Memory Hierarchy Design

1. How to evaluate Cache Performance. Explain various cache optimization categories.

The *average memory access time* is calculated as follows: a whole or part all time low level and high

Average memory access time = hit time + Miss rate x Miss Penalty.

Where Hit Time is the time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache), Miss Rate is the fraction of memory references not found in cache (misses/references) and Miss Penalty is the additional time required because of a miss.

The average memory access time due to cache misses predicts processor performance.

First, there are other reasons for stalls, such as contention due to I/O devices using memory and due to cache misses

Second, The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time.

CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock cycle time

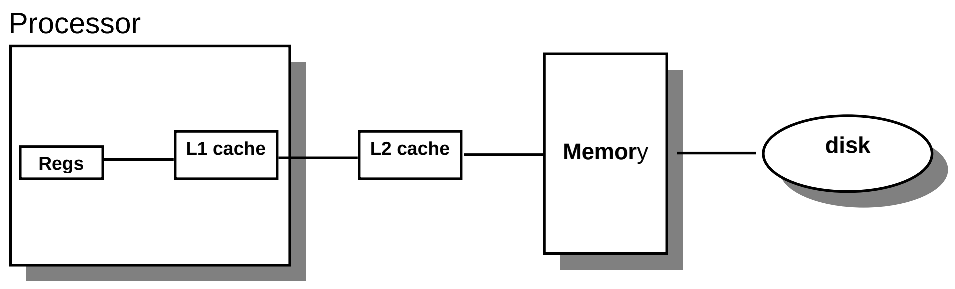
There are 17 cache optimizations into four categories:

1. Reducing the miss penalty: multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;
2. Reducing the miss rate larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations;
3. Reducing the miss penalty or miss rate via parallelism: nonblocking caches, hardware prefetching, and compiler prefetching;
4. Reducing the time to hit in the cache: small and simple caches, avoiding address translation, and pipelined cache access.

2. Explain various techniques for Reducing Cache Miss Penalty

There are five optimizations techniques to reduce miss penalty.

i) First Miss Penalty Reduction Technique: Multi-Level Caches



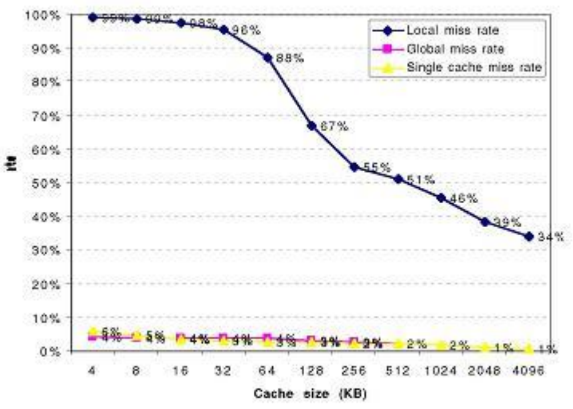
The First Miss Penalty Reduction Technique follows the Adding another level of cache

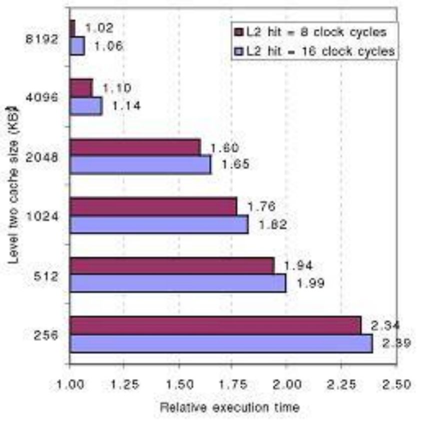
between the original cache and memory. The first-level cache can be small enough to match the clock cycle time of the fast CPU and the second-level cache can be large enough to capture many accesses that would go to main memory, thereby the effective miss penalty.
The definition of *average memory access time* for a two-level cache. Using the subscripts L_1 and L_2 to refer, respectively, to a first-level and a second-level cache, the formula is
Average memory access time = Hit time_{L1} + Miss rate_{L1} × Miss penalty_{L1}

and Miss penalty_{L1} = Hit time_{L2} + Miss rate_{L2} × Miss penalty_{L2}

so Average memory access time = Hit time_{L1} + Miss rate_{L1}× (Hit time_{L2} + Miss rate_{L2} × Miss penalty_{L2})

Local miss rate—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1} and for the second-level cache it is Miss rate_{L2}.
Global miss rate—The number of misses in the cache divided by the total num-ber of memory accesses generated by the CPU. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1} but for the second-level cache it is Miss rate_{L1} × Miss rate_{L2}.
This local miss rate is large for second level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.
Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second level cache.
Average memory stalls per instruction = Misses per instruction_{L1}× Hit time_{L2} + Misses per instruction_{L2} × Miss penalty_{L2}.
We can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache.
The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high.
Figures 5.10 and 5.11 show how miss rates and relative execution time change with the size of a second-level cache for one design.





ii) Second Miss Penalty Reduction Technique: Critical Word First and Early Restart

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally needs just one word of the block at a time. This strategy is impatience: Don’t wait for the full block to be loaded before sending the requested word and restarting the CPU. Here are two specific strategies:

Critical word first—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called *wrapped* fetch and *requested word first*.

