4

Time Performance

Time is nature's way to keep everything from happening all at once.

— John Archibald Wheeler

A number of software attributes are associated with the behavior of the system across time. The interactions between the various attributes are nontrivial and often involve tradeoffs. Therefore, before embarking on a project to improve the time performance of a program's operation, it is important to determine which of the time-related attributes we want to change. The most important attributes are

Latency Also referred to as *response time*, *wall clock time*, or *execution time*: the time between the start and the completion of an event (for example, between submitting a form and receiving an acknowledgment). Typically, individual computer users want to decrease this measure, as it often affects their productivity by keeping them idle, waiting for an operation to complete.

Throughput Also sometimes referred to as *bandwidth*: the total amount of work done in a given unit of time (for example, transactions or source code lines processed every second). In most cases, system administrators will want to increase this figure, as it measures the utilization of the equipment they manage.

Processor time requirements Also referred to as CPU *time*: a measure of the time the computer's CPU is kept busy, as opposed to waiting for data to arrive from a slower peripheral. This waiting time is important because in many cases, the processor, instead of being idle, can be put into productive use on other jobs, thus increasing the total throughput of an installation.

Real-time response In some cases, the operation of a system may be degraded or be incorrect if the system does not respond to an external event within a (typically short)

time interval. Such systems are termed real-time systems: *soft* real-time systems if their operation is degraded by a late response (think of a glitch in an MP3 player); *hard* real-time systems if a late response renders their operation incorrect (think of a cell phone failing to synchronize with its base station). In contrast to the other attributes we describe, real-time response is a Boolean measure: A system may or may not succeed in achieving it.

Time variability Finally, in many systems, we are less interested in specific throughput or latency requirements and more interested in a behavior that does not exhibit time variability. In a video game, for example, we might tolerate different refresh rates for the characters but not a jerky movement.

In addition, when we examine server-class systems, we are interested in how the preceding properties will be affected by changes in the system's workload. In such cases, we also look at performance metrics, such as *load sensitivity*, *capacity*, and *scalability*.

We started this chapter by noting that when setting out to work on the time-related properties of a program, we must have a clear purpose concerning the attributes we might want to improve. Although some operations, such as removing a redundant calculation, may simultaneously improve a number of attributes, other changes often involve tradeoffs. As an example, changing a system to perform transactions in batches may improve the system's throughput and decrease processor time requirements, but, on the other hand, the change will probably introduce higher latency and time variability. Some systems even make such tradeoffs explicit: Sun's JVM runtime invocation options[1] optimize the virtual machine performance for latency or throughput by using different implementations of the garbage collector and the locking primitives. Other programs, such as *tcpdump*[2] and *cat*,[3] provide an option for disabling block buffering on the output stream, allowing the user to obtain lower latency at the expense of decreased throughput. In general, it is easier to improve bandwidth (by throwing more resources at the problem) than latency; historically over every period in which technology doubled the bandwidth of microprocessors, memories, the network, or hard disk, the corresponding latency improvement was no more than by a factor of 1.2 to 1.4. An old network saying captures this as follows:

[1] `-server` and `-client`
[2] netbsdsrc/usr.sbin/tcpdump/tcpdump.c:191–197
[3] netbsdsrc/bin/cat/cat.c:104–106

> Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God.

In addition, when examining code with an eye on its performance, it is worthwhile to keep in mind other code attributes that may suffer as a result of any performance-related optimizations.

- Many efficient algorithms are a lot more complex than their less efficient counterparts. Any implementation that uses them may see its *reliability* and *readability* suffer. Compare the 135 lines of the C library's optimized quicksort implementation[4] against the relatively inefficient 10-line bubble sort implementation[5] that the X Window System server uses for selecting a display device. Clearly, reimplementing quicksort for selecting between a couple of different screens would have been an overkill. The use of bubble sort is justified in this case, although calling the C library's `qsort` function could have been another, perhaps even better, alternative. ⚠
- Some optimizations take advantage of a particular platform's characteristics, such as operating system–specific calls or specialized CPU instructions. Such optimizations will negatively impact the code's *portability*. For example, the assembly language implementation of the X Window System's VGA server raster operations[6] is suitable only for a specific CPU architecture and a specific compiler. ⚠
- Other optimizations rely on developing proprietary communication protocols or file storage formats, thus reducing the system's *interoperability*. As an example, a binary file format[7] may be more efficient to process but a lot less portable than a corresponding XML format.[8] ⚠
- Finally, performance optimizations often rely on exploiting special cases of a routine's input. The implementation of special cases destroys the code's *simplicity*, *clarity*, and *generality*. To convince yourself, count the number of assumptions made in the special-case handling of single-character key symbols in the internally used Xt library function `StringToKeySym`.[9] ⚠

[4] netbsdsrc/lib/libc/stdlib/qsort.c:48–182
[5] XFree86-3.3/xc/programs/Xserver/hw/xfree86/common/xf86Config.c:1160–1159
[6] XFree86-3.3/xc/programs/Xserver/hw/xfree86/vga256/enhanced/vgaFasm.h:77–286
[7] netbsdsrc/games/adventure/save.c
[8] argouml/org/argouml/xml/argo/ArgoParser.java
[9] XFree86-3.3/xc/lib/Xt/TMparse.c:837–842

For all those reasons, the first piece of advice all optimization experts agree on is: Don't optimize; you can see a number of more colorful renditions of the same principle in Figure 4.1. The second piece of advice invariably is: Measure before optimizing. Only by locating a program's bottlenecks will you be able to minimize the human programming cost and the reduction in source code quality typically associated with optimization efforts.

If you are lucky enough to work on a new software system rather than on the code of an existing one, you can apply many best practices from the field of *software performance engineering* throughout your product's lifecycle to ensure that you end up with a responsive and scalable system. This is a large field with a considerable body of knowledge; it would not be possible to cover it in this chapter, but a summary of its most important elements in this paragraph can give you a taste of what it entails (see the Further Reading section at the end of the chapter for more details). On the project management front, you should try to estimate your project's performance risk and set precise quantitative objectives for the project's critical use cases (for example, "a static web page shall be delivered in no more than 50 μs"). When modeling, you will benefit from assessing various design alternatives for your system's architecture so as to avoid expensive mistakes before you commit yourself to code. To do this, you need to build a performance model that will provide you with best- and worst-case estimates of your resource requirements (for example, "the response time increases linearly with the number of transactions). Your guide through this process will be measurement experiments that provide representative and reproducible results and software instrumentation that facilitates the collection of data. Here is an excerpt of measurement code, in its simplest form:[10]

```
void
sendfile(int fd, char *name, char *mode)
{
    startclock();
    [...]
    stopclock();
    printstats("Sent", amount);
}

static void
printstats(const char *direction, unsigned long amount)
{
    [...]
```

[10] netbsdsrc/usr.bin/tftp/tftp.c:95–195, 433–448

The First Rule of Program Optimization: Don't do it.

