and a garbage collection pass is required.

Step 2. When a new segment is to be allocated the total free space may be sufficient to hold a segment but the space may not be contiguous, which is a condition required for segmentation. A full discussion of the many techniques for compacting fragmented free space and de-allocated space, generally known as *garbage collection*, is outside the scope of this book. However, I want to provide a brief discussion of the issues involved. A comprehensive survey of garbage collection techniques can



A: Allocated, F: Free, G: Garbage.

FIGURE 3.39 Garbage collection.

be found in [COHE81]. Note that not all algorithms for managing free space are known as "garbage collectors."

One of many types of compacting garbage collection systems is illustrated by the example shown in Figure 3.39. In the initial state the map of memory shows areas that are of the three states noted above: free, allocated, and garbage. The state of each area is denoted by means of a tag in either the space lists or by two control bits that impart *coloring* to the three states.

The initial state of the memory is shown and consists of three allocated spaces, three free spaces, and two garbage spaces. The first step, to State 1, in the garbage collection process searches out the garbage spaces and turns them into free space by placing these spaces on the free space list. The tag bits are also changed as required, and no movement in memory takes place. The second step moves both the allocated and free spaces into two compact regions of real memory. Thus all of free space is in one compact space and is available for allocation.

Step 3. If garbage collection and compaction do not yield a space that is large enough for allocating the new segment, an already allocated segment must be evicted. The algorithms for making the selection for eviction are based on (1) selection of a segment of appropriate size, (2) the usage criteria, and (3) the choice of a modified or nonmodified segment (clean or dirty).

If a modified segment is chosen for eviction, it must be copied back onto the disk. The free space thus made available is placed on the free space list and allocation Step 1 is initiated. In the chosen segment is not

modified, it is only necessary to place the segment on the free space list and proceed with allocation Step 2.

From the above brief discussion of segment allocation it is clear that considerable operating system overhead is required. Hardware assist for this process consists of stack and block move support. The amount of overhead is proportional to the size of the segment being allocated and the state of the memory prior to the allocation. This overhead can be observed in Lotus 1-2-3 by the difference in time taken to return to the READY state after allocating small and large numbers of cells by, for example, a COPY command.

3.5.3 *Write Access*

Write accesses proceed just like read accesses. This is because virtual memory systems are early select and the presence or absence of a page or segment is determined before real memory is accessed.

If the write access is a hit, the write is made to the AU in real memory and the page table dirty bit, if used, is set. A dirty bit is used in conjunction with the valid bit to manage the de-allocation and allocation of pages or segments in real memory. Unlike caches, there are no known implementations of the equivalent of a write through strategy for a virtual memory system, because the time to write a single AU to the disk is quite large and would block the disk from other activity. Write buffers would not eliminate this problem. Thus, all virtual memory systems use a write back strategy.

If the write access is a miss, the missed page is allocated into real memory and the write operation is performed again. There are no known no-write-allocate strategies, like those of caches, with virtual memory systems. The reason for this is the same as for not using a write through strategy: the very long transport time makes writing a single AU to disk unjustifiable.

3.5.4 *Location of Tables*

The first virtual memory machine, the Atlas shown in Figure 3.15, had its page table in an associatively searched hardware register file. As the size of the virtual address space has increased, larger tables have been required and placed in real memory.

One might ask if the tables in real memory are addressed in real memory space or virtual space? To my knowledge, (except for Multics) the tables are never placed in just any virtual space but in dedicated pages (identity name translations) of virtual memory space. If the page tables are paged, there can be a deadlock in paging as one name transla-

tion is forced to wait on another name translation that is waiting on the first, and so on. Thus, access to page tables is mapped by identity and the time for name translation via a page table tree is eliminated. The RS/6000 accesses its table in real addresses without the identity translation step [IBM90].

In addition to storing the page tables in real memory (or an identity translation), if the system has a data cache the page tables may be cached as well. Caching of page tables can further reduce the name translation time, as is described in the model found in Section 3.6. There is no known published data on cache P_{miss} when storing page tables. It may well be that a page table cache could be a useful variation of caches. On the other hand, any chip area needed for this cache may be more usefully devoted to larger TLBs.

Section 3.4 discusses the Motorola MC88200 [MOTO90a], which provides a dual TLB facility. A separate TLB known as the *block address translation cache* (BATC) translates the virtual address into a pointer into a 512-Kbyte block of real memory. Management of the BATC is under program control as there is no recourse to a multilevel translation table if there is a TLB miss. With a miss, the operating system is invoked to perform all of the management functions and is supported by some special instructions for loading the BATC.

3.5.5 *Table Walking Methods*

The MC68040 multilevel direct page tables are stored in real memory, as shown in Figure 3.40. If a successful name translation cannot be accomplished in the TLB, the address is translated with the memory resident tables. The various address fields and control bits of the table entries are allocated into groups of the AUs. The use of real memory for page tables leads to the requirement that the tables must be traversed by fetching and decoding the entries of each valid level of the hierarchy, a function called *table walking*.

Early systems such as the GE-645 Multics [DENN72, ORGA72, MATI80] and at least one contemporary system, the MIPS R2000/R4000 [MIPS87], perform table walking with a program using the normal instruction set of the processor. For these systems, referencing each level of the page table requires a sequence of instructions such as

1. Load page entry.
2. Select valid bit.
3. Branch to 8 on not valid.
4. Select pointer field.
5. Extract Index value from address.
6. Concatenate to form memory address.

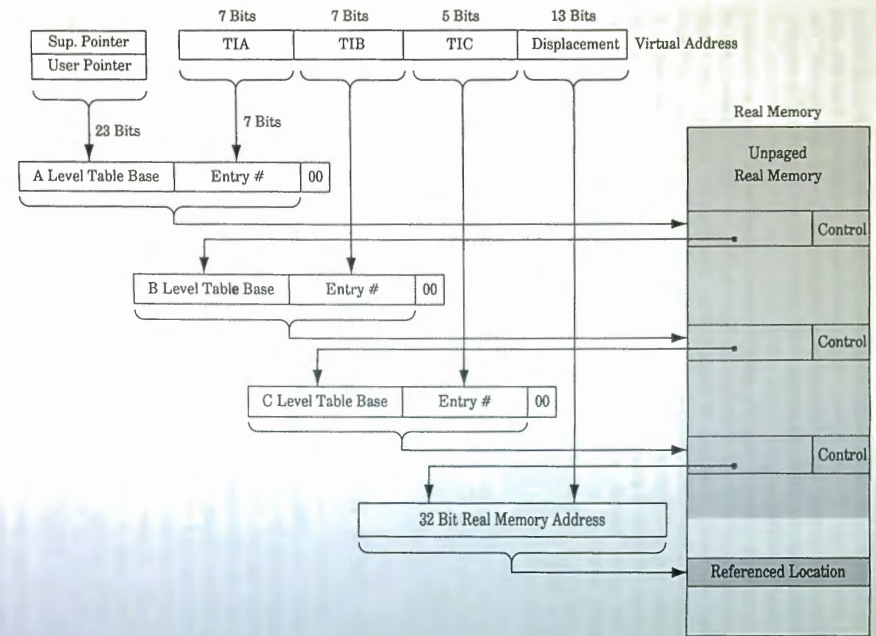


FIGURE 3.40 MC68040 multilevel page table storage.

7. Leaf node? no, branch to 1; yes branch to 9.
8. Fault exit.
9. Continue.

Each level of the page table will have an overhead of approximately eight instructions. A three-level system has an overhead of approximately 24 instructions or 30 memory cycles for each memory reference that is not translated in the TLB. The actual time required for a translation is a function of whether or not the page tables are cached and whether or not there is an instruction cache that may be holding the short program. Also, if the program is paged and the page is not "locked in" there could even be page faults during a table walk.

This software approach is still possible, but many modern memory management units have special logic that performs these functions. The logic is known as *table walking hardware,* and it significantly reduces the time required to access the page tables. Motorola calls this operation *translation table searching,* a function implemented in microcode.

The page table walking overhead burden is reduced by incorporating a state machine controller that implements the table walking function; an example of this controller is found in the MC88200 Memory Management Unit. Using hardware table walking eliminates most of the memory cycles and the possibility of cache misses and page faults. The net result is that the time required to translate is approximately one memory cycle per level of translation.

Inverted page tables, such as those used in the RS/6000, use a form of table walking in their linked list structure. For these systems, the control of searching down the list is vested totally in hardware. There are no known systems that are software-based. The reduction in weighted average name translation time with hardware over software table walking is dramatic and is modeled for the VAX 11/780 in Section 3.6.

3.5.6 *Instruction Set Support for Virtual Memory*

As I noted previously, a major portion of the work (but not the time) associated with servicing a page or segment fault is performed by the processor. Instruction set support is found in the three general areas of (1) table walking, (2) table management, and (3) interrupt support. For multitasking systems, the time to execute a table walk is important if it cannot be overlapped with other processing. For single-user virtual memory systems, the table walk time is small with respect to the disk time and can usually be ignored.

Table Walking Support

Table walking with software requires instruction set support to execute the algorithm noted above. The instruction set should include the usual field extraction, logical operations, and testing instructions found in general purpose instruction sets. In general, no additional instructions are required. Table walking code can be provided as a subroutine that is called when there is a miss on the TLB.

If page faults are frequent, a special subroutine call/return, designed for this purpose only, can eliminate most of the overhead for this rather special subroutine. It may also be helpful if the subroutine is locked into the cache to eliminate instruction cache misses.

Table Management Support

Table management support is required due to the large number of tables that must be loaded and stored under program control. The extension register(s) portion of the virtual address are loaded by the operating system with special load instructions. Special load instructions are also necessary for the segment registers for those systems that use this method for extending the effective address.

The IBM RS/6000 is one example of the special load instructions. There are a number of hardware registers collectively named *storage control registers*. First, there are 16 *segment registers* (SR); the most significant bit of each signifies if the segment is an I/O or processing (normal) segment. Another register is the *transaction ID register* (TID); this register contains a 16-bit segment identification number. Two other registers are named the *storage description registers* (SDR0 and SDR1). Special instructions are provided to load all of these registers. Note that these registers are never modified by a translation or a side effect of a translation; however, they can be modified as the result of a page fault. Thus, a store instruction must also be provided to store these tables for a context switch. Loads and stores are accomplished with move to sri (*mtsri*) and move from sri (*mfsri*) instructions that are moves between the special registers and the general purpose registers.

An inverted hash access system has other tables, the hash anchor table and page tables, that must be loaded and stored under program control. These tables are located in real memory, and the usual instructions that manipulate memory locations serve to load and store these tables. TLBs and other cachelike translation tables must be loaded and stored as well. The need to store the tables is a consequence of the fact that these tables can be modified between context switches.

Page Fault Interrupt Support

An executing program references memory with name translations via the TLB and the page tables assuming that the referenced pages are in real memory. If the page is not in real memory, the present bit in the translation system will trigger an interrupt, an action known as a *page fault* or *memory exception trap*. When there is a page fault, there are a number of tasks that must be accomplished. The following list is for a uniprogramming environment. Multiprogramming or multitasking requires other steps and is outside the scope of this book.

1. Trap the operating system.
2. Save the user and process state.
3. Write out a "dirty page" if necessary (virtual memory uses write back).
4. Allocate the new page to real memory.
5. Translate the virtual address to a disk file address.
6. Issue a read to the file system.
7. Read page into the allocated location.
8. Update translation tables and TLB.
9. Restore user and process state.
10. Resume processing.

The actual time required to execute these steps is not as great as the length of the list would suggest. For example, assume eight of the steps (not counting the transport times of steps 3 and 7) require an average of 1,000 instructions and each instruction requires 50 ns. The total time to service a page fault interrupt is 0.4 ms, a trivial part of the one or two 20-ms to 30-ms disk latencies.

3.5.7 *Memory Access Control*

This section discusses two issues: *protection* and *proper access*. Protection has to do with assuring that a reference to memory is authorized. Proper access has to do with the correctness of the access, as in not attempting to execute a floating point datum as an instruction. The issues of protection of objects or abstract data types and system access security are outside the scope of this book, and references to these subjects can be found in most books on operating systems.

Protection

Early research into the implementation of multiuser time sharing systems pointed to the need to protect programs and data from unauthorized access [DENN65, GRAH65, WILK68, GRAH72]. For reasons of

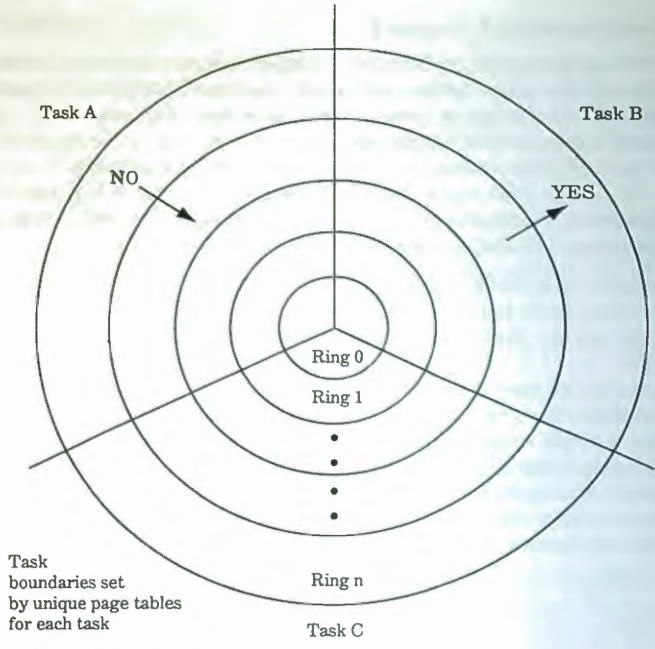


FIGURE 3.41 Rings of protection.

efficiency, however, it is desirable to permit sharing between users of programs and some data, a desiderata that introduced the need for protection. Graham [GRAH72] lists seven levels of protection for multiuser time sharing systems; each level requires additional hardware support for their implementation. The requirement for sharing and protection is just as important in processors that execute a number of tasks (multiprogramming or multitasking) that share resources.

The view of protection we have today has its roots in the MIT Project MAC and Multics. Graham [GRAH65] published the idea of *rings of protection,* which is shown in Figure 3.41. Briefly stated, Graham's idea is that a "process executing in ring i has no access whatever to any segment in ring j, where $j < i$." This view of protection leads immediately to the requirement that somewhere in the name translation system there must be a tag to denote the level of protection. Because control bits are somewhat limited in the page table entries, there is a limit to the number of rings of protection.

For paged virtual memory systems, protection is exerted at the page

level, or at the block level if block translation is employed. Examples of the use of control bits that provide protection can be found in a number of processors. The i386, i486, and Pentium devote two bits in the descriptor tables, which provide four levels of protection. These levels are: kernel, system services, custom extensions, and application. Two protection bits are also used in the page frame table entry of the RS/6000. In the most general implementation of rings of protection, if control is being transferred to a ring other than the current one, an interrupt occurs and the operating system is invoked to perform the proper housekeeping tasks.

Proper Access

Protecting memory from an improper access is the second issue of access control. One task should not be able to interfere with another task without mutual agreement. Four possible ways [SITE80] of providing memory protection are

1. bounds registers,
2. memory keys,
3. translation table keys,
4. distinct addresses spaces.

These four protection methods are discussed individually below. It should be noted, however, that combinations of the four methods are implemented in some specific computers.

Bounds registers were first used in nonvirtual memory machines. Bounds registers provide two pointers: one to an upper address limit, the other to a lower address limit. Bounds can also be specified with a starting address and length. If a reference is found to be outside these bounds, an error interrupt is signaled. The bounds registers are loaded and stored by the operating systems using privileged instructions.

For a system that will have a number of tasks resident in memory at one time, there may be a number of bounds register pairs, one for each task. The register pairs are addressed by a pointer that is set when a task is switched in. The use of bounds registers assumes that a task, program, and/or data, is allocated within a contiguous address space. Bounds registers permit sharing between tasks of an address space by identifying overlapping spaces by the bounds registers. Bounds registers, however, present restrictions with, for example, a single word containing a semaphore that is accessed by a number of tasks. The semaphore would have to be in the highest address of one task and in the lowest address in a second task. Access by three tasks is difficult.

For paged systems, protection is at the page or block level and for segmented systems at the segment level. For example, the segment

length field of a segment table operates in the fashion of a bounds register. An example of this is the block translation system of the PowerPC 601 that has a block partition of memory that is similar to a segment. A block name is presented to the block TLB and is checked for being within the starting address and the length of the block. Block length is encoded with six bits, rather than having an upper address, in the BAT registers discussed previously. The length of the block is a binary progression: 128 Kbytes, 256 Kbytes, 512 Kbytes, . . . , 8 Mbytes, an encoding designed to facilitate length checking [MOTO93].

Memory keys specify the type of access that is permitted to a memory location, page, or segment. Typical access types are: Unlimited, Read Only, Execute Only, and/or a task I.D. The executing program key is usually found in the processor status register that, in combination with the type of access, is compared to the "lock" associated with the memory and found in the name translation system. If the key matches the lock, the requested access is performed. There is usually a "skeleton key" that can be used by the operating system to access all of memory regardless of the key associated with each of the keyed memory spaces.

The memory keys are usually contained in the name translation maps and/or the TLB. Thus, an improper access is terminated before the read or write is actually performed. Memory keys permit access to noncontiguous page frames in real memory, unlike bounds registers. Sharing is more difficult, however, as sharing is at the block level. Shared variables may need to be relocated with a context switch increasing context switch overhead. If different accesses are legal for different tasks, the protection fields in the TLB must be changed when tasks are switched.

The PowerPC 601 illustrates the use of memory keys. The supervisor mode has the keyed accesses of read/write and read only, while the user mode has the keyed accesses of no access, read only, and read/write. The operating system can block user programs from accesses by evoking the no access key. This description is not complete; additional information can be found in [MOTO93].

Translation table keys provide protection by having an access table for each task. These tables, one for each task, contain the legal access methods for each page of real memory and permit different legal accesses for different tasks with a minimum of overhead. As there are multiple tables, only a switch of a pointer is needed to select the table for the new task. Images of the keys are found in the page tables and are checked along with the check of the virtual address tag.

Distinct address spaces can be used to provide protection by assuring that different tasks do not share the same virtual address space. This protection is accomplished by making the task name a part of the virtual address. With the task name in the most significant bits, there can be no

overlap between addresses in different tasks. One example of a distinct address space system is found in the ix86 processors that have segment registers associated with each type of access, for example, code and stack.

3.5.8 *Choice of Page Size*

This section addresses four tradeoffs in determining page size.

1. Reducing internal fragmentation.
2. Reducing disk/memory transfer time.
3. Matching the page size to the levels of the name translation tables.
4. Reducing the miss ratio.

Internal fragmentation was introduced in Section 3.1. When a page size is selected, there is a waste of usable memory space. First, because the allocation block is not an exact multiple of the page size, internal fragmentation occurs. Internal fragmentation is reduced with small pages because the wasted space is limited to a fraction of a smaller page. Second, when a program and its data is allocated to real memory, space must be provided for the page table entry of the allocated page. Page table space is not available for the program and data; it is pure overhead space. As the page size decreases, more pages must be allocated and the space used for page table entries increases. Thus, what page size will balance these two counterforces and minimize the loss of memory? Early work on this design problem, and the model derived below, is found in [WOLM65], who attributes the first version of the model to J. B. Kruskal. This model determines an optimum page size to minimize wasted real memory.

Assume that the number of AUs that must be allocated for a program (instructions and data) is n; the page size is p; and a AUs are required for each page table entry. A direct one-level page table, in memory, is assumed. Thus, the use of main memory by page tables and internal fragmentation represents a loss of real memory to the user.

Memory loss = loss due to internal fragmentation + loss for allocating a page table entry,

$$\text{Loss due to internal fragmentation} = \frac{p}{2},$$

$$\text{Loss for allocating a page table entry} \approx \frac{an}{p},$$

$$\text{Memory loss} = C = \frac{p}{2} + \frac{an}{p}.$$

The derivative of C with respect to p is taken and set to zero as

$$p_{\text{opt}} = \sqrt{2an}.$$

This model shows that in order to minimize the loss of real memory due to internal fragmentation and page table entries, large pages are best for large programs and data sets that are becoming more common. However, with lower cost memory resulting in larger real memory, internal fragmentation is not viewed to be the problem it was three decades ago.

As described in Section 3.3.1, today, page tables are allocated into pages themselves, making this model incorrect as the first allocation requires a full page. With the internal fragment loss of one-half page and with page tables allocated to pages that lose one-half page, the total loss is one page and is invariant with page size or the number of allocations. Thus, the size of a page is more-or-less not an issue as far as real memory loss is concerned.

Reducing disk/memory transfer time is the second tradeoff option for determining page size. Pages transported between the disk and real memory incur a significant overhead due to the long access time of the disk. The time per byte transferred is reduced as the page size is increased. As with the discussion of block size for caches found in Chapter 2, if the page size is very large, the transfer time per byte approaches the data transfer time with the access time overhead totally prorated. Note, however, that page sizes today are far from this limit. The larger main memories of modern computers and the reduction in disk access and transfer time, discussed in Chapter 5, tend to drive the page size upward.

Matching the page size to the levels of the name translation tables is a determining factor. The choice of page size can determine the number of levels in a direct multilevel page table. This issue is discussed in Section 3.3.1. Larger page sizes require fewer levels for name translation.

Reducing the TLB miss ratio is another consideration in the selection of a page size. The larger the page size, the smaller the TLB miss ratio and the smaller can be the TLB. This is the reason for the use of two TLBs for two different page sizes in a number of processors such as the MC88200 as described in Section 3.3. Larger page sizes led to smaller TLBs, which is an important consideration in some designs. A large page has another virtue because a cache can be placed in virtual memory

space. That is, the displacement in the virtual address addresses the cache while the virtual name translation is being performed. This cache architecture is discussed in Chapter 4.

From the above discussion, it can be seen that page size selection is a tradeoff process to achieve a balance between transport time, internal fragmentation, and name translation time. It is interesting to note that the flexibility in page size provided by the Motorola 6851 has been largely abandoned by designers for a fixed page size. However, the IBM RS/6000 gives a choice of two page sizes while the Digital Alpha provides page sizes of 8 Kbytes, 64 Kbytes, 512 Kbytes, and 4 Mbytes. The evolution of virtual memory systems has arrived at the consensus of a 4-Kbyte page size. The page sizes of some contemporary processors are given below.

VAX	512 bytes
IBM S/370	4096 bytes
RS/6000	4096 bytes
PowerPC 601	4096 bytes
i486	4096 bytes
Pentium	4096 bytes
MC88200	4096 bytes
MC88100	4096 bytes
R2000	4096 bytes
SPARC	8192 bytes

3.5.9 *Addressing the Disk*

This section discusses the way that the disk address is found when a new page is allocated following a page fault. Recall from Section 3.1 that the required name translation is referred to as using Map 3. This name translation problem is similar to the issues discussed in the preceding sections, which deal with translating virtual names to page frame addresses. Here we need to translate a virtual name to the disk addresses of a page, a process that requires a table(s) to hold translation information. Note that as with virtual memory, only page names need translating as the disk is never addressed at a lower level.