Early restart—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Generally these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. The problem is that given spatial locality, there is more than random chance that the next miss is to the remainder of the block. In such cases, the effective miss penalty is the time from the miss until the second piece arrives.

iii) Third Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer. With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss. The simplest way out of this is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority

over writes.
The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

iv) Fourth Miss Penalty Reduction Technique: Merging Write Buffer

This technique also involves write buffers, this time improving their efficiency. Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. As mentioned above, even write back caches use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the CPU's perspective; the CPU continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses can be checked to see if the address of this new data matches the address of the valid write buffer entry. If so, the new data are combined with that entry, called *write merging*.
If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.

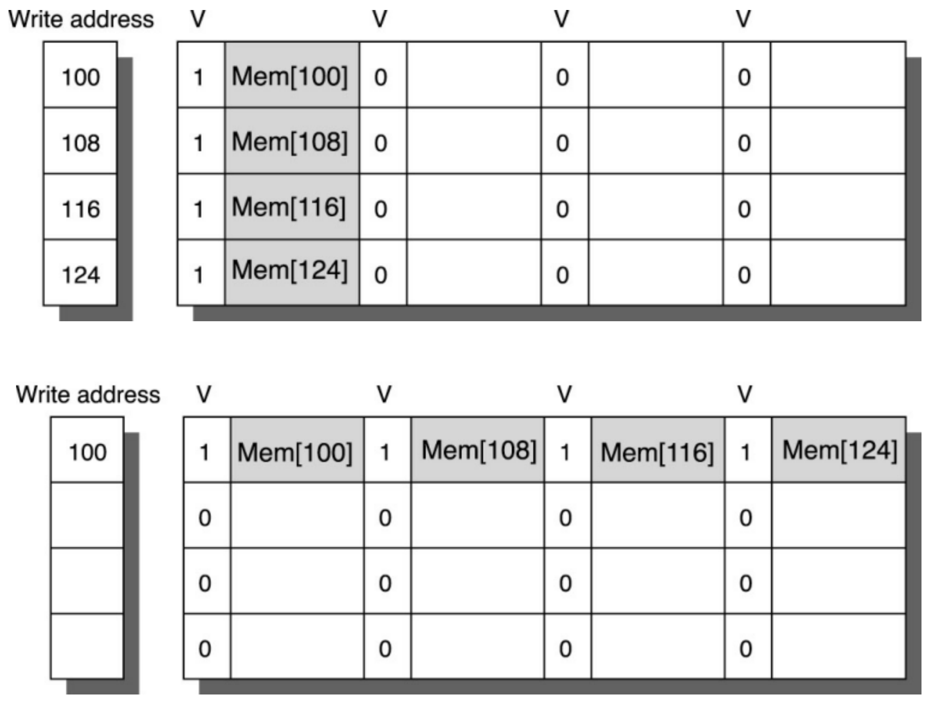


Figure 5.12

The optimization also reduces stalls due to the write buffer being full. Figure 5.12 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer. Figure 5.12. To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with valid bits (V) indicating whether or not the next sequential eight bytes are occupied in this entry. (Without write merging, the words to the right in the upper drawing would only be used for instructions which wrote multiple words at the same time.)

v) Fifth Miss Penalty Reduction Technique: Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.
Such "recycling" requires a small, fully associative cache between a cache and its refill path. Figure 5.13 shows the organization. This *victim cache* contains only blocks that are discarded from a cache because of a miss "victims" and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped. The AMD Athlon has a victim cache with eight entries.
Jouppi [1990] found that victim caches of one to five entries are effective at reducing misses, especially for small, direct-mapped data caches. Depending on the program, a four-entry victim cache might remove one quarter of the misses in a 4-KB direct-mapped data cache.

