LECTURA 1
EQUIPO 1

2 The Basics

This chapter reviews features that are found in all modern microprocessors: (i) instruction pipelining and (ii) a main memory hierarchy with caches, including the virtual-to-physical memory translation. It does not dwell on many details – that is what subsequent chapters will do. It provides solely a basis on which we can build later on.

← INICIO

2.1 Pipelining

Consider the steps required to execute an arithmetic instruction in the von Neumann machine model, namely:

1. Fetch the (next) instruction (the one at the address given by the program counter).
2. Decode it.
3. Execute it.
4. Store the result and increment the program counter.

In the case of a load or a store instruction, step 3 becomes two steps: calculate a memory address, and activate the memory for a read or for a write. In the latter case, no subsequent storing is needed. In the case of a branch, step 3 sets the program counter to point to the next instruction, and step 4 is voided.

Early on in the design of processors, it was recognized that complete sequentiality between the executions of instructions was often too restrictive and that parallel execution was possible. One of the first forms of parallelism that was investigated was the overlap of the mentioned steps between consecutive instructions. This led to what is now called *pipelining*.[1]

[1] In early computer architecture texts, the terms *overlap* and *look-ahead* were often used instead of *pipelining*, which was used for the pipelining of functional units (cf. Section 2.1.6).

2.1.1 The Pipelining Process

In concept, pipelining is similar to an assembly line process. Jobs A, B, and so on, are split into n sequential subjobs $A_1, A_2, \ldots, A_n$ ($B_1, B_2, \ldots, B_n$, etc.) with each A_i (B_i, etc.) taking approximately the same amount of processing time. Each subjob is processed by a different station, or equivalently the job passes through a series of *stages*, where each stage processes a different A_i. Subjobs of different jobs overlap in their execution: when subjob A_1 of job A is finished in stage 1, subjob A_2 will start executing in stage 2 while subjob B_1 of job B will start executing in stage 1. If t_i is the time to process A_i and $t_M = \max_i t_i$, then in steady state one job completes every t_M. Throughput (the number of instructions executed per unit time) is therefore enhanced. On the other hand, the latency (total execution time) of a given job, say L_A for A, becomes

$$L_A = nt_M, \quad \text{which may be greater than } \sum_{i=1}^{n} t_i.$$

Before applying the pipelining concept to the instruction execution cycle, let us look at a real-life situation. Although there won't be a complete correspondence between this example and pipelining as implemented in contemporary processors, it will allow us to see the advantages and some potential difficulties brought forth by pipelining.

Assume that you have had some friends over for dinner and now it's time to clean up. The first solution would be for you to do all the work: bringing the dishes to the sink, scraping them, washing them, drying them, and putting them back where they belong. Each of these five steps takes the same order of magnitude of time, say 30 seconds, but bringing the dishes is slightly faster (20 seconds) and storing them slightly slower (40 seconds). The time to clean one dish (the latency) is therefore 150 seconds. If there are 4 dishes per guest and 8 guests, the total cleanup time is $150 \times 4 \times 8 = 4800$ seconds, or 1 hour and 20 minutes. The throughput is 1 dish per 150 seconds. Now, if you enlist four of your guests to help you and, among the five of you, you distribute the tasks so that one person brings the dishes, one by one, to the second, who scrapes them and who in turn passes them, still one by one, to the washer, and so on to the dryer and finally to the person who stores them (you, because you know where they belong). Now the latency is that of the longest stage (storing) multiplied by the number of stages, or $40 \times 5 = 200$ seconds. However, the throughput is 1 dish per longest stage time, or 1 dish per 40 seconds. The total execution time is $40 \times 8 \times 4 + 4 \times 40 = 1360$ seconds, or a little less than 23 minutes. This is an appreciable savings of time.

Without stretching the analogy too far, this example highlights the following points about pipelining:

- In order to be effective, the pipeline must be *balanced*, that is, all stages must take approximately the same time. It makes no sense to optimize a stage whose processing time is not the longest. For example, if drying took 25 seconds instead of 30, this would not change the overall cleanup time.