— Michael A. Jackson

The Second Rule of Program Optimization—for experts only: Don't do it yet.

— Michael A. Jackson

Premature optimization is the root of all evil (or at least most of it) in programming.

— Donald E. Knuth

A fast program is just as important as a correct one—false!

— Steve McConnell

Optimizations always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught.

— Larry Wall

The key to performance is elegance, not battalions of special cases. The terrible temptation to tweak should be resisted unless the payoff is really noticeable.

— Jon L. Bentley and M. Douglas McIlroy

More computing sins are committed in the name of efficiency than for any other single reason—including blind stupidity.

— William A. Wulf

Improved efficiency comes at the cost of practically every other desirable product attribute.

— Martin Carroll and Margaret Ellis

When the "efficiency" programmers had trouble, they were loath to change their approach because they would then have to sacrifice some efficiency.

— Gerald M. Weinberg

Do not strive to write fast programs—strive to write good ones.

— Joshua Bloch

Figure 4.1 Experts caution against optimizing code

```
    printf("%s %ld bytes in %.1f seconds",
           direction, amount, delta);
    printf(" [%.0f bits/sec]", (amount*8.)/delta);
}
```

On the implementation front, when you consider alternatives, you should always use measurement data to evaluate them; then, when you write the corresponding code, measure its performance-critical components early and often, to avoid nasty surprises later on.

You will read more about measurement techniques in the next section. In many cases, a major source of performance improvements is the use of a more efficient algorithm; this topic is covered in Section 4.2. Having ruled out important algorithmic inefficiencies, you can begin to look for expensive operations that impact your program's performance. Such operations can range from expensive CPU instructions to operating system and peripheral interactions; all are covered in Sections 4.3–4.6. Finally, in Section 4.7, we examine how caching is often used to trade memory space for execution time.

Exercise 4.1 Choose five personal productivity applications and five infrastructure applications your organization relies on. For each application, list its most important time-related attribute.

Exercise 4.2 Your code provides a horrendously complicated yet remarkably efficient implementation of a simple algorithm. Although it works correctly, measurements have demonstrated that a simple call to the language's runtime library implementation would work just as well for all possible input cases. Provide five arguments for scrapping the existing code.

4.1 Measurement Techniques

Humans are notoriously bad at guessing why a system is exhibiting a particular time-related behavior. Starting your search by plunging into the system's source code, looking for the time-wasting culprit, will most likely be a waste of your time. The only reliable and objective way to diagnose and fix time inefficiencies and problems is to use appropriate measurement tools. Even tools, however, can lie when applied to the wrong problem. The best approach, therefore, is to first evaluate and understand the type of workload a program imposes on your system and then use the appropriate tools for each workload type to analyze the problem in detail.

4.1.1 Workload Characterization

A loaded program on an otherwise idle system can at any time instance be in one of the three different states:

1. Directly executing code. The time spent directly executing code is termed *user time* (*u*), denoting that the process operates in the context of its user.
2. Having the kernel execute code on its behalf. Correspondingly, the time the kernel devotes to executing code in response to a process's requests is termed *system time* (*s*), or *kernel time*.
3. Waiting for an external operation to complete. Operations that cause a program to wait are typically read or write requests to slow peripherals, such as disks and printers, input from human users, and communication with other processes, often over a network link. This time is referred to as *idle time*.

The total time a program spends in all three states is termed the *real time* (*r*) the program takes to operate, often also referred to as the program's wall clock time: the time we can measure using a clock on the wall or a stopwatch. The sum of the program's user and system time is also referred to as CPU time.

The relationship among the real, kernel, and user time in a program's (or complete system's) execution is an important indicator of its workload type, the relevant diagnostic analysis tools, and the applicable problem-resolution options. You can see these elements summarized in Table 4.1; we analyze each workload type in a separate section.

On Unix-type systems, you can specify a process as an argument of the *time*[11] command to obtain the user, system, and real time the process took to its completion. On Windows systems, the *taskmgr* command can list a process's CPU time and show a chart indicating the time the system spends executing kernel code. For nonterminating processes, you will have to obtain similar figures on Unix systems through the commonly available *top* command. On an otherwise unloaded system, you can easily determine a process's user and system time from the corresponding times of the whole system. When analyzing a process's behavior, carefully choose its execution environment: Execute the process either in a realistic setting that reflects the actual intended use or on an unloaded system that will not introduce spurious noise in your measurements.

[11] netbsdsrc/usr.bin/time

Table 4.1 Timing Profile Characterization, Diagnostic Tools, and Resolution Options

Timing Profile	$r \gg u + s$	$s > u$	$u \simeq r$
Characterization	I/O-bound	Kernel-bound	CPU-bound
Diagnostic tools	Disk, network, and virtual memory statistics; network packet dumps; system call tracing	System call tracing	Function profiling; basic block counting
Resolution options	Caching; efficient network protocols and disk data structures; faster I/O interfaces or peripherals	Caching; a faster CPU	Efficient algorithms and data structures; other code improvements; a faster CPU or memory system

4.1.2 I/O-Bound Tasks

Programs and workloads whose real time r is a lot larger than their CPU time $u + s$ are characterized as I/O-bound. Such programs spend most of their time idle, waiting for slower peripherals or processes to respond. Consider as an example the task of creating a copy of the word dictionary on a diskless system with an NFS-mounted disk and a 10Mb/s network interface:

```
$ /usr/bin/time cp /usr/share/dict/words wordcopy
        5.68 real         0.00 user         0.32 sys
```

It would be futile to try to analyze the *cp*[12] command, looking for optimization opportunities that would make it execute faster than the 5.68 seconds it took. The results of the *time* command indicate that *cp* spent negligible CPU time; for 94% of its clock time, it was waiting for a response from the NFS-mounted disk.