There are two ways of providing translation tables. First, the disk page address is contained in the page table entry [TANN84]. With this approach, the leaf page table entry would contain not only a field for the page frame address of the page in real memory but also a field that contains the address of the page on the disk. These disk address fields

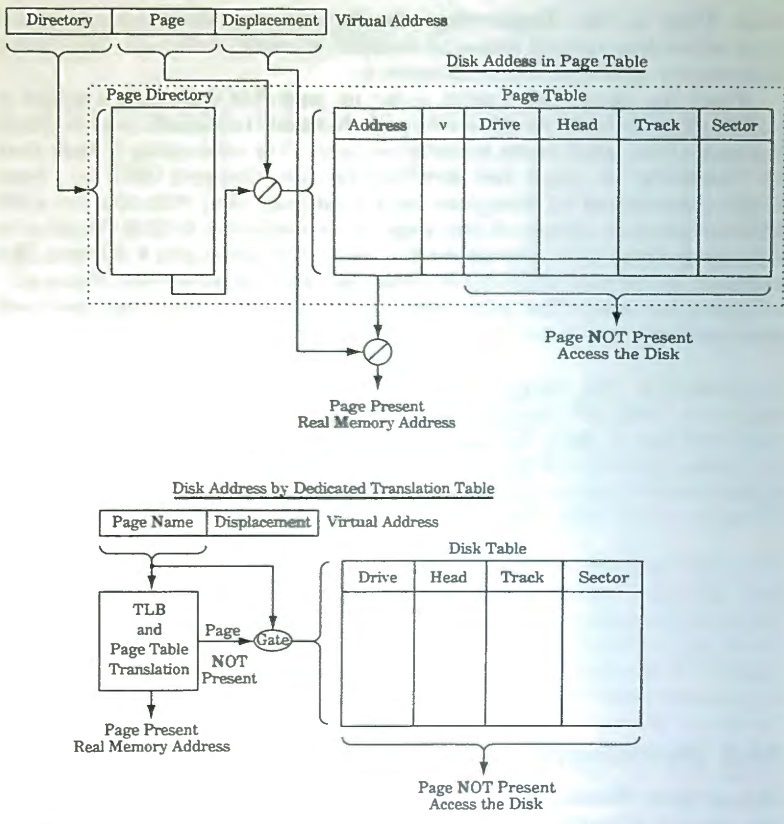


FIGURE 3.42 Disk translation table placement.

would have information on drive, head, track, record, and the like, which as Figure 3.42 indicates, is a relatively large number of bits.

A variation of this technique is found in the i386, i486, and Pentium [INTE90]. Because multilevel tables are direct with an entry for each page in virtual memory, the page table entries can be either the translation address or the disk address (MAP 2 or MAP 3). Only one map is valid at a time; the valid bits indicate which address is stored. When there is a page fault, the disk address is used to access the disk and is then replaced with the page frame address and the various control bits.

Neither version of the technique of placing disk address information

in the page tables is suitable for inverted page table name translation. The reason is that an inverted page table contains only the name translation information of pages resident in real memory. Thus, another technique for disk address translation is needed.

This second technique, also shown in Figure 3.42, starts anew, upon a page fault, with the virtual address that caused the page fault, and translates that address via a dedicated translation table (MAP 3). With this method, the same number of bits are required to address the disk, but they are managed in a separate table. The translation of the virtual address into the disk address can be via direct multilevel translation or the file management system of the operating system.

3.6 Virtual Memory Performance Issues and Models

There are two major performance issues with virtual memory systems: name translation time and page fault time. The time to service a page fault has been discussed previously; thus this section deals only with name translation time. A rather old annotated bibliography of virtual memory performance measurements is found in [PARM72].

Name translation time is the weighted average time needed to translate a name and is added to the access time of main memory for systems without a cache and is added to cache access time for real address caches, as with the VAX 11/780 discussed in Chapter 4. Name translation by accessing tables can be a lengthy process, and multilevel translation tables create a significant performance problem. Each virtual memory reference requires one or more real memory references to access the tables, and if page table walking hardware is not provided, a number of instructions must be executed by the processor to interpret the contents of each page table entry.

A simple model for the time required to make a name translation is given below. This model is based on the unified TLB design of the VAX 11/780 [SATY81, CLAR83, EMER84, CLAR85]. The steps of translating an address are

1. Search the TLB and use the page frame address on a hit.
2. If the TLB search fails, the address is translated with cached page table information. On a translation hit, the translation information is placed in the TLB and the name translation is restarted in the TLB. Note that while a normal cache access requires a translated address, microcode has untranslated access to the cache for name translation purposes.
3. If the cache translation fails, the name is translated with page table

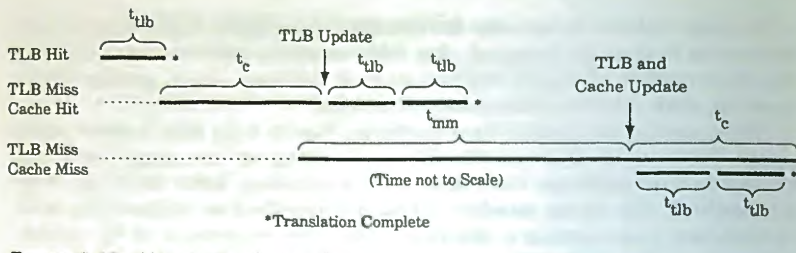


FIGURE 3.43 Name translation time.

information found in real memory. The translated information is placed in the cache and, in parallel, the TLB. The name translation is restarted in the TLB. Note that microcode table walking is provided for untranslated access to the page tables in real memory.

The VAX 11/780 memory performs name translation on data accesses and instruction accesses that miss the small instruction TLB. A timing diagram of the name translation process is shown in Figure 3.43.

The weighted average translation time (WATT) for a successful translation is

$$\text{WATT} = f_t \times t_{tlb} + f_c(t_c + 3t_{tlb}) + f_m(t_c + 3t_{tlb} + t_{mm})$$

where

f_t = fraction translated in TLB,
f_c = fraction translated in cache,
f_m = fraction translated in memory,
t_{tlb} = time for a TLB translation or update,
t_c = time for translation in the cache and a cache cycle (read or write),
t_{mm} = time for a translation in main memory.

The following assumptions are made:

$$2t_{tlb} \ll t_{mm}, \qquad f_t + f_c + f_m = 1.$$

Note that the fraction of translations performed at each level are equivalent to global hit ratios of multilevel caches. Clark [CLAR85] provides values for the parameters for this model derived from simulation. These parameters are

$$f_t = 0.97, \quad f_c = 0.012, \quad f_m = 0.018,$$
$$t_{tlb} = 1 \text{ clock}, \quad t_c = 6 \text{ clocks}, \quad t_{mm} = 18 \text{ clocks}.$$

Using these parameters in the model gives a weighted average name translation time of 1.56 clocks. For the VAX 11/780, the average number of clocks per instruction, without TLB miss delays, is approximately 10. This time includes the one clock for translating the data addresses through the TLB. Thus, for the parameters above, 0.56 clocks must be added to the 10 clocks of the basic execution time. TLB misses, therefore, add approximately 6% to the total execution time. This result compares favorably to the measured results of [CLAR85] of 5.1% to 8.0%, for various benchmarks. From the same parameters, 19.8 clocks are spent for each miss on the TLB. This result also compares favorably to the measured results of 21.0 to 22.1 clocks [CLAR85].

A name translation time of 1.56 clocks is a reasonable penalty for a processor that spends 10 clocks per instruction. For pipelined processors, however, discussed in Chapters 6 to 10, the clocks per instruction is approximately 1.5 (without translation delays). If there is an additional 0.64 clocks for name translation, the total execution time will be increased by approximately 30%. This is a major reason why real addressed caches are not used in most pipelined processors (described in Chapter 4).

Another issue can be addressed with this model. What is the performance benefit of hardware table walking? Assume that t_{mm} is 48 clocks rather than 18 with table walking. The weighted average translation time is 3.08 clocks rather than 1.56. The performance impact on the VAX 11/78 would be approximately 20% rather than 6%, a clear performance advantage for a processor with a cache addressed with real addresses.

3.6.1 *Published Page Fault Ratio Data*

This section discusses the identified published page fault ratios; unfortunately, little current information has been found. Rather old information on page fault ratios, as a function of page and real memory size (expressed as number of allocated pages), is provided by [CHU74] and excerpted in Table 3.13. Note that this data was taken in the era of small memory on the Sigma 7 and may not be representative of contemporary systems.

The presentation of this data is similar to other tables displaying miss ratio data. Diagonals from lower left to upper right are equal real memory sizes. This data shows very interesting properties. First, and this is to be expected, for any page size, as the size of real memory is increased permitting more pages to be allocated, the page fault ratio becomes constant at the value required to page in/out the working set. Once a working set is in memory, no further paging is required until the working set is paged out to the disk. When the memory is large, a larger

Memory Partition in Pages	Page Size in 24-Bit Words			
	64	128	256	512
8				.0018
16			.0012	.0002
32		.0012	.00002	.00001
64	.0012	.000035	.00002	
128	.00007	.000035		
256	.00007			

Note. FORTRAN compilation, LRU replacement.

TABLE 3.13 Page fault ratios.

page size will give a smaller page fault ratio because fewer page faults are required to page in the working set. Loading and storing the working set presents a minimum page fault ratio regardless of the page size, thus larger pages will reduce the page fault ratio.

Another interesting observation is that for a fixed memory size, the page size should also be as large as possible. For example, with a partition of 256 pages and a page size of 64 words, the fault ratio is 0.00007, while with a partition of 32 pages and a page size of 512 words, the fault ratio is 0.00001.

4

Memory Addressing and I/O Coherency

4.0 Overview

The issues addressed in this chapter are frequently viewed as related to multiprocessors. However, contemporary uniprocessors with caches, virtual memory, and concurrent input/output data transfers have the same characteristics and problems that must be solved. This chapter addresses these issues in the uniprocessor context.

In the 1970s, before virtual memory systems gained acceptance, processor designers debated the advantages and disadvantages of the type of address generated by the processor; should this address be real or virtual? Morris and Ibbett present the arguments in [MOOR79]. With the almost universal acceptance of virtual memory, however, the issues now become the nature of the address presented to the cache and the address domain of the I/O system. This chapter considers two issues: (1) the addresses presented to the cache and I/O in a virtual memory system, and (2) coherency (also known as consistency) maintenance between the various spaces in a computer and its memory. These issues are present in even rather simple systems and become quite complex with larger multiprocessors.

Virtual memory computers must cope with the coexistence of virtual and real addresses in the system. Problems arise from two sources: multiple virtual addresses and replicated memory spaces. Examples are

Multiple virtual addresses
- Concurrent processors, such as I/O and the main processor
- Multiple processes;

Replicated spaces	▪ Two processes accessing the same value but in two locations ▪ A value can be in the cache and main memory

For systems with multiple virtual addresses and replicated spaces, there are four cases of relationships, which are noted below couched in terms of *virtual address* and *real address*.

Case I. Multiple instances of the *same virtual address—same real address*. A normal and correct access results when the same virtual address issued from any process accesses the same real address or entity.

Case II. *different virtual addresses—different real addresses*. This situation is normal and presents no problems in the correct execution of a program.

Case III. *different virtual addresses—same real address*. This situation is called a *synonym* or *alias* and is illustrated by

Program 1, Virtual Address $A \rightarrow z$;
Program 2, Virtual Address $B \rightarrow z$.

Synonyms are viewed as either a virtue or a vice by different system designers. For example, the Multics system depends on synonyms for sharing and has a system call to provide a synonym for a given name [ORGA72]. The CMU system MACH also depends on synonyms. On the other hand, the prevention of synonyms is important to other systems. A discussion of the unique problems of synonyms in real address caches is found in [WHEE92].

Case IV. Multiple instances of the *same virtual address—different real addresses*. This situation has the potential for a *coherency problem* if the values in the different real addresses are different. For example,

Program 1, Name $A \rightarrow x$;
Program 2, Name $A \rightarrow y$.

If the values in x and y are different, the values are not coherent. Note that addresses x and y may be in the same physical memory, such as a cache, or in multiple physical memories, such as multiple caches. One solution to this problem is to include the process name in the virtual address thereby creating unique names for the variables, converting a Case IV into a Case II. This solution requires a longer virtual address and more tag bits but is consistent with the view of large virtual addresses, a technique used with the IBM S/38 [DAHL80] and AS/400. There are other solutions to ensuring coherency that are discussed in later sections in terms of cache coherency.

4.1 Cache Addressing, Virtual or Real

Chapter 2, on caches, takes the simple view that an address is presented to the cache. Chapter 3 introduced the concept of virtual and real addresses. This section discusses the relationships between virtual addresses and real address as applied to caches. Recall that a cache is used to reduce the latency of a memory system while a virtual memory system introduces a name translation step that can add latency to the memory system. It would seem, therefore, that caches and virtual memories are at odds with each other. The primary design problem addressed in this section is balancing synonym and coherency control with cache performance and cache size. Note that nonvirtual memory systems do not have the problems discussed in this section as all addresses are real.

Before proceeding further, I give a brief review of addresses that are present in a system. Figure 4.1 shows seven address forms of interest: virtual address, effective address, real address, cache address, TLB address, BTB/BTC address, and I/O address. A virtual address has a page name field that may be divided into two or more subfields for addressing translation tables and a displacement field. A real address has a page frame address, which is the translated page name, and a displacement field. The cache address consists of a sector name, field that may or may not align with the boundaries in a real address or a virtual address, a set index field, a block address field, and a displacement field. Note that in the discussion to follow, the term sector is used even if there is only one block per sector. Finally, the address used by the TLB, BTB or BTC, and the I/O system for reading and writing memory will be either real or virtual and have fields allocated by the details of the specific designs.

Are real or virtual addresses presented to the cache and what are the consequences of one or the other? The issue is more complicated than

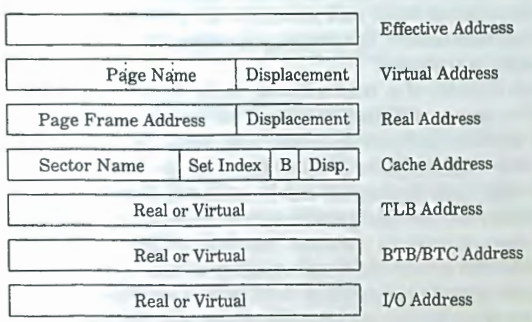


FIGURE 4.1 System address.

Set Index	Sector Name	Cache Name
Real	Real	Real address cache
Real	Virtual	No known implementation or name
Virtual	Real	Pipelined real address cache
Virtual	Virtual	Virtual address cache
Untranslated displacement		Restricted virtual cache

TABLE 4.1 Cache addressing options.

suggested by the simple choice of real or virtual addresses. Recall from Chapter 2 that a cache is addressed via two paths: the set index path and the sector name that is compared with the tag. These two paths can be either virtual or real, leading to four design options as shown in Table 4.1. The taxonomy of Table 4.1 is similar to that of [WU93].

The following sections will discuss these four design options. The real/virtual design is not considered.

4.1.1 *Real Address Caches*

Figure 4.2 shows the organization of a cache that is in real address space. The page name of the virtual address is translated via the TLB and page tables and then concatenated with the displacement. This process forms the real address, as shown. Fields of the real address are delineated to form the address into the cache: the sector name, the set index, block address if used, and the displacement. As the cache can be organized in any of the ways discussed in Chapter 2, the fields shown are for illustration only.

The sector name portion of the real address is stored in the cache tags and is compared to the sector name field. If a hit occurs, the data found in the cache is valid and is sent to the processor. If the comparison fails, a cache miss occurs, the memory is referenced with the real address, and the cache is updated.

A major benefit of a real address cache is that synonyms are not a problem. Two or more different virtual addresses cannot translate to the same real address location in either real memory or the cache. Another virtue of this organization is that there is no limit to the size of the cache. As many bits as desired can be selected from the real address for the cache address. The designer does not have to resort to extending the degree of associativity to increase the size of the cache. The only constraints on cache size is the total number of bits in the real address.

The most significant problem with a real address cache is its performance. With a cache in real address space, the virtual address must be translated before the cache can be accessed. The time to perform the

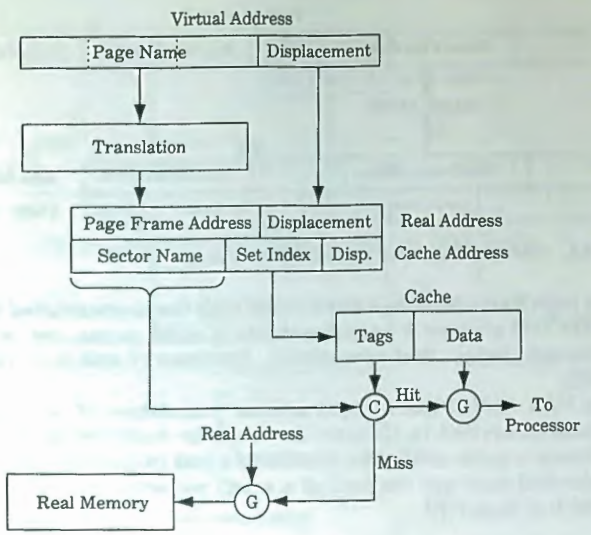


FIGURE 4.2 Real address cache.

page name translation via the TLB and/or page tables is added to the cache's access time. Since the TLB and the cache are usually constructed from the same technology, the cache's access time can be more than doubled. Recall from Chapter 3 that a typical address translation time can be 1.5 cycles. These cycles add directly to the cache access time. For processors such as the VAX 11/780 that have a CPI of 10 cycles, this overhead is not a significant loss of performance. The effective access time of a real address cache is

$$\text{Eff. } t_{ea} = (\text{WATT} + t_{ea})$$

where WATT = weighted average translation time.

However, note that because of unlimited cache address space, a large cache can be used, and the P_{miss} of the real address cache may be significantly lower than the P_{miss} of a virtual address cache. Reducing the P_{miss} of the cache may overcome some of the performance loss due to the serial name translation.

The cache addressing of the VAX 11/780 is a good example of a real address cache, as shown in Figure 4.3. The 30-bit virtual address is divided into a byte displacement field and a virtual page name field. The page name is translated via a TLB and page tables, which gives

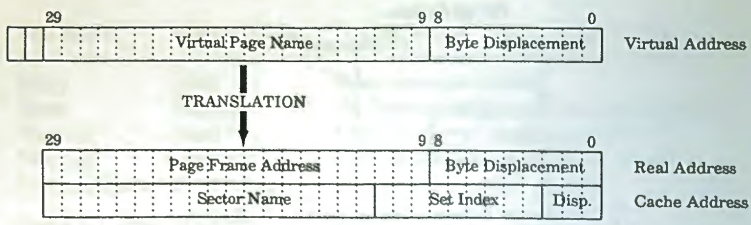


FIGURE 4.3 VAX 11/780 cache address generation.

the real page frame address concatenated with the untranslated displacement. The real address is partitioned into a sector name, set index, and displacement fields that constitute the cache address [LEVY80, LEON87].

The VAX 11/780 has a rapid address translation of an instruction page name (described in Chapter 3). Thus, for instruction fetches, the WATT delay is quite small. The benefits of a real cache (lack of synonyms and unlimited size) are realized at a small performance cost for a processor with a large CPI.

4.1.2 *Pipelined Real Caches*

From the above discussions, we see that a pure real address cache can have a significant performance problem. A form of cache that has the benefit of no synonyms of a real address cache and very small access latency has been called a *pipelined cache* [STON93] and is called a *pipelined real cache* in this book. As Figure 4.4 shows, the displacement field of the virtual address accesses the cache. In parallel with the cache access, a virtual page name is translated into a page frame address, a portion of which is compared to the page frame address that is stored in the cache's tag field. If the tags compare, the data is gated out of the cache to the processor. Otherwise, the real address accesses the main memory to fetch a new block into the cache. This cache organization permits the cache access and the translation operations to proceed in parallel.

The price paid for low latency and lack of synonyms is that the size of the cache is limited. The size limitation is bounded by "*the page size is larger than the cache size divided by the associativity*" [AGAR84]. For example, a byte displacement field of 10 bits and a set associativity of 4 limits the cache size to 4 Kbytes or AUs. While the logic paths via the cache and address translation are roughly equivalent, strict pipelining with latches may be required because a TLB miss can lengthen the translation time.

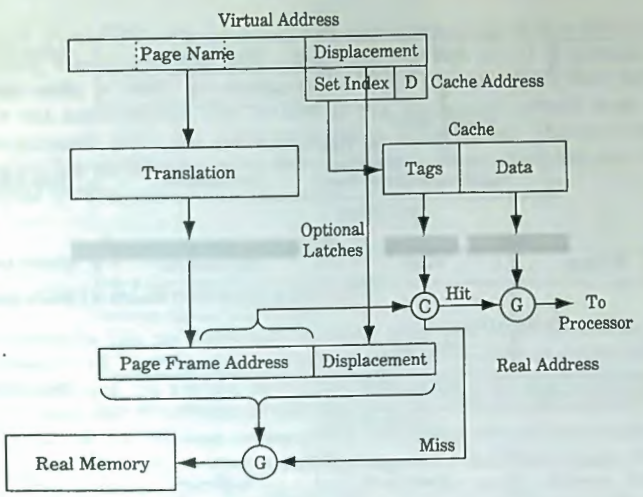


FIGURE 4.4 Pipelined real address cache.

If a constant stream of addresses is presented to a pipelined real address cache, this cache scheme will run very close to the pipeline rate of one cache access per clock. The effective clock period, however, will be determined by the maximum of the delays through the address translation path or the cache access time. The effective cache access time is

$$\text{Eff. } t_{ea} = \text{Max}[\text{WATT}, t_{ea}].$$

It may be unnecessary to use strict pipelining with latches; a technique called *maximum rate pipelining* does not use latches and is discussed in Chapter 6. The result of the address translation and the cache's output should arrive at the compare circuit at approximately the same time and then can gate out the data on a valid compare. Provisions must be made for long translations on a TLB miss.

Designers of the IBM S/360 pioneered the use of the pipelined real cache [MATI77]. The IBM 3090 has continued this cache design [TUCK86] as have cache designs in other processors. Figure 4.5 shows the pipelined real address cache address formulation for the IBM S/370/168, the Intel i486, and the IBM RS/6000.

The IBM S/360/168 cache is early select; the displacement provides the set index and displacement into a table of cache address information. The real sector names are found by translating the page name with the user ID via hash tables and comparing it with the real sector name tags

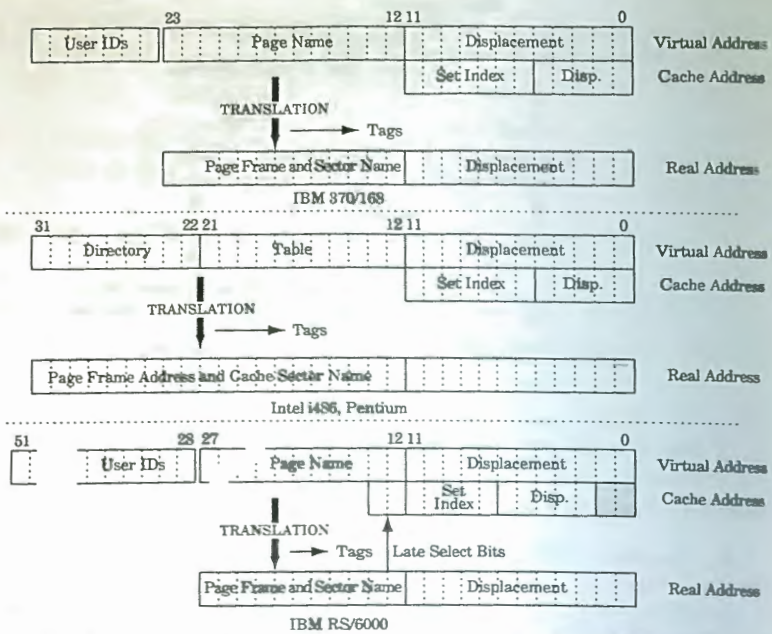


FIGURE 4.5 IBM 370/168, Intel i486, and Pentium, IBM RS/6000 cache address generation.

in the cache address cache. The output of the table is concatenated with the 5-bit displacement field to form the cache address itself.

The Intel i486 also uses the pipelined real cache organization. Although I am unfamiliar with implementation details of the i486, I believe that it does not have latches. The i486 cache is late select. The set index and displacement select the data from the cache sector that was identified by the translated directory and table names. For the i486, with a page size of 4 Kbytes and a 4-way set associative cache, a maximum cache size is 16 Kbytes.

