Chapter 2

Modern Processor and Memory Technology

Computation and the storage of data are inseparable concepts. In this chapter, we give a background on how they have evolved and how storage and processors are implemented in computers today. It presents aspects of modern computers that are important for achieving high performance.

We will use knowledge of these aspects when we begin the engineering effort in Chapter 5. We will use it both as guidelines for how to improve performance and to help us understand why performance was or was not improved.

The information provided in this chapter has been gathered from [Tan99], [Tan01], and [HP96] as well as from information available from hardware manufactures on the World Wide Web, such as [HSU$^+$01], [AMD02], and [MPS02]. There will be a lot of detailed information, so the chapter ends in a brief summary of considerations important when implementing high performance sorting algorithms.

2.1 Historical Background

Storage has been the limiting factor on computing, in a much stronger sense than actual raw processing power. Up until the early 1970's computers used magnetic core memory, invented by Jay Forrester in 1949. It was slow, cumbersome, and expensive and thus appeared in very limited quantities, usually no more than 4KB. Algorithms too complex to run in such a limited environment could not be realized and programmers were forced to use simpler and slower algorithms [Tan99, Sect. 6.1]. The solution was to provide a secondary storage. Programmers needing more space for their programs had to split them up manually in so-called overlays and explicitly program loading and storing of these overlays between secondary storage and main memory. Needless to say, much work was involved with the management of overlays.

The situation improved somewhat with the introduction of transistor-based dynamic random-access memory (DRAM, invented at IBM in 1966) by Intel® Corp. and static random-access memory (SRAM) by Fairchild Corp. both in 1970, however high price of memory still made code size an issue. A trend thus arose that useful functionality would be implemented directly in the CPU core; features should rather be implemented in hardware than software. Designs like this, for example, provided single instructions capable of doing what small loops does in higher level languages, such as copying strings.

Through the 1980's storage of large programs in memory became a non-issue. The complex instruction sets of the 1970's were still used, but they required several physically separate chips to implement (e.g. one for integer operations, one dedicated to floating-point math, and one memory-controller). This complexity prompted a change in design philosophy, toward moving features from hardware to software. The idea was that much of the work done by the more complex instructions in hardware, at the time, could be accomplished by several simpler instructions. The result was the Reduced Instruction Set Computer (RISC). After the RISC term had been established the term Complex Instruction Set Computer (CISC) was used rather pejoratively about computers not following this philosophy.

2.2 Modern Processors

Today's processors can rightfully be called neither RISC nor CISC; at the core, they are in essence all RISC-like. However, whether the processor is based on RISC design or not, have implications we may have to consider to this day. Though the RISC approach is architecturally compelling, the most popular architecture used today is the IA32. The key to its popularity is that it is backwards compatible with all its predecessors, commonly named x86, the first of which was the 8086. The 8086, in turn, was assembly-level compatible with its predecessor, the 8080, which was compatible with Intel's first "computer-on-chip", the 4004. The 4004, 8080, and the 8086 was all designed before the RISC philosophy became prevalent, so the successors of the 8086 are all, what we would call CISC. Today, the x86 processors are the only widely employed CISC processors so they have become synonymous in some sense with the CISC design philosophy. We will use the term RISC to denote processors with roots in the RISC design philosophy, such as the MIPS R4000-R16000, the IBM/Motorola PowerPC, the Digital/HP/Compaq Alpha, and the Sun Ultra Sparc and use the term CISC about processors implementing the IA32.

Common for all processors, RISC or CISC, since the 4004 are the designs with registers, arithmetic/logic unit (ALU) and clocks; each clock tick, operands stored in registers are passed through the ALU, producing the result of some operation performed on them that is then stored in a target register. The particular operation to be performed is decided by the instructions that make up the executing program.

2.2.1 Instruction Pipelining

When looking for ways to enhance performance of a processor, an important observation to make is that each instruction undergoes several steps during its execution. In a typical lifecycle of an instruction it is first fetched from the part of memory the code is stored, then decoded to determine which registers keep the needed operands or if operands are in memory. The instruction is then executed, and finally memory operations are completed and write-back of the result of the operation is completed [HP96, Chap. 3]. These tasks would typically be done in physically and functionally separate parts of the processor, inviting the idea that the work on the next instruction could proceed as soon it has finished the first stage. In computer architecture, this is known as pipelining. The first pipelined processors were simple ones with only a few stages. This remained the norm until RISC designers started to extend the concept. Figure 2-1 shows the stages of a typical RISC pipeline.

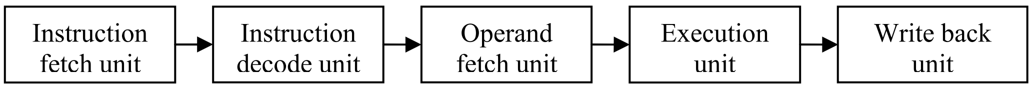

Figure 2-1. A typical RISC five-stage pipeline. While fetching instruction *n*, this pipeline is decoding instruction *n*-1, fetching operands for instruction *n*-2, and so on, all in the same clock cycle.

To make pipelining work efficiently, each stage of the instruction execution must complete in the same time; no instruction can proceed down the pipeline sooner than one can finish an individual stage. While this was one of the primary design goals of the RISC, it cannot be done in case of many of the complex instructions in CISCs. Pipelining can still improve performance, though. The idea is to implement a CISC using a micro-program, in essence emulating the complex instruction set, using a RISC core. This is accomplished by translating the complex instructions into simpler *micro operations* (µ-ops) that each take an equal amount of time to execute. This translation is simply done in additional stages along the pipeline, immediately after the instruction is fetched. Each instruction may result in several µ-ops, so each CISC instruction may take several cycles to complete. Furthermore, while RISC has load-from-/store-to-register instructions and only allows operations to be done on operands in register, CISC has complex addressing modes that allow instructions to operate, not only on operands in registers, but also on operands in memory. To accommodate for this additional address generation stages are inserted in the pipeline. In all, CISC tend to have longer, more complex pipelines than RISC, but the execution proceeds in the same way.