The diagnostic tools we use to analyze I/O-bound tasks aim to find the source of the bottleneck and any physical or operational constraints affecting it. The physical constraint could be lagging responses from a genuinely slow disk or the network;

[12] netbsdsrc/bin/cp

the corresponding operational constraints could be the overloading of the disk or the network with other requests that are not part of our workload. On Unix systems, the *iostat*,[13] *netstat*,[14] *nfsstat*,[15] and *vmstat*[16] commands provide summaries and continuous textual updates of a system's disk and terminal, network, and virtual memory performance. On Windows systems, the management console performance monitor (invoked as the *perfmon* command) can provide similar figures in the form of detailed charts. After we find the source of the bottleneck, we can either improve the hardware performance of the corresponding peripheral (by deploying a faster one in its place) or reduce the load we impose on it. Strategies for reducing the load include caching (discussed in Section 4.7) and the adoption of more efficient disk data structures or network protocols, which will minimize the expensive transactions. We discuss how these elements relate to specific source code instances in Section 4.5.

Analyzing the disk performance on the NFS server hosting the `words` file in our example using *iostat* shows the load on the disk to be quite low:

```
            ad0
 KB/t tps  MB/s
 0.00   0  0.00
32.80  15  0.47
27.12  24  0.63
37.31  26  0.94
73.60  10  0.71
35.24  33  1.14
25.14  21  0.51
 7.00   4  0.03
 0.00   0  0.00
```

The load never exceeded 1.14MB/s, well below even the lowly 3.3MB/s PIO[17] mode 0 transfer mode limit. Therefore, the problem is unlikely to be related to the actual disk I/O. However, using *netstat* to monitor the network I/O on the diskless machine does provide us with an insight:

[13] netbsdsrc/usr.sbin/iostat
[14] netbsdsrc/usr.bin/netstat
[15] netbsdsrc/usr.bin/nfsstat
[16] netbsdsrc/usr.bin/vmstat
[17] The programmed input/output mode is a legacy ATAPI hard disk data transfer protocol that supports data transfer rates ranging from 3.3MB/s (PIO mode 0) to 16.6MB/s (PIO mode 4). Modern ATAPI drives typically operate using the Ultra-DMA protocol, supporting transfer rates up to 133MB/s.

```
          input        (Total)           output
packets  errs      bytes    packets  errs      bytes colls
      1     0         60          1     0        250     0
    210     0     237744        204     0     230648   113
    417     0     515196        418     0     513722   324
    383     0     467208        402     0     496650   292
    368     0     451418        381     0     470212   259
    425     0     519588        430     0     515714   301
    400     0     488434        400     0     496816   287
      9     0       6106         15     0      11886     7
      1     0         60          1     0        138     0
```

The maximum network throughput attained is 7.9Mb/s,[18] which is very near the limit of what the particular machine's half-duplex 10Mb/s ethernet interface can deliver in practice. We can therefore safely say that the operation efficiency of the particular command invocation is bound by the capacity of the machine's network interface. Minimizing the network traffic (by adding, for example, a local disk and keeping copies of the data on it) or improving the network interface (for example, to 100Mb/s) will most probably correct the particular deficiency.

In some cases, a more detailed examination of the I/O operations may be required to locate the problem. Two tool categories that will provide such details are system call tracers and network packet-monitoring tools. On Unix systems, you will find such commands as *strace*, *dtrace*, *truss*, *ltrace*,[19] *tcpdump*, and *ethereal*;[20] for Windows systems, you will have to download such programs as *apispy* and *windump*. Your objective here is to examine either the sheer volume of the corresponding transactions or the time a single transaction takes to complete. Most tools provide options for time stamping each transaction, thus providing you with an easy way to reason about the program's behavior.

Consider, for example, the performance of the Apache *logresolve*[21] command. The command reads a web server log and replaces numeric IP addresses with the corresponding host name. Examining its operation with *time* reveals that it spends 99.99%[22] of its time sitting idle:

```
$ /usr/bin/time logresolve <httpd-access.log >/dev/null
     1230.55 real         0.04 user         0.03 sys
```

[18] Calculated as $8\,(519,588 + 515,714)\,/1,024^2$.
[19] freshmeat.net/projects/ltrace
[20] http://www.ethereal.com
[21] apache/src/support/logresolve.c
[22] Calculated as $100\,(1 - (0.04 + 0.03)\,/1,230.55)$.

The output of the *netstat* command also shows an (almost) idle network connection:

```
          input        (Total)           output
packets  errs      bytes    packets  errs      bytes colls
      7     0        486          8     0        108     0
     14     0        229         11     0        383     0
      3     0        336          3     0        324     0
      3     0        216          4     0        301     0
      3     0        667          3     0        216     0
      6     0         98          2     0        301     0
```

However, obtaining a network packet dump with *tcpdump* and examining the timestamps of a single name lookup operation reveal that this may require up to 150 ms:[23]

```
16:15:33.283221 istlab.dmst.aueb.gr.1024 > gns1.nominum.com.domain:
        9529 [1au] PTR? 105.199.133.198.in-addr.arpa. (57)
16:15:33.433305 gns1.nominum.com.domain > istlab.dmst.aueb.gr.1024:
        9529*- 1/2/0 (122) (DF) [tos 0x80]
```

Ignoring the effects of caching (which *logresolve* does perform),[24] we can easily see that processing a 10 million log file may require 17 days.[25] Thus, *logresolve*'s caching is certainly a worthwhile investment.

4.1.3 Kernel-Bound Tasks

Programs and workloads whose system time s is larger than their user time u can be characterized as *kernel-bound*. Strictly speaking, the kernel-bound tasks are also CPU-bound in the sense that improving the processor's speed will often increase their performance. However, we treat such programs as a separate class because they require different diagnostic and resolution techniques. Your objective when dealing with a kernel-bound task is first to determine what kernel system calls the task performs. A system call tracing utility, such as *strace* or *apispy*, will be your tool of choice here.[26] Such a program will provide you with a list of all the system calls a process has performed. As we discuss in Section 4.4, system calls are relatively expensive

[23] Calculated as $(433,305 - 283,221)/1,000$.
[24] apache/src/support/logresolve.c:32–34
[25] Calculated as $0.150 \times 10^7/60/60/24$.
[26] Don't confuse system call tracing with the program comprehension human activity with the same name we discuss in Section 7.2.

operations; therefore, you will then browse the system call list to see whether any calls could be eliminated by the use of appropriate user-level caching strategies.

