have the same microprocessor core. The 21264 has a much more complex execution engine, out-of-order instead of in-order (cf. Chapter 3). Because of this change in instruction execution, the designers felt that longer access times for L2 could be hidden better than in the 21164. Therefore, the chip has larger L1 caches and no L2 cache. In the successor to the 21264 (the 21364, which was never commercially produced because of the demise of DEC), the intent was to keep the 21264 execution model but to return to the cache hierarchy philosophy of the 21164, namely, small L1s and a very large L2 on chip. In the case of the Pentium, the L2 of Pentium II was on a separate chip, but it was "glued" to the processor chip, thus allowing fast access. In the Pentium 4, the I-cache has been replaced by a *trace cache* that has a capacity equivalent to that of an 8 KB I-cache. The design challenges and the advantages brought forth by trace caches will be discussed in Chapter 4.

Finally, we should remember that while we have concentrated our discussion on h and T_{cache}, we have not talked about the parameter T_{mem}, the time to access memory and send the data back to the cache (in case of a cache read miss). T_{mem} can be seen as the sum of two components: the time T_{acc} to access main memory, that is, the time to send the missing line address and the time to read the memory block in the DRAM buffer, plus the time T_{tra} to transfer the memory block from the DRAM to the L1 and L2 caches and the appropriate register. T_{tra} itself is the product of the bus cycle time T_{bus} and the number of bus cycles needed, the latter being L/w, where L is the (L2) line size and w is the bus width. In other words,

$$T_{mem} = T_{acc} + (L/w)T_{bus}$$

EXAMPLE 4 Assume a main memory and bus such that $T_{acc} = 5$, $T_{bus} = 2$, and $w = 64$ bits. For a given application, cache C1 has a hit ratio $h_1 = 0.88$ with a line size $L_1 = 16$ bytes. Cache C2 has a hit ratio $h_2 = 0.92$ with a line size $L_2 = 32$ bytes. The cache access time in both cases is 1 cycle. The average memory access times for the two configurations are

$$Mem1 = 0.88 \times 1 + (1 - 0.88)(5 + 2 \times 2) = 0.88 + 0.12 \times 9 = 1.96$$

$$Mem2 = 0.92 \times 1 + (1 - 0.92)(5 + 4 \times 2) = 0.92 + 0.08 \times 13 = 1.96$$

In this (contrived) example, the two configurations have the same memory access times, although cache C2 would appear to have better performance. Example 4 emphasizes that a metric in isolation, here the hit ratio, is not sufficient to judge the performance of a system component.

2.3 Virtual Memory and Paging

In the previous sections we mostly considered 32-bit architectures, that is, the size of programmable registers was 32 bits. Memory references (which, in general, are generated as the sum of a register and a small constant) were also 32 bits wide. The addressing space was therefore 2^{32} bytes, or 4 GB. Today some servers have main memories approaching that size, and memory capacities for laptops or personal workstations are not far from it. However, many applications have data

sets that are larger than 4 GB. It is in order to be able to address these large data sets that we have now 64-bit ISAs, that is, the size of the registers has become 64 bits. The addressing space becomes 2^{64} bytes, a huge number. It is clear that this range of addresses is much too large for main memory capacities.

The disproportion between addressing space and main memory capacity is not new. In the early days of computing, a single program ran on the whole machine. Even so, there was often not enough main memory to hold the program and its data for the whole run. Programs and data were statically partitioned into *overlays* so that parts of the program or data that were not used at the same time could share the same locations in main memory. Soon it became evident that waiting for I/O, a much slower process than in-core computations, was wasteful, and *multiprogramming* became the norm. With multiprogramming, more than one program is resident in main memory at the same time, and when the executing program needs I/O, it relinquishes the CPU to another program. From the memory management viewpoint, multiprogramming poses the following challenges:

- How and where is a program loaded in main memory?
- How does one program ask for more main memory, if needed?
- How is one program protected from another, for example, what prevents one program from accessing another program's data?