- A job must pass through all stages, and the order is the same for all jobs. Even if a dish does not need scraping, it must pass through that stage (one of your friends will be idle during that time).
- *Buffering* (holding a dish in its current partially processed state) between stages is required, because not all stages take exactly the same time.
- Each stage must have all the resources it needs allocated to it. For example, if you have only one brush and it is needed some of the time, by both the scraper and the washer there will be some stalling in the pipeline. This situation is a form of (structural) *hazard*.
- The pipeline may be disrupted by some internal event (e.g., someone drops a dish and it breaks) or some external event (one of your coworkers is called on the phone and has to leave her station). In these cases of *exception* or *interrupt*, the pipeline must be flushed and the state of the process must be saved so that when the exception or interrupt is removed, the operation can start anew in a consistent state.

Other forms of hazards exist in processor pipelines that do not fit well in our example: they are *data hazards* due to dependencies between instructions and *control hazards* caused by transfers of control (branches, function calls). We shall return to these shortly.

2.1.2 A Basic Five-stage Instruction Execution Pipeline

Our presentation of the instruction execution pipeline will use a RISC processor as the underlying execution engine. The processor in question consists of a set of registers (the register file), a program counter PC, a (pipelined) CPU, an instruction cache (I-cache), and a data cache (D-cache). The caches are backed up by a memory hierarchy, but in this section we shall assume that the caches are perfect (i.e., there will be no cache misses). For our purposes in this section, the state of a running process is the contents of the registers and of the program counter PC. Each register and the PC are 32 bits long (4 bytes, or 1 word).

Recall that a RISC processor is a load–store architecture, that is, all instructions except those involving access to memory have register or immediate operands. The instructions are of same length (4 bytes in our case) and can be of one of three types:

- Arithmetic–logical instructions, of the form $R_i \leftarrow R_j op R_k$ (one of the source registers R_j or R_k can be replaced by an immediate constant encoded in the instruction itself).
- Load–store instructions, of the form $R_i \leftarrow Mem[R_j + disp]$ or $Mem[R_j + disp] \leftarrow R_i$.
- Control instructions such as (conditional) branches of the form $br\ (R_j op R_k)$ *displ*, where a taken branch (R_j *op* R_k is true) sets the PC to its current value plus the *displ* rather than having the PC point to the next sequential instruction.

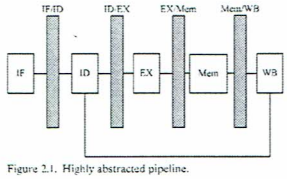

Figure 2.1. Highly abstracted pipeline.

As we saw at the beginning of this chapter, the instruction that requires the most steps is a load instruction, which requires five steps. Each step is executed on a different stage, namely:

1. Instruction fetch (IF). The instruction is fetched from the memory (I-cache) at the address indicated by the PC. At this point we assume that the instruction is not a branch, and we can increment the PC so that it will point to the next instruction in sequence.
2. Instruction decode (ID). The instruction is decoded, and its type is recognized. Some other tasks, such as the extension of immediate constants into 32 bits, are performed. More details are given in the following.
3. Execution (EX). In the case of an arithmetic instruction, an ALU performs the arithmetic or logical operation. In the case of a load or store instruction, the address $addr = R_j + disp$ is computed (*disp* will have been extended to 32 bits in the ID stage). In the case of a branch, the PC will be set to its correct value for the next instruction (and other actions might be taken, as will be seen in Section 2.1.4).
4. Memory access (Mem). In the case of a load, the contents of $Mem[addr]$ are fetched (from the D-cache). If the instruction is a store, the contents of that location are modified. If the instruction is neither a load nor a store, nothing happens during that stage, but the instruction, unless it is a branch, must pass through it.
5. Writeback (WB). If the instruction is neither a branch nor a store, the result of the operation (or of the load) is stored in the result register.

In a highly abstracted way, the pipeline looks like Figure 2.1. In between each stage and the next are the pipeline registers, which are named after the left and right stages that they separate. A pipeline register stores all the information needed for completion of the execution of the instruction after it has passed through the stage at its left.