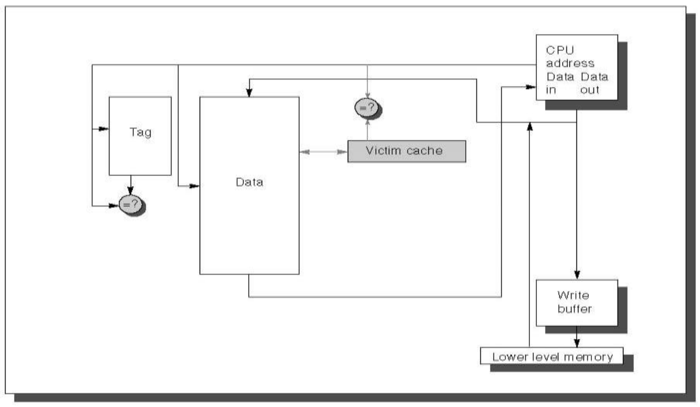


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

Summary of Miss Penalty Reduction Techniques

The first technique follows the proverb “the more the merrier”: assuming the principle of locality will keep applying recursively, just keep adding more levels of increasingly larger caches until you are happy. The second technique is impatience: it retrieves the word of the block that caused the miss rather than waiting for the full block to arrive. The next technique is preference. It gives priority to reads over writes since the processor generally waits for reads but continues after launching writes. The fourth technique is companion-ship, combining writes to sequential words into a single block to create a more efficient transfer to memory. Finally comes a cache equivalent of recycling, as a victim cache keeps a few discarded blocks available for when the fickle primary cache wants a word that it recently discarded. All these techniques help with miss penalty, but multilevel caches is probably the most important.

3. Explain various techniques to reduce miss rates?

The classical approach to improving cache behavior is to reduce miss rates, and there are five techniques to reduce miss rate. we first start with a model that sorts all misses into three simple categories:

Compulsory—The very first access to a block *cannot be* in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.

Capacity—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur be-cause of blocks being discarded and later retrieved.

Conflict—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur be-cause a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses*. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

i) First Miss Rate Reduction Technique: Larger Block Size

The simplest way to reduce miss rate is to increase the block size. Figure 5.16 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

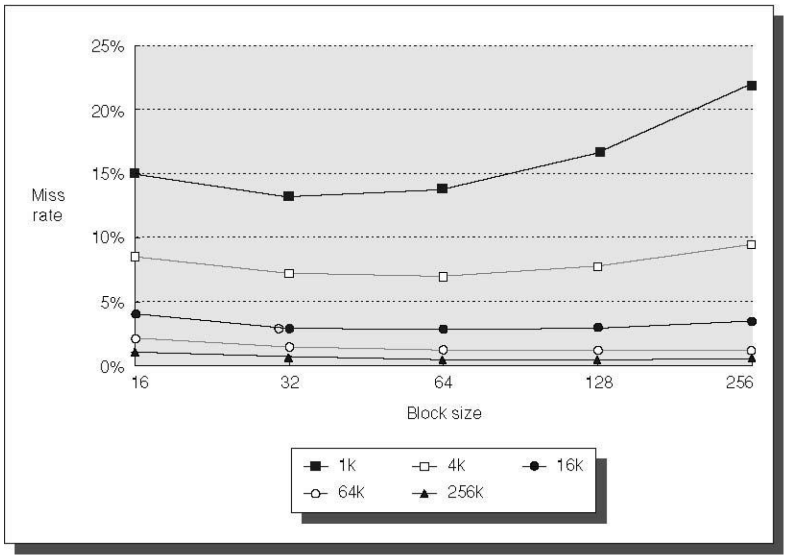


Figure 5.16 Miss rate versus block size for five different-sized caches.

At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it *increases* the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

ii) Second Miss Rate Reduction Technique: Larger caches

The obvious way to reduce capacity misses in the above is to increases capacity of the cache. The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers.

iii) Third Miss Rate Reduction Technique: Higher Associativity:

Generally the miss rates improves with higher associativity. There are two general rules of thumb that can be drawn. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the 8-way entries to the capacity miss, since capacity misses are calculated using fully associative cache. The second observation, called the *2:1 cache rule of thumb* and found on the front inside cover, is that a direct-mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N*/2. This held for cache sizes less than 128 KB.

iv) Fourth Miss Rate Reduction Technique: Way Prediction and Pseudo-Associative Caches

In *way-prediction*, extra bits are kept in the cache to predict the set of the *next* cache access. This prediction means the multiplexer is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.
The Alpha 21264 uses way prediction in its instruction cache. (Added to each block of the instruction cache is a set predictor bit. The bit is used to select which of the two sets to try on the *next* cache access. If the predictor is correct, the instruction cache latency is one clock cycle. If not, it tries the other set, changes the set predictor, and has a latency of three clock cycles.
In addition to improving performance, way prediction can reduce power for embedded applications. By only supplying power to the half of the tags that are expected to be used, the MIPS R4300 series lowers power consumption with the same benefits.
A related approach is called *pseudo-associative* or *column associative*. Accesses proceed just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of the memory hierarchy, a second cache entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find the other block in the "pseudo set."
Pseudo-associative caches then have one fast and one slow hit time—corresponding to a regular hit and a pseudo hit—in addition to the miss penalty. Figure 5.20 shows the krelative times. One danger would be if many fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache. The performance would then be *degraded* by this optimization. Hence, it is important to indicate for each set which block should be the fast hit and which should be the slow one. One way is simply to make the upper one fast and swap the contents of the blocks. Another danger is that the miss penalty may become slightly longer, adding the time to check another cache entry.