The reason performance is increased using a pipeline approach is that less work is done at each stage, so it can be completed faster. An *n*-stage pipeline has *n* instructions working in parallel. This is known as *instruction-level parallelism*. When an instruction proceeds from one stage to the next, it is temporarily stored in a *pipeline latch*, so it does not interfere with the previous instruction. Other than the increased delay incurred by such a latch, there is no reason each stage of the pipeline cannot be split into several simpler, shorter stages, thus making the completion of each stage faster.

Pipeline Hazards

The reason decreasing the complexity of each stage will not improve performance indefinitely, is the increase in the number of stages. Increasing the number of stages, increases the risk of *pipeline hazards*. These are situations, where the next instruction cannot in general proceed, until the previous instruction has either completed, or reached a certain point in the pipeline. There are three kinds of hazards [HP96, Sect. 3.3]:

- *Structural hazards* occur when hardware, such as the memory subsystem, does not support the execution of certain two instructions in parallel.
- *Data hazards* arise when a needed operand is not ready, because either the current instruction is not finished computing it, or it has not arrived from storage.

- *Control hazards* occur when it is simply not known which instruction should come next.

There are several approaches to overcome these problems, the simplest one perhaps being just to halt the pipeline until the hazard has passed. This approach, while expensive in that parts of the processor are simply sitting idle, works relatively well in case of a structural or data hazard. The reason is, these hazards pass in very few cycles. Consider a structural hazard occurring because an instruction is being fetched from memory at the same time another instruction is writing to memory. The memory may not accept that, but stalling the pipeline for one cycle, while one of the instructions proceeds, will resolve the hazard. Likewise will the missing result of one instruction be ready in a few clock cycles, if it is generated by an instruction ahead of it. That instruction must after all be close to completing the execution stage, if the next instruction is ready to execute. Conversely, if the processor is waiting for data from memory, nothing can be done about it in the pipeline.

Control hazards, however, can be much more severe. They typically occur due to branches in the code. Branches are generally indirect, conditional, or unconditional. Unconditional branches simply say that the next instruction is not the one following this one, but another predetermined one. These branches do not pose a problem, since they can be identified quickly and the fetch stage notified accordingly. Indirect branches typically occur when invoking virtual functions (or equivalently calling a function through a function pointer) or when returning from a function. The latter is usually sped up by keeping a dedicated buffer of call-site addresses so when returning from a function, the processor can quickly look up where is to continue. Both types of indirect branches are relatively infrequent so they do not pose a threat to the overall performance, even if the pipeline is stalled for many cycles. Conditional branches are not rare, however.

Algorithm 2-1.

```
void merge(int n, const float *A, const float *B, float *u)
{
    for( int i=0; i!=n; ++i )
        if( *A < *B )
            *u = *A, ++A, ++u;
        else
            *u = *B, ++B, ++u;
}
```

Consider the simple loop in Algorithm 2-1. After each iteration, a test is done to see if this was the *n*'th iteration and if not, branch back, and loop again. The test is one instruction and the branch is the one following it, but what should follow the branch instruction? That depends on the result of the test, but the result is not available until the test has gone all the way through the pipeline. Thus, conditional branches may block the fetching of instructions. If we were to make the pipeline longer, it will simply be stalled for a correspondingly longer time on an unconditional branch.

Branch Prediction

Realizing, that simply stalling the pipeline until the result of the test has been determined will make all stages sit idle, we might as well go ahead and proceed along one of the branches. If it turns out to be the wrong branch, the work done in the pipeline after the branch was futile, and must be discarded (through what is called a *pipeline flush*), but we have wasted no more time than if we had simply stalled the pipeline. On the other hand, if it was the right branch we have wasted no time at all! The next question is what branch to follow? Consider again the loop branch in Algorithm 2-1. In all but one case, the correct choice is to follow the branch back and repeat the loop. This leads to a simple scheme of *branch prediction* take the same branch as the last time. Using this simple scheme, the loop branch will actually only cause one structural hazard. Another simple scheme is to take a branch if it leads back in the instruction stream, and not to take it if it leads forward. The latter approach favors loops and if-statements that evaluate to true.

Most modern processors use both of these heuristics to predict which branch to take. Branch prediction units today are implemented using branch history table indexed by the least significant bits of the address of the branch instruction. Each entry contains a couple of bits that indicate whether the branch should be taken or not, and whether the branch was taken the last time or not.

Maintaining a table like this is known as *dynamic branch prediction* and it works rather well in practice. RISC was designed with pipelining in mind, so they typically also allow for static branch prediction. Static prediction is done by having two kinds of conditional branch instructions, one for branches that are likely taken, and one for unlikely ones. It is then up to the compiler to figure out, which on should be used. In case of Algorithm 2-1, an intelligent compiler may realize that it is dealing with a loop that may iterate for a while, and generate a “likely conditional branch” instruction to do the loop.

Conditional Execution

There are cases where branches are simply not predictable, however. If *A* and *B* dereferences to random numbers, the branch inside the loop in Algorithm 2-1 is unpredictable, and we would expect a misprediction every other iteration. With a ten-stage pipeline, that amounts to an average of five wasted cycles each iteration, due to structural hazards alone. The only solution to this is not to use branches at all. The assignment in the branch inside the loop of Algorithm 2-1, could like this: `*u=*A*(*A<*B)+*B*!(*A<*B)`, assuming usual convention of Boolean expressions evaluating to 1, when true and 0 otherwise.