The basic answer is that programs are compiled and linked as if they could address the whole addressing space. Therefore, the addresses generated by the CPU are *virtual addresses* that will be translated into *real*, or *physical*, addresses when referencing the physical memory. In early implementations, base and length registers were used, the physical address being the sum of the base register and the virtual address. Programs needed to be in contiguous memory locations, and an exception was raised whenever a physical address was larger than the sum of the base and length registers. In the early 1960s, computer scientists at the University of Manchester introduced the term *virtual memory* and performed the first implementation of a *paging system*. In the next section are presented the salient aspects of paging that influence the architecture of microprocessor systems. Many issues in paging system implementations are in the realm of operating systems and will only be glanced over in this book. All major O.S. textbooks give abundant details on this subject.

2.3.1 Paging Systems

The most common implementation of virtual memory uses paging. The virtual address space is divided into chunks of the same size called (virtual) *pages*. The physical address space, (i.e., the space used to address main memory), is also divided into chunks of the same size, often called physical pages or *frames*. The mapping between pages and frames is fully associative, that is, totally general. In other words, this is a relocation mechanism whereby any page can be stored in any frame and the contiguity restriction enforced by the base and length registers is avoided.

Before proceeding to a more detailed presentation of paging, it is worthwhile to note that the division into equal chunks is totally arbitrary. The chunks could

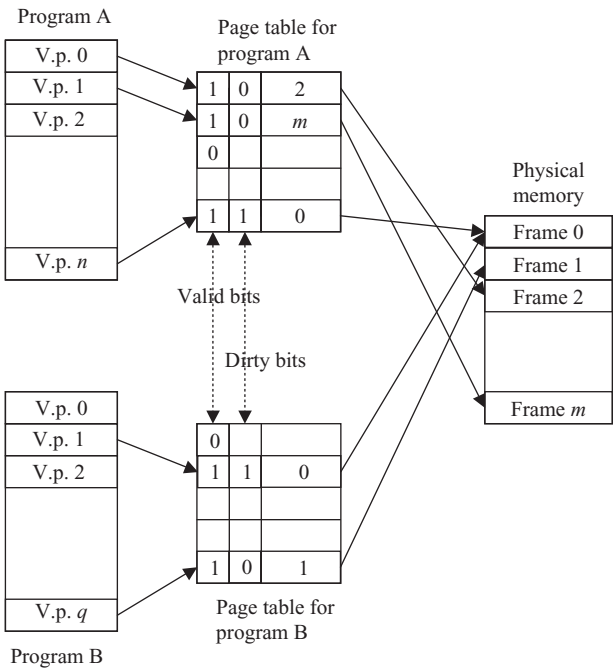

Figure 2.19. Paging system overview. Note: (1) in general n,q are much larger than m; (2) not all of programs A and B are in main memory at a given time, (3) page n of program A and page 1 of program B are shared.

be *segments* of any size. Some early virtual memory systems, such as those present in the Burroughs systems of the late 1960s and 1970s, used segments related to semantic objects: procedures, arrays, and so on. Although this form of segmentation is intellectually more appealing, and even more so now with object-oriented languages, the ease of implementation of paging systems has made them the invariable choice. Today, segments and *segmentation* refer to sets of pages most often grouped by functions, such as code, stack, or *heap* (as in the Intel IA-32 architecture).

Figure 2.19 provides a general view of paging. Two programs A and B are concurrently partially resident in main memory. A mapping device, or *page table* – one per program – indicates which pages of each program are in main memory and gives their corresponding frame numbers. The translation of virtual page number to physical frame number is kept in a *page table entry* (PTE) that, in addition, contains a *valid bit* indicating whether the mapping is current or not, and a *dirty bit* to show whether the page has been modified since it was brought into main memory. The figure shows already four advantages of virtual memory systems:

- The virtual address space can be much larger than the physical memory.
- Not all of the program and its data need to be in main memory at a given time.
- Physical memory can be shared between programs (multiprogramming) without much *fragmentation* (fragmentation is the portion of memory that is allocated and unused because of gaps between allocatable areas).
- Pages can be shared among programs (this aspect of paging is not covered in this book; see a book on operating systems for the issues involved in page sharing).