If we assume that accessing the caches takes slightly longer than the operations in the three other stages, that is, that the cache access takes 1 cycle, then a snapshot

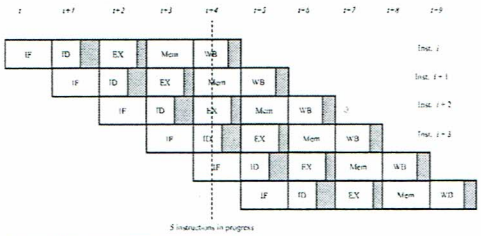

Figure 2.2. Snapshot of sequential program execution.

of the execution of a sequential program (no branches) is shown in Figure 2.2 (the shaded parts indicate that some stages are shorter than the IF and Mem stages).

As can be seen, as soon as the pipeline is full (time $t+4$), five instructions are executing concurrently. A consequence of this parallelism is that resources cannot be shared between stages. For example, we could not have a single cache unified for instruction and data, because instruction fetches, that is, accesses to the I-cache during the IF stage, occur every cycle and would therefore interfere with a load or a store (i.e., with access to the D-cache during the Mem stage). This interference would be present about 25% of the time (the average frequency of load–store operations). As mentioned earlier, some stage might be idle. For example, if instruction $i+1$ were an add, then at time $t+4$ the only action in the Mem stage would be to pass the result computed at time $t+3$ from the pipeline register EX/Mem where it was stored to the pipeline register Mem/WB.

It is not the intention to give a detailed description of the implementation of the pipeline. In order to do so, one would need to define more precisely the ISA of the target architecture. However, we shall briefly consider the resources needed for each stage and indicate what needs to be stored in the respective pipeline registers. In this section, we only look at arithmetic–logical and load–store instructions. We will look at control instructions in Section 2.1.4.

The first two stages, fetch (IF) and decode (ID), are common to all instructions. In the IF stage, the next instruction, whose address is in the PC, is fetched, and the PC is incremented to point to the next instruction. Both the instruction and the incremented PC are stored in the IF/ID register. The required resources are the I-cache and an adder (or counter) to increment the PC. In the ID stage, the opcode of the instruction found in the IF/ID register is sent to the unit that controls the settings of the various control lines that will activate selected circuits or registers and cache read or write in the subsequent three stages. With the presence of this control unit (implemented, for example, as a programmable logic array (PLA)),

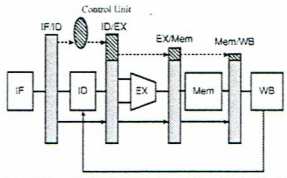

Figure 2.3. Abstracted view of the pipeline with the control unit.

the abstracted view of the pipeline becomes that of Figure 2.3. In addition to the opcode decoding performed by the control unit, all possible data required in the forthcoming stages, whether the instruction is an arithmetic–logical one or a load–store, are stored in the ID/EX register. This includes the contents of source registers, the extension to 32 bits of immediate constants and displacements (a sign extender is needed), the name of the potential result register, and the setting of control lines. The PC is also passed along.

After the IF and ID stages, the actions in the three remaining stages depend on the instruction type. In the EX stage, either an arithmetic result is computed with sources selected via settings of adequate control lines, or an address is computed. The main resource is therefore an ALU, and an ALU symbol for that stage is introduced in Figure 2.3. For both types of instruction the results are stored in the EX/Mem register. The EX/Mem register will also receive from the ID/EX register the name of the result register, the contents of a result register in the case of a store instruction, and the settings of control lines for the two remaining stages. Although passing the PC seems to be unnecessary, it is nonetheless stored in the pipeline registers. The reason for this will become clear when we deal with exceptions (Section 2.1.4). In the Mem stage, either a read (to the D-cache) is performed if the instruction is a load, or a write is performed if the instruction is a store, or else nothing is done. The last pipeline register, Mem/WB, will receive the result of an arithmetic operation, passed directly from the EX/Mem register, or the contents of the D-cache access, or nothing (in the case of a store), and, again, the value of the PC. In the WB stage the result of the instruction, if any, is stored in the result register. It is important to note that the contents of the register file are modified only in the last stage of the pipeline.