The expression is needlessly complicated, involving two floating-point multiplications; instead, a so-called predication bit is used. The test `*A<*B` is done, as when branches was used, and the result is put in a predication bit. Following the test comes two conditional move instructions, one to be completed only if the test was true, and the other only if the test was false. The result of the test is ready just before the first conditional move is to be executed, so no pipeline hazard arose. This way, instructions in both branches are fed through the pipeline, but only one has an effect. A primary limitation of the application of this approach is thus the complexity of what is to be executed conditionally.

Another limitation is that only simple conditions are supported, not arbitrary Boolean expressions or nested if-statements. Yet another is that typical architectures do not allow for all types of instructions to be executed conditionally, and for Algorithm 2-1 to be implemented without risk of severe hazards, we need both a conditional store instruction and a conditional increment or add instruction[1]. Conditional instructions are native to RISC, but have also been added to the x86, from Pentium Pro and on, though only in the form of conditional moves.

2.2.2 Super-Scalar Out-of-Order Processors

The high price of branch mispredictions prevents us from gaining performance by making the pipeline longer. Orthogonal to parallelizing by making the pipeline longer, is parallelizing by issuing more instructions. Realizing that the execution unit often is the bottleneck in the pipeline, we could operate several execution units in parallel. An architecture that employs multiple execution units is known as super-scalar. Super-scalar architectures were popular in the 1970s and 1980s culminating in Very Long Instruction Word (VLIW) designs that packed several instructions into one. A single instruction in a VLIW architecture could consist, for example, of an integer multiplication, a floating-point addition and a load. Each of these instructions would be executed in its own unit, but they would be fetched and decoded as a single instruction. The interest in VLIW has diminished somewhat, probably due to the limited parallelism in real-life code, and limited ability of compilers to express it explicitly.

Going super-scalar with an architecture that was not designed for it, takes quit a bit of work. The 80486 processor was the first CISC processor, designed by Intel, to feature a pipeline. Its successor, the Pentium, expanded on that featuring two parallel pipelines, one (the u pipeline) capable of executing all x86 instructions the other (the v pipeline) only capable of executing a few select ones. If an instruction was immediately followed by one capable of being executed in the v pipeline, they could be executed simultaneously. If the following instruction could not be executed in the v pipeline, the instruction would simply go down the u pipeline alone the following clock cycle. Not surprisingly, only code generated specifically with this setup in mind benefit.

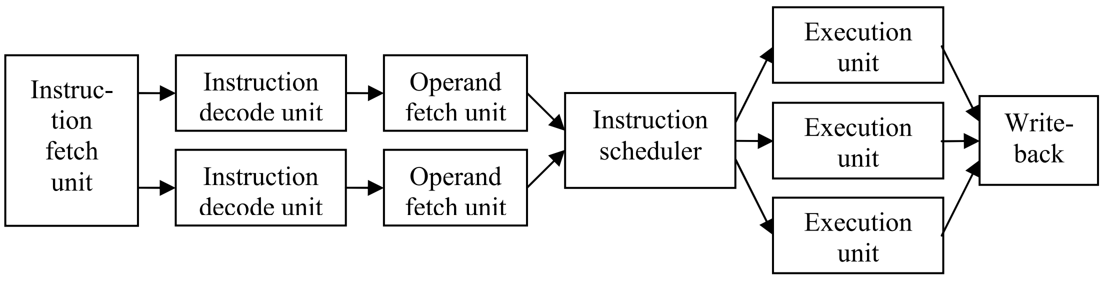

Figure 2-2. A multi-pipeline super-scalar processor.

The successor, the Pentium Pro, extended the number and capabilities of the execution units. It was capable of issuing several μ-ops per clock to the execution unit. Furthermore, the order of the individual instructions did not have to be maintained on entry into the execution units; it would be reestablished after the instructions had been executed. As long as one instruction did not depend on the result of one scheduled close before it, it would be executed in parallel with other instructions. A further advantage

[1] Conditional adds can be simulated with an unconditional add followed by a conditional and then an unconditional move, however that may be to much work.

of having multiple pipelines and execution units is that data and structural hazards not necessarily cause the entire processor to stall.

Register Renaming
It is actually possible to extract even more parallelism seemingly, from where no parallelism exists. Consider this C expression: `++j, i=2*i+1`. On an, admittedly contrived, one-register processor, the expression could be realized by the following instruction sequence:

```
load j    // load j into the register
inc       // increment the register
store j   // store register contents back in j
load i    // load i into the register
shl 1     // multiply by 2
inc       // add 1
store i   // and store result back in i
```

The first three instructions are inherently sequential, as are the last four. With only one register, the two blocks of code would step on each other's toes, were they to be executed in parallel. An out-of-order processor could execute the last four before the first three, but with no performance gain. If, however, the architecture had two registers, the first three instructions could then use one and the last four use the other, thus allowing them to be executed in parallel in almost half the time. This is exactly what register renaming does. Even though the architecture only allows the instructions to use one register, behind the scene instructions are mapped to different physical registers to extract parallelism. A so-called scoreboard keeps track of which scheduled instruction reads and writes which register, and allocates rename registers and execution units accordingly.

This approach works best, when the instruction set does not offer direct access to many registers. One may ask, why not simply add more general purpose registers, and let the compiler use them as best possible? The answer is simple: it will not make legacy code, compiled for fewer registers, run any faster.

2.2.3 State of the Art

The x86 architecture is still very much alive. It was extended with support for 64-bit addressing just last month. The Opteron, the first implementation with this new extension, has a 10K branch history table, three parallel decoder units, three address generating units (AGUs) and three arithmetic/logic units, as well as floating-point addition, a floating-point multiplication and a third float unit. Each pair of address generating and arithmetic/logic has an eight-entry instruction scheduler and the floating-point units have a 36-entry scheduler. Further, it doubles the number of general-purpose registers to 16 that code compiled specifically for the Opteron will be able to utilize.

