Chapter 1

Modern processors

In the "old days" of scientific supercomputing roughly between 1975 and 1995, leading-edge high performance systems were specially designed for the HPC market by companies like Cray, CDC, NEC, Fujitsu, or Thinking Machines. Those systems were way ahead of standard "commodity" computers in terms of performance and price. Single-chip general-purpose microprocessors, which had been invented in the early 1970s, were only mature enough to hit the HPC market by the end of the 1980s, and it was not until the end of the 1990s that clusters of standard workstation or even PC-based hardware had become competitive at least in terms of theoretical peak performance. Today the situation has changed considerably. The HPC world is dominated by cost-effective, off-the-shelf systems with processors that were not primarily designed for scientific computing. A few traditional supercomputer vendors act in a niche market. They offer systems that are designed for high application performance on the single CPU level as well as for highly parallel workloads. Consequently, the scientist and engineer is likely to encounter such "commodity clusters" first and only advance to more specialized hardware as requirements grow. For this reason, this chapter will mostly focus on systems based on standard cache-based microprocessors. *Vector computers* support a different programming paradigm that is in many respects closer to the requirements of scientific computation, but they have become rare. However, since a discussion of supercomputer architecture would not be complete without them, a general overview will be provided in Section 1.6.

1.1 Stored-program computer architecture

When we talk about computer systems at large, we always have a certain architectural concept in mind. This concept was conceived by Turing in 1936, and first implemented in a real machine (EDVAC) in 1949 by Eckert and Mauchly [H129, H131]. Figure 1.1 shows a block diagram for the *stored-program digital computer*. Its defining property, which set it apart from earlier designs, is that its instructions are numbers that are stored as data in memory. Instructions are read and executed by a control unit; a separate arithmetic/logic unit is responsible for the actual computations and manipulates data stored in memory along with the instructions. I/O facilities enable communication with users. Control and arithmetic units together with the appropriate interfaces to memory and I/O are called the *Central Processing Unit* (CPU). Programming a stored-program computer amounts to modifying instructions in memory,

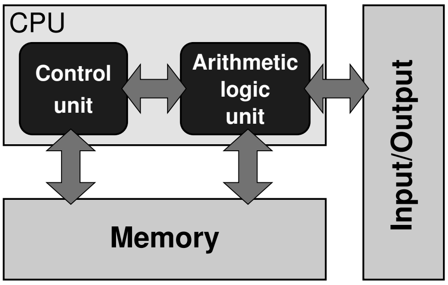


Figure 1.1: Stored-program computer architectural concept. The "program," which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

which can in principle be done by another program; a *compiler* is a typical example, because it translates the constructs of a high-level language like C or Fortran into instructions that can be stored in memory and then executed by a computer.

This blueprint is the basis for all mainstream computer systems today, and its inherent problems still prevail:

- Instructions and data must be continuously fed to the control and arithmetic units, so that the speed of the memory interface poses a limitation on compute performance. This is often called the *von Neumann bottleneck*. In the following sections and chapters we will show how architectural optimizations and programming techniques may mitigate the adverse effects of this constriction, but it should be clear that it remains a most severe limiting factor.

- The architecture is inherently sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory. The term *SISD* (Single Instruction Single Data) has been coined for this concept. How it can be modified and extended to support parallelism in many different flavors and how such a parallel machine can be efficiently used is also one of the main topics of this book.

Despite these drawbacks, no other architectural concept has found similarly widespread use in nearly 70 years of electronic digital computing.

1.2 General-purpose cache-based microprocessor architecture

Microprocessors are probably the most complicated machinery that man has ever created; however, they all implement the stored-program digital computer concept as described in the previous section. Understanding all inner workings of a CPU is out of the question for the scientist, and also not required. It is helpful, though, to get a grasp of the high-level features in order to understand potential bottlenecks. Figure 1.2 shows a very simplified block diagram of a modern cache-based general-purpose microprocessor. The components that actually do "work" for a running application are the arithmetic units for floating-point (FP) and integer (INT) operations

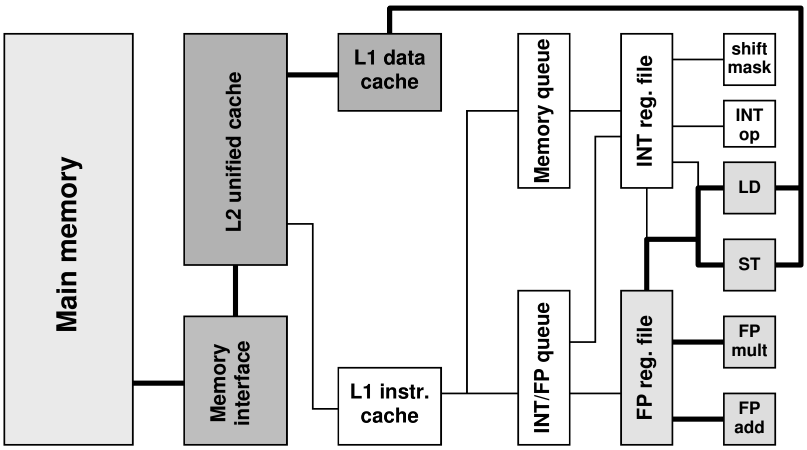


Figure 1.2: Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

and make up for only a very small fraction of the chip area. The rest consists of administrative logic that helps to feed those units with operands. CPU *registers*, which are generally divided into floating-point and integer (or "general purpose") varieties, can hold operands to be accessed by instructions with no significant delay; in some architectures, *all* operands for arithmetic operations must reside in registers. Typical CPUs nowadays have between 16 and 128 user-visible registers of both kinds. Load (LD) and store (ST) units handle instructions that transfer data to and from registers. Instructions are sorted into several *queues*, waiting to be executed, probably not in the order they were issued (see below). Finally, *caches* hold data and instructions to be (re-)used soon. The major part of the chip area is usually occupied by caches.

A lot of additional logic, i.e., branch prediction, reorder buffers, data shortcuts, transaction queues, etc., that we cannot touch upon here is built into modern processors. Vendors provide extensive documentation about those details [V104, V105, V106]. During the last decade, *multicore* processors have superseded the traditional single-core designs. In a multicore chip, several processors ("cores") execute code concurrently. They can share resources like memory interfaces or caches to varying degrees; see Section 1.4 for details.

1.2.1 Performance metrics and benchmarks

All the components of a CPU core can operate at some maximum speed called *peak performance*. Whether this limit can be reached with a specific application code depends on many factors and is one of the key topics of Chapter 3. Here we introduce some basic performance metrics that can quantify the "speed" of a CPU. Scientific computing tends to be quite centric to floating-point data, usually with "double preci-

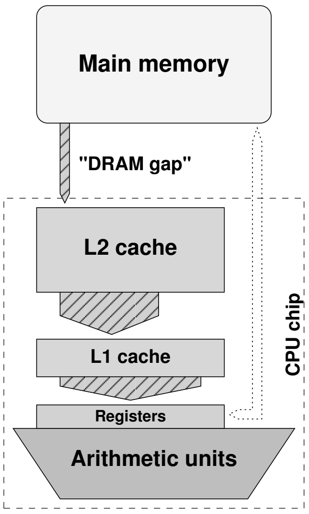
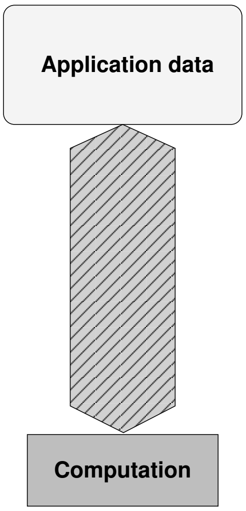


Figure 1.3: (Left) Simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. (Right) The "DRAM gap" denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.



sion" (DP). The performance at which the FP units generate results for multiply and add operations is measured in *floating-point operations per second* (Flops/sec). The reason why more complicated arithmetic (divide, square root, trigonometric functions) is not counted here is that those operations often share execution resources with multiply and add units, and are executed so slowly as to not contribute significantly to overall performance in practice (see also Chapter 2). High performance software should thus try to avoid such operations as far as possible. At the time of writing, standard commodity microprocessors are designed to deliver at most two or four double-precision floating-point results per clock cycle. With typical clock frequencies between 2 and 3 GHz, this leads to a peak arithmetic performance between 4 and 12 GFlops/sec per core.

As mentioned above, feeding arithmetic units with operands is a complicated task. The most important data paths from the programmer's point of view are those to and from the caches and main memory. The performance, or *bandwidth* of those paths is quantified in GBytes/sec. The GFlops/sec and GBytes/sec metrics usually suffice for explaining most relevant performance features of microprocessors.[1] Hence, as shown in Figure 1.3, the performance-aware programmer's view of a cache-based microprocessor is very data-centric. A "computation" or algorithm of some kind is usually defined by manipulation of data items; a concrete implementation of the algorithm must, however, run on real hardware, with limited performance on all data paths, especially those to main memory.

Fathoming the chief performance characteristics of a processor or system is one of the purposes of *low-level benchmarking*. A low-level benchmark is a program that tries to test some specific feature of the architecture like, e.g., peak performance or

[1]Please note that the "giga-" and "mega-" prefixes refer to a factor of 10^9 and 10^6, respectively, when used in conjunction with ratios like bandwidth or arithmetic performance. Since recently, the prefixes "mebi-," "gibi-," etc., are frequently used to express quantities in powers of two, i.e., 1 MiB=2^{20} bytes.

Listing 1.1: Basic code fragment for the vector triad benchmark, including performance measurement.

```
1  double precision, dimension(N) :: A,B,C,D
2  double precision :: S,E,MFLOPS
3
4  do i=1,N                          !initialize arrays
5    A(i) = 0.d0; B(i) = 1.d0
6    C(i) = 2.d0; D(i) = 3.d0
7  enddo
8
9  call get_walltime(S)              ! get time stamp
10 do j=1,R
11   do i=1,N
12     A(i) = B(i) + C(i) * D(i)     ! 3 loads, 1 store
13   enddo
14   if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
15 enddo
16 call get_walltime(E)              ! get time stamp
17 MFLOPS = R*N*2.d0/((E-S)*1.d6)    ! compute MFlop/sec rate
```

memory bandwidth. One of the prominent examples is the *vector triad*, introduced by Schönauer [S5]. It comprises a nested loop, the inner level executing a multiply-add operation on the elements of three vectors and storing the result in a fourth (see lines 10–15 in Listing 1.1). The purpose of this benchmark is to measure the performance of data transfers between memory and arithmetic units of a processor. On the inner level, three *load streams* for arrays `B`, `C` and `D` and one *store stream* for `A` are active. Depending on `N`, this loop might execute in a very small time, which would be hard to measure. The outer loop thus repeats the triad `R` times so that execution time becomes large enough to be accurately measurable. In practice one would choose `R` according to `N` so that the overall execution time stays roughly constant for different `N`.