The RS/6000 cache is another variation of the pipelined real cache. The 12-bit virtual address displacement accesses four 16-Kbyte caches in parallel as shown in Figure 2.20. Each of these four caches has 32 sets, 4 sectors with one block, and 128 bytes per sector. After the virtual page name and ID are translated, the two LSBs provide a late select of one of the four caches. Documentation on the RS/6000 indicates that latches are used to delineate the stages of this pipeline.

The PowerPC 601 changed the cache mapping of the RS/6000 by dropping the translated late select scheme. The 12-bit displacement is divided into a 6-bit set index and a 6-bit displacement; the cache is 8-way set associative. The 20 bits of the translated virtual page name are compared with the eight tags of the selected set. The organization of the cache is (64, 8, 1, 64) for a 32-Kbyte cache, half the size of the RS/6000 cache.

4.1.3 *Virtual Address Cache*

A solution to the performance problem with real address caches is the elimination of the translation latency by addressing the cache with virtual addresses, as shown in Figure 4.6. With this cache scheme, the virtual address is allocated into fields for accessing the cache. These fields are the usual ones: sector name, set index, and displacement. This scheme allows full flexibility in selecting these fields so that a cache of any size can be addressed. In parallel with the cache access, the page name is translated and concatenated with the displacement to form the real memory address.

Because page name translation is required only on a cache miss, the effective access time of this organization in the absence of cache misses

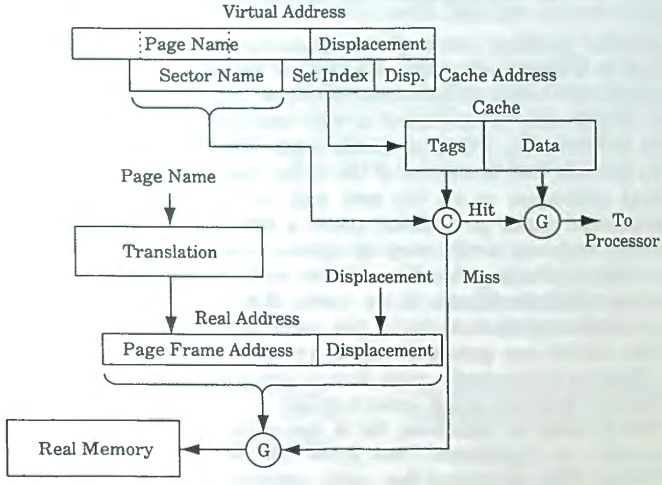


FIGURE 4.6 Virtual address cache.

is merely the access time of the cache.

$$\text{Eff. } t_{ea} = t_{ea}.$$

Synonyms are a major problem with a virtual address cache that does not use the entire page name for the sector name because two virtual names can refer to the same address in the cache. Figure 4.6 illustrates this problem in that not all of the page name is used as the sector name. A cache hit signifies that the AU has been stored in the cache under the page name that is currently being used and the AU is loaded into the processor. However, what happens if there is another access with a different page name that has the same LSBs and displacement but differs only in its MSBs? This access will use the same sector name and find the same AU in the cache.

What is to be done? A solution is to ensure that different MSBs of the page name are not used if a synonym will be a problem. M.D. Hill [HILL86] points out that most context switches call for a change of virtual address space. Thus there must be a guarantee against two processes using the same symbolic addresses that would permit the new process to read valid but incorrect data from the cache. One solution is to provide the capability to flush the cache on a context switch. Another solution is to extend the sector name to the length of the full virtual address. This is very expensive as the cache tags must be extended as well.

Another problem occurs if the operating system permits synonyms. If there is a cache miss, the miss is not conclusive evidence that the referenced AU is not in the cache; it can be there under a different page name. If the miss is processed, a new instance of the same variable is placed in the cache, which produces a potential coherency problem.

To prevent two instances of the value, there must be a search of all the real addresses of all the sets and sectors in the cache to see if the requested value is resident under a different virtual address. This procedure requires performing an inverse mapping of the cache sector name (that is, finding its real address) and comparing it to the inverse mappings of all the blocks in the cache. If this search of real addresses finds that the value is really in the cache, the tag must be adjusted so that the access can proceed without processing the miss. If the search fails, there is a true cache miss that is processed in the knowledge that a coherency problem is not being created.

The process of searching for a synonym in the cache can be accomplished in hardware; the Intel i860XP provides an example [INTE92a]. This processor has split instruction and data caches that both have support to prevent synonyms from becoming coherency prob-

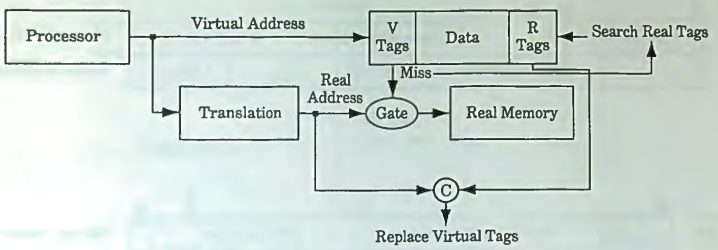


FIGURE 4.7 Intel i860XP caches.

lems. The caches are organized as (128, 4, 1, 32), and each sector has both virtual and real tags, as shown in Figure 4.7.

The cache is accessed with the index and sector name taken from the virtual address. If the tag matches, the cache access is normal for both a read or write. However, if the tag does not match, there is a potential miss, and a search is made of the real tags by comparing them with the now-available real address. If there is no match on the real sector name, there is a true miss and a bus cycle to the memory is initiated with the translated (real) address. Table 4.2 shows the action taken for four cases of read-write and hit-miss on the search of the real tags.

It can be seen that this system permits synonyms and ensures that there cannot be duplicate entries in the cache that can cause coherency problems.

The virtual address cache address fields of the i860 and SPUR processor [HILL86] are shown in Figure 4.8. The i860 has a very large virtual page, 2^{22} bytes, and an unextended virtual address that is consistent with the intended purpose of this processor. The caches are organized as (128, 4, 1, 32), and they carry a long (20 bits) tag.

Access	Real Tag Hit	Real Tag Miss
Read	Use entry and replace virtual tag with virtual page name that caused the miss	Place block returned from memory into cache and update both the virtual and real tags
Write	Write to block and replace virtual tag with the virtual page name that caused the miss	Use no-allocate policy and write direct to memory

TABLE 4.2 i860 synonym processing.

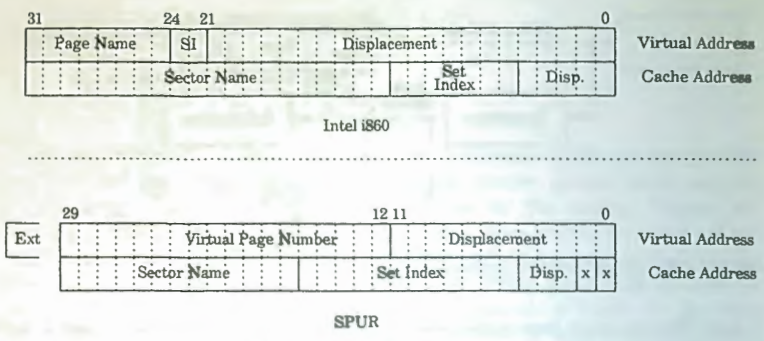


FIGURE 4.8 SPUR cache address generation.

The SPUR cache is organized as (4096, 32, 1) for 128 Kbytes and is word addressed. There are 13 tag bits with each sector along with a valid bit; the cache uses the write through policy. The two MSBs of the virtual address index into a table to select a virtual address extension of eight bits, giving a 38-bit global virtual address. This global virtual address is translated via tables if there is a cache miss.

D. Roberts et al. [ROBE90] and C. E. Wu [WU93] discuss the MIPS R6000 processors that use virtual first-level caches for instruction and data. However, the second-level cache is indexed by a translated real address while its tags are virtual. This hybrid scheme uses the second-level cache to help resolve synonyms and reduce the size of the on-chip TLB.

4.1.4 *Restricted Virtual Caches*

Restricted virtual caches are special cases of virtual caches. This organization combines the positive features of a virtual cache and a pipelined real cache. As shown in Figure 4.9, the cache set index plus displacement are constrained to be no longer than the displacement field of the virtual address. A portion of the full page name becomes the sector name of the cache address and is compared to the tags. A cache miss uses the translated virtual address to access the memory.

The advantage of this cache organization is performance because a page name translation is not required before the cache is accessed and pipeline delays of the pipelined real cache are eliminated. The problem, however, of synonyms remains and, as with pipelined real caches, the cache size is limited to the address span of the displacement (page size) times the degree of associativity. The size of the cache, once the page size is fixed, can be increased only by increasing the degree of associativity.

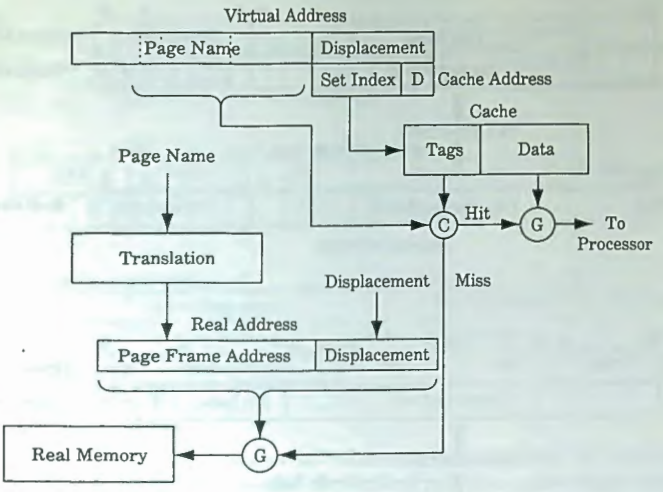


FIGURE 4.9 Restricted virtual cache.

Two examples of restricted virtual caches are the MC68030 and MC68040. Figure 4.10 shows the address generation for these two designs. These two caches are both on-chip and are accessed in parallel with the TLB. If a cache miss occurs, the off-chip memory is addressed by the real already-translated address.

The caches of the MC68040 use a smaller portion of the virtual address displacement for the set index and displacement. The MC68040 has a larger page—4 Kbytes or 8 Kbytes—than does the MC68030, and the full displacement field is not used for generating the cache address. The 6-bit set index is taken from the page displacement field of the virtual address. Each set consists of four words, each selected by the 2-bit displacement field. Note that the number of sets or the number of words in a set, or both, can be increased by a factor of 8 for future expansion.

4.1.5 *Summary*

Table 4.3 summarizes the characteristics of the four cache design options. There is no clear advantage to any one of these designs, as illustrated by the fact that all are used in contemporary processors. Note again that there are no known implementations of a cache with a real index and virtual tags.

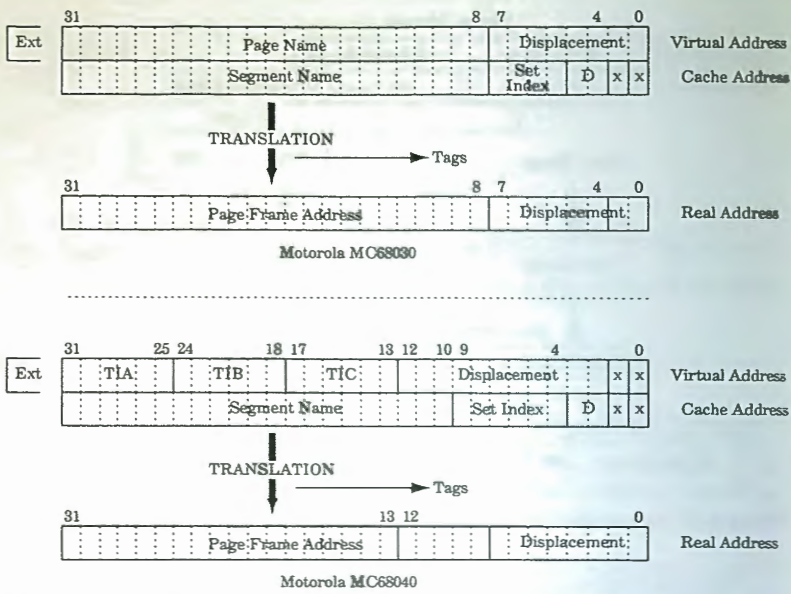


FIGURE 4.10 MC68030 and MC68040 restricted virtual caches.

Cache Type	Synonym Problem	Coherency Problem	Performance	Cache Address Size	Example System
Real	No	Yes	Translation delay	Unlimited	VAX 11/780
Pipelined real	No	Yes	Pipeline	Limited by degree of associativity	i486 PowerPC 601
Virtual	Yes	No	No delays	Unlimited	i860 SPUR
Restricted virtual	Yes	No	No delays	Limited by degree of associativity	MC68040

TABLE 4.3 Cache characteristics.

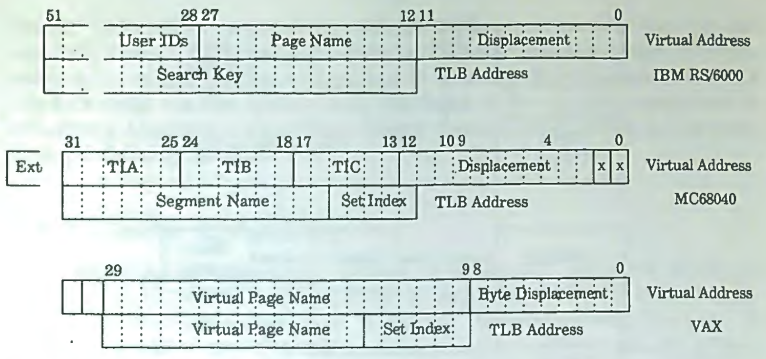


FIGURE 4.11 TLB address generation.

C.E. Wu [WU93] performed extensive simulations for evaluating the performance of these cache addressing options, cache organizations, and replacement policies. They conclude that pipelined real cache or the restricted virtual cache using MRU replacement provides the best choice for high-performance computers. These results are confirmed by the number of processors that use this cache today.

4.1.6 *TLB Addressing*

Translation Lookaside Buffers (TLB) are cache-like buffers that contain pretranslated page names. The purpose of a TLB is to reduce the latency of a page name translation when a reference to main memory is required. Because a TLB translates only the page name, the displacement field of a virtual address is usually (there are exceptions) not a component of the TLB address. Three examples of TLB addressing are shown in Figure 4.11.

The IBM RS/6000 has an associative TLB as shown in Figure 3.35. The 40-bit extended page name is used as the key to search the tag field of the TLB. The MC68040 has a 4-way set associative TLB with 16 sets. The set index and segment name are extracted directly from the virtual address. The VAX TLB has 32 sets and is 2-way set associative.

4.2 I/O System Addressing Design Issues

Section 4.1 discussed the issue of cache and TLB addressing with either real or virtual addresses. The same issue is present with the I/O system;

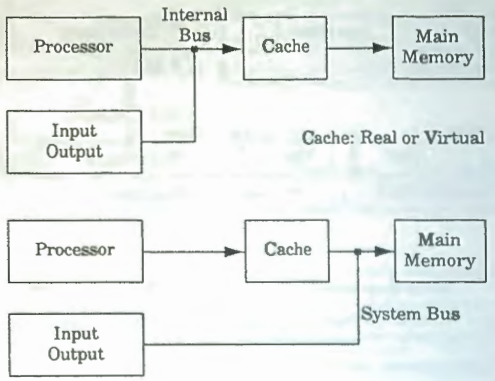


FIGURE 4.12 Data transfer paths.

should the I/O system be in real or virtual address space? The addressing issue is usually only a consideration for the case when an I/O process is writing into the system main memory and ultimately the I/O system must provide real addresses for storing data in the memory. As will be discussed in this section, designers must consider certain performance and coherency problems with I/O systems.

Before discussing I/O addressing, I will discuss the issue of routing data between the I/O processor and the memory. Figure. 4.12 shows two ways of routing data: (1) I/O via the cache and (2) I/O via the system bus to main memory.

Placement of the I/O transfer path impacts two areas: performance and coherency. The performance issue here is that if the I/O data path is via the cache, valuable cache cycles can be stolen from the processor. If the processors have a multiple cycle CPI (such as the Amdahl 470/v6), this is not a significant problem. But, with the I/O data path via the system bus to memory, bus cycles can be stolen that, with heavy bus loading, can impact the time for servicing a cache miss. Recall from Chapter 2 that bus utilization with a single processor can be approximately 50%.

With modern pipelined processors that can generate two cache accesses per clock, loading the bus seems to be a less serious problem than loading the cache. Data caches, however, are not as loaded as instruction caches and warrant investigation for data cache routing. In addition, contemporary microprocessors with on-chip caches transfer via the system bus—a practical consideration based on the need to minimize pin

count. Only older processors without on-chip caches transfer via the internal bus and cache. Given these considerations, the design of choice today is to route via the system bus to memory. For systems that route the I/O data via the system bus, the issue of the logical consistency or coherency becomes a significant design consideration which is discussed in the following sections.

4.2.1 *Cache Coherency*

The literature discusses the concept of coherency, discussed briefly in Section 4.0, in terms of shared-variable cache coherency for shared memory multiprocessors [CENS78, STEN90]. As pointed out, however, in [LEON87, GROC89, CRAW90], coherency problems can exist in uniprocessors with a concurrent I/O. Knuth [KNUT66] and Dijkstra [DIJK71] provide seminal references to the general issues of coherency. The following is an informal definition of a coherent system:

> A memory system is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address [CENS78].

van de Goor [VAND89] states that this definition is too weak to cover multiple-access problems found in multiprocessors or even processors with I/O channels. van de Goor's definition covers the case of a Read-Modify-Write operation that is sometimes used for synchronization, as in a Test and Set instruction (discussed in Section 6.7).

> A memory scheme is coherent if the value, returned on a LOAD instruction, is always the value of the latest STORE instruction to the same address; and a multiple-access operation has to be executed atomically—that is, excluding any other operation to the same address.

Based upon his definition, van de Goor gives two conditions that must be met to achieve coherency.

1. There is a single path to every AU.
2. There is a single copy of every AU.

Clearly both of these conditions can never be met in any reasonable computer system. The one path consideration is usually achieved, in part, for memory if the system contains only one memory bus. Dijkstra [DIJK71] states the requirement in another way, with regard to mutual exclusion, when he says: "The switch granting access to store on word

basis provides a built in mutual exclusion." Having only a single copy of every AU is difficult for systems with caches, register files, TLBs, and other such resources where an AU can reside in more than one space at a time.

Ensuring coherency and resolving true dependencies are closely related issues. Coherency relates to *values*, while true dependencies relate to *spaces* (discussed in Chapter 8). However, since values are bound to spaces during execution, the solutions to both problems revolve around managing spaces.

The potential for the lack of coherency in a shared memory multiprocessor with private caches is well documented. However, the potential coherency problems associated with an I/O process that reads and writes in memory concurrently with processor reads and writes are not covered as well. Consider the following operations and the related problem spaces.

Cache write through. As discussed in Chapter 2, if the memory system has a write buffer, the value in the buffer can be different from the value in the memory. Thus, reads from the cache must be checked against the value in the buffer.

Cache write back. This operation mode can only have a coherency problem if another processor (such as autonomous I/O systems) references the memory before the write-back has been accomplished.

A lack of coherency arises from the following situation: An AU has been fetched into the cache, loaded into the processor's register files, modified, and stored from the processor into the cache with a write-through policy. During the time from when the AU is modified in the processor to when the store into the main memory is completed, the AU is incoherent. That is, a read reference to the same main memory address by an I/O processor during this time will not produce the most current value of the AU. A write-back policy increases the time of incoherency.

Designers use four basic methods to maintain cache coherency for either a multiprocessor with multiple caches or a uniprocessor with a cache and concurrent I/O systems.

Method 1. *Noncached shared-variables.* Shared-variables are prohibited from being cached. Thus, all references to shared-variables must be served from the main memory. This method is sometimes called *static coherency protection.*

The VAX 8800 and the i486 are two examples of processors that use Method 1 to ensure that coherency problems do not occur with I/O operations. A buffer area in memory, addressed with real addresses, is reserved for I/O, and interlocks are provided to ensure that information in the buffer is not placed in the cache until the I/O operation is completed. These processors reserve 64-Kbyte buffers for I/O. When the I/O transfers are in the assigned buffer spaces, the system inhibits the

detection of possible false alarm coherency problems, as they cannot occur. Note that this method depends on the software to ensure that coherency problems do not occur.

Method 2. *One space for shared-variables.* Shared variables are cached in only one cache. This cache may be a dedicated cache or may be one of the multiple caches in the system.

This method requires that I/O be via the internal bus to/from the cache. The only known example of the use of this method is found in the Amdahl 470/v6. The performance penalty is too great for this method to be used with today's processors.

Method 3. *Write invalidate.* This method requires that a write to a shared-variable result in the invalidation of all other instances of that variable. Two forms of invalidation have been used: total invalidation of the cache and selective invalidation of the invalid AU [CASE78].

Method 4. *Write update (write broadcast).* This method requires that a write to a shared-variable be broadcast to all other instances of that variable, which are then updated.

Site [SITE80] generalizes Methods 3 and 4 by giving the design principle involved whenever buffers of any type (for example, caches, register files, and TLBs) are used to improve a processor's performance. This principle is: "If a datum is copied and the copy is to match the original at all times, then all changes to the original must cause the copy to be immediately updated or invalidated."

For both Methods 3 and 4, all caches must have the ability to detect a write from another processor to a variable currently resident and either invalidate that entry or update it. For the uniprocessor case with only one cache and one I/O port, the bus activity is observed to detect a write from the I/O port to the memory that should either invalidate or update the cache, a process called *bus snooping*. With the VAX, this action is called *watching the bus* [LEON87].

Methods 3 and 4 for maintaining cache coherency result in significant overhead to the memory system. If a write results in an update, all of the caches must halt and perform the update cycle even if the AU in the cache will not be later referenced. On the other hand, an invalidation strategy requires that the valid bits be turned off in all of the caches, causing a cache miss if the AU is later referenced. Weber [WEBE89] presents a study on invalidation patterns that shows 0.3 to 3.0 invalidations per shared-variable write for systems with 4 to 16 processors. Note that this data will not apply to the uniprocessor I/O coherency problem, which is the subject of this book, and no published data is known to exist.

If the caches are multilevel, the system must contain a method of identifying all of the caches that require invalidation or updating. The complexity of this task is reduced if multilevel inclusion is enforced.

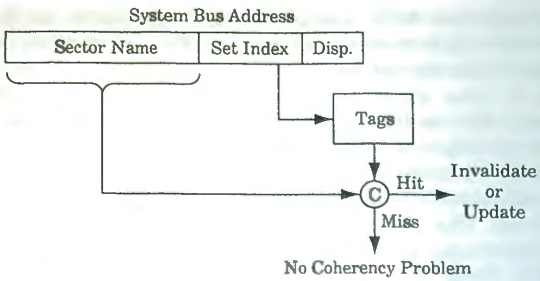


FIGURE 4.13 Snoopy controller.

4.2.2 *Snoopy Controllers*

As noted above, the problem of coherency exists in a uniprocessor with a cache and an I/O system with or without virtual memory capabilities. I/O reads-and-writes over the system bus have a policy that depends on the types of addresses used by the cache and the I/O system. The situations that will have a coherency problem can be detected by means of a *Snoopy Controller*, as shown in Figure 4.13.

When there is a write-to-memory by the I/O processor, a test must be performed to see if the address is present in the cache, indicating a possible cache coherency problem. With a snoopy controller, the system bus accesses a copy of the cache tags, or the tags themselves, with the set index and compares the accessed tag with the sector name. In general the snoopy controller is accessed by every bus. If there is a hit on the tags, the sector is resident in the cache and must be either invalidated or updated. If there is a miss, there is no coherency problem.

The snoopy controller function requires access to cache tags, a capability that can be provided in one of three ways. The design selection is based on cost and performance criteria.

Shared tags. One set of tags is shared between the users (I/O and processor) with the potential of a structural hazard. Concurrent accesses to the tags will result in a loss of performance, but sharing eliminates any tag coherency problem.