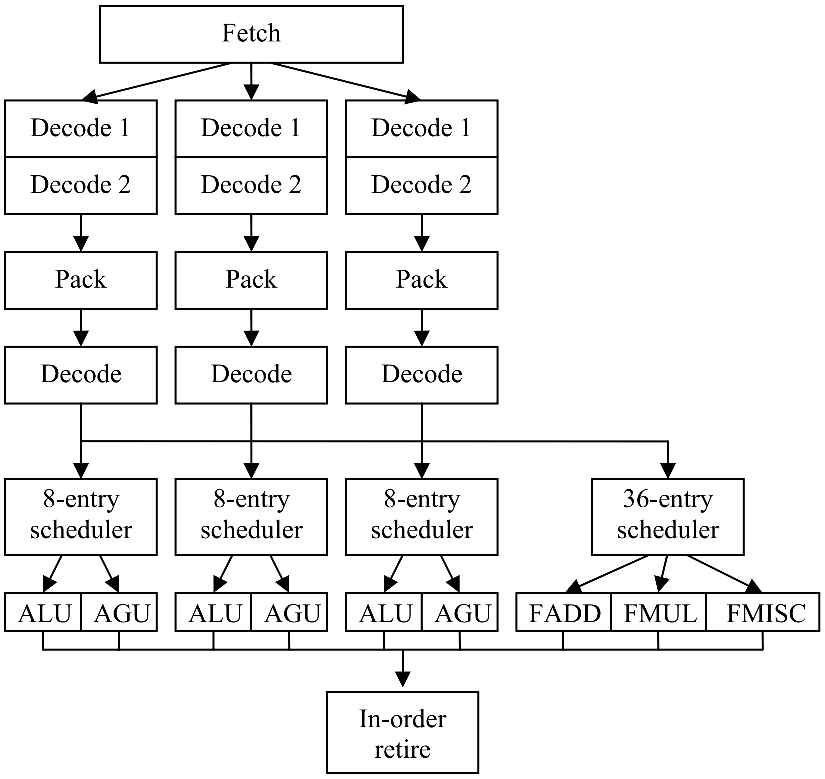

Figure 2-3. Core of the Opteron processor. The many decoder units are primarily due to the complexity of the instruction set.

Top of the line workstation processors from Intel today has a 20-stage pipeline, ALUs that work at twice the clock speed of the rest of the core, 128 rename registers (16 times the number directly available to the programmer/compiler). Their out-of-order logic can handle up to 126 µ-ops at a time. To minimize the risk of control hazards, they have 4000-entry branch history table and a 16-entry return address stack. RISC processors typically do not go to these extremes, but do also employ the types of optimizations described above.

2.3 Modern Memory Subsystems

With the introduction of DRAM and SRAM, a fundamental gap was established; DRAM chips can be manufactured using a single transistor per stored bit, while SRAM chips required six. DRAM chips however requires frequent refreshing, making it inherently slower. The question thus arose, should the computer have fast but expensive, thus limited in quantity, storage, or should it have cheap, slow storage and lots of it?

The question was easily answered up through the 1980s; CPUs were not much faster than DRAM chips so they were never starved of data, even if they used DRAM as primary storage. However, improvements in the transistor manufacturing process have since then made CPUs and SRAM chips a lot faster than DRAM chips. This, combined with the employment of super-scalar pipelined designs, have given CPUs and SRAM a factor of 25 speed increase over DRAM chips in clock speed alone, with a further factor of three or four from synchronization, setup, and protocol overhead. Feeding the processor directly from DRAM, could thus make data hazards stall the pipeline for more than 100 cycles, significantly more than the number of cycles wasted on e.g. a pipeline flush.

Performance enhancing techniques, such as those discussed in the previous section, can also be applied to storage. However, when dealing with memory in a request-response pattern, *latency* aside from mere throughput, is an important issue. Latency is the time it takes a memory request to be served. Introducing pipelines or parallel access banks in RAM, enabling the handling of several memory requests simultaneously, may increase the overall throughput, but it will also increase latency.

2.3.1 Low-level Caches

The problem is economical rather than technological; we know how to build fast storage, but we do not want to pay for it in the quantities we need. The solution is to use hybrid storage, that is, *both* SRAM and DRAM. The quantity of each is determined solely economically, but spending about as much money on SRAM as on DRAM seem to be a good stating point. That, in turn, will give a lot less SRAM than DRAM.

The now widely used technique for extending storage with small amounts of faster storage, is known as caching. The goal is to limit the number of times access to slow storage is requested and the idea is to use the small amount of SRAM to store the parts of data in DRAM that is requested. When a processor needs a word from memory, it does not get it from DRAM, but rather from the cache of SRAM. If the data was not in cache, it is copied from DRAM to cache, and the processor gets it from there. If it is, however, we can enjoy the full performance of SRAM. The hope is that most of the time, we only need the data stored in fast storage, the cache. The challenge is then to keep the needed data in cache and all else in RAM, so we rarely need to copy data from RAM to cache.

Design

The design of the system managing the cache should be as simple as possible, to provide for high responsiveness; however, a too naïve design may cache the wrong data. One important observation to make is that data, whose addresses are close, tend to be accessed closely in time. This is known as *the locality principle* [Tan99, p. 66], [HP96, p. 373]. One example is the execution stack. Stack variables are rarely far from each other. Another is a linear search through a portion of memory; after having accessed the i'th word, we will in the near future access the $i+1^{\text{st}}$ through, say, $i+16^{\text{th}}$ word, which are all stored close to each other.