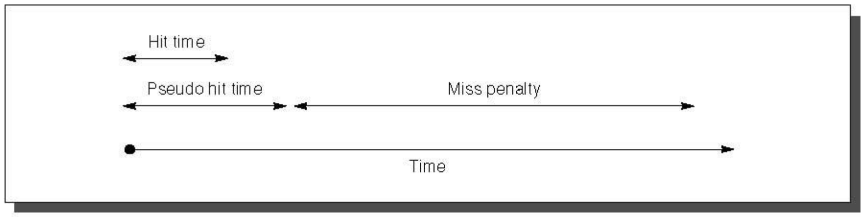


FIGURE 5.20 Relationship between a regular hit time, pseudo hit time, and miss penalty. Basically, pseudoassociativity offers a normal hit and a slow hit rather than more misses.

v) Fifth Miss Rate Reduction Technique: Compiler Optimizations

This final technique reduces miss rates without any hardware changes. This magical reduction comes from optimized software. The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data

misses.
Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses. Reordering the instructions reduced misses by 50% for a 2-KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8-KB cache.
Another code optimization aims for better efficiency from long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

Loop Interchange:
Some programs have nested loops that access data in memory in non-sequential order. Simply exchanging the nesting of the loops can make the code access the data in the order it is stored. Assuming the arrays do not fit in cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded.

```
/* Before */
for (j = 0; j < 100; j = j+1)
for (i = 0; i < 5000; i = i+1)
x[i][j] = 2 * x[i][j];

/* After */
for (i = 0; i < 5000; i = i+1)
for (j = 0; j < 100; j = j+1)
x[i][j] = 2 * x[i][j];
```

This optimization improves cache performance without affecting the number of instructions executed.

Blocking: This optimization tries to reduce misses via improved temporal locality. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (*row major order*) or column by column (*column major order*) does not solve the problem because both rows and columns are used in every iteration of the loop. Such orthogonal accesses mean the transformations such as loop interchange are not helpful.
Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced.
Summary of Reducing Cache Miss Rate

This section first presented the three C's model of cache misses: compulsory, capacity, and conflict. This intuitive model led to three obvious optimizations: larger block size to reduce compulsory misses, larger cache size to reduce capacity misses, and higher associativity to reduce conflict misses. Since higher associativity may affect cache hit time or cache power consumption, way prediction checks only a piece of the cache for hits and then on a miss checks the rest. The final technique is the favorite of the hardware designer, leaving cache optimizations to the compiler.

4. What is virtual memory? Write techniques for fast address translation

Virtual memory divides physical memory into blocks (called page or segment) and allocates them to different processes. With virtual memory, the CPU produces virtual addresses that are translated by a combination of HW and SW to physical addresses, which accesses main memory. The process is called memory mapping or address translation. Today, the two memory-hierarchy levels controlled by virtual memory are DRAMs and magnetic disks

Virtual Memory manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure 5.31 shows the mapping of virtual memory to physical memory for a program with four pages.