Consider, as an example, running the directory listing *ls* command to recursively list the contents of a large directory tree:

```
$ /usr/bin/time ls -lR >/dev/null
        4.59 real         1.28 user         2.73 sys
```

We can easily see that *ls* spends twice as much time operating in the context of the kernel than the time it spends executing user code. Examining the output of the *strace* command, we see that for listing 7,263 files, *ls* performs 18,289 system calls. The *strace* command also provides a summary display, which we can use to see the number of different system calls and the average time each one took:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- -------------
 42.52    5.604588         994      5638           lstat
 13.95    1.838295         842      2183           open
 12.38    1.632320         599      2727           fstat
 12.37    1.630742         747      2182           fchdir
 11.41    1.503261         690      2180           close
  2.94    0.387360         353      1096           getdirentries
[...]
```

Armed with that information, we can reason that *ls* performs an `lstat` or `fstat` system call for every file it visits.

Based on those results, we can look at the source to find the cause of the various `stat` calls:[27]

```
if (!f_inode && !f_longform && !f_size && !f_type &&
    sortkey == BY_NAME)
        fts_options |= FTS_NOSTAT;
```

The preceding snippet tells us that by omitting the "long" option, we will probably eliminate the corresponding `stat` calls. The improvement in the execution performance figures corroborates this insight:

```
$ /usr/bin/time ls -R >/dev/null
        1.08 real         0.28 user         0.77 sys
```

[27] netbsdsrc/bin/ls/ls.c:232–234

4.1.4 CPU-Bound Tasks and Profiling Tools

Having dealt separately with kernel-bound tasks, we can now say for the purpose of our discussion that CPU-bound tasks are those whose user time u is roughly equal to their real clock time r. These are the types of programs that can readily benefit from the algorithmic improvements we discuss in Section 4.2 and from some of the code improvements we present in Section 4.3.

The execution of most programs follows the 80/20 rule, also known as the *Pareto Principle*, after the nineteenth-century Italian economist Vilfredo Pareto, who, while studying the distribution of wealth and income, first expressed the often-occurring imbalance between causes and results. The law, applied to the distribution of a program's execution profile, states that 20% of the code often accounts for 80% of the execution time. It is therefore vitally important to locate the code responsible for the majority of the execution time and concentrate any optimization efforts on that area. (Financial analysts working on extracting actionable figures from aggregate data term this process *torturing the data until it confesses*.)

A *profiler* is a tool that analyzes where a program spends its execution time. By applying such a tool when our program runs on a representative input data set, we can easily isolate the areas that merit our further attention.

One approach for performing a profile analysis is *sampling*. At very short periodic intervals, the profiler interrupts the program's execution and keeps a note of the program's instruction pointer or stack trace. When the program has finished its execution, the accumulated data (typically counts of hits within predefined ranges) can be mapped to individual program functions and methods. The advantage of the sampling method is its efficiency and the relatively minor impact it has on the program's operation. On the other hand, the results obtained are coarse. If a method is called from two different contexts, we cannot find out which of the two contributed more to a program's runtime. Furthermore, the fixed-size address ranges that many sampling profilers use for counting the samples may result in routines lumped together or having their execution time misattributed. Finally, any sampling method may be subject to statistical bias.

A different approach involves having every routine call the profiler on its entry and exit. This is typically implemented by having the compiler generate appropriate calls in each routine's prologue and epilogue or by directly modifying the code before it gets executed. Many compilers for traditional compiled languages support an option for generating additional profiling code. A different approach, feasible in virtual machine environments, such as the JVM and Microsoft's CLR, is to have the

virtual machine execution environment register specific events (such as method calls or object allocations) with the profiler. The Java virtual machine tool interface (JVMTI) is a prime example of this approach—the *hprof* heap and CPU profiler supplied with the Java 2 SE 1.5 platform is a fully functional demonstration of a tool built on top of this interface. The disadvantage of call-monitoring profilers is the considerable effect they have on the program's operation. The program will typically run a lot slower—sometimes intolerably slower. In addition, the profiler calls interspersed on each and every call and return may affect the program's operation to an extent that renders the measurement results useless. Fortunately, in most cases, the Pareto Principle will make the code we are after stand out, even in the presence of the profiler interference.

Finally, a number of modern CPUs provide specialized hardware-based *event performance counters*. These registers are counters that get incremented every time a specific event occurs. Typical events are successful fetches and misses from the instruction or the data cache, mispredicted branches, misaligned data references, instruction fetch stalls, executed floating-point instructions, and resynchronizations at the microarchitecture level. By sampling these event performance counters in conjunction with the program counter, a specialized profiling tool, such as *oprofile*,[28] can generate reports that show which parts of the program contribute most to a specific event. We can thus, for example, use profiling through event performance counters to see which functions contribute most to data cache misses.

Typically, profiling is performed in two distinct steps. First, you run the program in a way that will produce raw profile data. This may involve a special compilation switch, such as `-pg` in many Unix compilers, or the invocation of the runtime environment (for example, the JVM) with appropriate flags. Many profiling systems allow you to aggregate the raw data of many runs, giving you the flexibility to create a profile data set from a series of representative program invocations. Whatever method you follow in this step, make sure that the data and operations you are profiling can be automatically and effortlessly repeated. A repeatable profiling data set will allow you to compare results from different runs, probably also *after* you have modified the program. In programs that present a user exclusively with a GUI, you may need to modify the program with hooks to allow its unattended operation. Such a modification will also come in handy for creating an automated test suite.

The second part of profiling often involves running a separate program to consolidate, analyze, and present the results gathered in the first phase. The Unix-based *gprof* tool is a typical example of such a program. The division between the data collection

[28] http://oprofile.sourceforge.net

Figure 4.2 EJP illustrates the Pareto Principle in the HSQLDB code

and the data analysis is required in order to minimize the effects the profiling will create to the running program. During the profiling phase, an efficient collection of raw data is preferable to a detailed analysis.

For a vivid illustration of the Pareto Principle in action, have a look at the profiling results shown in Figure 4.2. The profile is derived from inserting into an HSQLDB[29]

[29] http://hsqldb.sourceforge.net/

table 2,000 rows produced by listing a directory hierarchy with the *find* `-ls` command. We performed the profiling by using the *Extensible Java Profiler* (EJP).[30] To provide a repeatable profiling process, we steered away from the GUI-based `DatabaseManager` class, using HSQLDB's `ScriptTool` class, and specified an SQL script as part of the command line. The terminology used by the EJP for the two parts of the profiling operation is running the program under a *tracer* to collect the raw data and then invoking the GUI-based *presenter* for browsing the results. As you can see, exactly three, deeply nested, methods account for 81% of the program's total running time:

```
32.7% org.hsqldb.Table.insert(Object[], org.hsqldb.Session) (3.75 s)
21.7% org.hsqldb.Parser.getValue(int) (2.48 s)
26.8% org.hsqldb.Log.writeScript(boolean) (3.06 s)
```

Let us examine the way profiling results are typically presented. The results break down the time spent in a program across elements of its *call graph*. Every node in the graph corresponds to a single routine. Although Figure 4.2 presents the graph as a tree, it is important to appreciate that we are really talking about a graph; a routine can appear in multiple places in the tree listing, and there can even be cycles within the graph. What we typically find for a given routine in the profile report are:

- Its callers
- The other routines it calls
- The time spent executing code in the routine
- The time spent executing code in the routine and its descendants
- The preceding times as percentages of the total program execution

The reason we obtain this level of detail is that we are interested not only in which routine the program spends most of its time but also the call sequence leading to that point. In modular, structured programs, this data can contain important information. If, for example, we find out that a program spends 70% of its time allocating objects, we would like to know which of the object-allocation routines is mostly responsible for that overhead and optimize that routine, minimizing the calls to the object allocator. (Most probably, it would be difficult to optimize the actual object allocator.)

As an example of a CPU-bound task with a nontrivial call graph, consider the timing profile of the *sed* command when used to print words containing six-letter palindromes:

[30] http://ejp.sourceforge.net/

```
$ /usr/bin/time sed -n\
      "s/\(.\)\(.\)\(.\)\3\2\1/(\1\2\3-\3\2\1)/p"\
      /usr/share/dict/words
[...]
(col-loc)ation [...]
g(ram-mar) [...]
sh(red-der) [...]
s(nif-fin)g [...]
      203.59 real       194.27 user         1.63 sys
```

In this case, *sed* spent 95% of its time executing user code. Any reductions in the 203 seconds the command took to execute will have to come from algorithmic improvements in the code of *sed* and the libraries it relies on. To profile *sed*, we compiled it specifying the C compiler `-pg` option and then used the *gprof* command to create a report from the generated raw data file.

The format *gprof* uses for presenting the profile data can appear somewhat cryptic at first sight[31] but is well worth getting acquainted with, both because of the amount of useful data the report contains and because other programs, such as *ltrace*, also use it. For each routine, we get a listing containing its callers (parents), the routine's name, and the routines it calls (children). Figure 4.3 illustrates how the indented routine's name separates its callers from the routines it calls: In our example, the `vfprintf` general-purpose formatting function is called by the front-end functions `snprintf`, `sprintf`, and `fprintf`. For its operation, it calls `__sprint`, `localeconv`, `memchr`, and others. For callers, the `called/total` column represents the number of times the routine being examined was called from the routine on that line, followed by the total number of nonrecursive calls to it from all its callers—in our case, `snprintf` contributed 1 of the 5,908 calls to `vfprintf`. For the routine under examination, the `called+self` column contains the number of nonrecursive calls to that routine, followed by the number of recursive calls. Finally, for children, the `called/total` column contains the number of calls from the routine under examination, followed by the total number of all nonrecursive calls—in our case, 2 out of the 20,030 calls to `memchr` were made from the `vfprintf` body.

Starting at the top of the call graph for our *sed* invocation, we can see that no time was spent in `main`[32] but that 167 s were spent on its descendant, `process`[33] (the overhead of the profiler was another 28 s).

---

[31] In a retrospective paper on *gprof* [GKM04], its authors note: "All we can say for our layout is that after a while we got used to it."

[32] netbsdsrc/usr.bin/sed/main.c:112–164

[33] netbsdsrc/usr.bin/sed/process.c:94–263

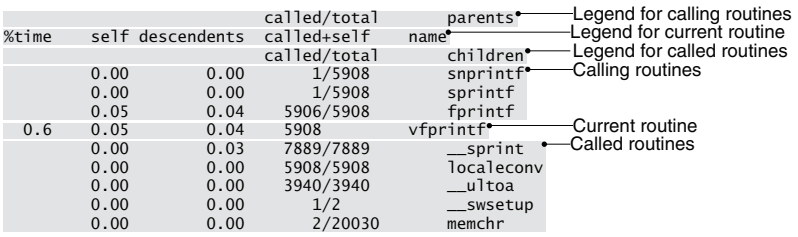

Figure 4.3 Example of *gprof* output for the `vfprintf` function

```
                          called/total         parents
%time    self dencendants called+self    name
                          called/total         children
                                               <spontaneous>
100.0    0.00      167.02                main
         0.61      166.41       1/1            process
         0.00        0.00       1/2            fclose
         0.00        0.00       1/1            compile
         0.00        0.00       1/1            add_compunit
         0.00        0.00       1/1            add_file
         0.00        0.00       2/2            getopt
         0.00        0.00       1/1            setlocale
         0.00        0.00       1/1            cfclose
         0.00        0.00       1/1            exit
```

Moving down five levels in the call graph profile, we see the main culprits. Both belong to the regular expression library.[34] The function `smatcher`,[35] which is called by `regexec`,[36] has two descendants that take 157 s. The actual execution of `smatcher` takes another 3.80 s. The numbers 3.80 and 157.44, appearing at the left of `regexec`, show how `smatcher`'s overhead is divided between its callers. Apparently, `smatcher` has only a single caller, `regexec`, and therefore all its overhead is attributed to this function. The overhead of the `smatcher`'s two descendants is divided between the `sfast`[37] and `sslow`[38] routines: `sslow` spends 14.8 s in its body out of a total 76.11 s spent on calls of it and its descendants; the corresponding numbers for `sfast` are 9.87 s and 46.59 s. Again, the figures here denote the time spent in `sslow` and `sfast`

[34] netbsdsrc/lib/libc/regex
[35] netbsdsrc/lib/libc/regex/engine.c:50, 140–299
[36] netbsdsrc/lib/libc/regex/regexec.c:165–191
[37] netbsdsrc/lib/libc/regex/engine.c:51, 699–783
[38] netbsdsrc/lib/libc/regex/engine.c:52, 790–869

as a result of being called by `smatcher`:

```
                        called/total         parents
%time    self dencendants  called+self    name
                        called/total         children
         3.80      157.44  235881/235881        regexec
 96.5    3.80      157.44  235881         smatcher
        14.80       76.11 2171863/2171863       sslow
         9.87       46.59 1321712/1321712       sfast
         8.16        0.00 1086032/1086032       sbackref
         0.47        1.43  218749/218760        malloc
         0.00        0.00     201/204           free
```

Moving another level down, we can now see how the 122.71 s spent on `sstep`[39] are divided between calls from `sfast` and `sslow`:

```
                        called/total         parents
%time    self dencendants  called+self    name
                        called/total         children
        46.59        0.00 9697679/25539316      sfast
        76.11        0.00 15841637/25539316     sslow
 73.5  122.71        0.00 25539316         sstep
```

Finally, we can also see that all the 14.8 s spent in the `sslow` body are attributed to its call from `smatcher`, and all the 76.11 s of its descendants are spent by `sstep`:

```
                        called/total         parents
%time    self dencendants  called+self    name
                        called/total         children
        14.80       76.11 2171863/2171863       smatcher
 54.4   14.80       76.11 2171863         sslow
        76.11        0.00 15841637/25539316     sstep
```

You can see the corresponding call graph in Figure 4.4. Each function node lists the time spent in the function's body and, below it in brackets, the time contributed by its dencendants. The same numbers (time spent directly in a called function and time spent in its dencendants) also appear as a sum on each edge as they propagate to the top of the graph.

In a number of cases, you will find that navigating the precise relationships of routines in a call graph is an overkill. A simple *flat profile* listing the routines and the time spent on each one may be enough to locate the program's hotspots. The following

[39] netbsdsrc/lib/libc/regex/engine.c:55, 886–998

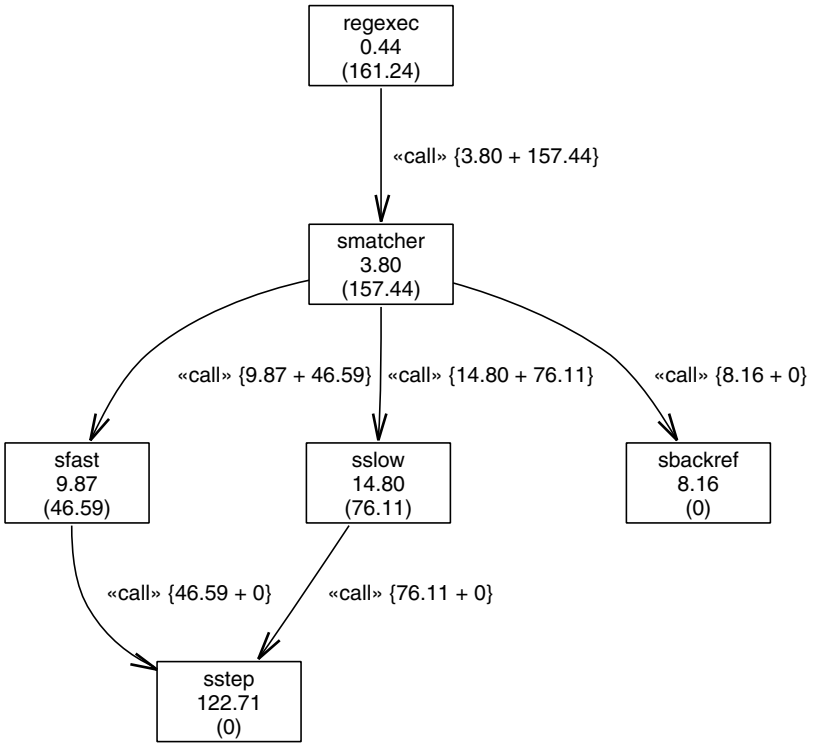

Figure 4.4 Propagation of processing times in a call graph

is the corresponding excerpt from the flat profile that *gprof* provides:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 62.8     122.71   122.71 25539316     0.00     0.00  sstep
 14.5     151.09    28.39                             .mcount
  7.6     165.90    14.80  2171863     0.01     0.04  sslow
  5.1     175.77     9.87  1321712     0.01     0.04  sfast
  4.2     183.93     8.16  1086032     0.01     0.01  sbackref
  1.9     187.73     3.80   235881     0.02     0.68  smatcher


[...]
  0.0     195.40     0.00        1     0.00     0.03  vfprintf
```

In most profile listings, routines are typically ordered by the percentage of total running time each routine takes. In the preceding excerpt, the column labeled "cumulative seconds" lists a running sum of the time taken by each routine and those listed above it. Note that the running sum does not correspond to any functional relationship between the routines it encompasses; it tells us only what part of a program's total running time is covered by the routines in question. Finally, note that the listing contains a routine titled `.mcount` that is not part of the program's source code. The `mcount` function[40] is the mechanism used for collecting the profiling information, and its appearance in the listing is simply an artifact of the profile-collection process.

In some cases, instead of profiling and examining the function calls in the entire program, we can obtain useful information by concentrating on the interactions between the program and its runtime library. Tools, such as *ltrace*, take advantage of the mechanisms used for dynamically linking application programs with their runtime library and trap all calls to the library, displaying them to the user. An important advantage of such programs is the fact that they can be directly applied on any dynamically linked executable program; in contrast to the *gprof* approach, no special compilation instructions are required beforehand to instrument the program.