A simple design that performs well in these scenarios consists of splitting memory into *lines*, each line consisting of several, say B, words of data. When a needed word is not in cache, the entire line it resides on is brought in. This eases the burden on the cache management system, in that it only needs to deal with lines, not single words. This design provides for good performance on directional locality, such as linear searches, where after a program has accessed a word at address i, it will access the word at address $i+d$. If the access to address i caused a new line to be brought in (caused a *cache miss*), the previous access, to address i-d, could not have been to the same line. The word at address i must thus be at one end of a line, regardless of whether d is negative or positive, so the next $B/|d|$ accesses must thus be within the line just brought in. The effect of this is that the time it takes to bring in the line is effectively amortized over the $B/|d|$ subsequent accesses. With d much smaller than B, this effect can be significant. Conversely, if d is larger than B, we do not gain from using a cache. By the way, it was the effect of a large d that caused Algorithm 1-1 to perform poorly, when the matrices became big. The effect splitting memory into lines effectively amortize the cost of accessing slow DRAM over the next $B/|d|$ accesses.

A system is needed to decide which line to overwrite, when a new line comes into cache. The ideal is to put it where the line, we are least like to use again it at. However, allowing lines to be placed at arbitrary positions in cache make them hard to locate, which in turn, reduces responsiveness. On the other hand, we are reluctant to let hardware requirements dictate where to put the data, since it may overwrite data we will be using soon. A more general approach is to divide the cache into *sets* of lines; a line may be placed arbitrarily within a set, but which set it must be put in, is dictated by a fixed mapping function, usually the least significant bits in the address. The structure can be seen in Figure 2-4.

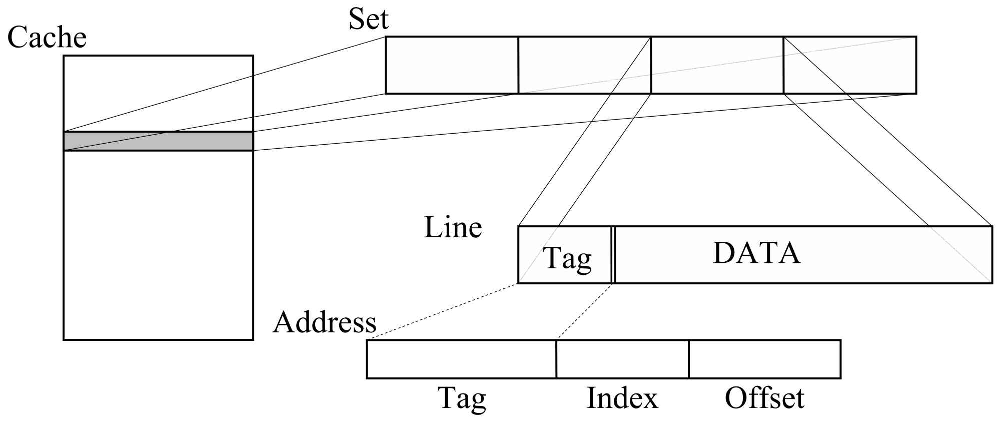

Figure 2-4. A four-way set associative cache.

When a word is needed, the index part of the address is used to locate the set. This is done quickly using a multiplexer. The tag part of the address is then checked against entries in the set. The tag of entries indicates exactly which line of memory is stored there. This comparison can be done in parallel with all entries in the set, however the more lines in a set, the more time is needed for the signal to propagate through. If a matching tag is found the word is in that line at an offset indicated be the least significant bits of the address. If it is not, however, the line is brought in from memory to replace one of the lines in the set. Which one to be replaced, is decided on an approximate *least recently used* (LRU) measure. The case of the cache consisting entirely of one set, known as a *fully associative* cache, allows lines to be placed at arbitrary positions in cache. Conversely, in *direct mapped* caches, a line can be placed only at the position dictated by the index part of its address in memory.

Miss Categories
It is sometimes useful to distinguish the different reasons for cache misses, so we may understand how to reduce their number. [HP96, Sect. 5.3] divides cache misses into three categories:

- *Compulsory*. The very first access to a word. By definition, the word will not be in cache and hence needs to be loaded.
- *Capacity*. If so much data is repeatedly accessed that the cache cannot contain it all, some lines have to be evicted, to be reloaded again later.
- *Conflict*. A variant of a capacity miss, occurring within a set; if data is accessed on lines that all map to the same set, a line from the set will have to be evicted and later retrieved.

Compulsory misses are usually unavoidable. Increasing the capacity to reduce capacity misses will also increase response time. To avoid this, the caching technique is often reapplied to itself, resulting in multiple levels of cache. The cache at the level closest to the CPU, named L1, is often designed as an intricate part of the core itself. It has very small capacity, but is highly responsive. The next level cache, L2, is significantly larger, but the overhead of locating data in it may be several more cycles. With this design, capacity misses only result in access to DRAM when the capacity of the L2 cache is exceeded and the L2 is checked only if a word is not in L1 cache. The L2 cache is often placed on the same chip as, but not embedded in the core of, the CPU. High-end processors may employ a third level of cache, sometimes shared with other processors in multiprocessor environments.

One consequence of using multiple processors is that cache contents can be changed externally. To check if the cache contents are being modified from another processor, a processor monitors the write requests on the memory bus (called *snooping*), and checks each of them to see if it affects data in its cache. This checking can interfere with normal cache operations. To avoid this, the contents of the L1, complete with data and tags, are often duplicated in L2. This way, the snooping mechanism can check write requests against the tags in L2 in parallel with the processor working on data in L1. A cache designed this way is said to have the *inclusion property* [HP96, p. 660-663,723-724]. As always, there is a tradeoff. Applying the inclusion property wastes space in the L2 cache, so depending on the relative sizes of the two levels, it may not be efficient. For example, the Duron, Athlon, and Opteron processors from Advanced Micro Devices (AMD) all have a total of 128kB L1 cache, making it inefficient to duplicate its contents in L2 cache. While AMD does not apply the inclusion property, most others do.