The aim of the masked-out call to the `dummy()` subroutine is to prevent the compiler from doing an obvious optimization: Without the call, the compiler might discover that the inner loop does not depend at all on the outer loop index `j` and drop the outer loop right away. The possible call to `dummy()` fools the compiler into believing that the arrays may change between outer loop iterations. This effectively prevents the optimization described, and the additional cost is negligible because the condition is always false (which the compiler does not know).

The `MFLOPS` variable is computed to be the MFlops/sec rate for the whole loop nest. Please note that the most sensible time measure in benchmarking is *wallclock time*, also called *elapsed time*. Any other "time" that the system may provide, first and foremost the much stressed CPU time, is prone to misinterpretation because there might be contributions from I/O, context switches, other processes, etc., which CPU time cannot encompass. This is even more true for parallel programs (see Chapter 5). A useful C routine to get a wallclock time stamp like the one used in the triad bench-

Listing 1.2: A C routine for measuring wallclock time, based on the `gettimeofday()` POSIX function. Under the Windows OS, the `GetSystemTimeAsFileTime()` routine can be used in a similar way.

```
#include <sys/time.h>

void get_walltime_(double* wcTime) {
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
}

void get_walltime(double* wcTime) {
  get_walltime_(wcTime);
}
```

mark above could look like in Listing 1.2. The reason for providing the function with and without a trailing underscore is that Fortran compilers usually append an underscore to subroutine names. With both versions available, linking the compiled C code to a main program in Fortran or C will always work.

Figure 1.4 shows performance graphs for the vector triad obtained on different generations of cache-based microprocessors and a vector system. For very small loop lengths we see poor performance no matter which type of CPU or architecture is used. On standard microprocessors, performance grows with `N` until some maximum is reached, followed by several sudden breakdowns. Finally, performance stays constant for very large loops. Those characteristics will be analyzed in detail in Section 1.3.

Vector processors (dotted line in Figure 1.4) show very contrasting features. The low-performance region extends much farther than on cache-based microprocessors,

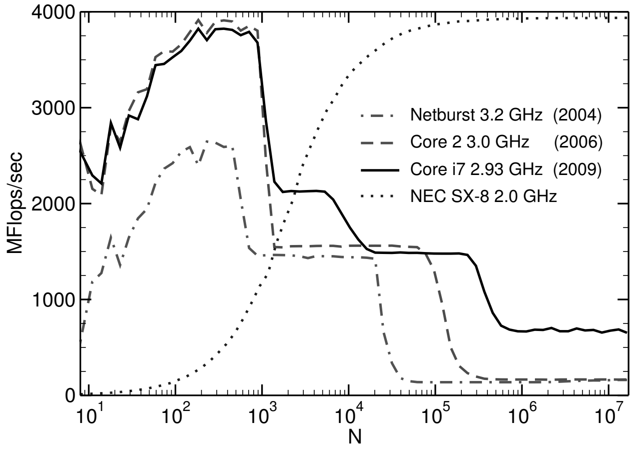


Figure 1.4: Serial vector triad performance versus loop length for several generations of Intel processor architectures (clock speed and year of introduction is indicated), and the NEC SX-8 vector processor. Note the entirely different performance characteristics of the latter.

but there are no breakdowns at all. We conclude that vector systems are somewhat complementary to standard CPUs in that they meet different domains of applicability (see Section 1.6 for details on vector architectures). It may, however, be possible to optimize real-world code in a way that circumvents low-performance regions. See Chapters 2 and 3 for details.

Low-level benchmarks are powerful tools to get information about the basic capabilities of a processor. However, they often cannot accurately predict the behavior of "real" application code. In order to decide whether some CPU or architecture is well-suited for some application (e.g., in the run-up to a procurement or before writing a proposal for a computer time grant), the only safe way is to prepare *application benchmarks*. This means that an application code is used with input parameters that reflect as closely as possible the real requirements of production runs. The decision for or against a certain architecture should always be heavily based on application benchmarking. Standard benchmark collections like the SPEC suite [W118] can only be rough guidelines.

1.2.2 Transistors galore: Moore's Law

Computer technology had been used for scientific purposes and, more specifically, for numerically demanding calculations long before the dawn of the desktop PC. For more than thirty years scientists could rely on the fact that no matter which technology was implemented to build computer chips, their "complexity" or general "capability" doubled about every 24 months. This trend is commonly ascribed to *Moore's Law*. Gordon Moore, co-founder of Intel Corp., postulated in 1965 that the number of components (transistors) on a chip that are required to hit the "sweet spot" of minimal manufacturing cost per component would continue to increase at the indicated rate [R35]. This has held true since the early 1960s despite substantial changes in manufacturing technologies that have happened over the decades. Amazingly, the growth in complexity has always roughly translated to an equivalent growth in compute performance, although the meaning of "performance" remains debatable as a processor is not the only component in a computer (see below for more discussion regarding this point).

Increasing chip transistor counts and clock speeds have enabled processor designers to implement many advanced techniques that lead to improved application performance. A multitude of concepts have been developed, including the following:

1. *Pipelined functional units*. Of all innovations that have entered computer design, pipelining is perhaps the most important one. By subdividing complex operations (like, e.g., floating point addition and multiplication) into simple components that can be executed using different functional units on the CPU, it is possible to increase instruction throughput, i.e., the number of instructions executed per clock cycle. This is the most elementary example of *instruction-level parallelism* (ILP). Optimally pipelined execution leads to a throughput of one instruction per cycle. At the time of writing, processor designs exist that feature pipelines with more than 30 stages. See the next section on page 9 for details.

2. *Superscalar architecture*. Superscalarity provides "direct" instruction-level parallelism by enabling an instruction throughput of more than one per cycle. This requires multiple, possibly identical functional units, which can operate currently (see Section 1.2.4 for details). Modern microprocessors are up to six-way superscalar.

3. *Data parallelism through SIMD instructions*. SIMD (*Single Instruction Multiple Data*) instructions issue identical operations on a whole array of integer or FP operands, usually in special registers. They improve arithmetic peak performance without the requirement for increased superscalarity. Examples are Intel's "SSE" and its successors, AMD's "3dNow!," the "AltiVec" extensions in Power and PowerPC processors, and the "VIS" instruction set in Sun's UltraSPARC designs. See Section 1.2.5 for details.

4. *Out-of-order execution*. If arguments to instructions are not available in registers "on time," e.g., because the memory subsystem is too slow to keep up with processor speed, out-of-order execution can avoid idle times (also called *stalls*) by executing instructions that appear later in the instruction stream but have their parameters available. This improves instruction throughput and makes it easier for compilers to arrange machine code for optimal performance. Current out-of-order designs can keep hundreds of instructions in flight at any time, using a *reorder buffer* that stores instructions until they become eligible for execution.

5. *Larger caches*. Small, fast, on-chip memories serve as temporary data storage for holding copies of data that is to be used again "soon," or that is close to data that has recently been used. This is essential due to the increasing gap between processor and memory speeds (see Section 1.3). Enlarging the cache size does usually not hurt application performance, but there is some tradeoff because a big cache tends to be slower than a small one.

6. *Simplified instruction set*. In the 1980s, a general move from the *CISC* to the *RISC* paradigm took place. In a CISC (Complex Instruction Set Computer), a processor executes very complex, powerful instructions, requiring a large hardware effort for decoding but keeping programs small and compact. This lightened the burden on programmers, and saved memory, which was a scarce resource for a long time. A RISC (Reduced Instruction Set Computer) features a very simple instruction set that can be executed very rapidly (few clock cycles per instruction; in the extreme case each instruction takes only a single cycle). With RISC, the clock rate of microprocessors could be increased in a way that would never have been possible with CISC. Additionally, it frees up transistors for other uses. Nowadays, most computer architectures significant for scientific computing use RISC at the low level. Although x86-based processors execute CISC machine code, they perform an internal on-the-fly translation into RISC "μ-ops."

In spite of all innovations, processor vendors have recently been facing high obstacles in pushing the performance limits of monolithic, single-core CPUs to new levels.

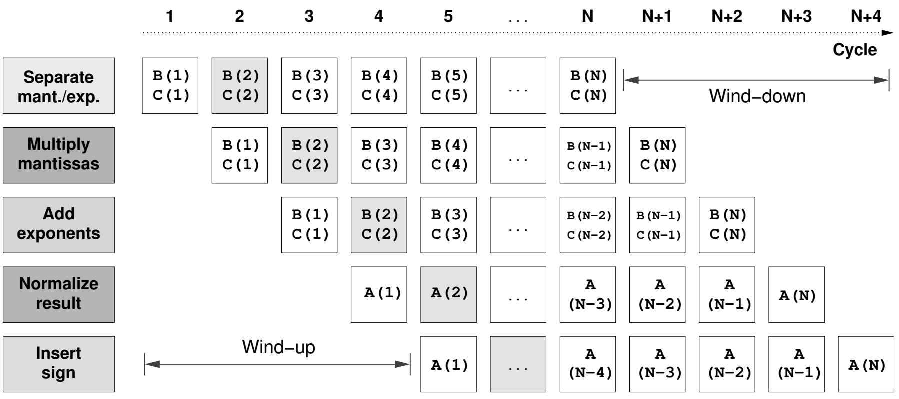


Figure 1.5: Timeline for a simplified floating-point multiplication pipeline that executes `A(:)=B(:)*C(:)`. One result is generated on each cycle after a four-cycle wind-up phase.

Moore's Law promises a steady growth in transistor count, but more complexity does not automatically translate into more efficiency: On the contrary, the more functional units are crammed into a CPU, the higher the probability that the "average" code will not be able to use them, because the number of independent instructions in a sequential instruction stream is limited. Moreover, a steady increase in clock frequencies is required to keep the single-core performance on par with Moore's Law. However, a faster clock boosts power dissipation, making idling transistors even more useless.

In search for a way out of this *power-performance dilemma* there have been some attempts to simplify processor designs by giving up some architectural complexity in favor of more straightforward ideas. Using the additional transistors for larger caches is one option, but again there is a limit beyond which a larger cache will not pay off any more in terms of performance. *Multicore* processors, i.e., several CPU cores on a single die or socket, are the solution chosen by all major manufacturers today. Section 1.4 below will shed some light on this development.

1.2.3 Pipelining

Pipelining in microprocessors serves the same purpose as assembly lines in manufacturing: Workers (functional units) do not have to know all details about the final product but can be highly skilled and specialized for a single task. Each worker executes the same chore over and over again *on different objects*, handing the half-finished product to the next worker in line. If it takes m different steps to finish the product, m products are continually worked on in different stages of completion. If all tasks are carefully tuned to take the same amount of time (the "time step"), all workers are continuously busy. At the end, one finished product per time step leaves the assembly line.