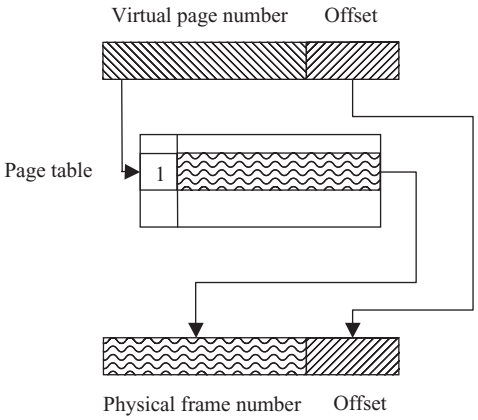

Figure 2.20. Virtual address translation.

When a program executes and needs to access memory, the virtual address that is generated is translated, through the page table, into a physical address. If the valid bit of the PTE corresponding to the virtual page is on, the translation will yield the physical address. Pages sizes are always a power of 2, (e.g., 4 or 8 KB), and naturally physical frames have the same size. A virtual address therefore consists of a virtual page number and an offset, that is, the location within the page (the offset is akin to the displacement in a cache line). Similarly, the physical address will have a physical frame number and the same offset as the virtual address, as illustrated in Figure 2.20. Note that it is not necessary to have virtual and physical addresses be of the same length. Many 64-bit architectures have a 64-bit virtual address but limit the physical address to 40 or 48 bits.

When the PTE corresponding to the virtual page number has its valid bit off, the virtual page is not in main memory. We have a *page fault*, that is, an exception. As we saw in Section 2.1.4, the current program must be suspended and the O.S. will take over. The page fault handler will initiate an I/O read to bring the whole virtual page from disk. Because this I/O read will take several milliseconds, time enough to execute billions of instructions, a *context switch* will occur. After the O.S. saves the process state of the faulting program and initiates the I/O process to handle the page fault, it will give control of the CPU to another program by restoring the process state of the latter.

As presented in Figure 2.20, the virtual address translation requires access to a page table. If that access were to be to main memory, the performance loss would be intolerable: an instruction fetch or a load–store would take tens of cycles. A possible solution would be to cache the PTEs. Although this solution has been implemented in some research machines, the usual and almost universal solution is to have special caches devoted to the translation mechanism with a design tailored for that task. These caches are called *translation look-aside buffers* (TLBs), or sometimes simply translation buffers.

TLBs are organized as caches, so a TLB entry consists of a tag and "data" (in this case a PTE entry). In addition to the valid and dirty bits that we mentioned previously, the PTE contains other bits, related to protection and to recency of access.

Table 2.2. Page sizes and TLB characteristics of two microprocessor families. Recent implementations support more than one page size. In some cases, there are extra TLB entries for a large page size (e.g., 4 MB) used for some scientific or graphic applications

Architecture	Page size	I-TLB	D-TLB
Alpha 21064	8 KB	8 entries (FA)	32 entries (FA)
Alpha 21164	8 KB	48 entries (FA)	64 entries (FA)
Alpha 21264	8 KB	64 entries (FA)	128 entries (FA)
Pentium	4 KB	32 entries (4-way)	64 entries (4-way)
Pentium II	4 KB	32 entries (4-way)	64 entries (4-way)
Pentium III	4 KB	32 entries (4-way)	64 entries (4-way)
Pentium 4	4 KB	64 entries (4-way)	128 entries (4-way)
Core Duo	4 KB	64 entries (FA)	64 entries (FA)

Because TLBs need to cache only a limited number of PTEs, their capacities are much smaller than those of ordinary caches. As a corollary, their associativities can be much greater. Typical TLB sizes and associativities are given in Table 2.2 for the same two families of microprocessors as in Table 2.1. Note that there are distinct TLBs for instruction and data, that the sizes range from 8 to 128 entries with either four-way (Intel) or full associativity (Alpha). Quite often, a fixed set of entries is reserved for the O.S. TLBs are writeback caches: the only information that can change is the setting of the dirty and recency of access bits.