The least costly method for providing a snoopy controller is for the I/O processor to steal a cache cycle, access the cache tags themselves via the index field, and compare the tags with the sector name. This method eliminates the cost of duplicating the cache tags but has the problem of reducing the system's performance due to cache cycle stealing. As a point of interest, if one directory of tags for the system exists as compared to a directory of each cache's tags, the system is called *directory based*.

Replicated tags. There is one set of tags for each user. There will be no structural hazards, but coherency of tag information itself must be maintained.

A data coherency check is made by accessing memories containing replicated cache tags that can be accessed in parallel with cache cycles. Each of the tag memories for each of the sets must be replicated. Obviously, the tag memory of the snoopy controller must contain an exact image of the cache tag memories in order for the tags to be coherent. Thus, when a change is made to the cache's content that modifies a tag field, this change must be reflected in all the snoopy controller's tags.

With a replicated snoopy tag memory, the tag memories must be as large as the total tag memory of the cache. For a high degree of associativity, this can be a significant burden on the snoopy memory, another reason for the preference of direct caches in contemporary processors. The IBM S/370/195 reduces the cost of its snoopy controller by snooping only the tags of one cache sector by the replacement policy, thus restricting I/O operations to the snooped cache sector.

Shared tags with multiports. Concurrent access can be performed without a loss in performance and without the tag concurrency problem.

The third approach to snoopy capability is to make the processor's cache tag memory multiported. This approach gives the performance of replicated tag memories at a relatively small cost. This approach also eliminates the tag memory coherency problem as there is only one copy of tag information. The Intel Pentium and the PowerPC 601 processors provide examples of multiported snoopy cache tag memory.

For the i486 and MC88200, snooping is used only for multiprocessor systems, not for I/O. For these processors, duplication of tags is an unacceptable cost over the tags of the on-chip data caches. Writes to memory should be infrequent for small multiprocessors and the snooping function using the cache tags themselves eliminates the problem of tag coherency with replicated tags. The Intel i486 and the MC88200 use invalidation with the consequence that a subsequent reference to the invalidated location by the processor will cause a cache miss. For processors with on-chip caches, invalidation is the preferred design when the processors are used in a multiprocessor configuration. The processor detecting the conflict broadcasts an invalidation signal to all the other processors.

I/O Snoopy Operation

The snoopy controller must detect four cases that are the combinations of Read or Write of memory and the addressed sector being cached or not cached. Each of these four cases calls for a different action, as summarized in Table 4.4. The snoopy controller must detect the presence

Operation	Sector in Cache?	
	Yes	No
Write to memory	Invalidate or update cache and write to memory	Write to memory
Read from memory	Write through cache: Read from memory Write back cache: Read from cache and write back	Read from memory

TABLE 4.4 I/O cache coherency.

of the sector in the cache, comprehend the type of I/O operation, and command the proper action.

The first case consists of an I/O write to memory and a sector that is not cached. In this case, no coherency problem exists and the write can proceed without a problem. The second case is an I/O write to memory and a sector that is cached. There are two basic design options: (1) write to memory and invalidate the sector in the cache, or (2) write to memory and update the sector in the cache.

The third case occurs when there is an I/O read from memory of a sector that is not in the cache. In this case, the read proceeds with the danger of a future coherency problem. Finally, the fourth case occurs when there is an I/O read from memory and a sector that is in the cache. If the cache is write through, the read is made from memory. If the cache is write back, the read is made from the cache. These policies ensure that the most current value is sent to the I/O system.

There are a number of papers describing the performance of various snoopy cache systems when used in multiprocessors [ARCH86] and [EGGE89]. To my knowledge, however, no similar published data exists on the performance for the uniprocessor case with I/O operations.

4.2.3 *DMA I/O Configurations*

With the above background, I turn to the design options for virtual memory systems with caches and concurrent DMA I/O, that is, the design options for addressing the cache and the type of address used by the DMA I/O system. The term DMA I/O is used here to signify not only DMA but programmable channel controllers as found in systems such as the S/360. Table 4.5 shows the four design options for real/virtual cache and real/virtual DMA I/O addressing. In the discussions to follow on these four design options, attention in the figures is focused on the addresses. The data routing for each of these cases is via the

Design Options	Cache	DMA I/O
1	Real	Real
2	Real	Virtual
3	Virtual	Real
4	Virtual	Virtual

TABLE 4.5 Cache and DMA I/O addressing.

system bus as shown in Figure 4.12. For expository purposes, Figures 4.14 to 4.17 show the snoopy controller with replicated tags.

Design Option 1, *real cache—real DMA I/O*. The cache is accessed with real addresses, as shown in Figure 4.14, and the DMA I/O processor writes into the memory with real addresses. A coherency problem can result if a DMA I/O port is writing into a cache sector of memory that is resident in the cache. If a DMA I/O-write is allowed to proceed, an incoherent situation can result between the memory and the cache. The bus snooper, indicated by *S*, observes all addresses passing across the bus and the addresses are compared to tags in the cache that are represented in the snoopy controller. If a match is found, the cache is directed to invalidate or update the sector in the cache.

Design Option 2, *real cache—virtual DMA I/O*. The cache is accessed with real addresses, as shown in Figure 4.15, while the DMA I/O processor uses virtual addresses. The virtual DMA I/O system addresses must be translated to give real memory addresses. This translation can be accomplished either by time sharing or multiporting one address translation system or by duplicating the address translation system. Time sharing one translation system may create a performance problem, while duplication has a cost and a coherency penalty because the duplicate tags must also be coherent.

Design Option 3, *virtual cache—real DMA I/O*. A virtual cache with

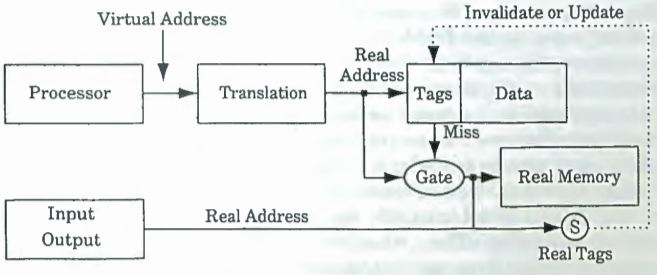


FIGURE 4.14 Real cache—real DMA I/O.

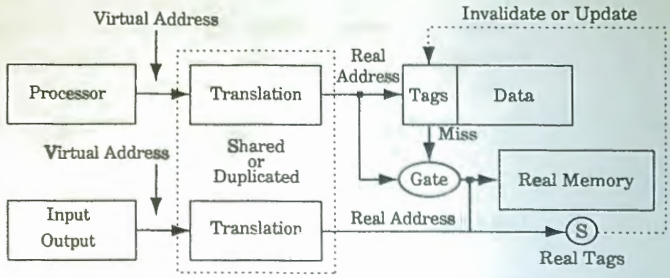


FIGURE 4.15 Real cache—virtual DMA I/O.

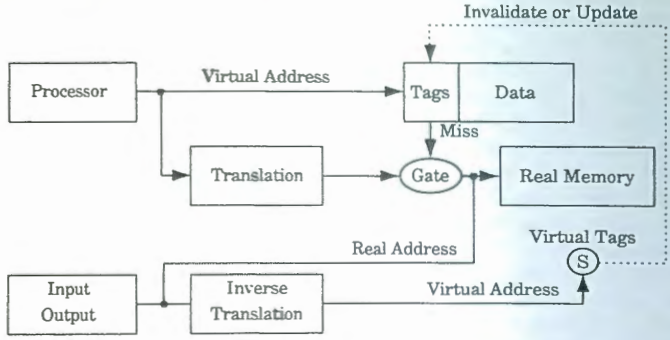


FIGURE 4.16 Virtual cache—real DMA I/O.

a real DMA I/O system presents a major design problem. Figure 4.16 shows this configuration. The cache is addressed with virtual addresses and the DMA I/O processor reads or writes into memory with real addresses that, in themselves, require no translation. A coherency problem, however, can exist if a sector is in the cache from the real address space being used by the DMA I/O. This problem is detected by performing an inverse address translation (real addresses to virtual addresses) that is applied to the tags of the snoopy controller. The cache sector is either invalidated or updated depending upon the detection of a coherency problem. The inverse translation system is usually called an *inverse translator* or a *reverse translation buffer*.

Design Option 4, *virtual cache—virtual DMA I/O*. Figure 4.17 illustrates this design in which both the cache and the DMA I/O processor use virtual addresses. Thus, the DMA I/O processor virtual addresses must be translated into real addresses for accessing memory. The ad-

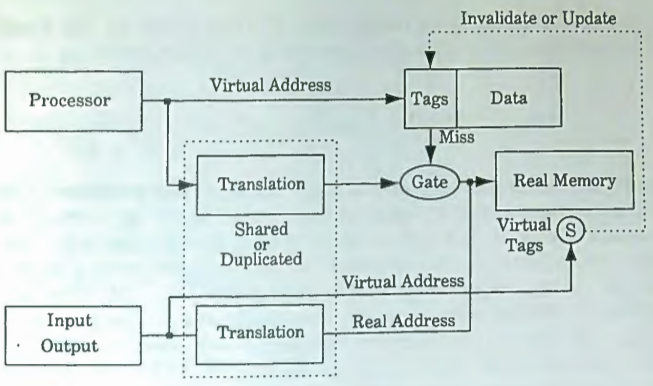


FIGURE 4.17 Virtual cache—virtual DMA I/O.

dress translation can be by a shared or replicated translation facility. Because the address translation system for the processor is used only on cache misses, it seems reasonable to share the translation facility. A shared system should have a minimum performance penalty plus the elimination of translation table coherency problems.

4.2.4 *Processor Control Over Snoopy Controllers*

For a number of reasons, the processor needs to be able to exert control over the snoopy controller. This is particlarly true for designs where the snoopy controller time shares the cache tags between the processor and DMA I/O. As discussed in Chapter 5, DMA I/O data transfer activity can be relatively high, in the range of one to eight bytes per instruction executed. Thus, there is the potential for a cache access conflict every eight or so instructions—a significant performance hit due to locking out the cache from the processor. For these systems, it is desirable to have the capability of inhibiting snooping to improve performance. Furthermore, for systems that have program control over the data cache for write through or write back, the snoopy controller must respond properly for invalidation or update on a snoopy hit, as noted in Table 4.4.

From the above discussion, the basic cache effective access time model from Chapter 2 can be modified, as follows, to comprehend shared

snoopy accesses of the data cache tags. The tag snoop by the processor is accounted for in the normalized access time of the cache as

$$t_{ea} = 1 + P_{miss}T + P_{sn} \times t_{sn}$$

where P_{sn} is the probability of a snoopy cache access concurrent with a normal cache access and t_{sn} denotes the time to snoop the cache, usually one cache cycle or 1. The net effect of shared snoopy cache access is to lengthen the effective cache access time, in the absence of a miss, from 1 cycle to $1 + P_{sn}$ cycles. High DMA I/O activity ($P_{sn} \rightarrow 1$) can result in a significant performance penalty. For cases where snoopy activity is this high, the snoopy tag memory should be replicated or multiported.

For systems with write through caches, all DMA I/O reads from memory can be accomplished without accessing the cache, as shown in Table 4.4. Only DMA I/O writes to memory need to access the cache. This policy has the potential for reducing the need to access the cache by approximately 50% and provides an argument in favor of write-through caches. If the cache, in the example above, is write through, the effective access time would be approximately 1.13 rather than the 1.27 with a write-back cache.

The MC68040 provides an example of a snoopy controller under control of the processor [MOTO89]. Within the status word there are two control bits (SC1, SC0) for the snoopy controller interpreted as shown in Table 4.6.

The MC68040 shares the cache tags with the snoopy controller. Thus there are two access paths and priority must be established for access; snooping has priority over data accesses from the processor. Also, the MC68040 snoopy controller provides functions to support multiprocessors that exceed the requirements of DMA I/O data transfers only. Interested readers should see [MOTO89] for additional details.

The PowerPC 601 shares the on-chip data cache tags with the snooping function. To reduce the cycle stealing from bus snoops, explicit control

Control		Requested Snoopy Operation	
SC1	SC0	Read Access	Write Access
0	0	Inhibit	Inhibit
0	1	Read from cache	Update cache, write mem.
1	0	Read from memory	Invalidate cache, write mem.
1	1	Reserved, inhibit	Reserved, inhibit

TABLE 4.6 MC68040 snoopy control.

must be asserted by the bus. There are two bus signals, TS' (transfer start) and GBL' (Global, and I/O signal), that must be simultaneously asserted to qualify a snoop operation. When a snoop is initiated and there is a hit on the tags, the processor emits signals to the bus that are used by other processors. Motorola [MOTO93] provides details on the behavior of loads for different bus operations and cache sector states. Similar information is provided for stores as well.

4.3 Other Considerations

This section will briefly discuss three other considerations with I/O systems: memory-mapped I/O, the MESI cache coherency protocol, and snooping on write queues. Some of these issues are relevant to multiprocessor systems and are only briefly discussed.

4.3.1 *Memory-Mapped I/O*

A number of processors augment the DMA I/O systems described above with memory-mapped I/O. This form of I/O reserves a region of the processor's address space for I/O control and data registers. Transfers to and from I/O can then be accomplished with the normal load and store instructions of the processor. The advantage of this form of I/O is low latency; the disadvantages are the loss of some address space, the use of program space to explicitly perform I/O, and complications to the bus so that the addresses can be recognized.

With memory-mapped I/O, the virtual addresses created by the processor must be translated into real addresses before they can be placed on the bus. Thus, the TLB will be called upon to hold an entry for I/O. The protection afforded in the TLB is available for I/O as well as normal data movement due to page faults. A significant design consideration is whether or not to cache the address space reserved for I/O. Write-back caches are unacceptable because of the long delay that can occur with an output operation.

The PowerPC 601 uses a bit in its segment registers (discussed in Chapter 3) to identify when an address is to memory or to a reserved space for memory-mapped I/O [MOTO93]. Likewise, the Pentium uses memory-mapped I/O while providing direct I/O instructions.

4.3.2 *Cache Coherency Protocols*

The previous discussion on the response of a cache to a snoopy hit assumes a simple model of cache behavior; the block is either valid or invalid [DUBO88]. This is an appropriate model for a simple write-

through cache in a uniprocessor because a sector can be in only one of two states.

As discussed previously, when the snoopy controller finds that a write to memory by the I/O will invalidate the contents of the cache, the cache sector can be either invalidated or updated depending upon the protocol selected by the designer.

With multiprocessors, the write-through protocol creates bus traffic that can be detrimental to the performance of the system, leading to the use of the write-back protocol. With a multiplicity of caches, the problem of coherency must be addressed, and a number of protocols have been investigated for this purpose. Some of these are: Write-Once protocol with four states, the Synapse protocol with three states, Berkeley protocol with four states, the Illinois protocol with four states, the Firefly protocol with three states, and the Dragon protocol with four states. These protocols are described by Archibald and Baer [ARCH86].

In the early 1980s, a working committee of IEEE started drafting a standard protocol that would be used with the IEEE Futurebus [GALL91] and a write-back protocol. The result of this early work is a five-state protocol called MOESI for Modified, Owned, Exclusive, Shared, or Invalid. Table 4.7 shows the state of the three sector control bits—valid, dirty, and shared—along with the five state names of the MOESI protocol. Note that if the sector is invalid, the clean and shared bits are don't cares.

The five-state protocol would be very expensive to implement for on-chip caches. Motorola's Greiner noted that the owned state could be eliminated if the modified and shared states were made illegal at the same time [GALL91]. This results in the MESI protocol that has been adopted as an IEEE standard and is considerably less costly to implement [IEEE90].

A sequence of events that must be taken for a read or write for each of the four states (eight events) is specified in the standard. The design of a cache consistency protocol is tied closely to the design of the bus

Sector Control Bits				
Valid	Dirty	Shared	MOESI	MESI
Valid	Clean	Shared	Shared	Shared
Valid	Clean	Not shared	Exclusive	Exclusive
Valid	Dirty	Shared	Owned	Not permitted
Valid	Dirty	Not shared	Modified	Modified
Invalid	X	X	Invalid	Invalid

TABLE 4.7 Cache sector coherency control.

because signals must be transmitted over the bus to inform the other processors about what is happening. Thus, there must be signal lines on the bus to permit a cache to signal to other processors a code, indicating the state of the cache sector and the transition that is being made.

The MESI protocol is used with the unified caches of the PowerPC 601 [MOTO93] and the internal data cache and external unified cache of the Pentium [INTE93, INTE93a]. The Pentium on-chip instruction cache implements only the invalid, shared portion of the protocol as the instruction cache is read only and does not have to contend with data.

4.3.3 *Snooping on Write Queues*

As discussed in Chapter 2, processor writes update the cache before the block is written back to memory over the bus. This action presents the potential for a coherency problem, and the PowerPC 601 is an example of how coherency is maintained [MOTO93]. The write queue (buffer) holds a dirty block (eight words or 32 bytes) that is being written out to memory as a result of a cache miss. Each entry of the write queue carries the virtual address for the write along with the block of data. A bus action from an I/O operation that is to be snooped on the processor is presented as a key to the addresses that are associatively searched. If there is a hit on a write queue element, the block is first loaded into memory before it is used as an update to the cache of the snooping master.

If the source of the snoop signal is a DMA I/O read operation, the transfer to memory is accomplished before the read. If the DMA I/O operation is a write, the DMA I/O write to memory is completed and the contents of the write queue are erased.

Configuration	DMA I/O via System Bus
Real cache—real DMA I/O	VAX*, IBM 370
Real cache—virtual DMA I/O	
Pipelined real cache—real DMA I/O	i386, i486, Pentium
Pipelined real cache—virtual DMA I/O	IBM RS/6000, PowerPC 601
Virtual cache—real DMA I/O	i860*
Virtual cache—virtual DMA I/O	SPARC II
Restricted virtual cache—real DMA I/O	
Restricted virtual cache—virtual DMA I/O	MC68030, MC68040

* Not snooped, software controls coherency.

TABLE 4.8 Example cache—DMA I/O configurations.

4.3.4 *Summary*

Consider the four methods of cache access, discussed previously, combined with the two variations of DMA I/O system addressing, real and virtual, giving eight possible system configurations. Table 4.8 shows these configurations and identifies known computers for each.

Six of the eight organizations are represented by actual implementations. I can only speculate on the reason for the omission of the real cache—virtual DMA I/O and restricted virtual cache—real DMA I/O.

5

Interleaved Memory and Disk Systems

5.0 Overview

This chapter discusses two important subjects, interleaved memory and disk systems, that are usually given only minor attention. Interleaved memory, which is used to provide high bandwidth, was of great importance prior to the advent of practical high-speed memories for caches. Currently, innovative architectures of modern DRAMS and various forms of interleaving are receiving design and research attention. Disk technology has had a remarkable life since the first commercial disks in 1956, the IBM 350 RAMAC. Disks have survived all predictions that the technology could go no further; indeed, cost and performance of disks have continued to improve.

5.1 Interleaved Memory

Interleaved memories provide high-memory bandwidth by distributing the read or write accesses over a number of memory modules, which is a form of parallelism. The array disk to be described in Section 5.2.3 is another instance of memory interleaving. Early research into interleaving was concurrent with and a "back up" to hierarchical memory research. In the late 1950s and 1960s no one was sure if cache technology development would be successful.

The first known reference to interleaving is found in a description of the IBM Stretch [BLOC59]: "The memories themselves are interleaved so that the first two memories have their address distributed modulo 2 and the other four are interleaved modulo 4."

A later reference to the IBM S/360 [FAGG64] points out that: "When

accesses are made to sequential storage addresses, the storage units operate in an interleaved fashion."

Pirtle [PIRT67] observes: "... the addresses are distributed over an n module memory so that address 0 is in $m(0)$, address 1 is in $m(1)$ and, in general, address X is in $m(i)$ where $i \equiv X$ modulo n."

We can see from these statements that there are two uses of the word *interleaved*. One use concerns the *units* or *modules*. Interleaved modules subdivide the module cycle time into clock periods that establish the streaming rate of the memory system. The degree of module interleaving is m and is formally defined as the number of requests that can be issued by the processor in one memory module cycle [BURN70]. Note that the clock period of the memory and the bus system are related as $m = t_m/t_c$ where t_m denotes the memory cycle time and t_c the bus or processor clock time. The other use of the term interleaving concerns *addresses*. Interleaving addresses across modules establishes the sequence of module access for a stream of requests. The degree of address interleaving is n, which is defined as the number of memory modules over which the addresses are interleaved [BURN70].

In this chapter, memory systems are described for $1 < m \leq n$. With many systems, the two degrees of interleaving are equal; that is, $m = n$. However, with some systems, such as those found with supercomputers, the degree of address interleaving is greater than the degree of module interleaving; that is, $m < n$, which is called *superinterleaving* and is discussed in Section 5.1.3.

Interleaving presents two questions of interest to researchers. First, can interleaved memories provide enhanced bandwidth to supply instruction and data streams to a processor? And second, what are the design parameters of an interleaved memory that supports vector reads and writes of supercomputers? With the success of caches, the first research objective has almost disappeared as an interesting topic. The second research topic is still an area of active research and is discussed in more detail in Section 5.1.3. Note that the IBM RT uses a noncache interleaved memory system [ROWL86] to provided instruction and data bandwidth.

The requirement for accessing vectors comes from two sources. The first source is found in the main memory references of supercomputers. Scientific programs frequently access arrays of data stored in memory as vectors. The second source is found in the interface between a cache and the main memory. A block of AUs are transferred between these two memories; this block is a vector of length 4 to 16 and is discussed in Chapter 2.

Interleaved memories are ideal for providing high bandwidth access to a long vector of data (either read or write). An interleaved memory requires two addresses: the address of the module and the address of

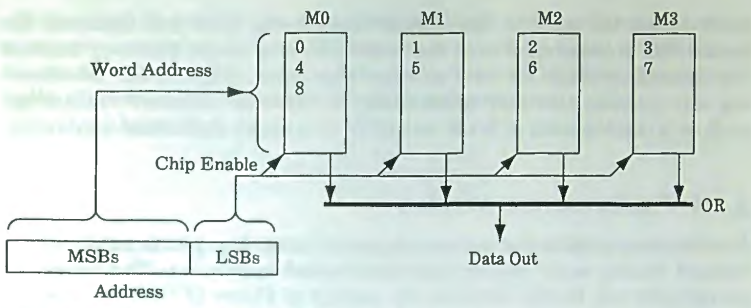


FIGURE 5.1 Four-way interleaved memory.

the word within the module. Low-order interleaving generates these two addresses by dividing the memory address by the address interleaving degree n: the quotient is the word address presented to each module and the reminder selects the module. This division by an integer power of two is a trivial operation for designs having a degree of interleaving that is an integer power of two.

Consider a memory of total size 2^s, with $m = n = 4$, as shown in Figure 5.1. The four modules are addressed by the two LSBs of the address, and the word within the addressed module is selected by the high-order bits. This process is called *low-order interleaving*.

Addressable units are stored in the modules according to the sequence

Module	Addresses
0	$0, 4, \ldots, 2^s - m + 0$
1	$1, 5, \ldots, 2^s - m + 1$
2	$2, 6, \ldots, 2^s - m + 2$
3	$3, 7, \ldots, 2^s - m + 3$

A major disadvantage of low-order interleaving is the difficulty of expanding the installed memory. If the installed memory is to be expanded by a factor of 2, for example, how is this to be done? Should the degree of interleaving be increased by 2, or should the module size be increased by a factor of 2? Increasing the interleaving factor requires a major hardware modification. Increasing the size of each of the modules is a rather straightforward task. However, both expansion methods become completely intractable if the expansion is something other than an integer power of 2.

Another form of interleaving is called *high-order interleaving*, in which the quotient selects the module and the remainder selects the

word within the module. High-order interleaving does not enhance the bandwidth of the system over the bandwidth of a single memory module but instead provides for easy memory expansion. High-order interleaving also permits the easy intermixing of different memory technology such as a combination of RAM and ROM in a small dedicated controller.