Complex operations like loading and storing data or performing floating-point arithmetic cannot be executed in a single cycle without excessive hardware requirements. Luckily, the assembly line concept is applicable here. The most simple setup is a "fetch–decode–execute" pipeline, in which each stage can operate independently of the others. While an instruction is being executed, another one is being decoded and a third one is being fetched from instruction (L1I) cache. These still complex tasks are usually broken down even further. The benefit of elementary subtasks is the potential for a higher clock rate as functional units can be kept simple. As an example we consider floating-point multiplication, for which a possible division into five "simple" subtasks is depicted in Figure 1.5. For a vector product `A(:)=B(:)*C(:)`, execution begins with the first step, separation of mantissa and exponent, on elements `B(1)` and `C(1)`. The remaining four functional units are idle at this point. The intermediate result is then handed to the second stage while the first stage starts working on `B(2)` and `C(2)`. In the second cycle, only three out of five units are still idle. After the fourth cycle the pipeline has finished its so-called *wind-up* phase. In other words, the multiply pipe has a *latency* (or *depth*) of five cycles, because this is the time after which the first result is available. From then on, all units are continuously busy, generating one result per cycle. Hence, we speak of a *throughput* of one cycle. When the first pipeline stage has finished working on `B(N)` and `C(N)`, the *wind-down* phase starts. Four cycles later, the loop is finished and all results have been produced.

In general, for a pipeline of depth m, executing N independent, subsequent operations takes $N+m-1$ steps. We can thus calculate the expected speedup versus a general-purpose unit that needs m cycles to generate a single result,

$$\frac{T_{\text{seq}}}{T_{\text{pipe}}} = \frac{mN}{N+m-1} \,, \tag{1.1}$$

which is proportional to m for large N. The *throughput* is

$$\frac{N}{T_{\text{pipe}}} = \frac{1}{1+\frac{m-1}{N}} \,, \tag{1.2}$$

approaching 1 for large N (see Figure 1.6). It is evident that the deeper the pipeline the larger the number of independent operations must be to achieve reasonable throughput because of the overhead caused by the wind-up phase.

One can easily determine how large N must be in order to get at least p results per cycle ($0 < p \leq 1$):

$$p = \frac{1}{1+\frac{m-1}{N_{\text{c}}}} \quad\Longrightarrow\quad N_{\text{c}} = \frac{(m-1)p}{1-p} \,. \tag{1.3}$$

For $p = 0.5$ we arrive at $N_{\text{c}} = m-1$. Taking into account that present-day microprocessors feature overall pipeline lengths between 10 and 35 stages, we can immediately identify a potential performance bottleneck in codes that use short, tight loops. In superscalar or even vector processors the situation becomes even worse as multiple identical pipelines operate in parallel, leaving shorter loop lengths for each pipe.

Another problem connected to pipelining arises when very complex calculations

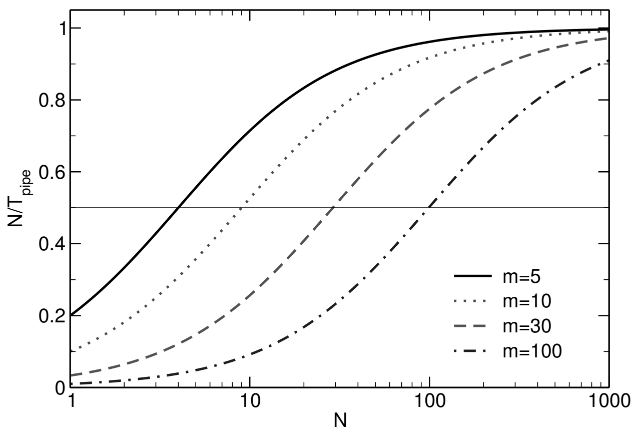


Figure 1.6: Pipeline throughput as a function of the number of independent operations. *m* is the pipeline depth.

like FP divide or even transcendental functions must be executed. Those operations tend to have very long latencies (several tens of cycles for square root or divide, often more than 100 for trigonometric functions) and are only pipelined to a small level or not at all, so that stalling the instruction stream becomes inevitable, leading to so-called *pipeline bubbles*. Avoiding such functions is thus a primary goal of code optimization. This and other topics related to efficient pipelining will be covered in Chapter 2.

Note that although a depth of five is not unrealistic for a floating-point multiplication pipeline, executing a "real" code involves more operations like, e.g., loads, stores, address calculations, instruction fetch and decode, etc., that must be overlapped with arithmetic. Each operand of an instruction must find its way from memory to a register, and each result must be written out, observing all possible interdependencies. It is the job of the compiler to arrange instructions in such a way as to make efficient use of all the different pipelines. This is most crucial for in-order architectures, but also required on out-of-order processors due to the large latencies for some operations.

As mentioned above, an instruction can only be executed if its operands are available. If operands are not delivered "on time" to execution units, all the complicated pipelining mechanisms are of no use. As an example, consider a simple scaling loop:

```
do i=1,N
  A(i) = s * A(i)
enddo
```

Seemingly simple in a high-level language, this loop transforms to quite a number of assembly instructions for a RISC processor. In pseudocode, a naïve translation could look like this:

```
loop:  load A(i)
       mult A(i) = A(i) * s
       store A(i)
       i = i + 1
```

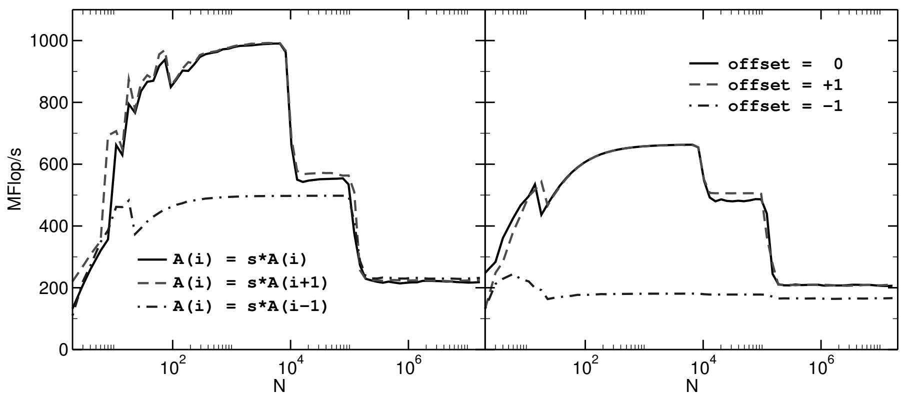


Figure 1.7: Influence of constant (left) and variable (right) offsets on the performance of a scaling loop. (AMD Opteron 2.0 GHz).

```
        branch -> loop
```

Although the multiply operation can be pipelined, the pipeline will *stall* if the load operation on `A(i)` does not provide the data on time. Similarly, the store operation can only commence if the latency for `mult` has passed and a valid result is available. Assuming a latency of four cycles for `load`, two cycles for `mult` and two cycles for `store`, it is clear that above pseudocode formulation is extremely inefficient. It is indeed required to *interleave* different loop iterations to bridge the latencies and avoid stalls:

```
loop:   load A(i+6)
        mult A(i+2) = A(i+2) * s
        store A(i)
        i = i + 1
        branch -> loop
```

Of course there is some wind-up and wind-down code involved that we do not show here. We assume for simplicity that the CPU can issue all four instructions of an iteration in a single cycle and that the final branch and loop variable increment comes at no cost. Interleaving of loop iterations in order to meet latency requirements is called *software pipelining*. This optimization asks for intimate knowledge about processor architecture and insight into application code on the side of compilers. Often, heuristics are applied to arrive at "optimal" code.

It is, however, not always possible to optimally software pipeline a sequence of instructions. In the presence of *loop-carried dependencies*, i.e., if a loop iteration depends on the result of some other iteration, there are situations when neither the compiler nor the processor hardware can prevent pipeline stalls. For instance, if the simple scaling loop from the previous example is modified so that computing `A(i)` requires `A(i+offset)`, with `offset` being either a constant that is known at compile time or a variable:

real dependency	pseudodependency	general version

`do i=2,N`
`  A(i)=s*A(`**`i-1`**`)`
`enddo`
	
	`do i=1,N-1`
	`  A(i)=s*A(`**`i+1`**`)`
	`enddo`
		`start=max(1,1-offset)`
		`end=min(N,N-offset)`
		`do i=start,end`
		`  A(i)=s*A(`**`i+offset`**`)`
		`enddo`

As the loop is traversed from small to large indices, it makes a huge difference whether the offset is negative or positive. In the latter case we speak of a *pseudodependency*, because `A(i+1)` is always available when the pipeline needs it for computing `A(i)`, i.e., there is no stall. In case of a real dependency, however, the pipelined computation of `A(i)` must stall until the result `A(i-1)` is completely finished. This causes a massive drop in performance as can be seen on the left of Figure 1.7. The graph shows the performance of above scaling loop in MFlops/sec versus loop length. The drop is clearly visible only in cache because of the small latencies and large bandwidths of on-chip caches. If the loop length is so large that all data has to be fetched from memory, the impact of pipeline stalls is much less significant, because those extra cycles easily overlap with the time the core has to wait for off-chip data.

Although one might expect that it should make no difference whether the offset is known at compile time, the right graph in Figure 1.7 shows that there is a dramatic performance penalty for a variable offset. The compiler can obviously not optimally software pipeline or otherwise optimize the loop in this case. This is actually a common phenomenon, not exclusively related to software pipelining; hiding information from the compiler can have a substantial performance impact (in this particular case, the compiler refrains from SIMD vectorization; see Section 1.2.4 and also Problems 1.2 and 2.2).

There are issues with software pipelining linked to the use of caches. See Section 1.3.3 below for details.

1.2.4 Superscalarity

If a processor is designed to be capable of executing more than one instruction or, more generally, producing more than one "result" per cycle, this goal is reflected in many of its design details:

- Multiple instructions can be fetched and decoded concurrently (3–6 nowadays).
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units (2–6). This is closely related to the previous point, because feeding those units requires code execution.
- Multiple floating-point pipelines can run in parallel. Often there are one or two combined multiply-add pipes that perform `a=b+c*d` with a throughput of one each.
- Caches are fast enough to sustain more than one load or store operation per

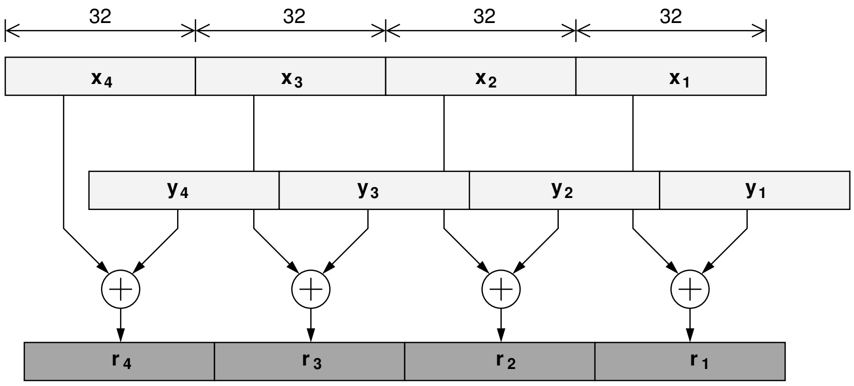


Figure 1.8: Example for SIMD: Single precision FP addition of two SIMD registers (x,y), each having a length of 128 bits. Four SP flops are executed in a single instruction.

cycle, and the number of available execution units for loads and stores reflects that (2–4).