Conflict misses are strongly related to the degree of associativity in the cache. A fully associative cache will not incur conflict misses, while direct mapped caches have a significant likelihood of incurring conflict misses. On a k-way set associative cache, a program must access data at no less than $k+1$ location, all with the same index part of the address and different tags, in order to incur capacity misses. With $k > 1$, this is reasonably unlikely, unless software is designed intentionally to do so. Paradoxically, software designed to minimize capacity misses tend inadvertently to increase accesses to data within the same set (see Section 6.1.2 or [XZK00]). Using direct mapped caches completely eliminates the need for tag matching and LRU approximation hardware, thus reducing the complexity significantly. Hence, direct mapped caches has been the design of choice among RISC designers, however, lately the risk of conflict misses has pushed RISC to adopt multi-way set associative caches. Now only very few processors have direct mapped caches and those that do have multi-way associative L2 caches. Among CISC, the Pentium 4 has a four-way set associative L1 cache and eight-way L2 while the latest x86 incarnation, the Opteron, has a 16-way set associative L2 cache.

Recall the structural pipeline hazard occurred when the fetch stage attempted to read an instruction from memory at the same time the execution stage read data from memory. Having one cache dedicated to instructions and one for data virtually eliminates the risk of structural hazards due to the fetch stage accessing memory at the same time as the execution unit, since the fetch and execute units each access their own cache. L1 cache is often split up this way, while L2 contains both instructions and data. This way, many instructions can be kept close to the CPU and the risk of both the execution and the fetch stage causing an L1 cache miss and accessing L2 in the same cycle, thus incurring a structural hazard, is virtually gone. Another reason for having separate instruction and data cache is that the access patterns for instructions tend to be more predictable than for data. The instruction cache can exploit this by e.g. load more than one line at a time.

2.3.2 Virtual Memory

Storage limitations not only had an influence on the central processing units; programmers tend in general to need more space than can be afforded. This is as much the case today as it is back in the middle of the last century. Indeed, the idea of using a memory hierarchy to cheaply provide a virtually infinite storage can be traced back to von Neumann in 1946 [HP96, Chap. 5], before even magnetic core memory was invented.

In the early 1960s, a group of researchers out of Manchester designed a system, known as virtual memory that provided a significant boost in storage capacity without needing the programmer to do anything. The idea was to work with the disk, usually considered secondary storage, as primary storage, but with random-access memory as a cache. Its implementation had very little hardware requirements, in fact, what they needed, they build themselves. It was designed primarily to ease the burden of managing the overlays used, when programs became too big to fit in memory and as an added benefit, it eased the use of large data sets too [Tan99, Chap. 6].

It has had at least two major impacts on the way programs are developed today. Firstly, programmers do not have to consider how much physical memory is present in the computer, the program is running on; a program designed to run on top of a virtual memory system will run on a computer with a gigabyte of memory as well as on a computer (telephone?) with only 16MB on memory. Secondly, for most applications, programmers do not need to consider the case of running out of memory.

Paging

The approach to implementing virtual memory is the same as implementing caches; the storage is split into blocks of words that are moved to and from cache atomically. In virtual memory terminology, these blocks are not called lines but *pages* and slots where pages can be stored in memory are called *page frames*. A process is allowed access to any address, not just the ones that fall within physical memory. Addresses used by the process are called *virtual addresses* and are taken from the *address space* of the process.

They are translated into the address in physical memory containing the desired word by the *memory management unit* in the processor.

The virtual to physical address translation is done using *page tables*. These tables are indexed, rather like the sets of the L1 and L2 caches, by the high order bits of the virtual address; however, unlike the low-level caches they do not contain the actual data. Rather, each *page table entry* contains the physical address of the page frame, where the page is located. There are no restrictions on what frame an entry must use, thus paging is fully associative.

When a virtual address cannot be translated to a physical, e.g. if the page is not in physical memory, the processor generates an interrupt, called a *page fault*; the processor it self is not responsible for bringing the page into memory. Since disk management is complicated and is highly dependant of the system used to manage files on disk, this is best left to the operating system. Unlike with lower-level caches the latency of the storage being cached (disk) is much higher than that of the cache, so we can afford to let software handle a miss.

Each process has its own address space and hence its own page table. To service a page fault, the operating system determines which process is faulting and which virtual address caused the fault. It then examines the page table of the process and locates the entry. When a page is in memory, the entry points to the frame containing it, but when it is not, the operating system is free to use it to point to a specific position in a specified file on disk. Using this information, the operating system reads the page from disk, stores it in a page frame, updates the page table accordingly, and lets the process continue running.

The maximum size of the address space of a process is dictated by the number of bits in the operands of address operations. A 32-bit processor thus supports processes using 2^{32}B = 4GB of virtual memory. To avoid flushing caches and the TLB (se below) on system calls, half of the address space is often mapped to kernel data structures. If the page size is 4kB, each process would need a page table of one million entries, each of about 32 bits, for a total of 4MB. While the tables are data structures stored in the virtual memory of the kernel, and thus may be paged, 4MB is still a lot of space. One solution is to use hierarchical page tables. A *page directory* of a thousand entries can point to page tables, each of a thousand entries mapping from 4MB of virtual memory to page frames. This way, if a process only uses 10MB of not too scattered storage, it would only require a page directory and three page tables. The downside is that two table entries need to be looked up per memory reference. The translation process can be seen in Figure 2-5.