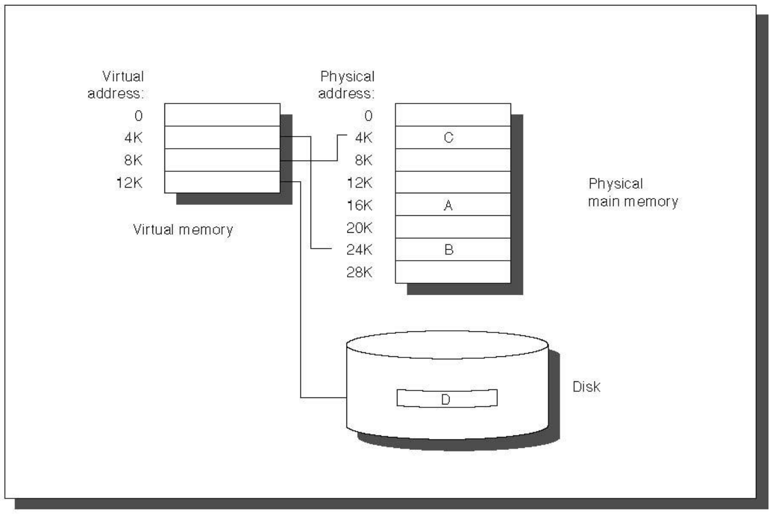
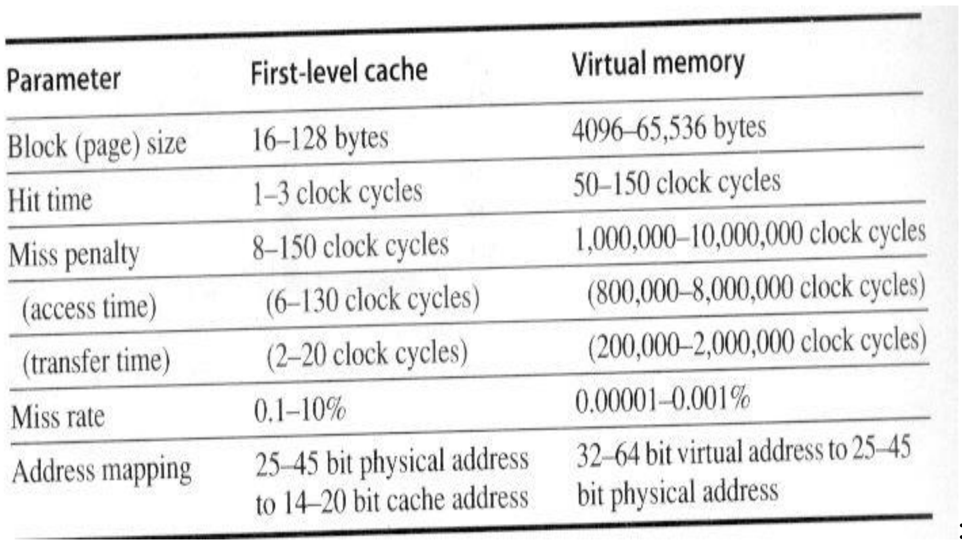


FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure 5.32



Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	50–150 clock cycles
Miss penalty	8–150 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–130 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–20 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit physical address to 14–20 bit cache address	32–64 bit virtual address to 25–45 bit physical address

:

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable-size locks, called *segments*. Pages are typically fixed at 4096 to 65,536 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure 5.33 shows how the two approaches might divide code and data.

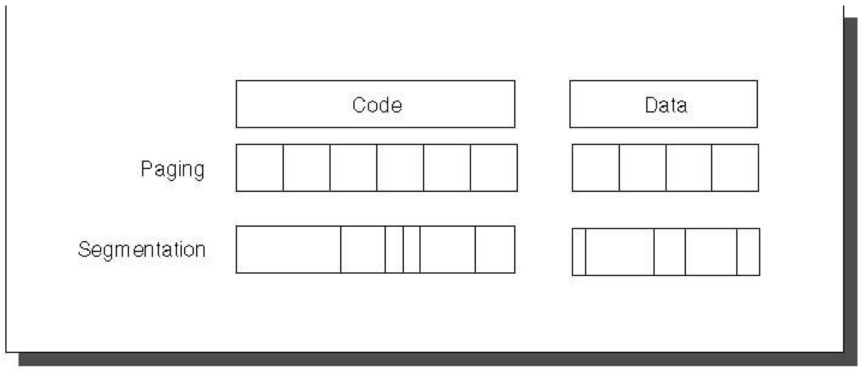


FIGURE 5.33 Example of how paging and segmentation divide a program.

The block can be placed anywhere in main memory. Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure 5.35).

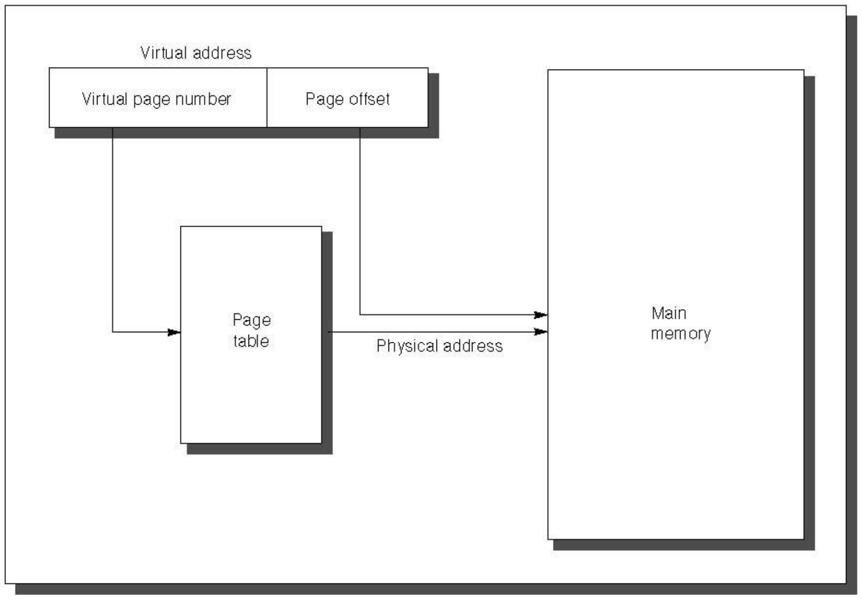


FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

This data structure, containing the physical page addresses, usually takes the form of a *page table.* Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry, the size of the page table would be $(2^{32}/2^{12}) \times 2^{2} = 2^{22}$ or 4 MB.