Superscalarity is a special form of parallel execution, and a variant of *instruction-level parallelism* (ILP). Out-of-order execution and compiler optimization must work together in order to fully exploit superscalarity. However, even on the most advanced architectures it is extremely hard for compiler-generated code to achieve a throughput of more than 2–3 instructions per cycle. This is why applications with very high demands for performance sometimes still resort to the use of assembly language.

1.2.5 SIMD

The SIMD concept became widely known with the first vector supercomputers in the 1970s (see Section 1.6), and was the fundamental design principle for the massively parallel *Connection Machines* in the 1980s and early 1990s [R36].

Many recent cache-based processors have instruction set extensions for both integer and floating-point SIMD operations [V107], which are reminiscent of those historical roots but operate on a much smaller scale. They allow the concurrent execution of arithmetic operations on a “wide” register that can hold, e.g., two DP or four SP floating-point words. Figure 1.8 shows an example, where two 128-bit registers hold four single-precision floating-point values each. A single instruction can initiate four additions at once. Note that SIMD does not specify anything about the possible concurrency of those operations; the four additions could be truly parallel, if sufficient arithmetic units are available, or just be fed to a single pipeline. While the latter strategy uses SIMD as a device to reduce superscalarity (and thus complexity) without sacrificing peak arithmetic performance, the former option boosts peak performance. In both cases the memory subsystem (or at least the cache) must be able to sustain sufficient bandwidth to keep all units busy. See Section 2.3.3 for the programming and optimization implications of SIMD instruction sets.

1.3 Memory hierarchies

Data can be stored in a computer system in many different ways. As described above, CPUs have a set of registers, which can be accessed without delay. In addition there are one or more small but very fast *caches* holding copies of recently used data items. *Main memory* is much slower, but also much larger than cache. Finally, data can be stored on disk and copied to main memory as needed. This a is a complex hierarchy, and it is vital to understand how data transfer works between the different levels in order to identify performance bottlenecks. In the following we will concentrate on all levels from CPU to main memory (see Figure 1.3).

1.3.1 Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die. The need for caches can be easily understood by realizing that data transfer rates to main memory are painfully slow compared to the CPU's arithmetic performance. While peak performance soars at several GFlops/sec per core, *memory bandwidth*, i.e., the rate at which data can be transferred from memory to the CPU, is still stuck at a couple of GBytes/sec, which is entirely insufficient to feed all arithmetic units and keep them busy continuously (see Chapter 3 for a more thorough analysis). To make matters worse, in order to transfer a single data item (usually one or two DP words) from memory, an initial waiting time called *latency* passes until data can actually flow. Thus, latency is often defined as the time it takes to transfer a zero-byte message. Memory latency is usually of the order of several hundred CPU cycles and is composed of different contributions from memory chips, the chipset and the processor. Although Moore's Law still guarantees a constant rate of improvement in chip complexity and (hopefully) performance, advances in memory performance show up at a much slower rate. The term *DRAM gap* has been coined for the increasing "distance" between CPU and memory in terms of latency and bandwidth [R34, R37].

Caches can alleviate the effects of the DRAM gap in many cases. Usually there are at least two *levels* of cache (see Figure 1.3), called *L1* and *L2*, respectively. L1 is normally split into two parts, one for instructions ("I-cache," "L1I") and one for data ("L1D"). Outer cache levels are normally *unified*, storing data as well as instructions. In general, the "closer" a cache is to the CPU's registers, i.e., the higher its bandwidth and the lower its latency, the smaller it must be to keep administration overhead low. Whenever the CPU issues a read request ("load") for transferring a data item to a register, first-level cache logic checks whether this item already resides in cache. If it does, this is called a *cache hit* and the request can be satisfied immediately, with low latency. In case of a *cache miss*, however, data must be fetched from outer cache levels or, in the worst case, from main memory. If all cache entries are occupied, a hardware-implemented algorithm *evicts* old items from cache and replaces them with new data. The sequence of events for a cache miss on a write is more involved and

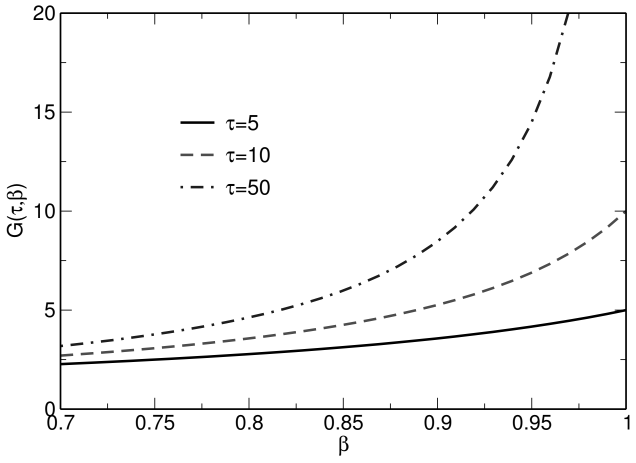


Figure 1.9: The performance gain from accessing data from cache versus the cache reuse ratio, with the speed advantage of cache versus main memory being parametrized by τ.

will be described later. Instruction caches are usually of minor importance since scientific codes tend to be largely loop-based; I-cache misses are rare events compared to D-cache misses.

Caches can only have a positive effect on performance if the data access pattern of an application shows some *locality of reference*. More specifically, data items that have been loaded into a cache are to be used again “soon enough” to not have been evicted in the meantime. This is also called *temporal locality*. Using a simple model, we will now estimate the performance gain that can be expected from a cache that is a factor of τ faster than memory (this refers to bandwidth as well as latency; a more refined model is possible but does not lead to additional insight). Let β be the *cache reuse ratio*, i.e., the fraction of loads or stores that can be satisfied from cache because there was a recent load or store to the same address. Access time to main memory (again this includes latency and bandwidth) is denoted by T_m. In cache, access time is reduced to $T_\mathrm{c} = T_\mathrm{m}/\tau$. For some finite β, the average access time will thus be $T_\mathrm{av} = \beta T_\mathrm{c} + (1-\beta)T_\mathrm{m}$, and we calculate an access performance gain of

$$G(\tau,\beta) = \frac{T_\mathrm{m}}{T_\mathrm{av}} = \frac{\tau T_\mathrm{c}}{\beta T_\mathrm{c} + (1-\beta)\tau T_\mathrm{c}} = \frac{\tau}{\beta + \tau(1-\beta)} \ . \qquad (1.4)$$

As Figure 1.9 shows, a cache can only lead to a significant performance advantage if the reuse ratio is relatively close to one.

Unfortunately, supporting temporal locality is not sufficient. Many applications show *streaming* patterns where large amounts of data are loaded into the CPU, modified, and written back without the potential of reuse “in time.” For a cache that only supports temporal locality, the reuse ratio β (see above) is zero for streaming. Each new load is expensive as an item has to be evicted from cache and replaced by the new one, incurring huge latency. In order to reduce the latency penalty for streaming, caches feature a peculiar organization into *cache lines*. All data transfers between caches and main memory happen on the cache line level (there may be exceptions from that rule; see the comments on nontemporal stores on page 18 for details). The advantage of cache lines is that the latency penalty of a cache miss occurs only on the first miss on an item belonging to a line. The line is fetched from memory as a whole; neighboring items can then be loaded from cache with much lower latency, increasing the *cache hit ratio* γ, not to be confused with the reuse ratio β. So if the application shows some *spatial locality*, i.e., if the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced. The downside of cache lines is that erratic data access patterns are not supported. On the contrary, not only does each load incur a miss and subsequent latency penalty, it also leads to the transfer of a whole cache line, polluting the memory bus with data that will probably never be used. The effective bandwidth available to the application will thus be very low. On the whole, however, the advantages of using cache lines prevail, and very few processor manufacturers have provided means of bypassing the mechanism.

Assuming a streaming application working on DP floating point data on a CPU with a cache line length of $L_\mathrm{c} = 16$ words, spatial locality fixes the hit ratio at $\gamma = (16-1)/16 = 0.94$, a seemingly large value. Still it is clear that performance is governed by main memory bandwidth and latency — the code is *memory-bound*. In order for an application to be truly *cache-bound*, i.e., decouple from main memory so that performance is not governed by memory bandwidth or latency any more, γ must be large enough so the time it takes to process in-cache data becomes larger than the time for reloading it. If and when this happens depends of course on the details of the operations performed.

By now we can qualitatively interpret the performance data for cache-based architectures on the vector triad in Figure 1.4. At very small loop lengths, the processor pipeline is too long to be efficient. With growing `N` this effect becomes negligible, and as long as all four arrays fit into the innermost cache, performance saturates at a high value that is set by the L1 cache bandwidth and the ability of the CPU to issue load and store instructions. Increasing `N` a little more gives rise to a sharp drop in performance because the innermost cache is not large enough to hold all data. Second-level cache has usually larger latency but similar bandwidth to L1 so that the penalty is larger than expected. However, streaming data from L2 has the disadvantage that L1 now has to provide data for registers as well as continuously reload and evict cache lines from/to L2, which puts a strain on the L1 cache's bandwidth limits. Since the ability of caches to deliver data to higher and lower hierarchy levels concurrently is highly architecture-dependent, performance is usually hard to predict on all but the innermost cache level and main memory. For each cache level another performance drop is observed with rising `N`, until finally even the large outer cache is too small and all data has to be streamed from main memory. The size of the different caches is directly related to the locations of the bandwidth breakdowns. Section 3.1 will describe how to predict performance for simple loops from basic parameters like cache or memory bandwidths and the data demands of the application.

Storing data is a little more involved than reading. In presence of caches, if data to be written out already resides in cache, a *write hit* occurs. There are several possibilities for handling this case, but usually outermost caches work with a *write-back* strategy: The cache line is modified in cache and written to memory as a whole when evicted. On a *write miss*, however, cache-memory consistency dictates that the cache line in question must first be transferred from memory to cache before an entry can be modified. This is called *write allocate*, and leads to the situation that a data write stream from CPU to memory uses the bus twice: once for all the cache line allocations and once for evicting modified lines (the data transfer requirement for the triad benchmark code is increased by 25% due to write allocates). Consequently, streaming applications do not usually profit from write-back caches and there is often a wish for avoiding write-allocate transactions. Some architectures provide this option, and there are generally two different strategies:

- *Nontemporal stores*. These are special store instructions that bypass all cache levels and write directly to memory. Cache does not get "polluted" by store streams that do not exhibit temporal locality anyway. In order to prevent excessive latencies, there is usually a small *write combine buffer*, which bundles a number of successive nontemporal stores [V104].