5.1.1 *Performance Models*

Performance models for various types of accessing patterns were developed during early research on interleaved memories. The following paragraphs will briefly describe the models of Flores [FLOR64], Hellerman [HELL67], and Burnett and Coffman [BURN70]. These models were created to assist designers evaluate the performance impact of various interleaved memories with different workloads.

There are three useful metrics for measuring and evaluating the performance of an interleaved memory. These are

1. Speedup, S = ratio of the bandwidth of an interleaved memory to the bandwidth of a single module.
2. Acceptance ratio, AR = steady-state ratio of accepted memory requests to total memory requests [CHEU86], $0 \leq AR \leq 1$. AR is an indicator of how effectively the memory design sustainable bandwidth is being used.
3. Mean acceptance ratio, MAR = acceptance ratio that prorates the latency of accessing the first module over all of the AUs accessed [CHEU86]; $0 \leq MAR \leq 1$.

Instruction and Data Streams

The models for estimating the performance of an interleaved memory are of three forms: (1) instruction references, (2) data references, and (3) combined references. The addressing patterns of these three streams are quite different and require different performance models. Note that the problems or conflicts associated with interleaved memory systems are special cases of *structural dependencies* (discussed in Chapter 8).

Flores [FLOR64] modeled the situation where a processor and I/O channels are contending for access to an interleaved memory. He modeled the time that a request must wait, waiting time, as a function of the degree of interleaving, the ratio of I/O time to processor time, and the fraction of time that the memory is busy. For an equal ratio of I/O time and as the memory busy time approaches one, the model gives the expected result that the speedup (ratio of waiting time of an interleaved memory to a single module memory) is equal to m, the number of interleaved modules.

The next published model is by Hellerman [HELL67]. This model

can be used to predict the performance of an interleaved memory when operands are being fetched. Hellerman assumed an equal probability of access to each module for the first fetch, followed by fetches to sequentially addressed modules until the Kth reference, as

$$S = \sum_{K=1}^{m} \frac{(m-1)!\,K^2}{(m-k)!\,m^K}$$

where K denotes the length of a sequence of reads ending in a nonsequential access. With an approximate solution

$$S = m^{0.56} \approx \sqrt{m} \quad \text{for } 1 \le m \le 45.$$

The implication of Hellerman's model is that if four memory modules are interleaved, the effective bandwidth is approximately twice the bandwidth of one module for the data streams of a typical general purpose computer.

Coffman and Burnett [COFF68] developed performance models for instruction, data, and combined streams. Their performance model for interleaved memories supplying an instruction stream is

$$\text{instruction speedup} = \frac{1-(1-\lambda)^m}{\lambda}$$

where λ, the probability that an instruction is a taken branch, is equal to $1/K$. For example, consider a program stream with $\lambda = 0.2$ and $m = 8$. The speedup over a single memory module is 4.16, while Hellerman's model suggests a speedup of 2.82 for a random data reference stream, which is not a surprising result. The Hellerman model should have poorer spatial locality than an instruction stream, hence a smaller speedup.

A data stream model is developed by Burnett and Coffman [BURN70] and described in [BAER80]. This model, called the α–β model, estimates speedup for a data request stream. The assumption is made that the first module is selected at random and that for each subsequent request there is a stationary probability $1-\lambda$ that the next module in sequence is accessed on the next request. Any one of the other modules is accessed with equal probability $\beta = (1-\lambda)/(m-1)$. This model, without a closed form solution, is solved numerically. This solution convincingly shows the advantage memories where $n > m$.

Models such as the α–β model are no longer useful to designers because most computers buffer instructions, and the data references are

for vectors with a stride of one. Models that have stride as a parameter are discussed in the following sections.

Vector Data Streams

Interleaved memory has been recognized as an ideal organization for providing high bandwidth for long vectors of reads or writes. Vector access of memory has a characterization parameter, *stride*, that is not present with the models above.

Stride is the difference between addresses of successive references. The addresses, starting at 0, for various strides are

Stride = 1 0,1,2,3, . . . ,
Stride = 2 0,2,4,6, . . . ,
Stride = 3 0,3,6,9, ...

A basic performance model assumes that a vector of consecutive AUs (stride = 1) is fetched from an interleaved memory. For a stride of one (illustrated in Fig. 2.12), the time per AU is equal to

$$t_{AU} = \frac{t_m}{t_c}\left(1 + \frac{m-1}{K}\right).$$

The speedup of the interleaved memory is

$$\text{speedup} = S = \frac{m}{1 + (m-1)/K}.$$

The time per AU model assumes that the first reference is to a module that is nonbusy. Subsequent references are to non-busy modules as guaranteed by the degree of interleaving and stride. Table 5.1 shows the performance limits of vector data streams on an interleaved memory over a single module memory.

From the speedup limit we can see that the speedup of an in-

	Vector Length K	
	1	∞
t_{AU}	t_m	t_m/m
Speedup	1	m

TABLE 5.1 Vector performance limits.

terleaved memory accessing a vector that has a stride of 1 is in the range $1 \leq S \leq m$.

The acceptance ratio and mean acceptance ratio indicate how well the memory system supports the demands placed upon it. For $AR = 1$, all requests are being serviced without delay. For $MAR = 1$, the vector length must be very long, $K \rightarrow \infty$, or the effective startup overhead must be reduced by concurrency.

5.1.2 *Reducing the Effect of Strides ≠ 1*

The models presented above show, for strides of 1, the influence that the degree of interleaving has on performance. As the models will show, strides other than 1 reduce the performance of an interleaved memory. For this reason, there is a body of research directed to decoupling the effect of stride and leaving only the degree of interleaving in the performance models. Most of the reported research was performed in the 1960s; however, there is renewed interest in this line of research today.

When low-order interleaved memories were applied to pipelined and array computers, performance problems were discovered. For example, if a matrix is stored by row in an interleaved memory and is read by column, the read *AR* may not be the same as the write *AR*. Instead of the addresses smoothly progressing from module to module as with the writes, there can be conflicts in the modules when the array is read. Figure 5.2 illustrates an example of this problem. In this example, a 4×4 matrix is stored by column in a 4-way interleaved memory. That is, successive AUs of each column are placed in different memory modules with a stride of 1. If the matrix is then read by row, each AU of a row is read from the same module with a stride of 4 resulting in $S = 1$ and $AR = 0.25$. The first row (1,1 1,2 1,3 and 1,4) is contained within memory module 0 and can be read no faster than the cycle time of one module.

The reduction in potential speedup when accessing this matrix can be generally attributed to conflicts resulting from the interactions of stride values and degree of interleaving. Reducing conflicts with interleaved memories has been a research topic for over two decades

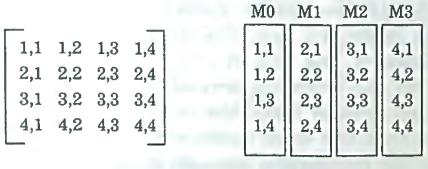


FIGURE 5.2 Matrix stored in interleaved memory.

Stride	S	AR
1	4	1
2	2	0.5
3	4	1
4	1	0.25
5	4	1

TABLE 5.2 Speedup and AR for various strides, $m = 4$.

[BUDN71, HARP91, RAU91]. Access conflicts can result in a significant performance decrease with vector processors when, for example, the output of one vector operation is stored by row and the next operation needs to read the matrix by column.

Conflictfree access is possible only if the stride is not a factor or a multiple of the degree of interleaving, except for a stride of 1. In the example above, the stride is 4 when reading by column and the degree of interleaving is also 4. A stride of 3 would be accessed with no degradation. Table 5.2 shows the acceptance ratio of a 4-way interleaved memory for various strides.

For strides of 1, 3, and 5, $AR = 1$ while for a stride of 4, $AR = 0.25$ because all of the references are to the same module. Several approaches have been proposed and used to solve the stride-induced conflict problem for interleaved memories having $m = n$. These approaches are discussed below.

1. Skewed addressing;
2. Dynamic skewed addressing;
3. Pseudorandom skewed addressing;
4. Prime number interleaving;
5. *Superinterleaving*, $m < n$ (discussed in Section 5.1.3).

Skewed Addressing

Skewed addressing maps an address into the modules in such a way as to reduce the conflicts [BUDN71]. An example of skewed storage is shown in Figure 5.3, which shows the 4×4 matrix mapped into skewed storage. If the matrix is stored by row, the AUs of row 1 are stored in M0, M1, M2, M3. However, the second row is skewed one memory module, beginning with M1 and wrapping around to M0. The second row is stored with an additional skew of 1 and the third with an additional skew of 1 and so on. Note that the word addresses from the high-order bits are unchanged from the nonskewed case. However, the low-order bits of the address must be mapped into the module selection bits.

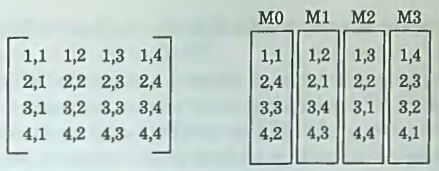


FIGURE 5.3 Skewed address interleaving.

The full speedup, $S = m$ and $AR = 1$, can now be obtained with either row or column accessing. Observe that there are strides that will interfere with the degree of interleaving and reduce S. For example, if one of the diagonals of the matrix is read, the AUs 2,2 and 3,3 are both found in M0 while 2,2 and 4,4 are both in M2, which results in $S = 2$ and $AR = 0.5$. The other diagonal finds all AUs in M3, resulting in $S = 1$ and $AR = 0.25$. Skewed storage requires that a skew factor be built into the hardware and used by all accesses. This poses a design problem for selecting the skew factor that will give the best performance over the anticipated workload.

Table 5.3 shows the addresses that needed to read the third column of the matrix with both normal and skewed (skew = 1) interleaving. The word address is the high-order bits, while the module address is the low-order bits. To read a column, the stride is four; for normal interleaving, the module address is unchanged as the word address is incremented. With skewed addressing, the word address increases as with normal addressing while the module address is formed by adding the skew factor to the current module address modulo the degree of interleaving. That is, add the skew factor to the module address but do not propagate a carry into the word address. The normal interleaved case works the

Interleaving Method			
Normal		Skewed	
Word	Module	Word	Module
00	10	00	10
01	10	01	11
10	10	10	00
11	10	11	01

TABLE 5.3 Skewed address interleaving.

same way: the skew of zero is added to the module address, keeping the module address unchanged.

An interesting application of skewed storage is found in [MATI 89]. This study describes the design of a cache that uses multiple chips, requiring skewed storage. With this design, reading words from the cache into the processor and loading a block from memory into the cache will not have stride conflicts.

Dynamic Skewed Addressing

Unlike the skew scheme discussed above, dynamic skewed addressing selects a skew for the anticipated access pattern of the data that is currently active [HARP91]. This technique assumes that the compiler can determine the skew that is added to the remainder of the address. The addition is done dynamically and can be implemented for a relatively low hardware cost.

In order for dynamic skewed storage to work properly, there must be some way to bind the skew value that is used for a write to a subsequent read of the same data. In other words, a read must have access to the corresponding write skew.

Pseudorandom Skewed Addressing

Rau [RAU91] points out that any scheme for reducing conflicts that depends upon a fixed skew mapping of any type is suspect because there are some address patterns that will conflict. As an alternative, he describes and evaluates pseudorandom skewing of addresses.

Two schemes for providing pseudorandom skewing are discussed. First, the module address is XORd with a key generated with a pseudorandom generator; and second, the module address is XORd with a polynomial whose coefficients are in the Galois Field. As with dynamic skewed addressing, the skew factor must be stored by a write and be available for future read operations.

Prime Number Interleaving

Recall that conflicts will occur if, and only if, the stride is a factor or a multiple of the degree of interleaving. Thus, by selecting m to be prime, the number of strides that can produce conflicts is significantly reduced. For example, if $m = 5$, then $AR = 1$ for all strides except 5, 10, 15,.... For many problems the stride is an even power of two, thus these strides can be process conflictfree.

The first known proposal for prime number interleaving in the literature is found in [BUDN71]. I believe that prime number interleaving was used on some early drum computers to assist in minimum latency

scheduling. The Burroughs Scientific Processor used prime number interleaving with $m = 17$ [LAWR82].

Yang [YANG93] describes a cache for vector processors with a prime number of sets rather than an even power of two. The cache is late select with the sets interleaved to achieve a high streaming rate. With prime sets, the addressing patterns of various strides is broken up, permitting the cache to stream without interference. Recall from Chapter 2 that a variation of the SPARC has a prime number of sectors [WEIS92].

The problem with prime number interleaving is that the address, as with all interleaved memory systems, must be divided by the prime number degree of interleaving. The quotient is the high-order word address into the modules while the remainder is the module select address. Division by an even power of two is much simpler than division by a prime number. However, it is possible with available VLSI technology to perform the division, so prime number interleaving may become a viable alternative. The Yang design is based on interleaving with a Mersenne prime number of the set $(3, 7, 15, \ldots, (2^c - 1))$, thus facilitating the generation of the cache address.

5.1.3 *Superinterleaving*

Another technique for reducing the effect of stride on speedup and AR of an interleaved memory is to provide more memory modules than the address degree of interleaving, that is, $m < n$; this method is called *superinterleaving*. Recall that for any m, the clock rate of the system is the memory module cycle time divided by m. By implementing a memory with $m < n$, the number of strides that will produce a conflict in a module is reduced.

Computers such as the CDC6600 have addresses interleaved over more memory modules than the module degree of interleaving; $m < n$. This technique is also used in the Cray-1 and other computers of the Cray family. For the CDC 6600, the memory module cycle time is 1,600 ns, the system clock is 100 ns, and there are 32 memory modules giving $m = 16$ and $n = 32$ [THOR70]. The Cray-1 has 16 50 ns modules with $m = 4$, $n = 16$, and a clock period of 12.5 ns.

Figure 5.4 illustrates, by example, the benefit of superinterleaving. For this example, the addresses are interleaved eight ways ($n = 8$) while the modules are interleaved four ways ($m = 4$). Recall that m is set by the clock period of the processor and the cycle time of the memory modules. Eight cases are shown as the stride is varied from 1 to 8. A form of reservation table is used to illustrate the operation of this memory; reservation tables will be described in greater detail in Chapter 6. A

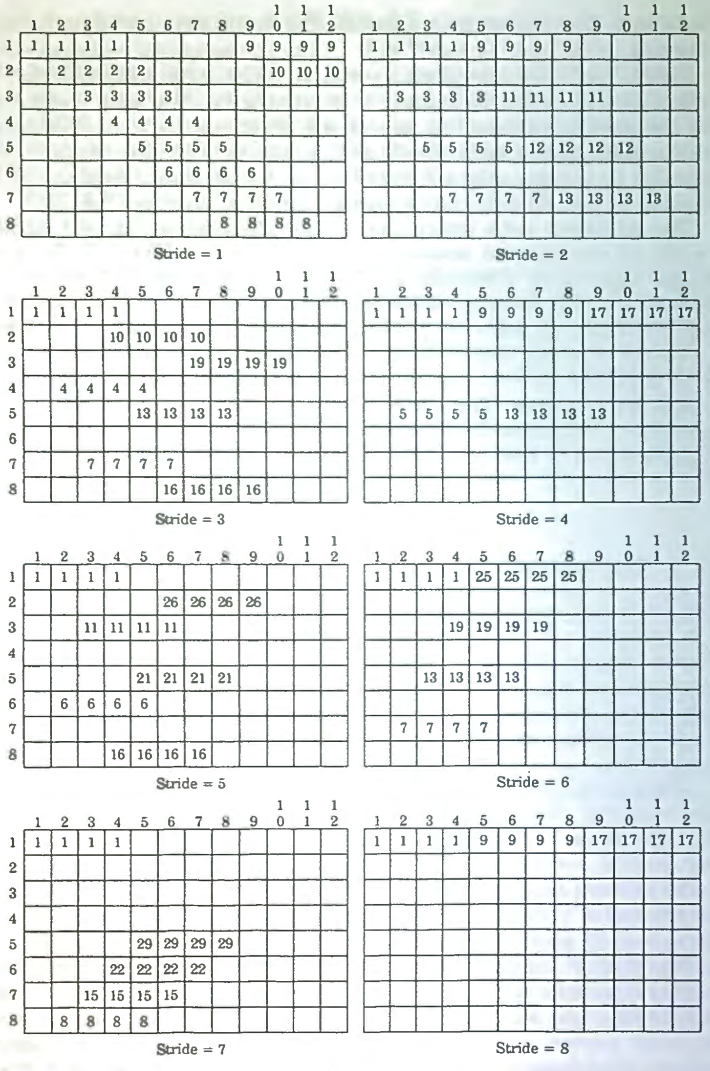


FIGURE 5.4 Superinterleaving example, $m = 4$, $n = 8$.

Stride	S	AR	Memory Utilization
1	4	1	0.5
2	4	1	0.5
3	4	1	0.5
4	2	0.5	0.25
5	4	1	0.5
6	4	1	0.5
7	4	1	0.5
8	1	0.25	0.125

TABLE 5.4 Superinterleaving performance, $m = 4$, $n = 8$.

reservation table is a display of space (the memory modules) on the vertical axis and time (in clocks periods) shown on the horizontal axis.

For the first case with a stride of one, the addresses are applied to modules 1, 2, 3, . . . , 8, 9, 10, ... Each module is busy for four clocks, indicated by the address number in the table. Because $m = 4$, each module is busy for four clocks when it is cycled. For this case, $AR = 1$.

Considers a stride of 3. The modules addressed are 1, 4, 7, . . . , 2, 5, ... The memory can sustain one reference per clock, $AR = 1$, after the initial 3 clocks of latency. Note that $AR = 1$ is obtained because each of the modules is used half of the time. In other words, there is a 2X overdesign in bandwidth that is devoted to ensuring that $AR = 1$ is sustainable.

Drawing from the information presented in Figure 5.4, Table 5.4 tabulates S, AR, and memory utilization for the different strides. For strides except four and eight, this memory has $S = 1$ and $AR = 1$. Interference resulting from a stride of four reduces to $S = 2$ and $AR = 0.5$, while a stride of eight will have $S = 4$ and $AR = 0.25$—a performance that is no better than a single module because all of the references are to one memory module.

The reservation tables clearly show how excess latent bandwidth is used to improve the performance for different strides. Except for the cases where the stride is a factor or multiple of n; the utilization of the memory modules is

$$\text{utilization} = \frac{\text{used bandwidth}}{\text{available bandwidth}} = \frac{AR \times m}{n}.$$

With this example of superinterleaving, if the strides are uniformly distributed, the weighted average AR is 0.84 as compared to the $m = n = 4$ case shown in Table 5.1 with a weighted average AR of 0.68.

Processor	Memory Words per Clock	Bus Words per Clock	Memory Utilization; $AR = 1$
Cray-1	16/4 = 4	1	0.25
Cray X-MP	32/4 = 8	3	0.375

TABLE 5.5 Cray memory system designs.

The Cray-1 memory is superinterleaved and has one port between the memory and its internal registers. The memory is organized $m = 4$, $n = 16$. This means that approximately 75% (12/16) of the latent bandwidth of the memory is wasted to reduce stride conflicts. The Cray X-MP extends the address interleaving to $n = 32$, which provides eightfold bandwidth over the capacity of one port [CHEN84]. With this overcapacity, three ports (two input and one output) to/from the internal registers can sustain continuous transfers for many strides. However, the Cray X-MP memory must support multiprocessors plus an I/O port per processor that reduces the bandwidth available for any one processor. The memory of these two processors is described in Table 5.5.

The memory words per clock indicates the maximum bandwidth of the memory. Bus words per clock represents the number of buses, and the maximum memory utilization is for $AR = 1$ and includes the effect of the number of memory ports.

Memories with one port to the processor and a relatively small interleaving factor, as is the case of the Cray-1, are relatively simple to implement. When the address interleaving is increased along with an increase in the number of ports to memory, the complexity of the interleaving schemes becomes costly to implement. The memory system of the TI ASC, for example, had eight memory modules ($m = n = 8$) and eight user ports, requiring two racks of equipment for the crossbar switch [CRAG89] (which is a very expensive system component).

Two of the major changes from the Cray-1 to the Cray X-MP are an increase in the number of memory modules from 16 to 32, and an increase in the number of ports from one to four (two read, one write, one I/O). The memory module cycle time is 36 ns, and with $n = 32$ the potential streaming rate out of the memory system is one word every 1.125 ns. Given the complexity of a full 32×4 crossbar switch, some hierarchical interconnection or multiplexing system is called for, as shown in Figure 5.5.

The 32 memory modules are grouped into four sections of eight modules each. The modules of a section pass data via eight *lines* (8 to 1 multiplexer-demultiplexer) that are then selected by a 4×4 crossbar switch. With this interconnection scheme, the potential bandwidth of the

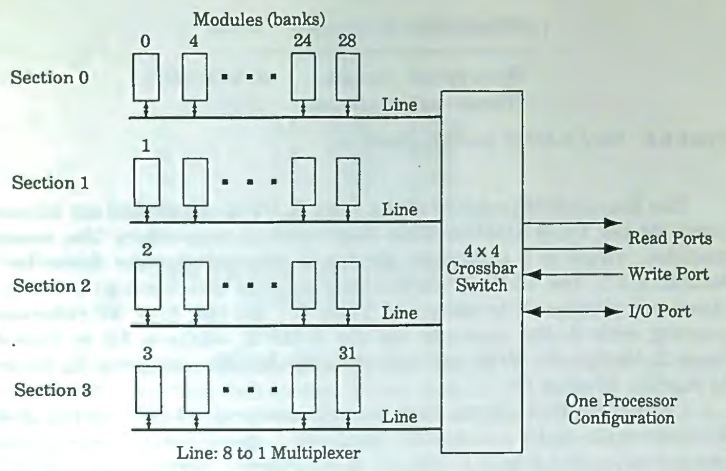


FIGURE 5.5 Cray X-MP-2, one processor, memory organization.

memory is reduced from 32 possible memory references per clock to 4 memory references per clock because of the constrictions imposed by the four lines. With four ports active, there is no excess bandwidth to mitigate the effects of conflicts. Note that for each processor in the system the four lines and the crossbar switch are replicated.

The crossbar switch of Figure 5.5 is assumed to be conflictfree for all accesses from all ports. However, conflicts exist with the Cray X-MP-2 memory in addition to the stride conflicts of the Cray-1. The conflicts are:

1. Module (bank) conflict due to access from one port—a *stride conflict*;
2. Module (bank) conflict due to access of an idle module by two or more ports—a *simultaneous bank conflict*;
3. Line conflict due to access by two or more ports—a *section conflict*.

The effects of these three types of conflicts have been extensively studied [CHEU84, CHEU86, OED85, WEIS92, CALA88, CALA88a]. Cheung and Smith [CHEU84] used reservation table analysis of memory conflicts to evaluate *MAR* values; excerpted results are shown in Table 5.6. Their reservation tables include memory banks and lines as resources. In all of the cases evaluated, stride is set to 1. Their results indicate that *MAR*s of 0.8 or better are achievable with the Cray X-MP-2 memory.

Conditions	*MAR*
Two-vector stream	0.975
Three-vector stream	0.80

TABLE 5.6 Cray X-MP-2 memory conflicts.

The line conflict problem of the Cray X-MP-2 suggested an improvement for the Cray X-MP-4. This improvement *renumbers* the memory modules, which is a technique similar to skewed storage described in Section 5.1.2. The address interleaving (module numbering) scheme for these two processors is shown in Table 5.7 for the first 32 references, starting with 0. For example, for the X-MP-2, address 13 is found in Bank 3, Module S1. With renumbering, the X-MP-4 address 13 is found in Bank 1, Module S3.

Cheu [CHEU86] shows that the renumbering scheme of the X-MP-4 improves the *MAR* over that of the X-MP-2. Specifically, this reassignment changes *MAR* from 0.929 to 1.0, an improvement of approximately 7%.