FIN

2.1.3 Data Hazards and Forwarding

In Figure 2.2, the pipeline is ideal. A result is generated every cycle. However, such smooth operation cannot be sustained forever. Even in the absence of exceptional conditions, three forms of hazards can disrupt the functioning of the pipeline. They are:

4. *Normalize and round off.* Shift the result mantissa right by one bit if there was a carry-out in the preceding step, and decrease the result exponent by 1. Otherwise, shift left until the most significant bit is a 1, increasing the result exponent by 1 for each bit shifted. Suppress the leftmost 1 for the result mantissa. Round off.

Step 3 (add mantissas) is a little more complex than it appears. Not only does a carry-out need to be detected when both operands are of the same sign, but also, when $D = 0$ and the two operands are of different signs, the resulting mantissa may be of the wrong sign. Its 2's complement must then be taken, and the sign of the result must be flipped.

This algorithm lends itself quite well to pipelining in three (or possibly four) stages: compare exponents, shift and add mantissas (in either one or two stages), and normalize and round off.

The algorithm for multiplication is straightforward:

1. *Add exponents.* The resulting exponent is $E = E_1 + E_2 - 127$ (we need to subtract one of the two biases). The resulting sign is positive if $S_1 = S_2$ and negative otherwise.
2. *Prepare and multiply mantissas.* The preparation simply means inserting 1's at the left of F_1 and F_2. We don't have to worry about signs.
3. *Normalize and round off.* As in addition, but without having to worry about a potential right shift.

Because the multiplication takes longer than the other two steps, we must find a way to break it into stages. The usual technique is to use a Wallace tree of carry–save adders (CSAs).[4] If the number of CSAs is deemed too large, as for example for double-precision f-p arithmetic, a feedback loop must be inserted in the "tree." In that case, consecutive multiplications cannot occur on every cycle, and the delay depends on the depth of the feedback.

With the increase in clock frequencies, the tendency will be to have deeper pipelines for these operations, because less gate logic can be activated in a single cycle. The optimal depth of a pipeline depends on many microarchitectural factors besides the requirement that stages must be separated by stable state components, that is, the pipeline registers. We shall return briefly to this topic in Chapter 9.

2.2 Caches

In our pipeline design we have assumed that the IF and Mem stages were always taking a single cycle. In this section, we take a first look at the design of the memory hierarchy of microprocessor systems and, in particular, at the various organizations

[4] CSAs take three inputs and generate two outputs in two levels of AND–OR logic. By using a logsum process we can reduce the number of operands by a factor 2/3 at each level of the tree. For a multiplier and a multiplicand of n bits each, we need $n - 2$ CSAs before proceeding to the last regular addition with a carry-lookahead adder. Such a structure is called a Wallace tree.

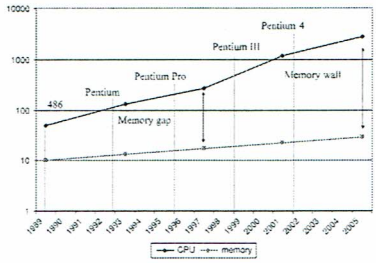

Figure 2.10. Processor–memory performance gap (note the logarithmic scale).

and performance of caches, the highest levels in the hierarchy. Our assumption was that caches were always perfect. Naturally, this is not the reality.

In early computers, there was only one level of memory, the so-called primary memory. The secondary memory, used for permanent storage, consisted of rotating devices, such as disks, and some aspects of it are discussed in the next section. Nowadays primary memory is a hierarchy of components of various speeds and costs. This hierarchy came about because of the rising discrepancy between processor speeds and memory latencies. As early as the late 1960s, the need for a memory buffer between high-performance CPUs and main memory was recognized. Thus, caches were introduced. As the gap between processor and memory performance increased, multilevel caches became the norm. Figure 2.10 gives an idea of the processor and memory performance trends during the 1990s. While processor speeds increased by 60% per year, memory latencies decreased by only 7% per year. The ratio of memory latency to cycle time, which was about 5:1 in 1990, became more than one order of magnitude in the middle of the decade (the *memory gap*) and more than two orders of magnitude by 2000 (the *memory wall*). Couched in terms of latencies, the access time for a DRAM (main memory) is of the order of 40 ns, that is, 100 processor cycles for a 2.5 GHz processor.