The memory reference process, say for a load instruction, is illustrated in Figure 2.21. After the ALU generates a virtual address, the latter is presented to the TLB.

In the case of a TLB hit with the valid bit of the corresponding PTE on and no protection violation, the physical address is obtained as a concatenation of a field in the PTE and the offset. At the same time, some recency bits in the TLB's copy of the PTE can be modified. In the case of a store, the dirty bit will be turned on. The physical address is now the address seen by the cache, and the remainder of the memory reference process proceeds as explained in Section 2.2. If there is a TLB hit and the valid bit is off, we have a page fault exception. If there is a hit and the valid bit is on but there is a protection violation (e.g., the page that is being read can only be executed), we have an access violation exception, that is, the O.S. will take over.

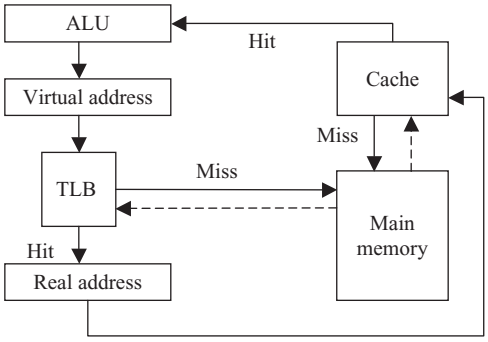

Figure 2.21. Abstracted view of the memory reference process.

In the case of a TLB miss, the page table stored in main memory must be accessed. Depending on the implementation, this can be done entirely in hardware, or entirely in software, or in a combination of both. For example, in the Alpha family a TLB miss generates the execution of a *Pal* (privileged access library) routine resident in main memory that can load a PTE from a special hardware register into the TLB. In the Intel Pentium architecture, the hardware walks the hierarchically stored page table until the right PTE is found. Replacement algorithms are LRU for the four-way set-associative Intel, and not-most-recently-used for the fully associative TLBs of the Alpha. Once the PTE is in the TLB, after an elapsed time of 100–1000 cycles, we are back to the case of a TLB hit. The rather rapid resolution of a TLB miss does not warrant a context switch (we'll see in Chapter 8 that this may not be the case for multithreaded machines). On the other hand, even with small TLB miss rates, the execution time can be seriously affected. For example, with a TLB miss rate of 0.1 per 1000 instructions and a TLB miss rate resolution time of 1000 cycles, the contribution to *CPI* is 0.1, the same as would arise from an L2 miss rate 10 times higher if the main memory latency were 100 cycles. This time overhead is one of the reasons that many architectures have the capability of several page sizes. When applications require extensive memory, as for example in large scientific applications or in graphics, the possibility of having large page sizes increases the amount of address space mapping that can be stored in the TLB. Of course, this advantage must be weighed against the drawback of memory fragmentation.

In the case of a page fault we have an exception and a context switch. In addition to the actions taken by the O.S. that are briefly outlined below, we must take care of the TLB, for its entries are reflecting the mapping of the process that is relinquishing the processor, not that of the incoming process. One solution is to invalidate the TLB (do not confuse this invalidation with turning off the valid bit in a PTE entry used to indicate whether the corresponding page is in main memory or not). A better solution is to append a *process ID number* (PID) as an extension to the tag in the TLB. The O.S. sets the PID when a program starts executing for the first time or, if it has been swapped out entirely, when it is brought back in. PIDs are recycled when there is an overflow on the count of allowable PIDs. There is a PID register holding the PID of the current executing process, and the detection of a hit or miss in the TLB includes the comparison of the PID in this register with the tag extension. On context switch, the PID of the next-to-execute process is brought into the PID register, and there is no need to invalidate the TLB. When a program terminates or is swapped out, all entries in the TLB corresponding to its PID number are invalidated.