- *Cache line zero*. Special instructions "zero out" a cache line and mark it as modified without a prior read. The data is written to memory when evicted. In comparison to nontemporal stores, this technique uses up cache space for the store stream. On the other hand it does not slow down store operations in cache-bound situations. Cache line zero must be used with extreme care: All elements of a cache line are evicted to memory, even if only a part of them were actually modified.

Both can be applied by the compiler and hinted at by the programmer by means of directives. In very simple cases compilers are able to apply those instructions automatically in their optimization stages, but one must take care to not slow down a cache-bound code by using nontemporal stores, rendering it effectively memory-bound.

Note that the need for write allocates arises because caches and memory generally communicate in units of cache lines; it is a common misconception that write allocates are only required to maintain consistency between caches of multiple processor cores.

1.3.2 Cache mapping

So far we have implicitly assumed that there is no restriction on which cache line can be associated with which memory locations. A cache design that follows this rule is called *fully associative*. Unfortunately it is quite hard to build large, fast, and fully associative caches because of large bookkeeping overhead: For each cache line the cache logic must store its location in the CPU's address space, and each memory access must be checked against the list of all those addresses. Furthermore, the decision which cache line to replace next if the cache is full is made by some algorithm implemented in hardware. Often there is a *least recently used* (LRU) strategy that makes sure only the "oldest" items are evicted, but alternatives like NRU (*not recently used*) or random replacement are possible.

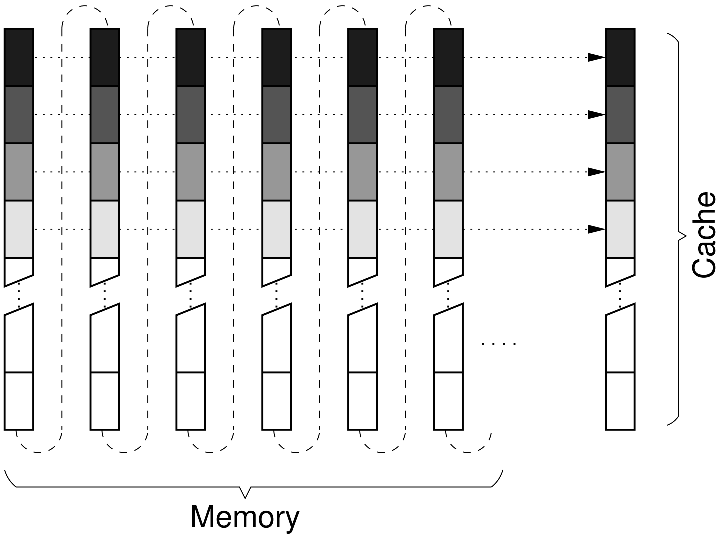


Figure 1.10: In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line (shaded boxes).

The most straightforward simplification of this expensive scheme consists in a *direct-mapped cache*, which maps the full cache size repeatedly into memory (see Figure 1.10). Memory locations that lie a multiple of the cache size apart are always mapped to the same cache line, and the cache line that corresponds to some address can be obtained very quickly by masking out the most significant bits. Moreover, an algorithm to select which cache line to evict is pointless. No hardware and no clock cycles need to be spent for it.

The downside of a direct-mapped cache is that it is disposed toward *cache thrashing*, which means that cache lines are loaded into and evicted from the cache in rapid succession. This happens when an application uses many memory locations that get mapped to the same cache line. A simple example would be a “strided” vector triad code for DP data, which is obtained by modifying the inner loop as follows:

```
do i=1,N,CACHE_SIZE_IN_BYTES/8
  A(i) = B(i) + C(i) * D(i)
enddo
```

By using the cache size in units of DP words as a stride, successive loop iterations hit the same cache line so that *every* memory access generates a cache miss, even though a whole line is loaded every time. In principle there is plenty of room left in the cache, so this kind of situation is called a *conflict miss*. If the stride were equal to the line length there would still be some (albeit small) N for which the cache reuse is 100%. Here, the reuse fraction is exactly zero no matter how small N may be.

To keep administrative overhead low and still reduce the danger of conflict misses and cache thrashing, a *set-associative cache* is divided into *m* direct-mapped caches

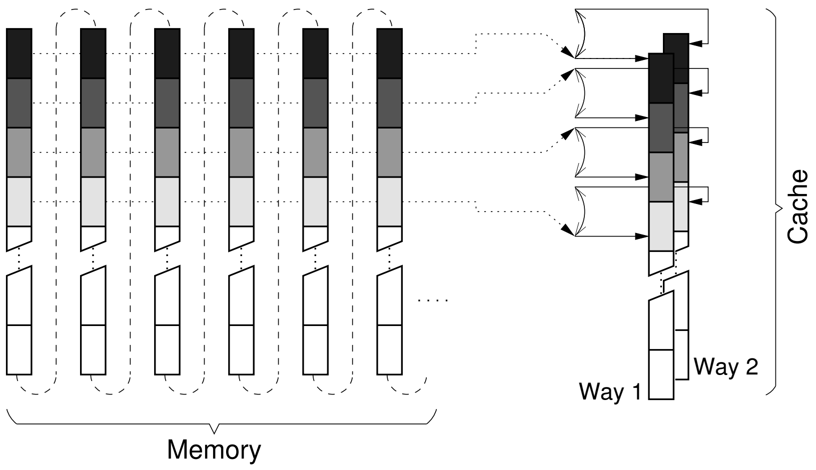


Figure 1.11: In an m-way set-associative cache, memory locations which are located a multiple of $\frac{1}{m}$th of the cache size apart can be mapped to either of m cache lines (here shown for $m = 2$).

equal in size, so-called *ways*. The number of ways m is the number of different cache lines a memory address can be mapped to (see Figure 1.11 for an example of a two-way set-associative cache). On each memory access, the hardware merely has to determine which way the data resides in or, in the case of a miss, to which of the m possible cache lines it should be loaded.

For each cache level the tradeoff between low latency and prevention of thrashing must be considered by processor designers. Innermost (L1) caches tend to be less set-associative than outer cache levels. Nowadays, set-associativity varies between two- and 48-way. Still, the *effective cache size*, i.e., the part of the cache that is actually useful for exploiting spatial and temporal locality in an application code could be quite small, depending on the number of data streams, their strides and mutual offsets. See Chapter 3 for examples.

1.3.3 Prefetch

Although exploiting spatial locality by the introduction of cache lines improves cache efficiency a lot, there is still the problem of latency on the first miss. Figure 1.12 visualizes the situation for a simple vector norm kernel:

```
do i=1,N
  S = S + A(i)*A(i)
enddo
```

There is only one load stream in this code. Assuming a cache line length of four elements, three loads can be satisfied from cache before another miss occurs. The long latency leads to long phases of inactivity on the memory bus.

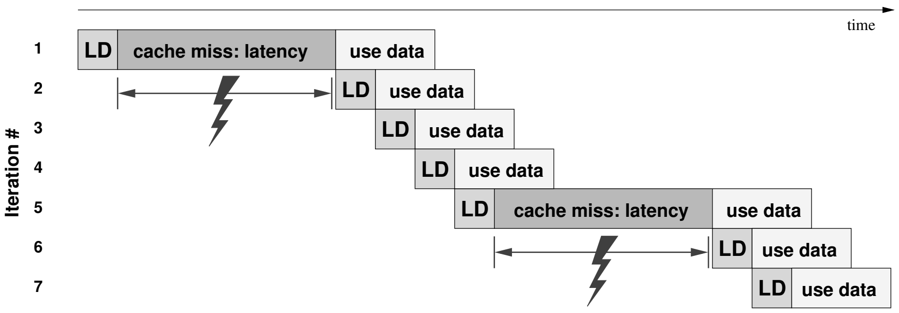


Figure 1.12: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.

Making the lines very long will help, but will also slow down applications with erratic access patterns even more. As a compromise one has arrived at typical cache line lengths between 64 and 128 bytes (8–16 DP words). This is by far not big enough to get around latency, and streaming applications would suffer not only from insufficient bandwidth but also from low memory bus utilization. Assuming a typical commodity system with a memory latency of 50 ns and a bandwidth of 10 GBytes/sec, a single 128-byte cache line transfer takes 13 ns, so 80% of the potential bus bandwidth is unused. Latency has thus an even more severe impact on performance than bandwidth.

The latency problem can be solved in many cases, however, by *prefetching*. Prefetching supplies the cache with data ahead of the actual requirements of an application. The compiler can do this by interleaving special instructions with the software pipelined instruction stream that "touch" cache lines early enough to give the hardware time to load them into cache asynchronously (see Figure 1.13). This assumes there is the potential of asynchronous memory operations, a prerequisite that is to some extent true for current architectures. As an alternative, some processors feature a *hardware prefetcher* that can detect regular access patterns and tries to read ahead application data, keeping up the continuous data stream and hence serving the same purpose as prefetch instructions. Whichever strategy is used, it must be emphasized that prefetching requires resources that are limited by design. The memory subsystem must be able to sustain a certain number of *outstanding prefetch operations*, i.e., pending prefetch requests, or else the memory pipeline will stall and latency cannot be hidden completely. We can estimate the number of outstanding prefetches required for hiding the latency completely: If T_ℓ is the latency and B is the bandwidth, the transfer of a whole line of length L_c (in bytes) takes a time of

$$T = T_\ell + \frac{L_\mathrm{c}}{B} \ . \tag{1.5}$$

One prefetch operation must be initiated per cache line transfer, and the number of cache lines that can be transferred during time T is the number of prefetches P that

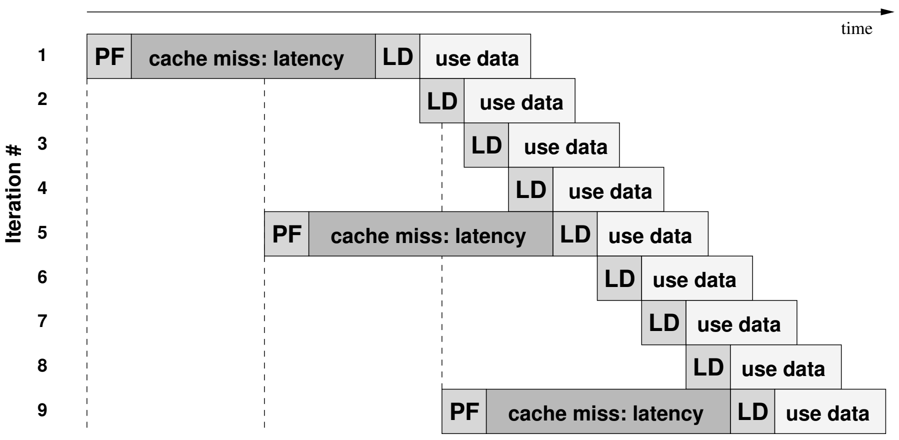


Figure 1.13: Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.

the processor must be able to sustain (see Figure 1.13):

$$P = \frac{T}{L_c/B} = 1 + \frac{T_\ell}{L_c/B} \; . \tag{1.6}$$

As an example, for a cache line length of 128 bytes (16 DP words), $B =$ 10 GBytes/sec and $T_\ell = 50$ ns we get $P \approx 5$ outstanding prefetches. If this requirement cannot be met, latency will not be hidden completely and the full memory bandwidth will not be utilized. On the other hand, an application that executes so many floating-point operations on the cache line data that they cannot be hidden behind the transfer will not be limited by bandwidth and put less strain on the memory subsystem (see Section 3.1 for appropriate performance models). In such a case, fewer outstanding prefetches will suffice.