To reduce address translation time, computers use a cache dedicated to these address translations, called a *translation look-aside buffer*, or simply *translation buffer*. They are described in more detail shortly.

With the help of Operating System and LRU algorithm pages can be replaced whenever page fault occurs.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory, and some-times paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost is far too dear.

One remedy is to remember the last translation, so that the mapping process is skipped if the current address refers to the same page as the last one. A more general solution is to again rely on the principle of locality; if the accesses have locality, then the *address translations* for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely re-quires a second access to translate the data. This special address translation cache is referred to as a *translation look-aside buffer* or TLB, also called a *translation buffer* or TB.

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't be-have properly. Note that this dirty bit means the corresponding *page* is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. The operating system resets these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits. Figure 5.36 shows the Alpha 21264 data TLB organization, with each step of a translation labeled. The TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

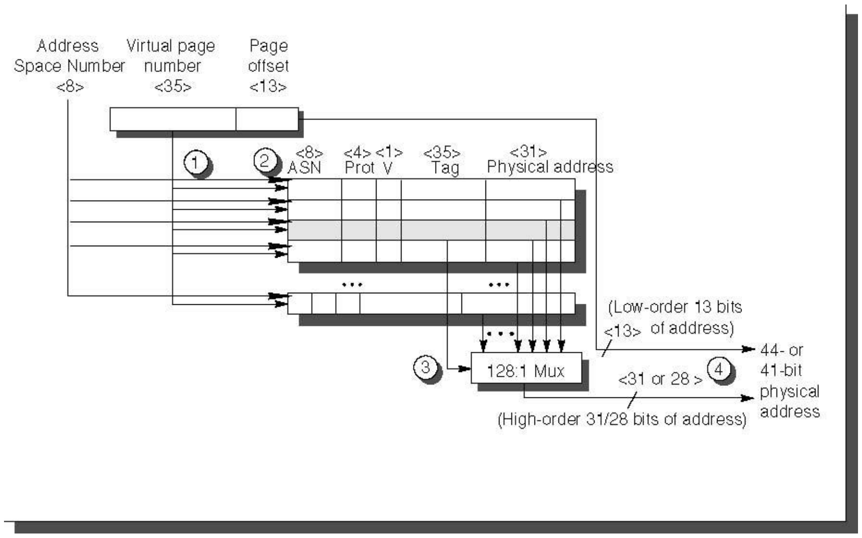


Figure 5.36 Operation of the Alpha 21264 data TLB during address translation.

Selecting a Page Size: The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- A larger page size can allow larger caches with fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- The number of TLB entries are restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

5. Explain how virtual memory is protected explain with example.

Multiprogramming forces to worry about usage of virtual memory. So Protection is required for virtual memory concept. The responsibility for maintaining correct process behavior is shared by designers of the computer and the operating system. The computer designer must ensure that the CPU portion of the process state can be saved and restored. The operating system designer must guarantee that processes do not interfere with each others' computations.

The safest way to protect the state of one process from another would be to copy the current information to disk. However, a process switch would then take seconds—far too long for a time-sharing environment. This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time.

Protecting Processes: The simplest protection mechanism is a pair of registers that checks every ad-dress to be sure that it falls between the two limits, traditionally called *base* and *bound*. An address is valid if

$$\text{Base} \le \text{Address} \le \text{Bound}$$

In some systems, the address is considered an unsigned number that is always added to the base, so the limit test is just

$$(\text{Base} + \text{Address}) \le \text{Bound}$$

If user processes are allowed to change the base and bounds registers, then users can't be protected from each other. The operating system, however, must be able to change the registers so that it can switch processes. Hence, the computer designer has three more responsibilities in helping the operating system designer protect processes from each other:

- Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a *kernel* process, a *supervisor* process, or an *executive* process.
- Provide a portion of the CPU state that a user process can use but not write. This state includes the base/bound registers, a user/supervisor mode bit(s), and the exception enable/disable bit. Users are prevented from writing this state because the operating system cannot control user processes if users can change the address range checks, give themselves supervisor privileges, or disable exceptions.
- Provide mechanisms whereby the CPU can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the sys-tem call, and the CPU is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