Note that the number of interleaved addresses per line, called

Cray X-MP-2

	Bank							
	0	1	2	3	4	5	6	7
Section 0	0	4	8	12	16	20	24	28
Section 1	1	5	9	13	17	21	25	29
Section 2	2	6	10	14	18	22	26	30
Section 3	3	7	11	15	19	23	27	31

Cray X-MP-4

	Bank							
	0	1	2	3	4	5	6	7
Section 0	0	1	2	3	16	17	18	19
Section 1	4	5	6	7	20	21	22	23
Section 2	8	9	10	11	24	25	26	27
Section 3	12	13	14	15	28	29	30	31

TABLE 5.7 Cray X-MP-2 and X-MP-4 address interleaving.

	Number of Sections (*NS*)			
NBPS	2	4	8	16
1	0.5	0.25	0.125	
2	1.5	0.75	0.38	
4	3.5	1.75	0.88	
8	7.5	3.75	1.88	0.94
16	15.5	7.75	3.87	1.93

TABLE 5.8 Cray X-MP memory latency (clocks).

number of banks per section (*NBPS*), can be 1 (X-MP-2), 2, 4 (as shown above), or 8. Another parameter is the *number of sections* (*NS*). The selection of *NBPS* for a given *NS* can have a pronounced effect on the performance of the memory system. However, improving *MAR* by module renumbering works at the expense of increasing the latency (sometimes called *startup delays*) while transient conflicts are resolved. Simulation results on the latency for various *NBPS* and *NS* from [CALA88] are given in Table 5.8 for a two-port system.

The change in interleaving for the Cray X-MP-4 results in an additional 1.25 clocks of latency. Similar results for a three-port access show that the Cray X-MP-2 has a delay of 0.67 clocks and that the Cray X-MP-4 has a delay of 5.11 clocks. Latency is reduced as the number of sections increases. This is not a surprising result because if the number of sections is equal to the number of modules, then a full crossbar switch is specified. For one section, all memory modules are connected by one line or a multiplexer offering the maximum potential for conflict from references from another port.

The memory designer must face the issue of balancing *MAR* and latency. As discussed in Chapter 11, for short vectors latency should be minimized while for long vectors *MAR* should be minimized. Additional recent research on the issue of high-performance interleaved memory for vector processors is found in [CHEN91] and [HARP91].

An interesting comparison of the memory configurations for various members of the Cray X-MP family is shown in Table 5.9.

The data in this table is extracted from [THOM86] and shows the number of memory modules in **bold** and the number of modules per port in *italics*. For example, a four-processor configuration with 8 Mwords of memory has 32 memory modules. The number of modules/port is based on three ports per processor; the I/O port is ignored. For example, the four-processor system with 64 memory modules has 12 ports and 5.33 modules per port.

	Memory Modules No. of Processors			Modules/Port No. of Processors		
Memory Size	1	2	4	1	2	4
16 Mwords		**32**	**64**		*5.33*	*5.33*
8 Mwords	**32**	**32**	**32**	*10.6*	*5.33*	*2.66*
4 Mwords	**16**	**16**		*4.33*	*2.66*	
2 Mwords	**16**	**16**		*4.33*	*2.66*	
1 Mwords	**16**			*4.33*		

TABLE 5.9 Cray X-MP memory modules and modules/port.

With $n = 4$, the observation can be made that some models of this processor do not have enough memory modules to support the number of ports. Any module/port value less than 4 is marginal in memory bandwidth and may suffer performance degradation for some benchmarks. The X-MP-4 with 8 Mwords of memory can be a marginal performer.

5.1.4 *Interleaving for Multiport Access*

Memory module interleaving is also used to permit concurrent access to a memory when two or more addresses are not to the same module. This capability is a feature of the Cray memories described in Section 5.1.3. However, a simpler system is used in the Intel Pentium for its data cache [ALPE93]. This processor is a superscalar implementation (discussed in Chapter 10) that has the need to perform two cache transactions in one clock period. This memory system is discussed in this chapter because the technique is applicable to multi-access memories in general, not just to caches for superscalar processors.

The basic idea is that with addresses interleaved across m modules ($m = n$), accesses to adjacent address will not conflict and can be processed in parallel. Only when there is a module conflict will one of the requests be deferred. A simplified block diagram of the Pentium data cache is shown in Figure 5.6. This cache has 8 modules, $m = n = 8$, with each module organized 256×4 bytes. The total cache capacity is 8 Kbytes.

Two addresses can be presented at the same time to the cache. If the addresses are not the same, the two addressed memory banks are cycled and the two AUs are gated out through the multiplexers. If there is a conflict, the U access has priority and the V access stalls for one cycle. The designers of the Pentium considered a dual-port cache as well

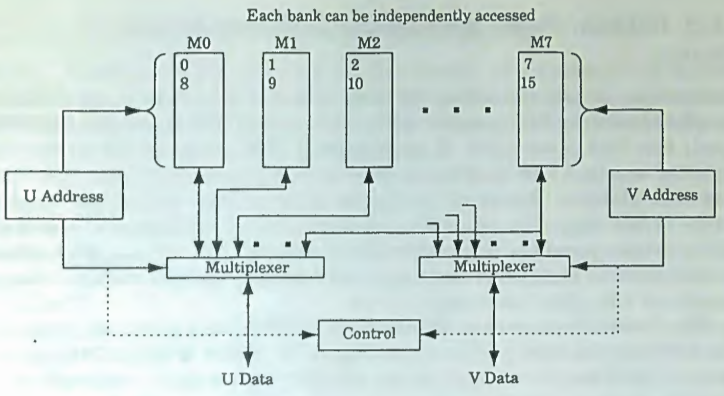


FIGURE 5.6 Pentium interleaved data cache.

as this interleaved cache and selected this design based on its smaller area and reduced complexity for handling data dependencies. This cache is discussed further in Section 10.2.2.

5.1.5 *Cache-Memory Transfers*

The use of interleaved memory as a technique for reducing the transport time of a block to/from a cache is discussed in Section 2.2.1 and Figure 2.13. Because a block is a vector with a stride of one, an efficient transfer is achieved. For example, the IBM S/360/85 has a 4-way interleaved core memory, each module having a word length of 16 bytes and cycle time of 1.04 μs. The processor and cache have a cycle of 80 ns. The interleaving is specifically designed to transfer a 64-byte block. After the first module cycle is initiated, the second module read starts 80 ns later, then the third, then the fourth. The first 16 bytes of the block are transferred at the end of the access portion of the cycle, and each of the other 16 bytes comes 80 ns later. For this design, $a = 1.04$ ns and $b = 80$ ns. Note that the parameters a and b are defined in Chapter 2 in terms of clocks, not time.

The performance effect of block size and interleaving is modeled in [SMIT87]. The length of the block can have a critical impact on the cache performance if the cache is blocked during a block load. However, if one of the methods of load forward or fetch bypass is used, this performance penalty can be reduced.

5.1.6 *Nibble, Page, and Static Column Mode Drams*

This section briefly describes memory chips that achieve an efficient transfer similar to that possible with interleaving. Nibble-mode DRAMS supply four bits in one cycle. A page-mode DRAM accesses the array and holds the accessed row in a buffer that is then accessed by the row (low-order bits) address. The size of the buffer is the square root of the number of bits in the chip. For example, a 1 M-bit device will have a 1024-bit buffer, which provides a possible block size of 128 bytes. The static column mode is similar to the page-mode DRAM and provides a faster transfer of bits after accessing the row.

The Texas Instruments TMS44C256 DRAM serves as an example of a nibble-mode device, shown in Figure 5.7. The TMS44C256 is organized 256 Kbits × 4 bits (1 M-bit DRAM). Internally it consists of 4, 256-Kbit arrays as shown in Figure 5.6. This device is supplied in three speed ranges; this example uses the highest speed one.

A read cycle is initiated with the presentation of the address (18 bits), and each of the four arrays are read. As shown in the cut-out, each array is organized 512 × 512 bits. Nine bits select a word in each array, and the output bit is selected by the other nine bits. The read cycle places 512 bits in each of four buffers for a total of 2 Kbits. These bits are gated into the output, either as a 4-bit nibble or, if in the page mode, up to 512 nibbles in sequence.

The performance model for the nibble, page, and static column devices is the same as that of interleaved memory. That is, there is a time a required to fetch the first AU and then a time b to fetch each of the successive AUs. The total transport time increases monotonically with the block size unless another fetch is required to start the cycle over again. For the TMS44C256, the read and write cycle time is 150 ns and the read or write page mode is 50 ns. The parameters for computing the transport time are $a = 150$ ns and $b = 50$ ns.

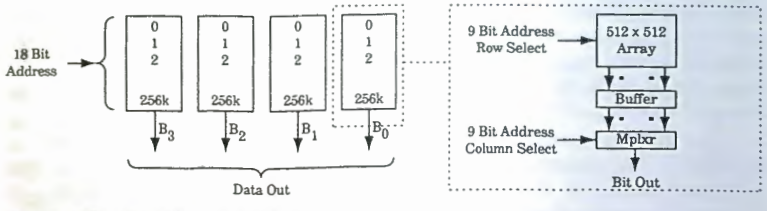


FIGURE 5.7 DRAM organization.

5.1.7 *Wide-Word Interleaved Modules*

Another method for increasing the bandwidth of a memory is to increase the width, in bits, of a memory module to a multiple of the addressable unit (2X, 4X, . . .). In addition, a number of these modules can be module interleaved for even greater bandwidth.

Consider a memory module with a width of two AUs and with four of these modules interleaved as shown in Figure 5.8. The matrix of Figure 5.2 is stored in this memory as shown. The high-order address bits select the word in each of the modules; the low-order address bits select the module and the left-hand or right-hand word of the addressed module.

If the matrix is accessed by row, the maximum bandwidth is obtained because the stride is one. Under these conditions, the speedup is $S = m \times W$; where W = number of AUs per memory word = 2, $S = 4 \times 2 = 8$.

On the other hand, there can be addressing patterns that will reference only one module. Under these conditions, $S = 1$. Thus we see that $1 \leq S \leq (m \times W)$.

Experience has shown that wide-word interleaving yields a speedup that is similar to module interleaving. Expressing this relationship in the terms of Hellerman's model for random data addressing patterns, the effect of interleaving is $\sqrt{m}$ and the effect of having a wide-word is $\sqrt{W}$. Furthermore, the effect of these to enhancement steps is multiplicative;$S = \sqrt{(m \times W)}$.

It is interesting to note that von Neumann [BURK46] proposed wide-word interleaving as a memory organization. The memory is discussed in Chapter 2 in terms of being a queue. In the simple example given in

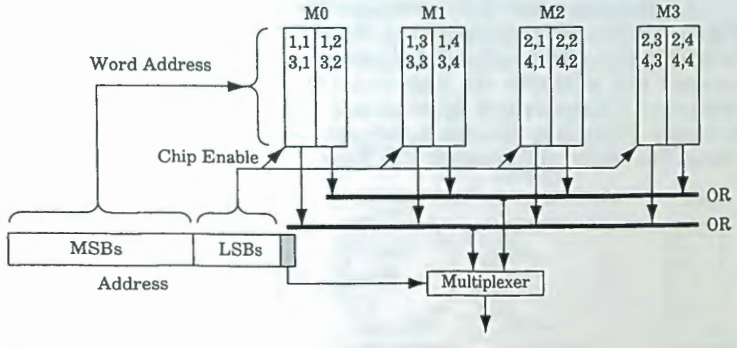


FIGURE 5.8 Wide-word interleaving.

Chapter 2, three memory cycles are required to read five instructions. A memory that is only 20 bits wide would require five memory cycles. Thus, the speedup is 5/3 or 1.666, which is rather close to $\sqrt{1 \times 2} = 1.414$. The simple model and example give speedup predictions that are reasonably close to each other.

The most compelling reason for wide-word memories today is to reduce the cache miss transport time. Wide-word memories complement wide busses, reducing the number of clocks required to transfer a block to/from a cache. Recall from section 2.2.1 that the memory modules of the IBM S/360/85 are eight bytes wide. Thus only four clocks are needed to transport a 32-byte block from the memory to the cache.

5.2 Disk Systems

The input/output of a computer has always been a bottleneck in system performance. During the early years, punched cards were the primary I/O media, followed later by magnetic tape. With the advent of the first commercial disk in 1956, cards have disappeared and magnetic tape has become archival media. The IBM 350 RAMAC, introduced in 1956, had a capacity of 40 Mbits (not bytes!) on 100 surfaces with an access time of 0.5 seconds [MATI 77]. Access to the RAMAC is random when compared to magnetic tape, and the name RAMAC is taken from Random Access Memory with the then-common suffix AC. Even though magnetic drums were used for mass storage and virtual memory as early as 1948, they were severely limited in capacity and the RAMAC type disk was a welcome replacement.

A disk is organized in a hierarchy of data storage areas described in Figure 5.9. The basic storage unit is the *sector*, which contains 32 Kbytes to 4 Kbytes. A sector is the unit of transfer between the disk and main memory and is usually the page in a virtual memory system. A track consists of 4 Ksectors to 8 Ksectors, and there are 30 to 2,000 tracks on a surface. Note that the disk sector and cache sector are not the same thing. The tracks are accessed with Read/Write (R/W) heads on an arm

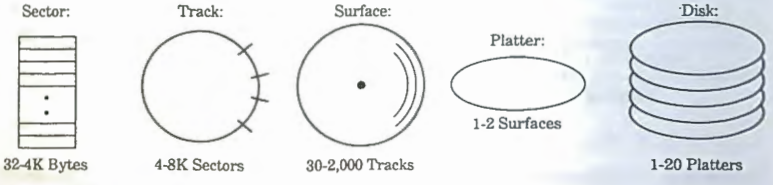


FIGURE 5.9 Disk storage hierarchy.

positioned over a track by an actuator. Each platter has one or two surfaces, and 1 to 20 surfaces make a disk unit. Needless to say, these hierarchy numbers change with time, generally increasing, and are only a guide to the relative dimensions.

Another component of disk organization concerns the relationship between the arms with their R/W heads and the tracks; each surface has at least one arm with its R/W head. Disk systems with more than one arm (either more than one arm for one surface or more than one surface each with one arm) can have either independent arm movement (independent actuators) or the arms can be ganged together in groups. The tracks under a ganged group of heads are known as a *cylinder*, a term taken from drum technology. A cylinder consists of the tracks that are under the heads at one time. Note that for a system with independent arm movement, cylinders can be virtualized by positioning the arms over the tracks of a cylinder.

The discussion above highlights heads that have the combined function of read and write. However, disk technology is moving in the direction of independent read and write heads. The reason for this is that recording and reading are no longer complementary with the higher areal densities and purpose-designed heads are required as well as accompanying arms and positioning circuits.

Two examples of disk organization hierarchy are shown in Table 5.10: the Cray DD-49 disk unit, as reported in [KATZ89], and the EX-2, 5-1/4" floppy reported by [SARG86]. The first is a high-capacity disk for a supercomputer. These parameters reflect formatted disks.

Disk surface areal density (bits/inch × tracks/inch) has increased at a rate of 26% per year since 1971, and these increases have translated almost directly into cost reduction. However, even with improvements in both latency and transfer rate of the early disks as compared to magnetic tape, disk systems remain a limiting factor in computer system performance because of their mechanical characteristics that limit lat-

	Cray EX-1	Cray EX-2 (floppy)
Bytes/Sector	42	512
Sectors/Track	4096	15
Tracks/Surface	443	80
Surfaces/Platter	2	2
Platters/Disk	8	1
Total Capacity	1.2 Gbytes	1.2 Mbytes

TABLE 5.10 Example disk parameters.

ency reductions. In this chapter several issues concerning disk systems are addressed, including disk latency, bandwidth requirements, performance, cost projections, interleaved disks (RAID), and disk caches.

5.2.1 *Disk Requirements*

There is very little solid information published on the question of the disk bandwidth and storage requirements based upon workloads that can serve as a design guide [GOLD87]. The usual design approach is to assume that all of the disk parameters are inadequate to the task and the designer should provide the needed capacity and then just provide the best performance possible with the selected disks. In fact, most of the research in the use of disks has been directed toward this second step: overcoming the inadequacies of the disks. For example, [MILL91] reports on measurements of I/O activity made on a Cray Y-MP 8/832 when executing scientific benchmarks. Unfortunately, most of the data is referenced to CPU time and are therefore of limited use for design guidance.

Five parameters are commonly used to characterize a disk system.

1. Capacity. Bytes of storage (formatted).
2. Bandwidth. Bytes transferred/unit of time.
3. Service rate. Number of service requests satisfied/unit of time.
4. Response time. Time between the start and completion of an event.
5. Cost. $/Mbyte.

The reader should keep in mind that the requirements for disk systems are different for mainframes and single-user computers. In the former case, capacity and service rate are of prime importance; for the latter, response time is of prime importance. These differences are discussed in the following paragraphs.

Capacity

A study by Goldstein [GOLD87] indicates, for mainframe computers, that the average disk storage capacity is 3.7 Gbytes per MIPS; see Figure 5.10. The plot shows the capacity per MIPS from a customer survey taken in 1985. This data is for large mainframe applications and does not reflect the requirements of work stations and other single-user computer systems. However, servers should have approximately the same disk capacity per MIPS as mainframes.

Other data from Goldstein's study, reflected in Table 5.11, shows that the capacity index has remained relatively constant at 3.7 Gbytes/MIPS since 1980.

The capacity of the hard disk of a typical personal computer has

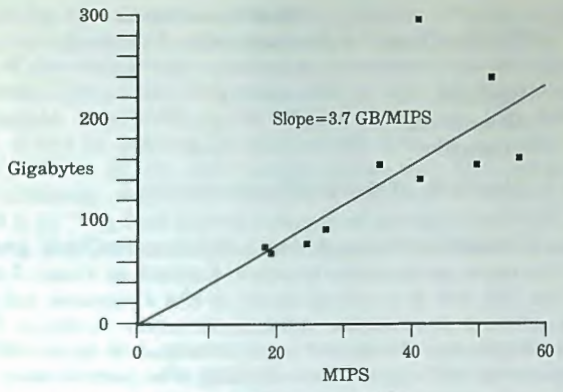


FIGURE 5.10 Mainframe disk capacity versus MIPS.

Year	GB/MIPS
1972	1.5
1974	2.0
1976	2.6
1978	3.5
1980	3.7
1982	3.6
1983	3.7

TABLE 5.11 Mainframe disk capacity versus year of survey.

grown from 5 Mbytes in 1988 to 250 Mbytes in 1994, which is an absolute capacity growth rate of 32% per year. A typical PC of 1980 and 1994 both have approximately 5 Mbytes per MIPS; installed hard disk capacity is tracking the increase in processing rate. Note, however, that the PC ratio of 5 Mbytes per MIPS is 1/740 that of the Goldstein mainframe ratio noted above. We can see from this information that there can be a major difference in ratio of disk capacity to MIPS depending upon the particular use and application of the computer.

Bandwidth

One of the famous computer design "laws" is the Amdahl Law, also called the CASE/AMDAHL law [HENN90, AKEL91], that states that a balanced system should provide a bit of I/O transfer for every instruc-

Problem Type	Bits Transferred per Instruction Executed
Scientific	1
Commercial	2
Interactive	4–8

TABLE 5.12 I/O bandwidth requirements.

tion executed. However, I cannot find a reference to this law in the published literature attributable to either Amdahl or Case. I can only conclude that this law is a rule-of-thumb in the computer art with no recognized reference to an author. A balanced system is one in which all of the queue depths are stable and are approximately equal to one and one component is not significantly limiting the performance of the system.

The definition of bandwidth in Chapter 2 is changed when referring to Amdahl's law in that the normalizing function is an executed instruction rather than time. Taken at its face value, Amdahl's law means that a processor executing at the rate of 10 MIPS will require an average I/O transfer of 10 million bits (1.25 Mbytes) per second. And, because the transfers would be in bursts, the peak transfer rate will be much higher.

Another view of the bandwidth requirement is given by Matick [MATI77] and shown in Table 5.12. The data shown is a composite view of data published by other researchers in 1970 and 1971 and is quite old.

Notice that for scientific processing, one bit is transferred for each instruction executed, which is the same value cited as Amdahl's Law. However, for interactive processing, the I/O requirement is 4 to 8 times more demanding, and transaction processing should have approximately the same requirement. As interactive and transaction processing increase in importance, the need to provide high I/O bandwidth is an important design problem.

A small amount of contemporary data exists on this issue. An I/O demand of 0.73 bits per instruction has been reported for a VAX 8800 executing a mix of batch, system, and interactive tasks [CLAR88]. For a scientific program, an atmospheric simulation model on a Cray-2, 0.32 bytes per instruction are required [CATL92].

Data presented in [KATZ89] shows that the bandwidth of a single-disk system is in the range of 0.3–3.0 Mbytes per second. Based upon one bit per instruction, these disks are capable of supporting approximately 1 MIPS of interactive processing. Thus, a single disk is inadequate for high-performance processors in most programming environments.

Current research concerns techniques for improving disk system

bandwidth by spreading the data over a number of disks, or disk array. An early survey of disk array research is found in [KATZ89]. A simple example of the disk array technique is to consider that 32 disks are used to store data. If the basic disk bandwidth per disk is 2 Mbytes per second, the aggregate bandwidth is 64 Mbytes per second. The technique of arraying disks is similar to interleaving memory modules discussed previously. Some of the major design issues with disk arrays are data layout, buffering, reliability, and error recovery (discussed further in Section 5.2.3). The first known example of a commercial disk array can be found in the Connection Machine Data Vault [KATZ89].

Latency

The deficiency in disk bandwidth or transfer rate is usually masked by the very long latency of a disk access. Latency of a disk consists of two components: (1) seek time and (2) rotational latency. Seek time is the time needed to position the head under the desired track and consists of the arm start time, traverse time, and stop time. Complex hardware mechanisms have been designed to reduce seek time and improve the positioning accuracy of the head. Nevertheless, seek time can be in the range of 2 ms to 10 ms depending upon the number of tracks to be crossed.

In addition to hardware approaches to reducing seek time, arm scheduling algorithms can be employed to reduce effective seek time. Effective seek time reduction can be achieved by taking requests off the queue such that the request for a close track is served before a request for a distant one. For heavily loaded systems that have a large number of requests in the queue, arm scheduling works well. However, arm scheduling does not always improve seek time. For example, systems that are balanced with regards to processing and I/O should have an average queue depth of less than one; consequently, arm scheduling becomes ineffective [KIM 87]. Also, arm scheduling is of little benefit to single-user systems unless there are background batch jobs executing that will fill the request queue.

Rotational latency is the time required for the desired sector to reach the R/W head and is set by the rotational speed of the disk. Rotational latency is, on average, one half of the disk's rotation time. Many disks rotate at 3,600 RPM, giving an average rotational latency of 8.3 ms.

One approach to improving the rotational latency of a disk is *rotational positioning sensing* (RPS). One use of RPS is to schedule disk transfers of the queue based on serving the first sector that will be under the heads, which is similar to arm scheduling. Another use of RPS is to improve the utilization of the disk channels. With RPS, after the access command is presented to the disk, the channel is disconnected and made

available for another access. Then the position of the disk is sensed and as the desired sector approaches the head the channel is reconnected. RPS is discussed further in Section 5.2.2.

Ng points out that there is a potential problem with RPS when used to improve channel utilization [NG91]. Because of the great difference between latency and data transfer time, large systems disconnect the channel after a request has been made to the disk for an access. Therefore, the channel is free to transfer data to/from another disk. However, there is a possibility that the channel is still connected to the other disk and cannot be restored to the disk that released it. When this happens, an RPS*miss* occurs, requiring another complete rotation before the request can be serviced. The latency of an RPS*miss* can be greater that the seek time. Ng discusses four methods by which the problem with RPS*miss* can be reduced. These include three forms of data redundancy and a dual actuator approach.

The dual actuator approach has a heritage in the paging drums of the 1960s, which had two heads per track and zero seek time because each track had a head and the average rotational latency was cut in half to approximately 4 ms. Data redundancy is also used on paging drums with only one head per track; redundant data is placed 180 degrees apart. Thus, one copy is always within a half rotation of the head. Another trick for eliminating rotational latency is to have the sector size equal a full rotation of the drum. The sector is transferred into a random access memory; consequently, the transfer could start at any point in the sector. Paging drums are no longer used because of the availability of low-cost RAM used as either main memory or as a disk cache.