A detailed description of the page fault handler is outside the scope of this book. In broad terms, the handler must (these actions are not listed in order of execution):

- Reserve a frame from a free list maintained by the O.S. for the faulting page.
- Find out where the faulting page resides on disk (this disk address can be an extension to the PTE).
- Invalidate portions of the TLB and maybe of the cache (cf. Section 2.2.1).

- Initiate a read for the faulting page.
- Find the page(s) to replace (using some replacement algorithm) if the list of free pages is not long enough (a tuning parameter of the O.S.). In general, the replacement algorithm is an approximation to LRU.
- Perform cache purges and/or invalidations for cache lines mapping to the page(s) to be replaced.
- Initiate a write to the disk of dirty replaced page(s).

When the faulting page has been read from the disk, an I/O interrupt will be raised. The O.S. will take over and, among other actions, will update the PTE of the page just brought in memory.

Figure 2.21 implies that the TLB access must precede the cache access. Optimizations allow this sequentiality to be relaxed. For example, if the cache access depends uniquely on bits belonging to the page offset, then the cache can be indexed while the TLB access is performed. We shall return to the topic of *virtually indexed* caches in Chapter 6. Note that the larger the page size, the longer the offset and therefore the better the chance of being able to index the cache with bits that do not need to be translated. This observation leads us to consider the design parameters for the selection of a page size.

Most page sizes in contemporary systems are the standard 4 or 8 KB with, as mentioned earlier, the possibility of having several page sizes. This choice is a compromise between various factors, namely, the I/O time to read or write a page from or to disk, main memory fragmentation, and translation overhead.

The time t_x to read (write) a page of size x from (to) disk is certainly smaller than the time t_{2x} to transfer a page of size $2x$, but $t_{2x} < 2t_x$. This is because t_x is the sum of three components, all of which are approximately the same: *seek time*, that is, the time it takes for the disk arm to be positioned on the right track; *rotation time*, that is, the time it takes for the read–write head to be positioned over the right sector; and *transfer time*, that is, the time it takes to transfer the information that is under the read–write head to main memory. The seek time, which ranges from 0 (if the arm is already on the right track) up to 10 ms, is independent of the page size. Similarly, the rotation time, which is on the average half of the time it takes for the disk to perform a complete rotation, is also independent of the page size. Like the seek time, it is of the order of a few milliseconds: 3 ms for a disk rotating at 10,000 rpm. There remains the transfer time, which is proportional to the page size and again can be a few milliseconds. Thus, having large page sizes amortizes the I/O time when several consecutive pages need to be read or written, because although the transfer time increases linearly with page size, the overhead of seek and rotation times is practically independent of the page size.

Another advantage of large page sizes is that page tables will be smaller and TLB entries will map a larger portion of the addressing space. Smaller page tables mean that a greater proportion of PTEs will be in main memory, thus avoiding the double jeopardy of more than one page fault for a given memory reference (see the exercises). Mapping a larger address space into the TLB will reduce the number of

Table 2.3. Two extremes in the memory hierarchy

	L1 cache	Paging System
Line or page size	16–64 bytes	4–8 KB
Miss or fault time	5–100 cycles 5–100 ns	Millions of cycles 3–20 ms
Miss or fault rate	0.1–0.01	0.0001–0.000001
Memory size	4–64 KB	A few gigabytes (physical) 2^{64} bytes (virtual)
Mapping	Direct-mapped or low associativity	Full generality
Replacement algorithm	Not important	Very important

TLB misses. However, page sizes cannot become too large, because their overall transfer time would become prohibitive and in addition there would be significant memory fragmentation. For example, if, as is the custom, the program's object code, the stack, and the heap each start on page boundaries, then having very large pages can leave a good amount of memory unused.