The goal of a memory hierarchy is to keep close to the ALU the information that is needed presently and in the near future. Ideally, this information should be reachable in a single cycle, and this necessitates that part of the memory hierarchy, a cache, be on chip with the ALU. However, the capacity of this on-chip SRAM memory must not be too large, because the time to access it is, in a first approximation, proportional to its size. Therefore, modern computer systems have multiple levels of caches. A typical hierarchy is shown in Figure 2.11 with two levels of

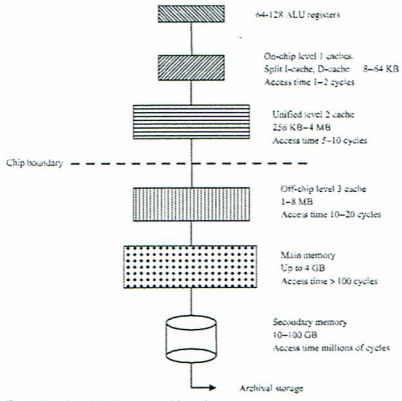

Figure 2.11. Levels in the memory hierarchy.

on-chip caches (SRAM) and one level of off-chip cache (often a combination of SRAM and DRAM), and main memory (DRAM).

Note that the concept of caching is used in many other aspects of computer systems: disk caches, network server caches, Web caches, and so on. Caching works because information at all these levels is not accessed randomly. Specifically, computer programs exhibit the *principle of locality*, consisting of *temporal locality*, whereby data and code used in the past are likely to be reused in the near future (e.g., data in stack, code in loops), and *spatial locality*, whereby data and code close (in terms of memory addresses) to the data and code currently referenced will be referenced again in the near future (e.g., traversing a data array, straight-line code sequences).

Caches are much smaller than main memory, so, at a given time, they cannot contain all the code and data of the executing program. When a memory reference is generated, there is a lookup in the cache corresponding to that reference: the I-cache for instructions, and the D-cache for data. If the memory location is mapped in the cache, we have a *cache hit*; otherwise, we have a *cache miss*. In the case of a hit, the content of the memory location is loaded either in the IF/EX pipeline register in the case of an instruction, or in the Mem/WB register in the case of a load, or else the

content of the cache is modified in the case of a store. In the case of a miss, we have to recursively probe the next levels of the memory hierarchy until the missing item is found. For ease of explanation, in the remainder of this section we only consider a single-level cache unless noted otherwise. In addition, we first restrict ourselves to data cache reads (i.e., loads). Reads for I-caches are similar to those for D-caches, and we shall look at writes (i.e., stores) separately.

2.2.1 Cache Organizations

Caches serve as high-speed buffers between main memory and the CPU. Because their storage capacity is much less than that of primary memory, there are four basic questions on cache design that need to be answered, namely:

1. When do we bring the content of a memory location into the cache?
2. Where do we put it?
3. How do we know it's there?
4. What happens if the cache is full and we want to bring the content of a location that is not cached? As we shall see in answering question 2, a better formulation is: "What happens if we want to bring the content of a location that is not cached and the place where it should go is already occupied?"

Some top-level answers follow. These answers may be slightly modified because of optimizations, as we shall see in Chapter 6. For example, our answer to the first question will be modified when we introduce prefetching. With this caveat in mind, the answers are:

1. The contents of a memory location are brought into the cache *on demand*, that is, when the request for the data results in a cache miss.
2. Basically the cache is divided into a number of cache entries. The mapping of a memory location to a specific cache entry depends on the *cache organization* (to be described).
3. Each cache entry contains its name, or *tag*, in addition to the data contents. Whether a memory reference results in a hit or a miss involves checking the appropriate tag(s).
4. Upon a cache miss, if the new entry conflicts with an entry already there, one entry in the cache will be replaced by the one causing the miss. The choice, if any, will be resolved by a replacement algorithm.