Applications with heavy demands on bandwidth can overstrain the prefetch mechanism. A second processor core using a shared path to memory can sometimes provide for the missing prefetches, yielding a slight bandwidth boost (see Section 1.4 for more information on multicore design). In general, if streaming-style main memory access is unavoidable, a good programming guideline is to try to establish long continuous data streams.

Finally, a note of caution is in order. Figures 1.12 and 1.13 stress the role of prefetching for hiding latency, but the effects of bandwidth limitations are ignored. It should be clear that prefetching cannot enhance available memory bandwidth, although the transfer time for a single cache line is dominated by latency.

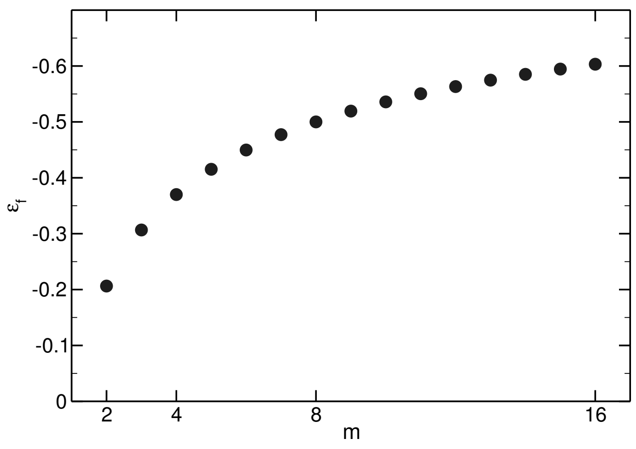


Figure 1.14: Required relative frequency reduction to stay within a given power envelope on a given process technology, versus number of cores on a multicore chip (or package).

1.4 Multicore processors

In recent years it has become increasingly clear that, although Moore's Law is still valid and will probably be at least for the next decade, standard microprocessors are starting to hit the "heat barrier": Switching and leakage power of several-hundred-million-transistor chips are so large that cooling becomes a primary engineering effort and a commercial concern. On the other hand, the necessity of an ever-increasing clock frequency is driven by the insight that architectural advances and growing cache sizes alone will not be sufficient to keep up the one-to-one correspondence of Moore's Law with application performance. Processor vendors are looking for a way out of this power-performance dilemma in the form of *multicore* designs.

The technical motivation behind multicore is based on the observation that, for a given semiconductor process technology, power dissipation of modern CPUs is proportional to the third power of clock frequency f_c (actually it is linear in f_c and quadratic in supply voltage V_cc, but a decrease in f_c allows for a proportional decrease in V_cc). Lowering f_c and thus V_cc can therefore dramatically reduce power dissipation. Assuming that a single core with clock frequency f_c has a performance of p and a power dissipation of W, some relative change in performance $\varepsilon_p = \Delta p/p$ will emerge for a relative clock change of $\varepsilon_f = \Delta f_\mathrm{c}/f_\mathrm{c}$. All other things being equal, $|\varepsilon_f|$ is an upper limit for $|\varepsilon_p|$, which in turn will depend on the applications considered. Power dissipation is

$$W + \Delta W = (1+\varepsilon_f)^3 W \ . \tag{1.7}$$

Reducing clock frequency opens the possibility to place more than one CPU core on the same die (or, more generally, into the same package) while keeping the same *power envelope* as before. For m "slow" cores this condition is expressed as

$$(1+\varepsilon_f)^3 m = 1 \quad \Longrightarrow \quad \varepsilon_f = m^{-1/3} - 1 \ . \tag{1.8}$$

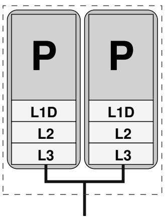


Figure 1.15: Dual-core processor chip with separate L1, L2, and L3 caches (Intel "Montecito"). Each core constitutes its own cache group on all levels.

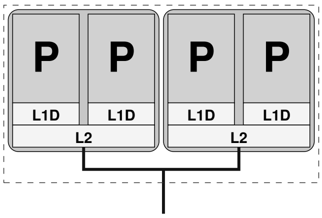


Figure 1.16: Quad-core processor chip, consisting of two dual-cores. Each dual-core has shared L2 and separate L1 caches (Intel "Harpertown"). There are two dual-core L2 groups.

Each one of those cores has the same transistor count as the single "fast" core, but we know that Moore's Law gives us transistors for free. Figure 1.14 shows the required relative frequency reduction with respect to the number of cores. The overall performance of the multicore chip,

$$p_m = (1+\varepsilon_p)pm \;, \tag{1.9}$$

should at least match the single-core performance so that

$$\varepsilon_p > \frac{1}{m} - 1 \tag{1.10}$$

is a limit on the performance penalty for a relative clock frequency reduction of ε_f that should be observed for multicore to stay useful.

Of course it is not trivial to grow the CPU die by a factor of m with a given manufacturing technology. Hence, the most simple way to multicore is to place separate CPU dies in a common package. At some point advances in manufacturing technology, i.e., smaller structure lengths, will then enable the integration of more cores on a die. Additionally, some compromises regarding the single-core performance of a multicore chip with respect to the previous generation will be made so that the number of transistors per core will go down as will the clock frequency. Some manufacturers have even adopted a more radical approach by designing new, much simpler cores, albeit at the cost of possibly introducing new programming paradigms.

Finally, the over-optimistic assumption (1.9) that m cores show m times the performance of a single core will only be valid in the rarest of cases. Nevertheless, multicore has by now been adopted by all major processor manufacturers. In order to avoid any misinterpretation we will always use the terms "core," "CPU," and "processor" synonymously. A "socket" is the physical package in which multiple cores (sometimes on multiple chips) are enclosed; it is usually equipped with leads or pins so it can be used as a replaceable component. Typical desktop PCs have a single socket, while standard servers use two to four, all sharing the same memory. See Section 4.2 for an overview of shared-memory parallel computer architectures.

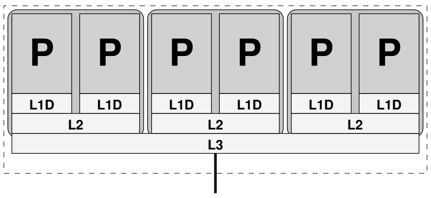


Figure 1.17: Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores (Intel "Dunnington"). L2 groups are dual-cores, and the L3 group is the whole chip.

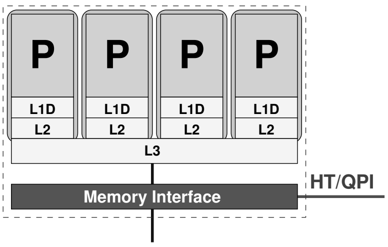


Figure 1.18: Quad-core processor chip with separate L1 and L2 and a shared L3 cache (AMD "Shanghai" and Intel "Nehalem"). There are four single-core L2 groups, and the L3 group is the whole chip. A built-in memory interface allows to attach memory and other sockets directly without a chipset.

There are significant differences in how the cores on a chip or socket may be arranged:

- The cores on one die can either have separate caches (Figure 1.15) or share certain levels (Figures 1.16–1.18). For later reference, we will call a group of cores that share a certain cache level a *cache group*. For instance, the hexa-core chip in Figure 1.17 comprises six L1 groups (one core each), three dual-core L2 groups, and one L3 group which encompasses the whole socket.

  Sharing a cache enables communication between cores without reverting to main memory, reducing latency and improving bandwidth by about an order of magnitude. An adverse effect of sharing could be possible cache bandwidth bottlenecks. The performance impact of shared and separate caches on applications is highly code- and system-dependent. Later sections will provide more information on this issue.

- Most recent multicore designs feature an integrated memory controller to which memory modules can be attached directly without separate logic ("chipset"). This reduces main memory latency and allows the addition of fast intersocket networks like HyperTransport or QuickPath (Figure 1.18).

- There may exist fast data paths between caches to enable, e.g., efficient cache coherence communication (see Section 4.2.1 for details on cache coherence).

The first important conclusion one must draw from the multicore transition is the absolute necessity to put those resources to efficient use by *parallel programming*, instead of relying on single-core performance. As the latter will at best stagnate over the years, getting more speed for free through Moore's law just by waiting for the new CPU generation does not work any more. Chapter 5 outlines the principles and limitations of parallel programming. More details on dual- and multicore designs will be revealed in Section 4.2, which covers shared-memory architectures. Chapters 6 and 9 introduce the dominating parallel programming paradigms in use for technical and scientific computing today.

Another challenge posed by multicore is the gradual reduction in main memory bandwidth and cache size available per core. Although vendors try to compensate these effects with larger caches, the performance of some algorithms is always bound by main memory bandwidth, and multiple cores sharing a common memory bus suffer from contention. Programming techniques for traffic reduction and efficient bandwidth utilization are hence becoming paramount for enabling the benefits of Moore's Law for those codes as well. Chapter 3 covers some techniques that are useful in this context.

Finally, the complex structure of shared and nonshared caches on current multicore chips (see Figures 1.17 and 1.18) makes communication characteristics between different cores highly *nonisotropic*: If there is a shared cache, two cores can exchange certain amounts of information much faster; e.g., they can synchronize via a variable in cache instead of having to exchange data over the memory bus (see Sections 7.2 and 10.5 for practical consequences). At the time of writing, there are very few truly "multicore-aware" programming techniques that explicitly exploit this most important feature to improve performance of parallel code [O52, O53].

Therefore, depending on the communication characteristics and bandwidth demands of running applications, it can be extremely important where exactly multiple threads or processes are running in a multicore (and possibly multisocket) environment. Appendix A provides details on how *affinity* between hardware entities (cores, sockets) and "programs" (processes, threads) can be established. The impact of affinity on the performance characteristics of parallel programs will be encountered frequently in this book, e.g., in Section 6.2, Chapters 7 and 8, and Section 10.5.

1.5 Multithreaded processors

All modern processors are heavily pipelined, which opens the possibility for high performance *if* the pipelines can actually be used. As described in previous sections, several factors can inhibit the efficient use of pipelines: Dependencies, memory latencies, insufficient loop length, unfortunate instruction mix, branch misprediction (see Section 2.3.2), etc. These lead to frequent pipeline bubbles, and a large part of the execution resources remains idle (see Figure 1.19). Unfortunately this situation is the rule rather than the exception. The tendency to design longer pipelines in order to raise clock speeds and the general increase in complexity adds to the problem. As a consequence, processors become hotter (dissipate more power) without a proportional increase in average application performance, an effect that is only partially compensated by the multicore transition.