As an example, consider the output *ltrace* generates when applied on the *paste* command:

```
$ ltrace paste expr.c paste.c >/dev/null

fgets("/*\t$NetBSD: expr.c,v 1.5 1997/07"..., 2049,
      0x08049260) = 0xbffff418
strchr("/*\t$NetBSD: expr.c,v 1.5 1997/07"..., '\n') = "\n"
printf("%s", "/*\t$NetBSD: expr.c,v 1.5 1997/07"...) = 62
fgets("/*\t$NetBSD: paste.c,v 1.4 1997/1"..., 2049,
      0x080493e8) = 0xbffff418
strchr("/*\t$NetBSD: paste.c,v 1.4 1997/1"..., '\n') = "\n"
```

Note how each call to `fgets`,[41] which will read characters looking for a newline, is immediately followed by a call to `strchr`.[42] If we were performance-tuning the *paste* program, it would have been a relatively easy task to combine the `fgets` and `strchr` calls, thus eliminating a redundant pass over each line read.

When developing software for an embedded system application (say, an MP3 player device), the tools we've discussed so far may not be available on your devel-

[40] netbsdsrc/lib/libc/gmon/mcount.c
[41] netbsdsrc/usr.bin/paste/paste.c:147
[42] netbsdsrc/usr.bin/paste/paste.c:156

opment platform. Nevertheless, time performance in embedded application domains is in many instances a critical concern. In such cases, we have to resort to simpler and lower-level techniques. Toggling discrete I/O signal lines before and after a process runs, while monitoring the line on an oscilloscope or logic analyzer, is a simple way to measure the process's runtime. Alternatively, if our hardware or operating system has a time counter, we can always store the counter's value before and after the execution of the code we're examining. This last approach is also useful for embedding profiling functionality into program code. Make it a habit to instrument performance-critical code with permanent, reliable, and easily accessible time-measurement functionality. This will allow you to measure the impact of your changes, based on hard facts rather than guesswork. As an example, the following code excerpt is used to measure a serial line's throughput in the Unix *tip* remote-connection program:[43]

```
time_t start_t, stop_t;
start_t = time(0);
while (1) {
    [...]
}
stop_t = time(0);
if (boolean(value(VERBOSE)))
    if (boolean(value(RAWFTP)))
        prtime(" chars transferred in ", stop_t-start_t);
    else
        prtime(" lines transferred in ", stop_t-start_t);
```

Exercise 4.3 Familiarize yourself with the code-profiling capabilities of your development environment by locating the bottlenecks in three different applications you are working on.

Exercise 4.4 Use the *swill* embedded web server library[44] and a modified version of the `mcount` function[45] to create a web interface for examining the operation of long-running programs.

Exercise 4.5 Enhance the *ltrace* implementation to provide a summary of library call costs.

[43] netbsdsrc/usr.bin/tip/cmds.c:288–371
[44] systems.cs.uchicago.edu/swill
[45] netbsdsrc/lib/libc/gmon/mcount.c

4.2 Algorithm Complexity

In a program that is CPU time–bound, the underlying algorithm is, by far, the most important element in determining its running time. The algorithm's behavior is especially significant if we care about how the program's performance will vary when the number of elements it will process changes. Computer scientists have devised the so-called *O-notation* (also referred to as the *big-Oh notation*) for classifying the running times of various algorithms. This notation expresses the execution time of an algorithm, ignoring small and constant terms in the mathematical formulas involved. Thus, for an algorithm that can process N elements in $O(N)$ time, we know that its processing time is linearly proportional to the number of elements: Doubling the number of elements will roughly double the processing time. On the other hand, doubling the number of elements would not affect the running time of an $O(1)$ algorithm and would increase the running time by a factor of 4 for an $O(N^2)$ algorithm ($2^2 = 4$). Note that in our analysis, we never express concrete running times, only relative algorithm efficiency classifications. When classifying the performance of algorithms, keep in mind their ranking from better to worst:

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^3) < O(2^N)$$

Note that the list is not complete; it presents only some common reference points. In Figure 4.5, you can see how the number of operations and the execution time change, depending on the algorithm's performance characteristics and the number of elements. We have assumed that each operation consists of a couple of thousand instructions and would therefore take about 1μs on a modern CPU. To provide a meaningful range, we have used a logarithmic scale on both axes. Note the following:

- An $O(\log N)$ algorithm, such as *binary search*,[46] will execute in a very small fraction of a second for 10 million (10^7) elements
- An $O(N)$ algorithm, such as a linear search, will execute in about 10 s for the same number of elements
- Even an $O(N \log N)$ algorithm—for example, *quicksort*[47]—will provide adequate performance (a couple of hours) for a batch operation on the same elements

[46] netbsdsrc/lib/libc/stdlib/bsearch.c
[47] netbsdsrc/lib/libc/stdlib/qsort.c

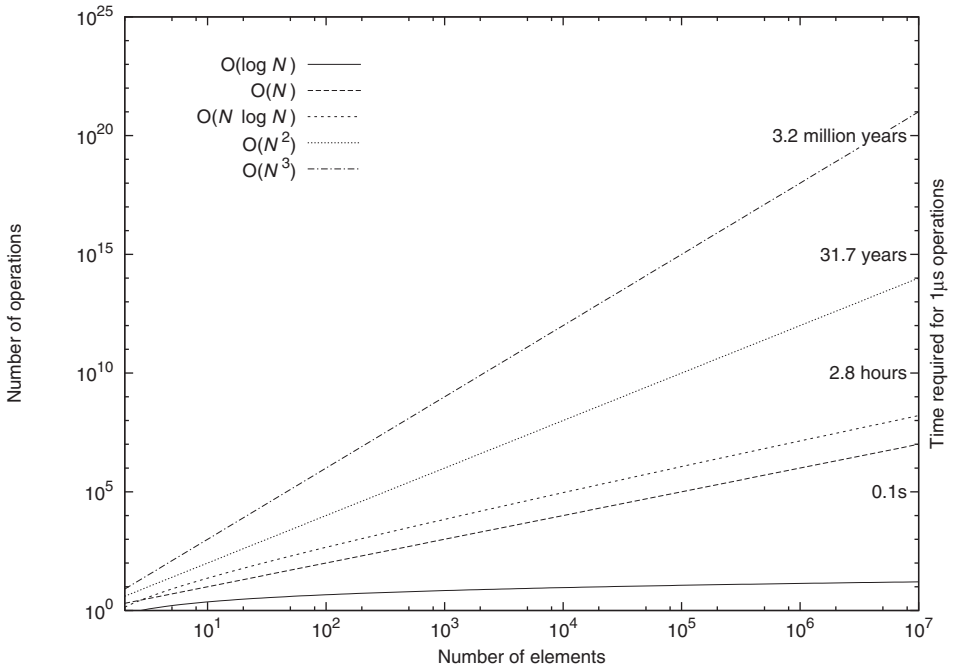

Figure 4.5 Relative performance of some common algorithm classes

- The $O(N^2)$ algorithm—for example, a doubly nested loop—will take more than 10 days when faced with more than 1 million (10^6) elements
- The $O(N^3)$ algorithm will encounter the same limit at around 10,000 (10^4) elements
- All algorithms provide acceptable performance when processing up to 50 elements

For all these cases, keep in mind that there is sometimes a big difference between the *average-case complexity* of an algorithm and its *worst-case complexity*. The classic example is the *quicksort* algorithm: A naive implementation has an average-case complexity of $O(N \log N)$ but a worst-case complexity of $O(N^2)$.