A generic cache organization is shown in Figure 2.12. A cache entry consists of an address, or tag, and of data. The usual terminology is to call the data content of an entry a *cache line* or a *cache block*.[5] These lines are in general several words (bytes) long; hence, a parameter of the cache is its *line size*. The overall cache capacity, or *cache size*, given in kilobytes (KB) or megabytes (MB), is the product of the number of lines and the line size, that is, the tags are not counted in the overall

[5] We will use *line* for the part of a cache entry and *block* for the corresponding memory image.

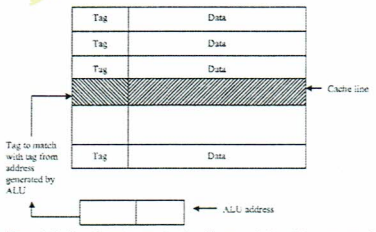

Figure 2.12. Generic cache organization. If (part of) the address generated by the ALU matches a tag, we have a hit; otherwise, we have a miss.

cache size. A good reason for discarding the tags in the terminology (but not in the actual hardware) is that from a performance viewpoint they are overhead, that is, they do not contribute to the buffering effect.

Mapping of a memory location, or rather of a sequence of memory words of line-size width that we will call a *memory block*, to a cache entry can range from full generality to very restrictive. If a memory block can be mapped to any cache entry, we have a *fully associative* cache. If the mapping is restricted to a single entry, we have a *direct-mapped* cache. If the mapping is to one of several cache entries, we have a *set-associative* cache. The number of possible entries is the set-associative *way*. Figure 2.13 illustrates these definitions.

Large fully associative caches are not practical, because the detection of a cache hit or miss requires that all tags be checked in the worst case. The linear search can be avoided with the use of *content-addressable memories* (CAMs), also called associative memories. In a CAM, instead of addressing the memory structure as an array (i.e., with an index), the addressing is by matching a key with the contents of (part of) all memory locations in the CAM. All entries can be searched in parallel. A matching entry, if any, will raise a flag indicating its presence. In the case of a fully associative cache, the key is a part of the address generated by the ALU, and the locations searched in parallel are the tags of all cache entries. If there is no match, we have a miss; otherwise, the matching entry (there can be only one in this case) indicates the position of the cache hit.

While conceptually quite attractive, CAMs have for main drawbacks that (i) they are much more hardware-expensive than SRAMs, by a factor of 6 to 1, (ii) they consume more power, and (iii) they are difficult to modify. Although fully associative hardware structures do exist in some cases, e.g., for write buffers as we shall see in this section, for TLBs as we shall see in the next section, and for other hardware data structures as we shall see in forthcoming chapters, they are in general very small, up to 128 entries, and therefore cannot be use for general-purpose caches.

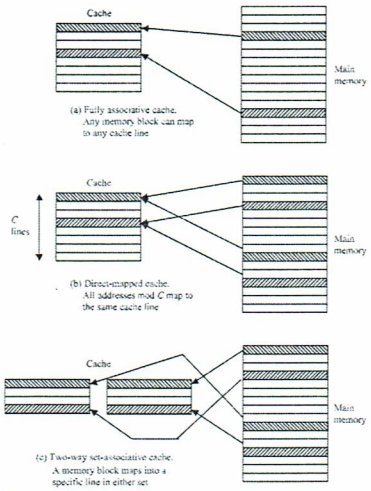

Figure 2.13. Cache organizations.

In the case of direct-mapped caches, the memory block will be mapped to a single entry in the cache. This entry is *memory block address mod C*, where *C* is the number of cache entries. Since in general-purpose microprocessors *C* is a power of 2, the entry whose tag must be checked is very easy to determine.

In the case of an *m*-way set-associative cache, the cache is divided into *m banks* with *C/m* lines per bank. A given memory block can be mapped to any of *m* entries, each of which is at address *memory block address mod C/m* in its bank. Detection of a cache hit or miss will require *m* comparators so that all comparisons can be performed concurrently.