A Paged Virtual Memory Example: The Alpha Memory Management and the 21264 TLB

The Alpha architecture uses a combination of segmentation and paging, providing protection while minimizing page table size. With 48-bit virtual addresses, the 64-bit address space is first divided into three segments: *seg0* (bits 63 - 47 = 0...00), *kseg* (bits 63 - 46 = 0...10), and *seg1* (bits 63 to 46 = 1...11). kseg is re-served for the operating system kernel, has uniform protection for the whole space, and does not use memory management. User processes use seg0, which is mapped into pages with individual protection. Figure 5.38 shows the layout of seg0 and seg1. seg 0 grows from address 0 upward, while seg1 grows downward to 0. This approach provides many advantages: segmentation divides the address space and conserves page table space, while paging provides virtual memory, relocation, and protection.

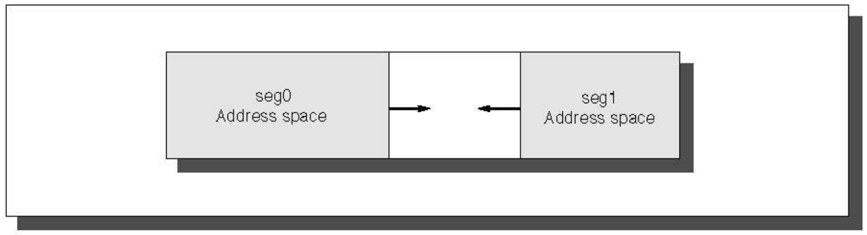


FIGURE 5.38 The organization of seg0 and seg1 in the Alpha. User processes live in seg0, while seg1 is used for portions of the page tables. seg0 includes a downward growing stack, text and data, and an upward growing heap.

The Alpha uses a three-level hierarchical page table to map the address space to keep the size reasonable. Figure 5.39 shows address translation in the Alpha. The addresses for each of these page tables come from three "level" fields, labeled level1, level2, and level3. Address translation starts with adding the level1 address field to the page table base register and then reading memory from this location to get the base of the second-level page table. The level2 address field is in turn added to this newly fetched address, and memory is accessed again to determine the base of the third page table. The level3 address field is added to this base address, and memory is read using this sum to (finally) get the physical address of the page being referenced. This address is concatenated with the page offset to get the full physical address. Each page table in the Alpha architecture is constrained to fit within a single page. The first three levels (0, 1, and 2) use physical addresses that need no further translation, but Level 3 is mapped virtually. These normally hit the TLB, but if not, the table is accessed a second time with physical addresses.

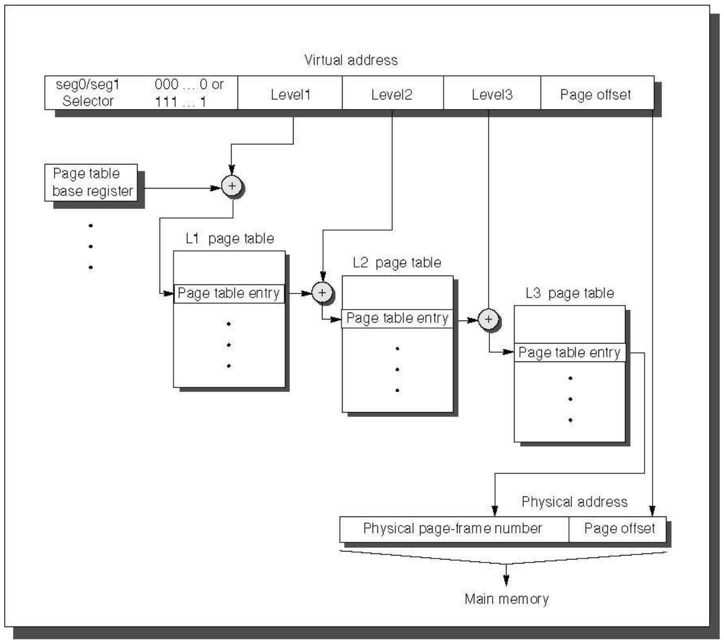


Figure 5.39 The mapping of an Alpha virtual address.

The Alpha uses a 64-bit *page table entry* (*PTE*) in each of these page tables. The first 32 bits contain the physical page frame number, and the other half includes the following five protection fields:
Valid—Says that the page frame number is valid for hardware translation

User read enable—Allows user programs to read data within this page

Kernel read enable—Allows the kernel to read data within this page

User write enable—Allows user programs to write data within this page

Kernel write enable—Allows the kernel to write data within this page

In addition, the PTE has fields reserved for systems software to use as it pleases. Since the Alpha goes through three levels of tables on a TLB miss, there are three potential places to check protection restrictions. The Alpha obeys only the bottom-level PTE, checking the others only to be sure the valid bit is set.