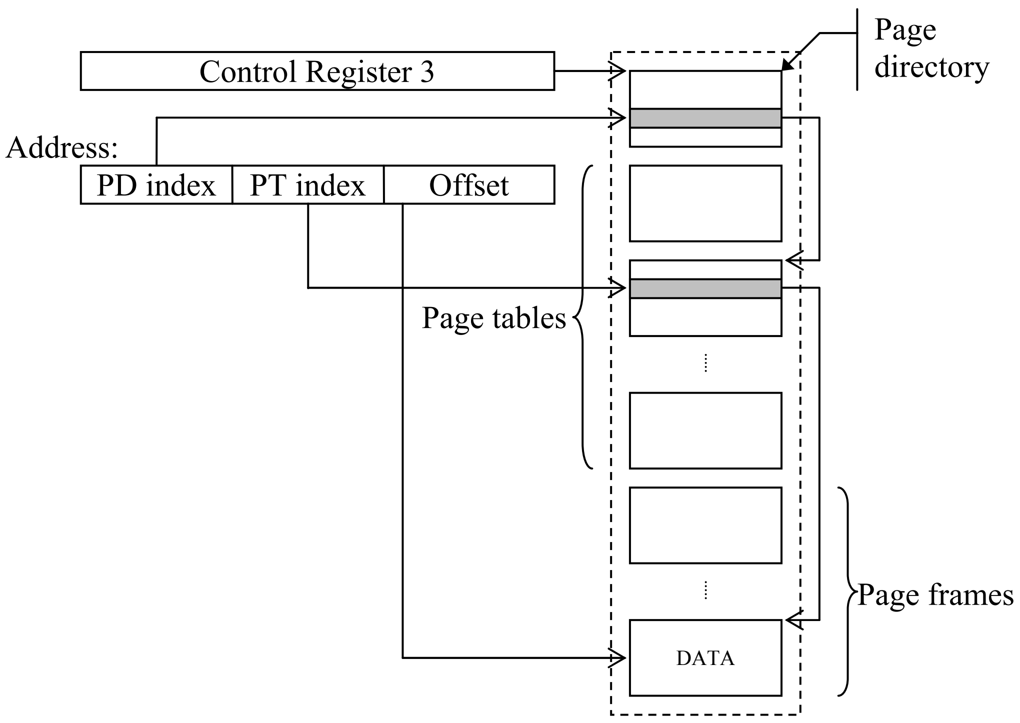

Figure 2-5. Virtual to physical address translation with two levels of page tables. A register in the processor (incidentally called CR3 in x86 terminology) locates the page directory.

Page Replacement Strategies

When a page is brought in from disk and memory is full, it needs to replace some other page in memory. Operating the disk to get the page typically takes in the order of milliseconds, while access to RAM takes in the order of tens of nanoseconds. Thus, we could execute millions of instructions in the same time it takes the page to get from disk to memory. One would think some of that time would be well spend contemplating exactly which page should be evicted, leading to very good page replacement strategies.

However, good replacement strategies, such as evicting the least recently used page, depend on information about pages (their *age*) being updated not only when they are evicted but also every time they are referenced, and we do not want to sacrifice any performance on all the references that do not cause page faults. To keep performance from suffering, non-faulting memory references must be handled entirely in hardware and making hardware update an LRU data structure would be infeasible. In fact, most hardware only keeps information on whether a page has been accessed and whether its contents have been changed in the *referenced bit* and the *dirty bit* respectively [Tan01, Sect. 4.3]. Therefore, while we could spend thousands, even millions of cycles finding the best page to evict, we only have two bits of information per page to base the decision on.

How these two bits are used, differ from operating system to operating system. A frequently used design includes a dedicated process, in UNIX called a *page daemon*, that scans pages and frees those deemed not needed. The hope is that the amount of free space can be kept above some minimum, so that when a page fault occurs, we are not forced to deal with finding a page to evict. When the daemon frees a page, it is not necessarily evicted from memory rather it is put on a free list. Dirty pages on the free list are written to disk, so that when a page fault occurs, the incoming page can be placed in any of the free pages. If a page on the free list is needed, and not overwritten by an incoming page, it can be retrieved. This is known a minor page fault, as opposed to major page faults that involve disk access.

In version 2.2 of the Linux kernel, the page daemon does not do much other than scan and free pages. Its state consisted of the number of the page it inspected when it last went to sleep and it would start from there, scanning pages in memory and simply evict all pages that were not marked as having been referenced. All pages that had been referenced would have their referenced bit cleared, and if it is clear the next time the process comes by, it will get evicted [Tan99, Sect. 10.4]. This results in a mere *not recently used* (NRU) page replacement strategy. Version 2.4, like 2.0, uses an approach much as many other operating systems do, known as *aging*. The implementation requires the operating system to keep a table of the age of each allocated page in the system. When a page is first read in, its age is zero. Each time the page daemon passes it and its referenced bit is not set, its age is increased. If the referenced bit is set, its age is reset to zero. The choice of pages to free is now based on the age of the pages; the oldest ones are freed first, which yields a strategy closer resembling LRU replacement.

File Mapping

Hardware and software is now in place that allows the operating system to place disk-based data in memory, when a program accesses that memory. When a process allocates some memory, writes data in it, and it is paged out, the operating system knows where to get it, should the process need it again.

A slightly different take on this is to let the process specify where the data should come from, in case it is not in memory; it could specify a region in its address space and a file in the file system and when it accesses data in that region, the operating system reads in a corresponding page from the file. This way a program can read and write files without using standard input/output system calls such as `read`/`write`/`seek`. This feature is known as memory mapping of files and is provided by almost all operating systems today, indeed most operating systems use it to load in binary files to be executed.

Translation Look-aside

As can be seen in Figure 2-5, much work is involved in any memory reference, not only the ones that causes page faults. To access a word through its virtual address, aside from accessing the word it self, two additional memory accesses are needed to locate the word. Even though this is all done though hardware, given that DRAM is about two orders of magnitude slower than the CPU, this is a significant overhead. Indeed, the page tables them selves may be paged out, and so the accesses just to get the address of the word may cause page faults and disk access. This does not, however, happen frequently; a page full of page table entries covers (with the figures from above) 4MB of physical memory and any access to those 4MB would reset the age of the page containing the page table, making it unlikely it will get paged out, if it were in fact in use.