A cache is therefore completely defined by three parameters: The number of cache lines, *C*; the line size *L*; and the associativity *m*. Note that a direct-mapped cache can be considered as one-way set-associative, and a fully associative cache as

C-way associative. Both *L* and *C/m* are generally powers[6] of 2. The cache size, or capacity, *S* is $S = C \times L$; thus either (C,L,m) or (S,L,m), a notation that we slightly prefer, can be given to define the *geometry* of the cache.

When the ALU generates a memory address, say a 32-bit byte address, how does the hardware check whether the (memory) *reference* will result into a cache hit or a cache miss? The memory address generated by the ALU will be decomposed into three fields: *tag*, *index*, and *displacement*. The *displacement d*, or *line offset*, indicates the low-order byte within a line that is addressed. Since there are *L* bytes per line, the number of bits needed for *d* is $d = \log_2 L$, and of course these bits are the least significant bits of the address. Because there are *C/m* lines per bank, the *index i* of the cache entry at which the *m* memory banks must be probed requires $i = \log_2(C/m)$ bits. The remaining *t* bits are for the tag. In the first-level caches that we are considering here, the *t* bits are the most significant bits.

EXAMPLE 3: Consider the cache (S,m,L) with $S = 32KB$, $m = 1$(direct-mapped), and $L = 16B$. The number of lines is $32 \times 1024/16 = 2048$. The memory reference can be seen as the triplet (t,i,d), where:

The displacement $d = \log L = \log 16 = 4$.
The index $i = \log(C/m) = \log 2048 = 11$.
The tag $t = 32 - 11 - 4 = 17$.

The hit–miss detection process for Example 3 is shown in Figure 2.14.

Let us now vary the line size and associativity while keeping the cache capacity constant and see the impact on the tag size and hence on the overall hardware requirement for the implementation of the cache. If we double the line size and keep the direct-mapped associativity, then *d* requires one more bit; the number of lines is halved, so *i* is decreased by 1; and therefore *t* is left unchanged. If we keep the same line size but look at two-way set associativity ($m = 2$), then *d* is left unchanged, *i* decreases by 1 because the number of lines per set is half of the original, and therefore *t* increases by 1. In addition, we now need two comparators for checking the hit or miss, or more generally one per way for an *m*-way cache, as well as a multiplexer to drive the data out of the right bank. Note that if we were to make the cache fully associative, the index bits would disappear, because $i = \log(C/C) = \log 1 = 0$.

From our original four questions, one remains, namely: What happens if on a cache miss all cache entries to which the missing memory block maps are already occupied? For example, assume a direct-mapped cache. Memory blocks *a* and *b* have addresses that are a multiple of *C* blocks apart, and *a* is already cached. Upon a reference to *b*, we have a cache miss. Block *b* should therefore be brought into the cache, but the cache entry where it should go is already occupied by block *a*. Following the principle of locality, which can be interpreted as favoring the most recent references over older ones, block *b* becomes cached and block *a* is evicted.

[6] *m* is not necessarily a power of 2. In that case, if *m'* is the smallest power of 2 larger than *m*, then there are *m* banks of *C/m'* lines.

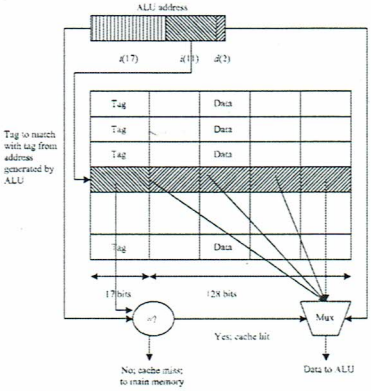

Figure 2.14. Hit and miss detection. The cache geometry is (32 KB,1,16).

In the case of an *m*-way set-associative cache, if all *m* cache entries with the same mapping as the block for which there is a miss are occupied, then a *victim* to be evicted must be selected. The victim is chosen according to a *replacement algorithm*. Following again the principle of locality, a good choice is to replace the line that has been not been accessed for the longest time, that is, the *least-recently used* (LRU). LRU replacement is easy to implement when the associativity *m* is small, for example $m \leq 4$. For the larger associativities that can happen in second- or third-level caches, approximations to LRU can be used (Chapter 6). As long as the (maybe two) most recently used (MRU) lines are not replaced, the influence of the replacement algorithm on cache performance is minimal. This will not be the case for the paging systems that we discuss in the next section.