A Segmented Virtual Memory Example: Protection in the Intel Pentium

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks, and before a segment register could be loaded the corresponding segment had to be in physical memory. Intel's dedication to virtual memory and protection is evident in the successors to the 8086 (today called IA-32), with a few fields extended to support larger addresses. This protection scheme is elaborate, with many details carefully designed to try to avoid security loopholes.

The first enhancement is to double the traditional two-level protection model: the Pentium has four levels of protection. The innermost level (0) corresponds to Alpha kernel mode and the outermost level (3) corresponds to Alpha user mode. The IA-32 has separate stacks for each level to avoid security breaches between the levels.

The IA-32 divides the address space, al-lowing both the operating system and the user access to the full space. The IA-32 user can call an operating system routine in this space and even pass parameters to it while retaining full protection. This safe call is not a trivial action, since the stack for the operating system is different from the user's stack. Moreover, the IA-32 allows the operating system to maintain the protection level of the *called* routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user process to ask the operating system to access something indirectly that it would not have been able to access itself. (Such security loopholes are called *Trojan horses*.)

Adding Bounds Checking and Memory Mapping

The first step in enhancing the Intel processor was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, as in the 8086, segment registers in the IA-32 contain an index to a virtual memory data structure called a *descriptor table*. Descriptor tables play the role of page tables in the Alpha. On the IA-32 the equivalent of a page table entry is a *segment descriptor*. It contains fields found in PTEs:

A *present bit*—equivalent to the PTE valid bit, used to indicate this is a valid translation

A *base field*—equivalent to a page frame address, containing the physical address of the first byte of the segment

An *access bit*—like the reference bit or use bit in some architectures that is helpful for replacement algorithms

An *attributes field*—specifies the valid operations and protection levels for operations that use this segment

There is also a *limit field*, not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure 5.41 shows examples of IA-32 segment descriptors.

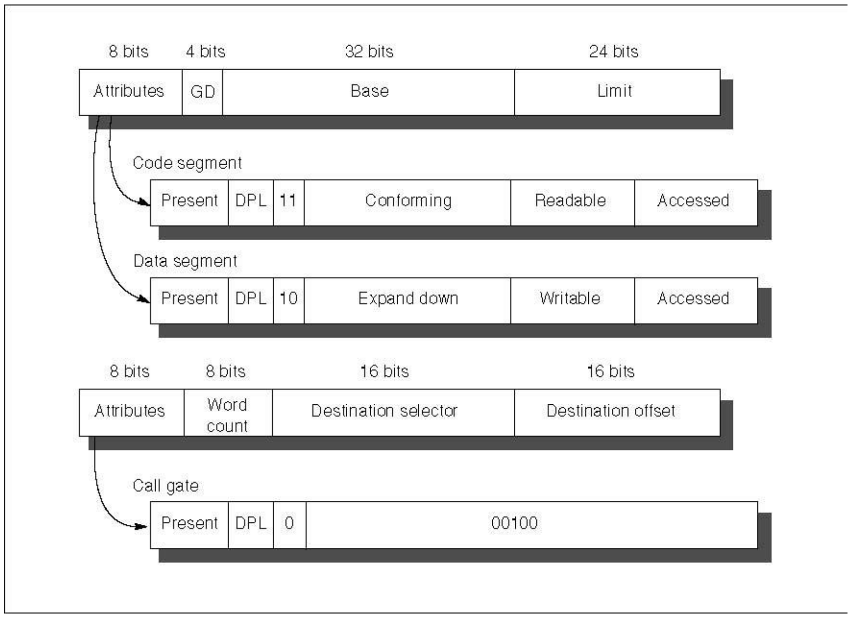


Figure 5.41 The IA-32 segment descriptors are distinguished by bits in the attributes field. *Base*, *limit*, *present*, *readable*, and *writable* are all self-explanatory.

IA-32 provides an optional paging system in addition to this segmented addressing. The upper portion of the 32-bit address selects the segment descriptor and the middle portion is an index into the page table selected by the descriptor.

Adding Sharing and Protection: To provide for protected sharing, half of the address space is shared by all processes and half is unique to each process, called *global address space* and *local address space*, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global descriptor table, while a descriptor for a private segment is placed in the local descriptor table.

Dr. Dharavath Ramesh Hari Nandan
Assistant Professor
Department of Computer Science & Engg.
Member IEEE, ACM, ISTE, IAENG
Indian Institute of Technology, Dhanbad
Email: drramesh@iitism.ac.in