The total latency (seek plus rotation) for one actuator of a mainframe class disk has decreased from approximately 30 ms in 1985 to approximately 15 ms in 1990—a reduction rate of 13% per year. Even if the rate of decrease in total latency continues to the year 2000, the latency will still be approximately 4 ms—the same as that obtained with paging drums.

In addition to arm scheduling and RPS, the use of buffers or disk caches has significantly improved the latency of disks where either (1) software prefetching is effective or (2) the pages demonstrate temporal locality. A hypothetical system, circa 1965, is shown at the top of Figure 5.11. This system transfers I/O directly into and out of main memory. Using the Matick data of 1 million bits per 1 million instructions for scientific programs and assuming that the sector size is 8 Kbits, the sector accesses per million instructions is

$$\text{accesses/million inst.} = \frac{1 \times 10^6}{8} \times 10^3 = 125.$$

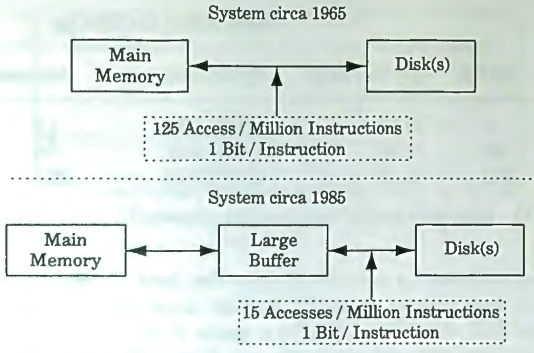


FIGURE 5.11 Disk requirements reduced with disk buffers.

The systems of the 1960s had small buffers that only smoothed the flow of data between the main memory and the disk. However, as the buffers increased in size, the "miss rate" on these buffers decreased and the traffic on the bus between the buffer and the disk decreased. The transition is now underway to convert the buffers into hardware-managed "disk caches." The benefits of buffers can be found in Goldstein's data from a 1985 survey, which shows 15 disk accesses per second are required for each MIPS or 15 accesses/million instructions; see the bottom of Figure 5.11. The 1985 system has a buffer between the main memory and the disk that operates similar to a cache and has a P_{miss} estimated to be $15/125 = 0.12$.

Note that the number of bits of real-disk I/O and the number of real-disk accesses per instruction is reduced by the use of buffers only to the extent that files have temporal locality. The buffers decouple many of the disk accesses and transfers from the physical disk by servicing them in the buffer, hence reducing the number of real-disk accesses/instruction by a factor of eight over this twenty-year period. This reduction is 11% each year, which is close to the 10% reported by Goldstein for the period 1980–1985.

Because of increasing instruction processing rates, ever higher access rates and I/O bandwidth are required. Table 5.13 shows projections of bytes/instruction, bytes/second, and accesses/second for the disk system (that is, buffer accesses of the disk). This projection is based on an increase in MIPS of 30% per year and a decrease in I/O bytes per instruction of 10% per year resulting from buffering. By the year 2000, the disk must be capable of 6 Mbytes/sec and 225 accesses/sec.

The developments in disk arrays will ensure that the bandwidth requirements are achieved. However, research into short latency disks

		Disk, After Buffering		
Year	MIPS	Bytes/Inst.	Bytes/Second	Accesses/Sec.
1985	5	0.10	500,000	15
1990	20	0.06	1,200,000	36
2000	300	0.02	6,000,000	225

TABLE 5.13 Disk bandwidth requirement projection.

has not yet indicated a way to achieve the goal shown in the table. As with caches and virtual memory, buffers must still be loaded with the working set and, this time, may be a major determinant in the response time of a system, regardless of the use of a buffer. Long response time or disk latency is particularly detrimental to single-user workstations and PCs.

Service Rate

Service rate (also called *throughput* and *bandwidth*) is defined as "the amount of work completed in a specific interval of time" [DONO72]. There are two views of service rate: The theoretical maximum service rate of a system and the actual service rate of a system that is receiving work requests. The latter is always less than the former. Factors such as the number of disk channels and load on the CPU play significant roles in setting the service rate of a disk system.

The combination of latency, bandwidth, and scheduling produce a service rate measure of the disk in terms of transactions/second. The first commercial disks, circa 1965, could perform approximately 3 accesses per second. Referring again to the data of [KATZ89], for the disks listed, the maximum service rate is in the range of 0.8 to 50.0 I/Os per second per actuator, not counting queue time.

Analytic modeling techniques have been developed that permit designers to evaluate trial designs and to assess tentative system changes to improve performance. Buzen developed a canonical model of a processor system [BUZE71, DENN78] that is used to evaluate processor and disk configurations and the level of multiprogramming. The issues discussed in the next paragraphs concern how the mean service rate varies as the system parameters are changed.

The Buzen model, as shown in Figure 5.12, is called the *central server model*. The central server model has one CPU and m I/O devices with service rates m_i. As a task completes processing, it is put into one of the I/O queues or exits and is replaced by an identical task that is placed on the CPU queue. The probabilities of these paths are given by

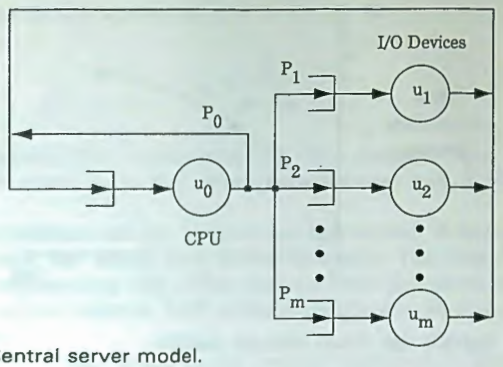


FIGURE 5.12 Central server model.

P_m. After an I/O event, the job is placed on the CPU queue. The I/O devices are assumed to be disks, drums, or a combination of the two. The response time solution to this model is found by recursively evaluating the response as the degree of multiprogramming is increased from 1 to 2 and so on. In addition to response time, the utilization of each of the I/O devices and the CPU can be determined.

Allen [ALLE80] provides an example of the use of the central server model for a system with a CPU and two I/O devices. The service rates and transition probabilities are

$$\mu_0 = 100/\text{sec}, \quad P_0 = 0.1;$$
$$\mu_1 = 25/\text{sec}, \quad P_1 = 0.2;$$
$$\mu_2 = 40/\text{sec}, \quad P_2 = 0.7.$$

The level of multiprogramming is set to 4. A proposed change is to replace I/O device #2 with a faster device; what are the original and the improved service rates? The results from the Buzen Central Server model are shown in Figure 5.13. By doubling the service rate of the disk #2 from 40 to 80 I/Os per second, the service rate of the system increases from 5.17 to 7.43 tasks per second—an increase of 43%. Other system changes can be investigated, such as a faster processor and/or increasing the performance of the slowest I/O device. Note that the asymptotic service rate of this system, with I/O devices that are infinitely fast, is 10 tasks per second as 10 passes through the CPU are required to complete a task. With these very fast disks, the system becomes CPU bound.

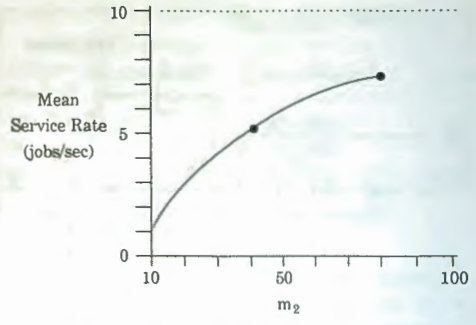


FIGURE 5.13 Service rate, mean jobs per second.

Analysis of systems performing conventional file-based I/O clearly indicates the detrimental effect of slow disk systems on total system performance. Reddy and Banerjee [REDD89] examined various alternative configurations of multiple disks and evaluated their performance for various scientific workloads. Akella and Siewiorek [AKEL91] survey various performance models. They measure the performance of VAX processors in relation to the measured performance of their models.

Response Time

Response time (also called *latency*) is the time between the start and completion of an event. The techniques of queuing theory are useful in evaluating response time for certain classes of systems. A thorough discussion of queuing theory is outside the scope of this book. However, I want to discuss a simple model; interested readers can consult [JAIN92].

For transaction processing and batch systems, response time is an important service parameter. While the system owner is interested in service rate and system utilization, users want rapid response time. A large transaction system may process thousands of transactions per second, and each transaction may require tens of disk accesses. Thus, the service rate of the disk system is an important parameter in satisfying a response time specification.

An open system queue model can provide an estimate of the response time of a transaction-processing system. The model assumes that there is an infinite pool of requesters and that when a request is serviced, the requester leaves the system. The arrival rates and service rates have Poision distribution. Transaction-processing systems, as well as grocery checkout and airline service counters, can be analyzed by this model.

The mean time in the system is

$$t_q = \frac{1}{\mu - \lambda}$$

where t_q denotes the mean time in the system, that is, queue time + service time; μ is the mean service rate; and λ is the mean arrival rate.

Two observations can be made about this model. If the arrival rate approaches zero, the mean time in the system is $1/\mu$, and if the mean arrival rate approaches the mean service rate, the mean time in the system approaches infinity. Both of these results can be observed in real life situations.

We can apply this model to the system described in the discussion on service rate, that is, a system with $\mu = 5.17$ tasks per second. If tasks are arriving at $\lambda = 2$ requests per second, the mean time in the system for a task is 0.315 seconds. A task is in the queue for 0.122 seconds and is processed for 0.193 seconds. With the infinitely fast disks, μ becomes 10 and the mean time in the system increases to 0.125 seconds. With this model, a system designer can trade off system cost with customer response time.

Note that the open system model can give misleading results if not properly applied. The assumption of an infinite source of requesters and no return to the input is not valid for many systems. For example, a system with a number of terminals must be modeled by a closed system model. Denning [DENN78] provide an excellent treatment of closed system models.

Disk Cost Projections

The areal density of disk recording increased from 0.002 Mbits/in^2 in 1957 to 10 Mbits/in^2 in 1980—an increase of approximately 43% per year [HARK81] that continues today. The cost of disk storage is shown in Figure 5.14, using data taken from IBM prices to the user that do not reflect the cost for PC related disks today. The slope of the curve is indicative of the reductions in price that have resulted from the stimulus of cost reductions of PC disks. Note that the figure uses dollars of the year. As inflation has been approximately 5% per year over the period 1965–1990, the reductions in disk price in constant dollars have been an astounding 25% per year. Note that 1 Mbyte of Model 2311 disk cost $3,300 in 1964. By 1993, the user price of the IBM 9377 disk array is $6.64 per Mbyte. There is little reason to doubt that similar cost reductions will continue at approximately the same rate over the next 10 years.

The lowest cost per Mbyte in the 1990s is found in disks used in

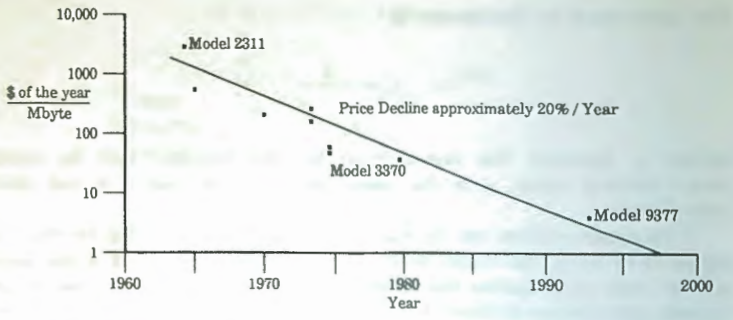


FIGURE 5.14 Disk cost history.

personal computers; high-volume applications costs are one-third to one-half those shown in Figure 5.14. Replacement disk systems for PCs cost approximately 50¢ per Mbyte in 1995. The major reason for this difference in cost, I believe, is that the cost learning curve that has been at work in the semiconductor industry is at work in the disk industry as well. Another reason for this difference is that high-volume PCs are sold with overhead and profit margins significantly lower than those found with mainframe computers.

Summary of Disk Projections

Goldstein's [GOLD87] survey, for moderate to large commercial MVS installations, shows that disk system response time in 1980 was in the range of 40 to 60 ms; an average of 46 ms is used in his studies. One survey shows response time averaged 30 ms—a decrease of 9% per year. It is projected that the trend to shorter response times will continue. Goldstein believes that a reduction of 13% per year in average response time will be required by systems and is possible to be obtained.

Goldstein combined his projections of 1987 and made a forecast of the performance of an "average installation" using a 30% per year growth rate in the installed MIPS. His projections are shown in Table 5.14.

The reduction in response time for the systems of 1990 and 1995 is predicated on caching. These caches are assumed to have a hit ratio of approximately 0.9. For any of the years, the installed disk capacity is in excess of a single drive and requires an aggregation of disk drives. The 510 Gbytes of 1990 requires 68 IBM 3380 units. Large computer installations often speak of their disk "farms" in terms of "acres of disks."

	1985	1990	1995
Installed MIPS	35	130	483
Gbytes of Disk	130	510	2092
Accesses/Sec (λ)	525	1387	3078
Response Time (t_q)	30 ms	15 ms	4.9 ms
Disk Access Time	34 ms	22 ms	22 ms
Acc/Sec/Gbyte	4.1	2.7	1.8

TABLE 5.14 Average installation disk system projections.

An approach to the design of arrays of small disks is discussed in Section 5.2.3.

5.2.2 *Disks in File I/O and Virtual Memory*

Up to this point, disks have been discussed in terms of a peripheral device. There are two ways that disks are connected to the processor and memory: (1) a file I/O system in which files of variable length are transferred or (2) as the lowest level in a virtual memory system in which pages are transferred. This section briefly discusses the issues of these two uses of disks.

The earliest computer systems used magnetic tape mass storage organized as files, usually of variable length. These files were transferred to/from the main memory under an operating system that managed files. With the advent of disk storage in the mid 1960s, the concept of a variable-length file was difficult to incorporate into operating systems. Because disks are physically divided into a hierarchy (sector, track, and surface), the sector is a natural addressable unit and is the unit of transfer between the tape, disk, and main memory; the sector is similar to a page in a virtual memory system. The operating system allocates a file to a set of consecutive sectors on the disk, hence wasting a portion of the last sector.

Files were user-defined collections of sectors that were managed in much the same way that pages are managed in real memory in a virtual memory system. The system must allocate space for new files on the disk, manage free space, and collect the garbage when blocks are de-allocated. The subject of allocation and de-allocation is discussed in Chapter 3, and reference to a book on operating systems is suggested for further details.

The most widely used operating system today, MS-DOS, is, as its name implies, a disk operating system. The user creates files that are managed, with considerable user intervention, by the operating system.

It is quite likely that disk file operating systems will continue to be used for many years. In fact, as discussed in Chapter 3, the disk file operating system is used in some virtual memory systems to map the virtual address to the disk when a page fault occurs.

The first virtual memory system, Atlas, used a drum for low-level memory backed up with tape. Drum technology was rapidly replaced by disks as they became available. With a virtual memory system, the concept of a file can, in theory, disappear into the very large virtual address space. That is, as every addressable unit is within the scope of the address space, no files are required. The IBM System/38 is an example of a fileless system. The elimination of a conventional file system leads to all sorts of problems that are outside the scope of this book; however, I want to mention a few.

One problem with a fileless system is the naming of virtual pages. In a system with a very long virtual address, a page name is assigned when a page is allocated and that name is never used again. The S/38 has a 55-bit virtual page name, and if 1 million names are assigned every second, over 1,000 years would elapse before the pool of names is exhausted.

Programming systems such as LISP assume a large name space. These systems and multiprocessor configurations of large name space machines present the problem of managing names. For example, how are new names assigned and how does one program that is calling a procedure with a name find the location of that name in the system? A distributed database known as the *namespace database* is used with a network of LISP machines to solve the problem of a very large name space [BROM87].

Another problem with a pure virtual memory system is backup and recovery. With a file-based I/O system, files can be periodically archived on magnetic tape or other removable media. If the system should fail, the system is restored from the backup material. With a virtual memory system, how does one back up an address space of 2^{64}, as is the case with the S/38? Furthermore, how is garbage collection (discussed in Chapter 3) performed on a very large virtual address space?

The problems of virtual memory described above lead to the design of systems that employ their disks in a hybrid configuration. That is, virtual memory demands that paged systems are implemented on top of conventional disk I/O. An example of this hybrid trend is found in the IBM RS/6000 and the various interactive systems implemented on top of MS-DOS, such as Windows and Lotus 1-2-3. I believe that these hybrid systems will be the implementation of choice for many years. There are just too many problems with fileless pure virtual memory system implementation for them to be viable.

Interconnection Topology

I now consider the issue of the connection between the processor and the disk(s). Figure 5.15 shows three typical disk system organizations [KIM86]. A single-disk system is shown in Figure 5.15(a); disk requests arrive at a rate λ, pass through the channel, and are serviced by the disk. The disk must have a service rate greater than λ, otherwise the queue in the processor will overflow. The model described above for the open system queue model applies, approximately, to this system.

Figure 5.15(b) shows a multiple-disk system with a single channel and multiple queues in the processor. By scheduling requests to the disks, the requests can be evenly distributed across the disks in a fashion similar to interleaved memory modules. Because the disks all share a single channel, rotational positioning sensing (RPS) is required to inform the queues when a request can be released to a particular disk. RPS sensing permits seek, latency, and RPS misses to be overlapped.

Figure 5.15(c) shows a system where more than one disk shares a channel, that is, a common system configuration. There is a queue in the processor for each disk, and disk requests are taken off the queues as each disk completes its current transaction. Disk requests arrive at the rate λ and are placed in the proper queues. Experience has shown that in a multiple-disk configuration there is a concentration of requests to a few of the disks, which is a distribution of requests known as *skew* [KIM86]. Reported skews for an eight-disk system are 0.388, 0.225, 0.153, 0.12, 0.068, 0.054, 0.01, and 0.001 for each of the eight disks (the conditions or operating system are unknown but are believed to be a data base machine). Thus the requests serviced by each disk are λ Skew(i). If the requests to the disks are balanced, the skew for each disk would be 0.125 for the eight-disk system. Three of the disks are overused and five are underused, leading to a diminution of potential response time of the system. Models show that a system with balanced skews gives the lowest weighted average response, as would be expected.

Shortest Access Time First Scheduling

A form of scheduling that was used prior to RPS with disks is *shortest access time first scheduling* (SATF), which was used with paging drum storage in early multiprogrammed systems [DENN67]. This scheduling technique is now routinely used in disk systems. A typical paging drum performed reads and writes, bit-parallel, with one R/W head per track; there is no head positioning time, and the tracks are divided into sectors of 1,024 bits. For a 32-bit word and with 32 R/W heads, each sector has 4 Kbytes.

Each sector has a queue, as shown in Figure 5.16, and the sectors

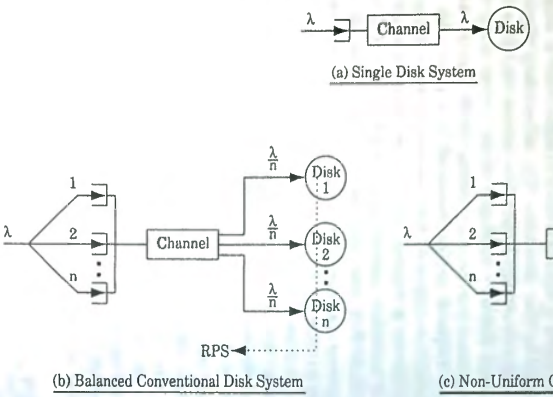


FIGURE 5.15 Disk system organizations.

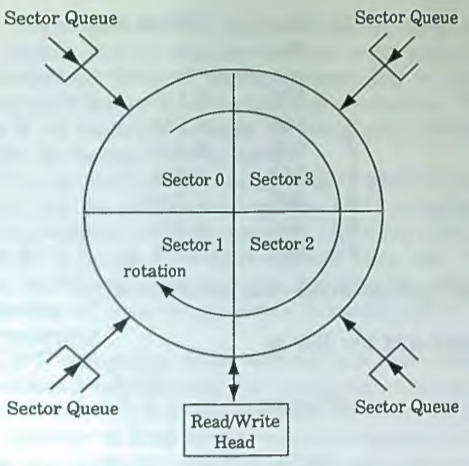


FIGURE 5.16 SATF scheduling.

rotate under the R/W heads in order $(1, 2, \ldots, n, 1, 2, \ldots)$. When, for example, sector 2 rotates to the point that the head could read or write from/to that sector, a request would be taken from the #2 queue and serviced. For a fully loaded system that has requests in all the queues, the drum would be continuously reading or writing, thereby achieving its maximum utilization.

A simple drum or disk without SATF scheduling has an average rotational latency equal to half the rotation time or an average service rate of two requests per revolution or a latency of one half-rotation. With SATF scheduling, the rotational latency is reduced because sectors are served as soon as possible but not necessarily in the order received by the drum system [FULL75]. Fuller and Baskett, whose work is modified by Pohm [POHM81], give an approximate service rate model of a paging drum using SATF scheduling:

$$\mu_d = \frac{s}{t_r}\left[1 - e^{\frac{-(J_d+1)}{s+1}}\right]$$

where μ_d is the paging drum service rate; s is the number of sectors; t_r is the rotation time of the drum; and J_d is the mean number of requests in the disk queues, $1 \leq J_d \leq 10$.

J_d	Service Rate	Utilization
1	106	0.110
2	155	0.161
3	201	0.210
4	244	0.254
5	286	0.298
6	324	0.337
7	361	0.375

TABLE 5.15 Paging drum service rate and utilization.

The utilization of the disk is

$$\text{utilization} = 1 - e^{-\frac{(J_d+1)}{(s+1)}}.$$

An example of the results obtained from this paging drum model for $t_r = 0.0166$ seconds (3600 RPM) and $s = 16$ is shown in Table 5.15. The number of requests in the queues, J_d, is varied from one to seven.

As the load increases, the service rate increases from 106 transactions per second to 361 transactions per second—a 3× increase in service rate. Note that the simple model for one queue is 2/0.0166 = 120 transactions per second, which approximately agrees with the Fuller model results for $J_d = 1$. The utilization of the paging drum also increases by a factor of three from 0.110 to 0.375. For very large values of J_d, the service rate approaches $s/t_r \approx 16/0.0166 \approx 960$, and the utilization approaches 1.

As noted previously, SATF scheduling only improves the service rate if there is a full queue of requests to be serviced. SATF scheduling does nothing to reduce the latency or increase the service rate for an interactive single-user system that has only a small mean number of requests in the disk queue, say, 0 or 1. SATF scheduling can be used on moving head disks if there is more than one request in the queue for the current track or cylinder. Note that sector scheduling is similar to the balanced system of Figure 5.15(b), where each R/W head is equivalent to a disk.

Multiprogrammed Systems

Large mainframe computers or servers are generally multiprogrammed systems for which the processing rate is usually more important than latency or response time. The first, known to me, multiprogramming

system is the Burroughs B5000. It was built in 1960 and was also the first commercial virtual memory system with pure segmentation [LEVI84]. A restricted form of multiprogramming is found today in systems that support foreground/background processing. The UNIX operating system is an example because its foreground process is usually interactive while its background is batch.

The basic idea of multiprogramming is that when an executing program must wait for an I/O operation, the CPU switches to another program that is resident in main memory. Another form of multiprogramming switch is based on a fixed elapsed time, say every 100 ms. The purpose of multiprogramming is to increase the utilization of the system. The number of resident programs is known as the *level* (*degree*) *of multiprogramming*.

Increased CPU utilization is not without cost. To be effective, there is processor overhead not required for a uniprogramming system such as context switching and queue management. In addition, there must be sufficient real memory to hold the additional programs and their data. Also there must be enough I/O channels (either real or shared) to support the concurrent I/O transactions generated by the resident programs.