To aid in the complicated translation of virtual addresses, processors often provide an on-chip cache of recently translated addresses, called the *translation look-aside buffer* (TLB). Whenever a memory operation is executed, the TLB is checked to see if it falls within a page that has recently been translated. If it does, the physical address is looked up in the TLB and used to carry out the memory operation. If it is not, however, a *TLB miss* occurs, and the page tables are consulted with the overhead described above. As a cache, some TLBs are fully associative while others are two- or four-way set associative, though rarely direct mapped.

Memory operations can now be divided into three categories: a TLB hit, a TLB miss, and a page fault. As stated above, page faults should be handled by the operating system and a TLB hit should be handled with no overhead by hardware. However, whether a TLB miss should be handled by software or hardware is not clear; provided a good TLB implementation, TLB misses should be relatively rare and the added overhead of a software miss handler compared to the three hardware memory lookups may not have much of an impact.

The CISC approach is not to care about hardware complexity and thus, the IA32 clearly specifies the formats of the page directory and page tables, so that a TLB miss can be handled entirely in hardware. This will be the fastest way of handling a TLB miss, but also the most complex, costing transistor space that could be used for bigger TLBs or bigger caches. The MIPS RISC processor takes the opposite approach, where a TLB miss is handled much like a page fault. This way, the processor only needs to be concerned with looking up physical addresses in the TLB. When a TLB miss occurs on a memory request, the processor generates an interrupt and expects the operating system to provide a TLB entry so that request will not miss on the TLB. This provides for much greater flexibility in choosing the page table data structures. For example, the operating system can keep its own software TLB with much greater capacity, than the one in the memory management unit. The downside to use software TLB miss handlers is, of course, that it takes significantly longer to service the miss. Maybe we will soon see second level hardware TLBs caching those translations that cannot fit in first level.

2.3.3 Cache Similarities

Common for all levels of cache in modern computers, is that they consist of a number of blocks of words. Each block is transferred from one level to the next atomically. Say a cache has a capacity of M words and its blocks a capacity of B words. A process can repeatedly access B contiguous words incurring only one memory transfer and M contiguous words incurring roughly M/B memory transfers. The number of transfers is inversely proportional to the size of the block transferred, so in designing the cache it is tempting have large blocks. One negative side effect of this is that it will increase the transfer time and the price of a transfer must be paid if only one word is needed. Another is that it decreases the granularity of the cache. Assuming adequate associativity, a process can repeatedly access M words at M/B *distinct* locations, again incurring M/B transfers. This flexibility is a good thing and increasing B will reduce it. Never the less, high-end workstation and server operating systems favor larger page sizes to maximize throughput.

The above description extends to the translation look-aside buffer as well. Say the buffer has T entries. With $M = TB$, B being the size of a page, we can access B

contiguous words incurring at most one TLB miss and M contiguous words or words at up to $M/B = T$ distinct locations, incurring at most $M/B = T$ TLB misses.

2.4 Summary

Achieving high performance in modern computers today comes down to keeping the pipeline fed. Two major obstacles to achieving this are control and data hazards, caused by unpredictable branches and slowly reacting storage.

Unpredictable branches are expensive, especially in long pipeline designs of CISCs, while predictable ones need not be. They can cost between ten and twenty cycles, between five and ten on average. Conditional execution can be used to avoid branches. However, the limited set of instructions capable of being executed conditionally limits its usefulness. Conditional moves can be used in such problems as selecting a distinct element, say the smallest, from a set of elements. As pipelines get longer and longer, more precise branch prediction units are needed. Recent pipelines of 12+ stages are accompanied by very capable branch prediction units, so that unless branch targets are completely random, branches are executed almost for free. Indeed, if the branch test is independent of instructions executed prior, it can be evaluated in a separate execution unit in parallel with other instructions, and if the branch was predicted correctly, the branch does not interfere with execution of the other instructions. In Algorithm 2-1, for example, the evaluation of `++i` and `i!=n` can be executed in parallel with the loop body, so when the branch is (correctly) predicted taken, some execution units proceeds with the loop body while one execution unit increments `i` and verifies that `i!=n`, while the loop body is executing at full speed.

Non-local storage accesses may take very long time. Requests go through several layers of cache, and the further they have to go, the longer the pipeline has to wait. If the requested data is in L1 cache, the pipeline is usually stalled for no more than one cycle and for no more than 5-10 cycles, if it is in L2. If it is not in cache, the delay depends on whether the address can be easily translated through the TLB. If it can, the delay may be as low as 100 cycles; otherwise, it will be two-three times as much or in case of software TLB miss handlers, significantly more. If a page fault occurs, the process will likely be stalled for tens millions of cycles.

Managing the cache needs to be fast, so no level provides exact least recently used block replacement, but all provide an approximation. The increased complexity of multi-way set associative caches has lead RISC processor designers to stick with direct-mapped caches for a long time; only the very latest generations of RISC processors, such as the UltraSparc III, the PowerPC 4, and the MIPS R10000 use multi-way set associative caches. The Pentium 4 has an eight-way set associative L2 cache and even a four-way set associative L1 cache and the Opteron has 16-way set associative L2 cache. With this increased flexibility, conflict misses should become less predominant.

On the positive side, if there is any parallelism in the code, we can expect the processor to detect and exploit it. Executing code like `i=i+1, j=2*j+1` may thus not be any more expensive than merely incrementing `i`, simply because there is no dependency between instructions updating `i` and the instructions updating `j`, so they can be scheduled in different execution units.