For this reason, *threading* capabilities are built into many current processor designs. *Hyper-Threading* [V108, V109] or *SMT* (Simultaneous Multithreading) are frequent names for this feature. Common to all implementations is that the *architectural state* of a CPU core is present multiple times. The architectural state comprises

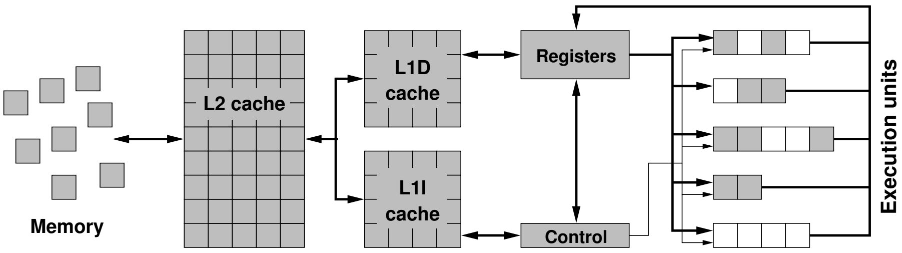


Figure 1.19: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor without SMT. White boxes in the execution units denote pipeline bubbles (stall cycles). Graphics by courtesy of Intel.

all data, status and control registers, including stack and instruction pointers. Execution resources like arithmetic units, caches, queues, memory interfaces etc. are not duplicated. Due to the multiple architectural states, the CPU appears to be composed of several cores (sometimes called *logical processors*) and can thus execute multiple instruction streams, or threads, in parallel, no matter whether they belong to the same (parallel) program or not. The hardware must keep track of which instruction belongs to which architectural state. All threads share the same execution resources, so it is possible to fill bubbles in a pipeline due to stalls in one thread with instructions (or parts thereof) from another. If there are multiple pipelines that can run in parallel (see Section 1.2.4), and one thread leaves one or more of them in an idle state, another thread can use them as well (see Figure 1.20).

It important to know that SMT can be implemented in different ways. A possible distinction lies in how fine-grained the switching between threads can be performed on a pipeline. Ideally this would happen on a cycle-by-cycle basis, but there are designs where the pipeline has to be flushed in order to support a new thread. Of course, this makes sense only if very long stalls must be bridged.

SMT can enhance instruction throughput (instructions executed per cycle) if there is potential to intersperse instructions from multiple threads within or across

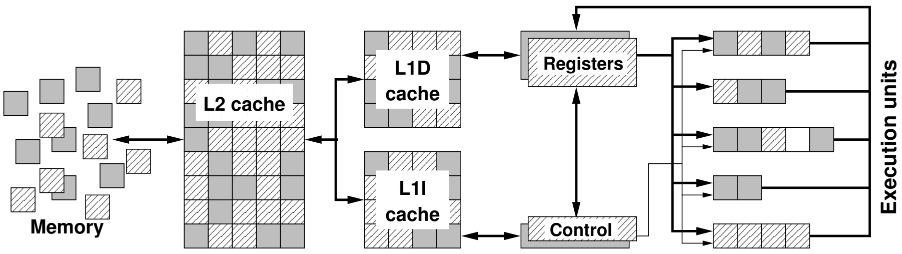


Figure 1.20: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor with fine-grained two-way SMT. Two instruction streams (threads) share resources like caches and pipelines but retain their respective architectural state (registers, control units). Graphics by courtesy of Intel.

pipelines. A promising scenario could arise if different threads use different execution resources, e.g., floating-point versus integer computations. In general, well-optimized, floating-point centric scientific code tends not to profit much from SMT, but there are exceptions: On some architectures, the number of outstanding memory references scales with the number of threads so that full memory bandwidth can only be achieved if many threads are running concurrently.

The performance of a single thread is not improved, however, and there may even be a small performance hit for a single instruction stream if resources are hardwired to threads with SMT enabled. Moreover, multiple threads share many resources, most notably the caches, which could lead to an increase in capacity misses (cache misses caused by the cache being too small) if the code is sensitive to cache size. Finally, SMT may severely increase the cost of synchronization: If several threads on a physical core wait for some event to occur by executing tight, "spin-waiting" loops, they strongly compete for the shared execution units. This can lead to large synchronization latencies [132, 133, M41].

It must be stressed that operating systems and programmers should be aware of the implications of SMT if more than one physical core is present on the system. It is usually a good idea to run different program threads and processes on different physical cores by default, and only utilize SMT capabilities when it is safe to do so in terms of overall performance. In this sense, with SMT present, affinity mechanisms are even more important than on multicore chips (see Section 1.4 and Appendix A). Thorough benchmarking should be performed in order to check whether SMT makes sense for the applications at hand. If it doesn't, all but one logical processor per physical core should be ignored by suitable choice of affinity or, if possible, SMT should be disabled altogether.

1.6 Vector processors

Starting with the Cray 1 supercomputer, vector systems had dominated scientific and technical computing for a long time until powerful RISC-based massively parallel machines became available. At the time of writing, only two companies are still building and marketing vector computers. They cover a niche market for high-end technical computing with extreme demands on memory bandwidth and time to solution.

By design, vector processors show a much better ratio of real application performance to peak performance than standard microprocessors for suitable "vectorizable" code [S5]. They follow the SIMD (Single Instruction Multiple Data) paradigm which demands that a single machine instruction is automatically applied to a — presumably large — number of arguments of the same type, i.e., a vector. Most modern cache-based microprocessors have adopted some of those ideas in the form of SIMD instruction set extensions (see Section 2.3.3 for details). However, vector computers have much more massive parallelism built into execution units and, more importantly, the memory subsystem.

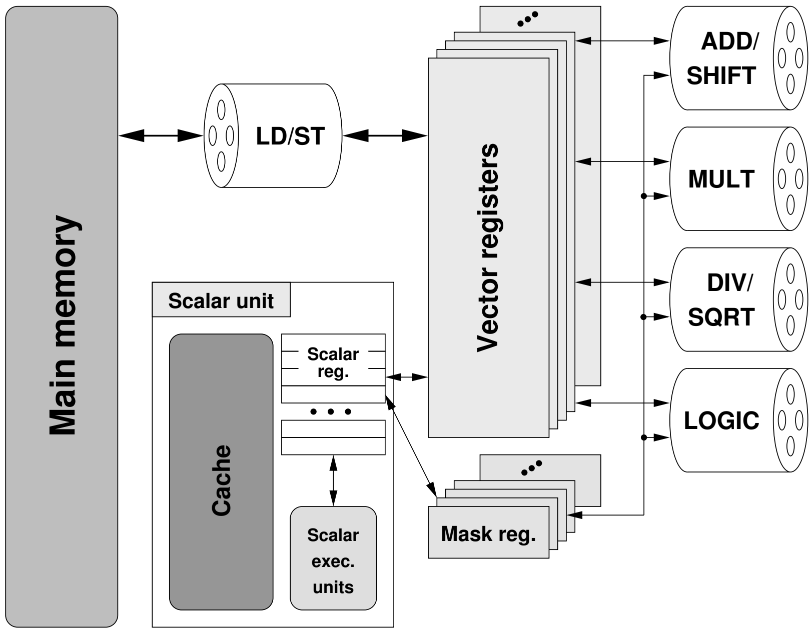


Figure 1.21: Block diagram of a prototypical vector processor with 4-track pipelines.

1.6.1 Design principles

Current vector processors are, quite similarly to RISC designs, register-to-register machines: Machine instructions operate on *vector registers* which can hold a number of arguments, usually between 64 and 256 (double precision). The width of a vector register is called the *vector length* L_v. There is a pipeline for each arithmetic operation like addition, multiplication, divide, etc., and each pipeline can deliver a certain number of results per cycle. For MULT and ADD pipes, this number varies between two and 16 and one speaks of a *multitrack pipeline* (see Figure 1.21 for a block diagram of a prototypical vector processor with 4-track pipes). Other operations like square root and divide are significantly more complex, hence the pipeline throughput is much lower for them. A vector processor with single-track pipes can achieve a similar peak performance per clock cycle as a superscalar cache-based microprocessor. In order to supply data to the vector registers, there is one or more load, store or combined load/store pipes connecting directly to main memory. Classic vector CPUs have no concept of cache hierarchies, although recent designs like the NEC SX-9 have introduced small on-chip memories.

For getting reasonable performance out of a vector CPU, SIMD-type instructions must be employed. As a simple example we consider the addition, of two arrays: `A(1:N)=B(1:N)+C(1:N)`. On a cache-based microprocessor this would result in a (possibly software-pipelined) loop over the elements of `A`, `B`, and `C`. For each index, two loads, one addition and one store operation would have to be executed, together with the required integer and branch logic to perform the loop. A vector CPU can issue a single instruction for a whole array if it is shorter than the vector length:

```
vload V1(1:N) = B(1:N)
vload V2(1:N) = C(1:N)
vadd V3(1:N) = V1(1:N) + V2(1:N)
vstore A(1:N) = V3(1:N)
```

Here, V1, V2, and V3 denote vector registers. The distribution of vector indices across the pipeline tracks is automatic. If the array length is larger than the vector length, the loop must be *stripmined*, i.e., the original arrays are traversed in chunks of the vector length:

```
do S = 1,N,Lv
  E = min(N,S+Lv-1)
  L = E-S+1
  vload V1(1:L) = B(S:E)
  vload V2(1:L) = C(S:E)
  vadd V3(1:L) = V1(1:L) + V2(1:L)
  vstore A(S:E) = V3(1:L)
enddo
```

This is done automatically by the compiler.

An operation like vector addition does not have to wait until its argument vector registers are completely filled but can commence after some initial arguments are available. This feature is called *chaining* and forms an essential requirement for different pipes (like MULT and ADD) to operate concurrently.

Obviously the vector architecture greatly reduces the required instruction issue rate, which had however only been an advantage in the pre-RISC era where multi-issue superscalar processors with fast instruction caches were unavailable. More importantly, the speed of the load/store pipes is matched to the CPU's core frequency, so feeding the arithmetic pipes with data is much less of a problem. This can only be achieved with a massively banked main memory layout because current memory chips require some cycles of recovery time, called the *bank busy time*, after any access. In order to bridge this gap, current vector computers provide thousands of memory banks, making this architecture prohibitively expensive for general-purpose computing. In summary, a vector processor draws its performance by a combination of massive pipeline parallelism with high-bandwidth memory access.

Writing a program so that the compiler can generate effective SIMD vector instructions is called *vectorization*. Sometimes this requires reformulation of code or inserting source code directives in order to help the compiler identify SIMD parallelism. A separate scalar unit is present on every vector processor to execute code which cannot use the vector units for some reason (see also the following sections) and to perform administrative tasks. Today, scalar units in vector processors are much inferior to standard RISC or x86-based designs, so vectorization is absolutely vital for getting good performance. If a code cannot be vectorized it does not make sense to use a vector computer at all.