Write Strategies

In our answers to the top-level questions we have assumed that the memory references corresponded to reads. When the reference is for a write, (i.e., a consequence of a store instruction), we are faced with additional choices. Of course, the steps taken to detect whether there is a cache hit or a cache miss remain the same.

In the case of a cache hit, we must certainly modify the contents of the cache line, because subsequent reads to that line must return the most up-to-date information. A first option is to write only in the cache. Consequently, the information in the cache might no longer be the image of what is stored in the next level of the memory hierarchy. In these *writeback* (or *copyback*) caches we must have a means to indicate that the contents of a line have been modified. We therefore associate a *dirty bit* with each line in the cache and set this bit on a write cache hit. When the line becomes a victim and needs to be replaced, the dirty bit is checked. If it is set, the line is written back to memory; if it is not set, the line is simply evicted. A second option is to write both in the cache and in the next level of the hierarchy. In these *write-through* (or *store-through*) caches there is no need for the dirty bit. The advantage of the writeback caches is that they generate less memory traffic. The advantage of the write-through caches is that they offer a consistent view of memory.

Consider now the case of a cache miss. Again, we have two major options. The first one, *write-allocate*, is to treat the write miss as a read miss followed by a write hit. The second, *write-around*, is to write only in the next level of the memory hierarchy. It makes a lot of sense to have either write-allocate writeback caches or write-around write-through caches. However, there are variations on these two implementations, and we shall see some of them in Chapter 6.

One optimization that is common to all implementations is to use a *write buffer*. In a write-through cache the processor has to wait till the memory has stored the data. Because the memory, or the next level of the memory hierarchy, can be tens to hundreds of cycles away, this wait is wasteful in that the processor does not need the result of the store operation. To circumvent this wait, the data to be written and the location where they should be written are stored in one of a set of temporary registers. The set itself, a few registers, constitutes the write buffer. Once the data are in the write buffer, the processor can continue executing instructions. Writes to memory are scheduled as soon as the memory bus is free. If the processor generates writes at a rate such that the write buffer becomes full, then the processor will have to stall. This is an instance of a structural hazard. The same concept can be used for writeback caches. Instead of generating memory writes as soon as a dirty replacement is mandated, the line to be written and its tag can be placed in the write buffer. The write buffer must be checked on every cache miss, because the required information might be in it. We can take advantage of the fact that each entry in the buffer must carry the address of where it will be written in memory, and transform the buffer into a small fully associative extra cache. In particular, information to be written can be coalesced or overwritten if it belongs to a line already in the buffer.

FIN

The Three C's

In order to better understand the effect of the geometry parameters on cache performance it is useful to classify the cache misses into three categories, called the three C's:

1. Compulsory (or cold) misses: This type of misses occurs the first time a memory block is referenced.

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_{trf} to transfer the memory block from the DRAM to the L1 and L2 caches and the appropriate register. T_{trf} 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.

INICIO

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

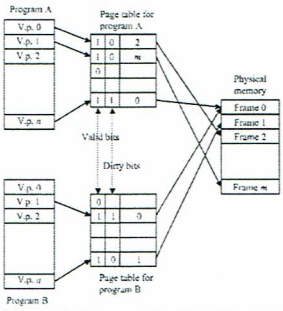

Figure 2.19. Paging system overview. Note: (1) in general n,q are much larger than m; (2) not all of programs A and B are in main memory at a given time, (3) page n of program A and page 1 of program B are shared.

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).

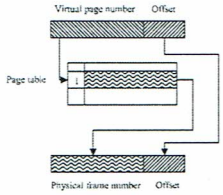

Figure 2.20. Virtual address translation.

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.

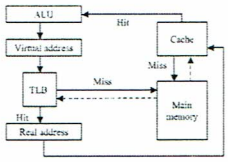

Figure 2.21. Abstracted view of the memory reference process.

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	Millions of cycles
	5–100 ns	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.