Cache and TLB geometries as well as the choice of page sizes are arrived at after engineering compromises. The parameters that are used in the design of memory hierarchies are now summarized.

2.3.2 Memory Hierarchy Performance Assessment

We can return now to the four questions we asked at the beginning of Section 2.2.1 regarding cache design and ask them for the three components of the memory hierarchy that we have considered: cache, TLB, and main memory (paging). Although the answers are the same, the implementations of these answers vary widely, as do the physical design properties, as shown in two extremes in Table 2.3. To recap:

1. When do we bring the contents of a missing item into the (cache, TLB, main memory)?

The answer is "on demand." However, in the case of a cache, misses can occur a few times per 100 memory references and the resolution of a cache miss is done entirely in hardware. Miss resolution takes of the order of 5–100 cycles, depending on the level of the memory hierarchy where the missing cache line (8–128 bytes) is found. In the case of a TLB, misses take place two orders of magnitude less often, a few times per 10,000 instructions. TLB miss resolution takes 100–1000 cycles and can be done in either hardware or software. Page faults are much rarer and occur two or three orders of magnitude less often than TLB misses, but page fault resolution takes millions of cycles to resolve. Therefore, page faults will result in context switches.

2. Where do we put the missing item?

In the case of a cache, the mapping is quite restrictive (direct mapping or low set associativity). In the case of a paging system, the mapping is totally general (we shall see some possible exception to this statement when we look at *page coloring* in Chapter 6). For TLBs, we have either full generality or high set associativity.

3. How do we know it's there?

Caches and TLB entries consist of a tag and "data." Comparisons of part of the address with the tag yield the hit or miss knowledge. Page faults are detected by indexing page tables. The organization of page tables (hierarchical, inverted, etc.) is outside the scope of this book.

4. What happens if the place where the missing item is supposed to go is already occupied?

A replacement algorithm is needed. For caches and TLBs, the general rule is (approximation to) LRU with an emphasis on not replacing the most recently used cache line or TLB PTE entry. For paging systems, good replacement algorithms are important, because the page fault rate must be very low. O.S. books have extensive coverage of this important topic. Most operating systems use policies where the number of pages allocated to programs varies according to their working set.

Memory hierarchies have been widely studied because their performance has great influence on the overall execution time of programs. Extensive simulations of paging systems and of cache hierarchies have been done, and numerous results are available. Fortunately, simulations that yield hit rates are much faster than simulations to assess *IPC* or execution times. The speed in simulation is greatly helped by a property of certain replacement algorithms such as LRU, called the *stack property*. A replacement algorithm has the stack property if, for (say) a paging system, the number of page faults for a sequence of memory references for a memory allocation of x pages is greater than or equal to the number of page faults for a memory of size $x + 1$. In other words, the number of page faults is a monotonically nonincreasing function of the amount of main memory. Therefore, one can simulate a whole range of memory sizes in a single simulation pass. For cache simulations, the stack property will hold set by set. Other means of reducing the overall simulation time of cache hierarchies and generalizing the results to other metrics and other geometries include mechanisms such as filtering the reference stream in a preprocessing phase and applying clever algorithms for multiple associativities.

Finally, while LRU is indeed an excellent replacement algorithm, it is not optimal. The optimal algorithm for minimizing the number of page faults (or cache misses on a set by set basis) is due to Belady. Belady's algorithm states that the page to be replaced is the one in the current resident set that will be accessed further in the future. Of course, this optimal algorithm is not realizable in real time, but the optimal number of faults (or misses) can be easily obtained in simulation and thus provide a measure of the goodness of the replacement algorithm under study.