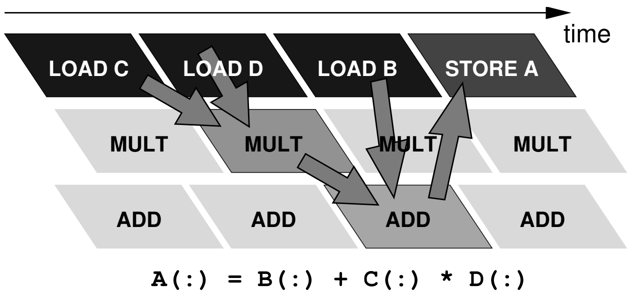


Figure 1.22: Pipeline utilization timeline for execution of the vector triad (see Listing 1.1) on the vector processor shown in Figure 1.21. Light gray boxes denote unused arithmetic units.

1.6.2 Maximum performance estimates

The peak performance of a vector processor is given by the number of tracks for the ADD and MULT pipelines and its clock frequency. For example, a vector CPU running at 2 GHz and featuring four-track pipes has a peak performance of

$$2\ (\text{ADD+MULT}) \times 4\ (\text{tracks}) \times 2\ (\text{GHz}) = 16\ \text{GFlops/sec}\ .$$

Square root, divide and other operations are not considered here as they do not contribute significantly because of their strongly inferior throughput. As for memory bandwidth, a single four-track LD/ST pipeline (see Figure 1.21) can deliver

$$4\ (\text{tracks}) \times 2\ (\text{GHz}) \times 8\ (\text{bytes}) = 64\ \text{GBytes/sec}$$

for reading or writing. (These happen to be the specifications of an NEC SX-8 processor.) In contrast to standard cache-based microprocessors, the memory interface of vector CPUs often runs at the same frequency as the core, delivering more bandwidth in relation to peak performance. Note that above calculations assume that the vector units can actually be used — if a code is nonvectorizable, neither peak performance nor peak memory bandwidth can be achieved, and the (severe) limitations of the scalar unit(s) apply.

Often the performance of a given loop kernel with simple memory access patterns can be accurately predicted. Chapter 3 will give a thorough introduction to balance analysis, i.e., performance prediction based on architectural properties and loop code characteristics. For vector processors, the situation is frequently simple due to the absence of complications like caches. As an example we choose the vector triad (see Listing 1.1), which performs three loads, one store and two flops (MULT+ADD). As there is only a single LD/ST pipe, loads and stores and even loads to different arrays cannot overlap each other, but they can overlap arithmetic and be chained to arithmetic pipes. In Figure 1.22 a rhomboid stands for an operation on a vector register, symbolizing the pipelined execution (much similar to the timeline in Figure 1.5). First a vector register must be loaded with data from array `C`. As the LD/ST pipe starts filling a vector register with data from array `D`, the MULT pipe can start performing arithmetic on `C` and `D`. As soon as data from `B` is available, the ADD pipe can compute the final result, which is then stored to memory by the LD/ST pipe again.

The performance of the whole process is obviously limited by the LD/ST pipeline; given suitable code, the hardware would be capable of executing four times

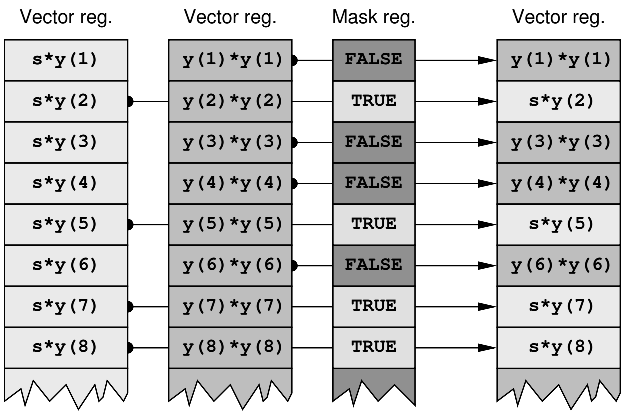


Figure 1.23: On a vector processor, a loop with an if/else branch can be vectorized using a mask register.

as many MULTs and ADDs in the same time (light gray rhomboids), so the triad code runs with 25% of peak performance. On the vector machine described above this amounts to 4 GFlops/sec, which is completely in line with large-N data for the SX-8 in Figure 1.4. Note that this limit is only reached for relatively large N, owing to the large memory latencies on a vector system. Apart from nonvectorizable code, short loops are the second important stumbling block which can negatively impact performance on these architectures.

1.6.3 Programming for vector architectures

A necessary prerequisite for vectorization is the lack of true data dependencies across the iterations of a loop. The same restrictions as for software pipelining apply (see Section 1.2.3), i.e., forward references are allowed but backward references inhibit vectorization. To be more precise, the offset for the true dependency must be larger than some threshold (at least the vector length, sometimes larger) so that results from previous vector operations are available.

Branches in inner loop kernels are also a problem with vectorization because they contradict the “single instruction” paradigm. However, there are several ways to support branches in vectorized loops:

- *Mask registers* (essentially boolean registers with vector length) are provided that allow selective execution of loop iterations. As an example we consider the following loop:

```
do i = 1,N
  if(y(i) .le. 0.d0) then
    x(i) = s * y(i)
  else
    x(i) = y(i) * y(i)
  endif
enddo
```

A vector of boolean values is first generated from the branch conditional using

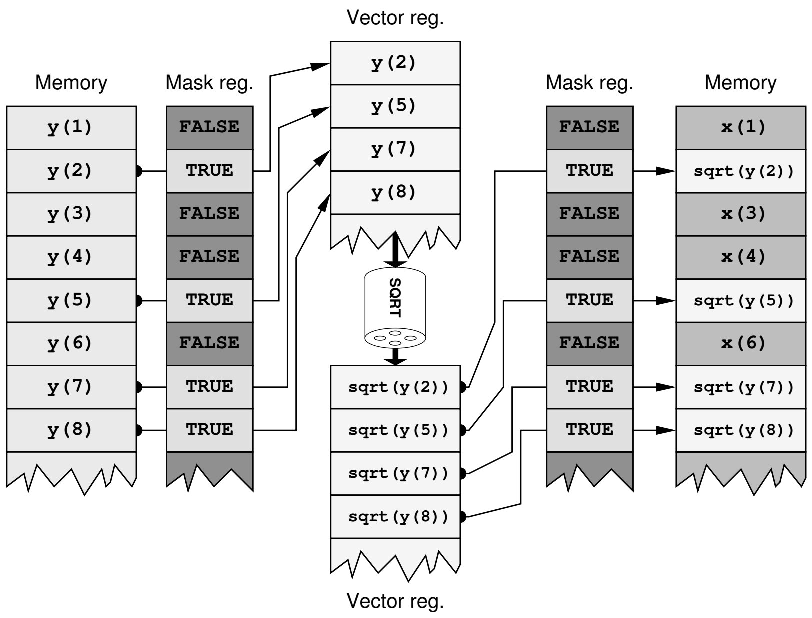


Figure 1.24: Vectorization by the gather/scatter method. Data transfer from/to main memory occurs only for those elements whose corresponding mask entry is true. The same mask is used for loading and storing data.

the logic pipeline. This vector is then used to choose results from the *if* or *else* branch (see Figure 1.23). Of course, both branches are executed for all loop indices which may be waste of resources if expensive operations are involved. However, the benefit of vectorization often outweighs this shortcoming.

- For a single branch (no *else* part), especially in cases where expensive operations like divide and square roots are involved, the *gather/scatter* method is an efficient way to vectorization. In the following example the use of a mask register like in Figure 1.23 would waste a lot of compute resources if the conditional is mostly false:

```
do i = 1,N
  if(y(i) .ge. 0.d0) then
    x(i) = sqrt(y(i))
  endif
enddo
```

Instead (see Figure 1.24), all needed elements of the required argument vectors are first collected into vector registers (gather), then the vector operation is executed on them and finally the results are stored back (scatter).

Many variations of this “trick” exist; either the compiler performs vectorization automatically (probably supported by a source code directive) or the code is rewritten to use explicit temporary arrays to hold only needed vector data. Another variant uses *list vectors*, integer-valued arrays which hold those indices for which the condition is true. These are used to reformulate the original loop to use indirect access.

In contrast to cache-based processors where such operations are extremely expensive due to the cache line concept, vector machines can economically perform gather/scatter (although stride-one access is still most efficient).

Extensive documentation about programming and tuning for vector architectures is available from vendors [V110, V111].

Problems

For solutions see page 287 *ff.*

1.1 *How fast is a divide?* Write a benchmark code that numerically integrates the function

$$f(x) = \frac{4}{1+x^2}$$

from 0 to 1. The result should be an approximation to π, of course. You may use a very simple rectangular integration scheme that works by summing up areas of rectangles centered around x_i with a width of Δx and a height of $f(x_i)$:

```
double precision :: x, delta_x, sum
integer, parameter :: SLICES=100000000
sum = 0.d0 ; delta_x = 1.d0/SLICES
do i=0,SLICES-1
  x = (i+0.5)*delta_x
  sum = sum + 4.d0 / (1.d0 + x * x)
enddo
pi = sum * delta_x
```

Complete the fragment, check that it actually computes a good approximation to π for suitably chosen Δx and measure performance in MFlops/sec. Assuming that the floating-point divide cannot be pipelined, can you estimate the latency for this operation in clock cycles?

1.2 *Dependencies revisited.* During the discussion of pipelining in Section 1.2.3 we looked at the following code:

```
do i = ofs+1,N
  A(i) = s*A(i-ofs)
enddo
```

Here, `s` is a nonzero double precision scalar, `ofs` is a positive integer, and `A` is a double precision array of length `N`. What performance characteristics do you expect for this loop with respect to `ofs` if `N` is small enough so that all elements of `A` fit into the L1 cache?

1.3 *Hardware prefetching.* Prefetching is an essential operation for making effective use of a memory interface. Hardware prefetchers in x86 designs are usually built to fetch data up until the end of a memory page. Identify situations in which this behavior could have a negative impact on program performance.

1.4 *Dot product and prefetching.* Consider the double precision dot product,

```
do i=1,N
  s = s + A(i) * B(i)
enddo
```

for large N on an imaginary CPU. The CPU (1 ns clock cycle) can do one load (or store), one multiplication and one addition per cycle (assume that loop counting and branching comes at no cost). The memory bus can transfer 3.2 GBytes/sec. Assume that the latency to load one cache line from main memory is 100 CPU cycles, and that four double precision numbers fit into one cache line. Under those conditions:

(a) What is the expected performance for this loop kernel without data prefetching?

(b) Assuming that the CPU is capable of prefetching (loading cache lines from memory in advance so that they are present when needed), what is the required number of outstanding prefetches the CPU has to sustain in order to make this code bandwidth-limited (instead of latency-limited)?

(c) How would this number change if the cache line were twice or four times as long?

(d) What is the expected performance if we can assume that prefetching hides all the latency?