Consider the issue of the real memory required to support multiprogramming. With a virtual memory system, memory capacity does not, at first, seem to be a problem. With a very long virtual address, each resident program has an almost unlimited memory space. However, as the real memory is finite, programs are competing for the limited real memory space and excessive paging may result. This causes another program to be switched into execution, which in its turn causes more paging until the system fails due to thrashing or lockout. Multiprogramming will not work unless there is enough real memory to support the degree of multiprogramming.

The capacity of the I/O system is also an issue. The executing program calls for an I/O operation, the next program is switched in, and it calls for an I/O operation and so on. Thus there must be sufficient I/O channels and bandwidth to support the degree of multiprogramming if 100% CPU utilization is to be achieved. Some smart I/O channels start the disk access for a read, disconnect while waiting for the disk latency, and can process another access. Nevertheless, the number of disks and channels (real or virtual) must be equal to the degree of multiprogramming and the accesses must be distributed over these disks; otherwise the queues will build up in front of one or more of the disks and further delay the access. The foreground/background type of multiprogramming may not suffer from the problem of large queues and blocking because the foreground may be inactive due to the user's thinking or otherwise not using the processor. There are a number of techniques for scheduling

the swapping of jobs that are outside the scope of this book. Reference to most books on operating systems will find a description of these techniques.

5.2.3 *Disk Arrays*

Disk arrays are receiving extensive attention from both university and industrial researchers. This attention results from the desire to overcome some of the aforementioned deficiencies of disks (cost, bandwidth, latency).

> A *disk array* is a grouping of a number of physical disks that makes these appear to applications as a single logical disk (paraphrased from [KATZ89]).

Various researchers [PATT88, HENN90] have observed that the cost learning curve has overtaken the law of scale with disk technology. In other words, the cost reductions due to vastly greater manufacturing volume produce a lower cost per bit than the reduction in cost due to the design and manufacturing of large scale disks.

In addition to potential cost advantages, disk arrays can provide, in some situations, a reduction in latency and an increase in bandwidth. If disk requests are interleaved across a number of disks in a balanced organization, a significant reduction in response time is possible if there are a number of active disk I/O requests. Also, because there are more than one active R/W head, the bandwidth is increased and the transport time can be reduced. Thus, proposals have been made to configure large-capacity disk systems from arrays of relatively small disk modules. If the overhead of interconnection can be controlled, an array system will have a lower cost than a single-disk system.

Consider the problem of designing a 100-MIPS computer installation requiring 3.7×10^9 bytes per MIPS. If this installation is served by IBM 3380 class disks having 7.5×10^9 byte capacity, approximately 50 disks are required. On the other hand, if an array of 300×10^6 byte modules are used, approximately 1,250 disks are required. Thus, disk array design must comprehend arraying large numbers of disks to support mainframe computers.

Arrays of disks will also have a higher transfer bandwidth because of the number of R/W heads that can be active at one time. The bandwidth increase is a function of the design approach, to be discussed below. It should go without saying that as the bandwidth of the disks is increased with arrays of disks that the bandwidth of the channel must also in-

crease. In all of the discussions to follow it is assumed that channel bandwidth is not a limiting factor.

There are a number of instances of arraylike operation with conventional drums and disks. Special purpose drums that provide equal bandwidth for access of a matrix [CRAG68] and operating systems that use more than one disk channel are found on supercomputers such as SCOPE OS [JOHN84]. Thus the current thinking on disk arrays builds upon these early efforts. A recent product announcement from IBM is for the IBM 9337 Disk Array Subsystem [IBM 92], which offers up to seven 3.5-inch disk drives. One model uses 542-Mbyte disks while the other uses 970-Mbyte disks. The implementation is RAID-5, which is discussed later in this section.

Disk Array Taxonomy

There are a number of design options for a disk array that are described in a taxonomy with three dimensions shown in Figure 5.17. These dimensions are (1) the *degree of interleaving*, which concerns the layout of the sectors on the disks; (2) the rotation of the disks, which can be synchronous or asynchronous; and (3) R/W heads, which can be positioned either independently or as a group. This disk array taxonomy is based on [KATZ89].

There are three possible design cases of rotation and arm movement (shown in Figure 5.17) that are discussed later. There is no known implementation of a synchronous disk with independent arm movement. Two definitions from [KATZ89] apply to disk arrays.

Stripe unit (SU) is the unit of data interleaving, that is, the amount of data that is placed on one disk before data is placed on the next disk. The stripe unit may be as small as one byte [KIM86] and as large as a disk sector. In terms of interleaved memory discussed in Section 5.1, a stripe unit is an addressable unit.

Data stripe (DS) is a sequence of logically consecutive stripe units.

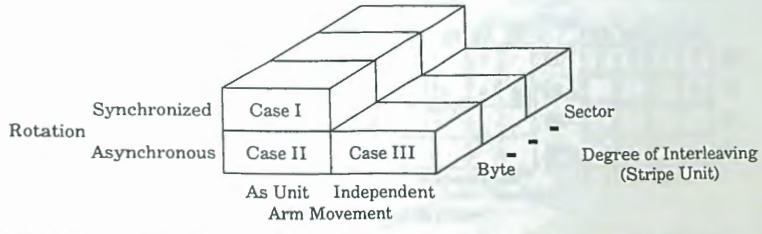


FIGURE 5.17 Disk array design options.

A logical I/O request to a disk array corresponds to a data stripe. In terms of an interleaved memory (discussed earlier), a data stripe is a vector of AUs with a stride of 1.

Degree of interleaving (m) is equal to the number of disks in the system over which the stripe unit is stored. This term is equivalent to the interleaving factor, m (discussed in Section 5.1 in reference to interleaved memory modules).

Degree of Interleaving

The degree of interleaving depends on the number of disks in the array and the size of the stripe unit. In other words, at any one time a number of R/W heads can be active depending on the number of disks in the array. The stripe unit can be, for example, a byte, word, quadword, or sector. The selection of the stripe unit size is a tradeoff between the number of disks in the array, the size of the data stripe and whether it is of fixed or variable length.

Figure 5.18 shows, for expository purposes, a disk array system with four disks, with one track per disk, and with four 1-Kbyte segments per track. The stripe unit is a sector. There are obviously more tracks, but only one track per disk is active at the time in this example.

Assume that a data stripe or file consisting of five stripe units, or five sectors, is to be written to the disk array. There are four possible assignments of these five sectors to the four disks; this assignment is similar to skewed interleaving of memory modules. The time required

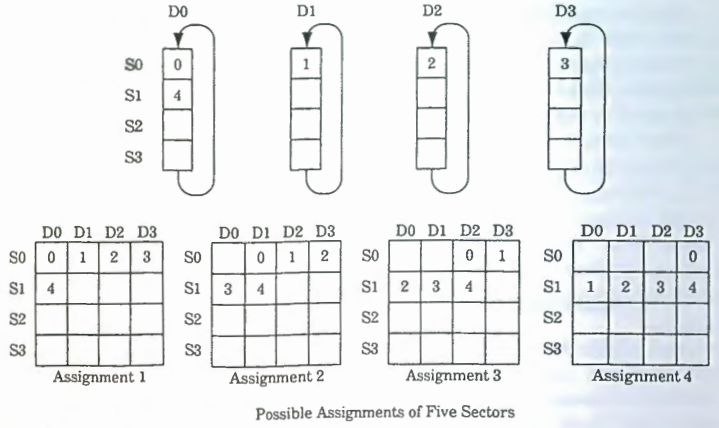


FIGURE 5.18 Disk array layout.

to write these five segments is the latency of the disks (rotation, head position) plus the time to transfer two segments rather than the time to transfer five segments if all of the segments are on one track of one disk. Note that the four disks are assumed to be accessed in parallel. Another assumption is that the disks are rotating synchronously, that is, the same sectors of all disks are under the R/W head at the same time. The other case (to be discussed later) is for asynchronous rotation.

Case I. Synchronous Disk With As-Unit Arm Movement

The synchronous disk organization, as shown in Figure 5.19, is an organization similar to the single-disk system of Figure 5.14(a). Fiducial marks are provided to the drive controller in order to maintain synchrony and to identify the beginning of the first sector, similar to RPS. The m heads are reading or writing in parallel, and the same sector is under the R/W heads of each disk at the same time. The access delays of head positioning and rotation are in parallel. The channel issues a request and, when the correct sector appears under the R/W heads, the request is honored. Kim [KIM86] named this system organization *synchronous disk interleaving* and assumes that the stripe unit is a byte—an assumption that is not a requirement for this organization.

The number of segment read/write times is

$$\text{segment times} = \left\lceil \frac{\text{data stripe size}}{\text{stripe unit size} \times \text{sector size} \times \text{number of disks}(m)} \right\rceil$$

For example, if a data stripe is 16 Kbytes, a stripe unit is 1 Byte, the sector size is 1 Kbyte, and $m = 4$, the data stripe can be transferred in four sector times as compared to sixteen sector times if disk interleaving is not used.

A conversion buffer is required to format a data stripe into the serial

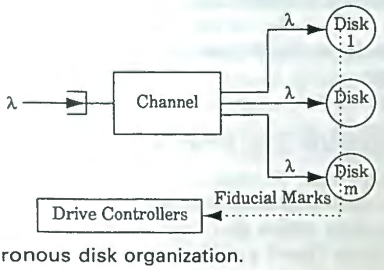


FIGURE 5.19 Synchronous disk organization.

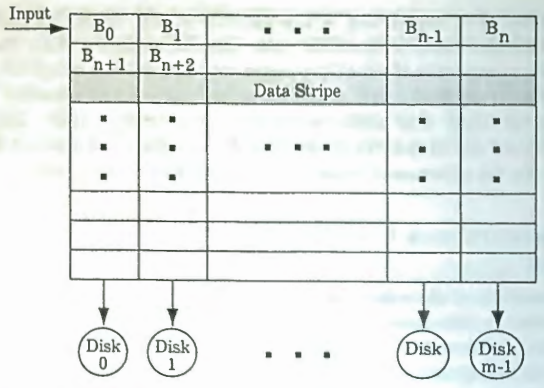


FIGURE 5.20 Conversion buffer.

streams required for each disk, as shown in Figure 5.20. This buffer design assumes that the incoming data stripe is serial by byte and the stripe unit is a byte. After the buffer is loaded horizontally, the buffer is then unloaded vertically into the disk units [KIM86].

There are a number of implementation problems with synchronous disks. The paramount one is providing synchronization of the disk spindles themselves. Kim [KIM86] discusses the problems of using synchronous motors driven from a common clock with feedback control (the fiducial marks noted in Figure 5.19). Another problem is coping with sectors that go bad after the disk array is placed in service. Reliability and failures are discussed later in this section.

Kim provides extensive model results on performance parameters such as service time, peak transfer rates, queuing delays, disk utilization, and weighted average response time. The effect of block (data stripe) size on response time and service rate is also evaluated [KIM86]. Chen and Patterson [CHEN90] have also investigated the impact of the size of the stripe unit on the service rate of this organization. The most significant parameter is the degree of concurrency of the input requests to the disk system.

Case II. Asynchronous Disk With As-Unit Arm Movement

An asynchronous disk organization is shown in Figure 5.21. With this system, after the data stripe has been loaded into a conversion buffer, the access operations (head positioning) of the disks are initiated and

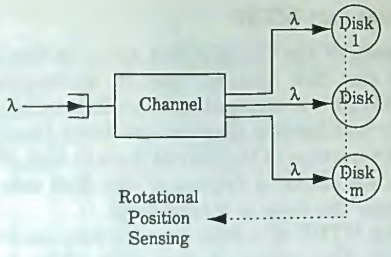


FIGURE 5.21 Asynchronous disk organization.

RPS informs the channel when each sector of each disk is available to receive its allocated stripe units. This system organization has the major advantage of not requiring synchronization of the spindles and the disadvantage of potentially lower performance due to the lack of concurrency in the read/write operations between the disks.

The conversion buffer of Figure 5.20 is modified so that the stripe units are read or written to/from the sectors of the disk at the correct time. This system organization has been named *asynchronous disk interleaving* by M. Y. Kim et al. [KIM87] and *disk striping* by [SALE86].

Case III. Asynchronous Disk With Independent Arm Movement

This system organization not only has asynchronous disk rotation but each arm of each disk can be independently positioned. This system organization appears to provide the greatest possibility for concurrency and therefore should improve the disk system's performance. This system organization is also the most consistent with the concept of an array of independent disk modules as would be created by racking up a multiplicity of PC disk modules.

Kim and Tantawi [KIM87] modeled this system configuration, and they point out that the effective rotational latency approaches that of the disk having the longest latency for a data stripe. That is, the expected seek plus rotational latency is greater than that expected of a synchronous system with unit arm movement. This result is intuitive in that the complete data stripe cannot be transferred until the last stripe units are transferred. Another variant of this system organization is named *declustering* under the assumption that the stripe unit is a sector [LIVN87].

Data Redundancy and MTBF

A direct consequence of the use of a disk array is the danger of reducing the reliability of the disk system, that is, decreasing the *mean time between failure* (MTBF). A conventional multiple-disk system, as shown in Figure 5.12(b), will have a disk failure from time to time, with the consequence that a portion of the stored data is lost. When the disks are arrayed as described above, a failure of one disk can corrupt all of the stored data because the data is interleaved.

In general, the MTBF of a disk array system is lower than that of a single unified disk. The reason is that disk units have roughly equal MTBFs. For example, the IBM 3380, a large mainframe disk, has an MTBF of 52,000 hours while the Conner CP3100, a smaller PC class disk, has an MTBF of 30,000 hours [KATZ89].

Katz illustrates the MTBF problem by comparing the MTBF of the IBM 3380 to an array of CP3100 disks that have the same storage capacity. Equal capacity, not considering redundancy, is achieved with 75 of the CP3100 disks that have an MTBF of 16.6 days. The single IBM 3380 has an MTBF of 2,166 days. The design of an array of disk modules must anticipate a high failure rate and the corruption of all or part of the stored data. Thus, an important and vital consideration in disk array design is providing a high level of data reliability in the face of poor hardware reliability or low MTBF.

The goal of schemes to cope with the low MTBF of disk arrays is to "fail soft," that is, operation continues while a repair is made. Fail soft schemes all depend upon some form of redundancy. A simple example is the use of error correcting codes in memory or communications systems. The design issue thus becomes one of choosing the best form of redundancy. There is an extensive body of knowledge on the use of redundancy for error detection and correction to give fail soft operation [SIEW82].

Kim [KIM86] proposed expanding the conversion buffer of Figure 5.20 to include horizontal *error correcting codes* (ECC). Check words are stored in an added disk along with vertical end-of-block check bytes. The described form of *X-Y* check bits has been used extensively in magnetic tape drives for many decades [MATI77]. With EBCDIC encoding, there is a parity bit for each byte (giving nine tracks) and a longitudinal parity byte recorded at the end of the record.

Patterson et al. have created a taxonomy of six disk array configurations that can be used to achieve data reliability or fail soft operation [PATT88, KATZ 89]. The term RAID (redundant array of inexpensive disks) is used to describe these systems briefly discussed in the following paragraphs. The six RAID configurations are shown in Figures 5.22 and 5.23.

RAID 1. Basically this reliability technique is simple redundancy—a traditional approach to the reliability of any disk system, arrayed or not. RAID 1 is also known as *mirrored disks* and *shadowing*. All reads

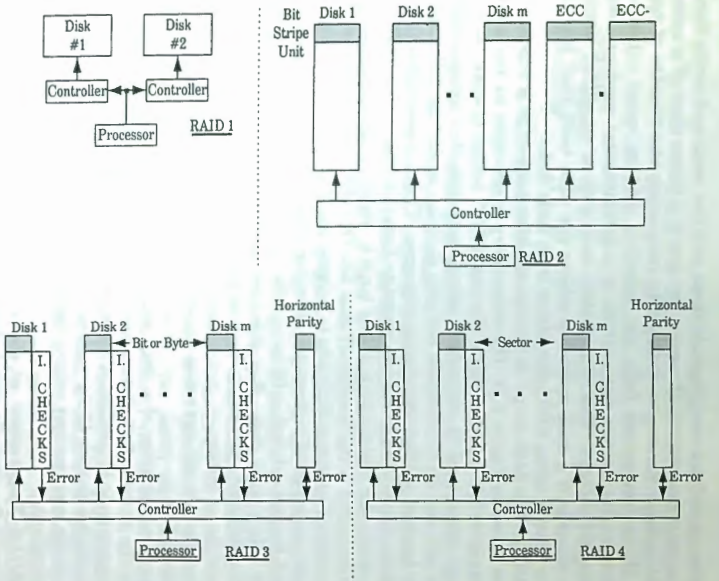


FIGURE 5.22 RAID 1 to RAID 4 designs.

and writes go to/from two identical disks, and the read and write bandwidth is not compromised. However, the cost is 2 times that of a nonredundant system. Errors are detected by comparing the results of both reads. If the comparison fails, the operating system must intervene to determine the correct data, which is a fundamental problem with duplicate redundancy. This organization may give a marginal performance increase if a read operation selects the disk unit that has the shortest latency using RPS information from both disks.

RAID 2. The data stripe is interleaved bitwise across a number of disks as shown in Figure 5.22. The redundancy is provided by horizontal ECC bits stored in separate disks. The use of ECC across a wide memory word has been used in computers using semiconductor memories for several decades. ECC identifies not only that an error has been made but the location of the bit(s) that are in error. The number of bits, thus ECC disks, required to detect two errors and correct one error (DEDSEC) must satisfy the relationship $2^c \geq d + c + 1$ where c is the number of check bits and d is the number of data bits [HAMM50]. A discussion of the number of required ECC bits is outside the scope of this book, but there are many good references. However, the degree of redundancy required for this scheme is $O(\log_2 m)$. For example, if the data stripe is applied across eight disks, four disks are needed to store the DEDSEC bits. The DEDSEC bits must themselves be interleaved because of the potential for a failure in one of the redundant disks. Simple parity for detecting but not correcting an error can be accomplished with one redundant disk.

Because a fraction of a data stripe will reside in a sector and the data in the sectors of all disks participate in the generation of the ECC bits, writes must be conducted as Read-Modify-Write operations, which reduce the bandwidth of the system. Read-Modify-Write operations are required for any memory where the memory word is larger than the addressable unit.

RAID 3. This method, as shown in Figure 5.22, of providing data correction in the face of a disk failure is a simplification of RAID 2. The stripe unit is a bit or byte. Because there is significant data checking provided on each of the disk drives, it is not necessary to have full ECC capability as with RAID2, simple parity will suffice. The redundant disk contains simple parity that is used to identify and correct the bit(s) in the disk that has failed or produced an error. The failed disk is identified by the internal checking hardware in each of the disks.

For example, assume a byte is interleaved across eight disks with even parity in the parity disk, as shown in Table 5.16. If disk 4 fails, the value of the bit is unknown: it could be a one or zero. The fact that the disk has failed is known, and the parity bit is 1 while there is an

	Disk								
	1	2	3	4	5	6	7	8	*P*
Normal	1	0	1	1	1	0	0	1	1
Disk 4 fails	1	0	1	x	1	0	0	1	0
Restored	1	0	1	1	1	0	0	1	1

TABLE 5.16 RAID 3 example.

even number of remaining good bits. From this information, the true value of 1 can be reconstructed.

The RAID 3 scheme requires Read-Modify-Write operations to compute the parity bit. However, the incremental hardware cost is small as the degree of redundancy is fixed at one disk module for any degree of disk interleaving.

RAID 4. This redundancy method is a modification of RAID 3 in that the stripe unit is a sector rather than a bit or byte. However, RAID 4 does not interleave across the disks, thus there is no improvement in transfer rate for a single stripe unit compared to a single disk. Some of this performance loss can be recovered with independent head movement that permits more then one read/write to be in process at once. For writes to the disk, the Read-Modify-Write operation is still required. Parity is provided at the byte level in the parity disk, and internal checking of each disk unit is used to identify the location of the data failure.

RAID 5. This redundant disk array system, shown in Figure 5.23, distributes the contents of the parity disk of RAID 3 or RAID 4 across all of the disks. The internal checking of RAID 3 and RAID 4 is not shown. A fraction of all of the disks is used to store parity bits rather than a separate disk; the amount of storage is the same even though a parity disk is not used. There are two failure modes that can be handled. First, a disk can fail that does not contain the parity of the accessed data stripe, this case is identical to RAID 3 or 4. For example, for the stripe unit identified •, if disk 1 fails, the data can be recovered because parity is not lost. Second, if the disk that fails contains the parity of the accessed data stripe, the correct data can still be recovered since only the parity bit is unknown. For example, if disk m fails the identified stripe unit can be recovered. As noted previously, the IBM 9337 is a RAID 5 system.

RAID 6. This disk array system extends the two-dimensional organization (one dimension of disks and one dimension of sector) of RAID 5

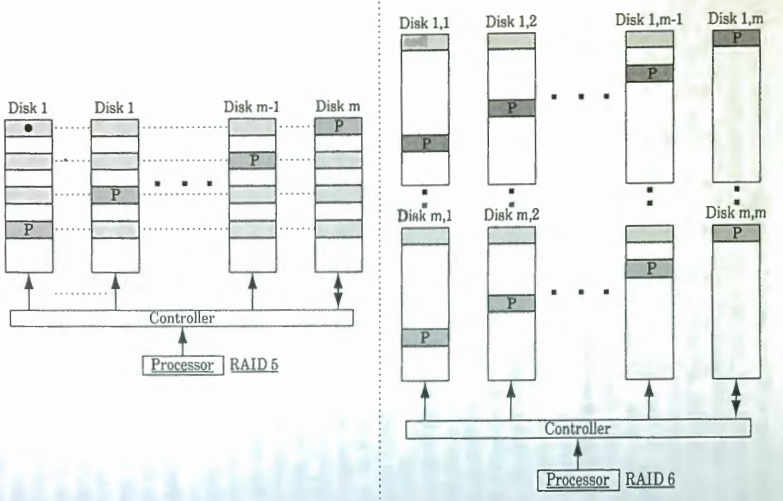


FIGURE 5.23 RAID 5 and RAID 6 designs.

to the three-dimensional (two dimensions of disks and one dimension of sectors). Each disk now contains both row and column parity plus the parity in the segments. Thus, RAID 6, due to its two-dimensional redundancy, can sustain two disk failures and still function. The RAID 6 redundant system checks the data in the column and does not depend on the internal checking hardware. Multidimensional disk arrays and their redundancy codes are discussed in [GIBS89].

5.2.4 *Disk Caches*

The very long latency of disks leads to significant delays for servicing page faults and requests for I/O. As with caches that effectively reduce the latency of main memory, a method for reducing the effect of disk latency is via use of the disk cache. Discussion in Section 5.2.1 shows how the use of software buffers has reduced the number of actual accesses to the disk. The question addressed in this section is whether or not a hardware-managed cache would be effective in reducing latency while transparent to the operating system.

I have no knowledge of specific disk cache implementations, thus I will speculate on their organization. A disk cache would probably be organized early select, direct access. The disk cache sector would probably be a disk track composed of a number of disk sectors that would be the disk cache blocks. The reason for this organization is that the transfer of a track from the disk to the cache sector would not have rotational latency because the read can start at any point of the track and can be mapped into the random access cache sector. A transfer from the cache that evicts a sector to the disk can also start at any point in the track since the source is a random access memory. A track size segment has been used with some paging drums to eliminate rotation latency.

A major design issue is where to place the disk cache in the processor-disk path. Figure 5.24 shows the typical topology for connecting disks to a processor. There can be a multiplicity of channels, a multiplicity of storage controllers, and a multiplicity of string controllers, each connected to a multiplicity of disk units.

The options for placing a disk cache are: (1) main memory, (2) channels, (3) storage controllers, (4) string controllers, or (5) at each of the disks. As the location of the caches is moved toward the processor, more disk accesses are handled by the cache and the utilization of the cache increases. For example, when placed at the main memory, all disk references pass through the cache. If placed at one of the disks, only the accesses to that disk are handled. Because of disk skew, it can be argued that the disk cache should be placed at the disk that has the heaviest