Let us now see some concrete examples of how you can determine an algorithm's performance from its implementation.

The following code is used in the Apache web server for determining the port of a URI request, based on the URI's scheme:[48]

```
static schemes_t schemes[] = {
    {"http", DEFAULT_HTTP_PORT},
    {"ftp", DEFAULT_FTP_PORT},
    {"https", DEFAULT_HTTPS_PORT},
    [...]
    {"prospero", DEFAULT_PROSPERO_PORT},
    {NULL, 0xFFFF}              /* unknown port */
};

API_EXPORT(unsigned short)
ap_default_port_for_scheme(const char *scheme_str)
{
    schemes_t *scheme; [...]

    for (scheme = schemes; scheme->name != NULL; ++scheme)
        if (strcasecmp(scheme_str, scheme->name) == 0)
            return scheme->default_port;
    return 0;
}
```

Note that if the `schemes` array contains N schemes, the body of the `for` loop will be executed at most N times; this is the best *guarantee* we can express about the algorithm's behavior. Therefore, we can say that this *linear search* algorithm for locating a URI scheme's port is $O(N)$. Keep in mind that this guarantee is different from predicting the algorithm's average performance. On many web servers, the workload is likely to consist mostly of HTTP requests, which can be satisfied with exactly a single lookup. In fact, the `schemes` table is ordered by the expected frequency of each scheme, to facilitate efficient searching. However, in the general case, a loop executed N times expresses an $O(N)$ algorithm.

i

Now consider the code of the NetBSD standard C library implementation for asserting the class of a given character (uppercase, lowercase, digit, alphanumeric, etc.) via the `isupper`, `islower`, `isdigit`, `isalnum`, `islower`, and similar macros:[49–51]

```
#define _U      0x01
#define _L      0x02
```

[48] apache/src/main/util_uri.c:72–73
[49] netbsdsrc/include/ctype.h:47–74
[50] netbsdsrc/lib/libc/gen/ctype_.c:55–75
[51] netbsdsrc/lib/libc/gen/isctype.c:54–59

```
#define _N      0x04
#define _S      0x08
[...]
const unsigned char _C_ctype_[1 + _CTYPE_NUM_CHARS] = {
    [...]
    _C,     _C|_S,  _C|_S,  _C|_S,  _C|_S,  _C|_S,  _C,     _C,
    _C,     _C,     _C,     _C,     _C,     _C,     _C,     _C,
    _C,     _C,     _C,     _C,     _C,     _C,     _C,     _C,
    _S|_B,  _P,     _P,     _P,     _P,     _P,     _P,     _P,
    _P,     _P,     _P,     _P,     _P,     _P,     _P,     _P,
    _N,     _N,     _N,     _N,     _N,     _N,     _N,     _N,
    _N,     _N,     _P,     _P,     _P,     _P,     _P,     _P,
    _P,     _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U,
    _U,     _U,     _U,     _U,     _U,     _U,     _U,     _U,
    [...]
};
const unsigned char *_ctype_ = _C_ctype_;

int
isalnum(int c)
{
    return((_ctype_ + 1)[c] & (_U|_L|_N));
}
```

The preceding code defines a number of constants that can be binary-ored together to represent the properties of a given character (`_U`: uppercase, `_L`: lowercase, `_N`: number, `_S`: space). The `_C_ctype` array contains the corresponding value for each character. As an example, the value for the character 'A' is `_U|_X`, meaning that it is an uppercase character and a valid hexadecimal digit. The function implementation of `isalnum` then tests only whether the array position for the corresponding character `c` contains a character classified as uppercase, lowercase, or digit. Irrespective of the number of characters N in the array, the algorithm to classify a given character will perform its task through a single array lookup, and we can therefore characterize it as an $O(1)$ algorithm. In general, on N elements, any operation that does not involve a loop, recursion, or calls to other operations depending on N expresses an $O(1)$ algorithm.

For an algorithm with much worse performance characteristics, consider the X Image Extension library code that performs a convolution operation on a picture element. This operation, often used for reducing a picture's sampling artifacts, involves processing the elements with the values of a square kernel:[52]

[52] XFree86-3.3/xc/lib/XIE/elements.c:783–786

```
for (i = 0; i < ksize; i++)
    for (j = 0; j < ksize; j++)
        *fptr++ = _XieConvertToIEEE (
            elemSrc->data.Convolve.kernel[i * ksize + j]);
```

If the kernel's dimension is `ksize`, the outer loop will execute the inner loop `ksize` times; the innermost statement will therefore be executed `ksize * ksize` times. Thus, the preceding algorithm requires $O(N^2)$ operations for a convolution kernel whose dimension is N. It is easy to see that if the preceding example used three nested loops, the algorithm's cost would be $O(N^3)$. Thus, the general rule is that K nested loops over N elements express an $O(N^K)$ algorithm. i

Note that when we are expressing an algorithm's performance in the O notation, we can omit constant factors and smaller terms. Consider the loop sequence used for detecting the existence of two same-value pixels:[53]

```
for (i = 0; i < count - 1; i++)
    for (j = i + 1; j < count; j++)
        if (pixels[i] == pixels[j])
            return False;
```

For $N =$ `count`, the inner part of the loop will be executed

$$\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2} = \frac{N^2 - N}{2}$$

times. However, we express the preceding function as simply $O(N^2)$, conveniently omitting the $1/2$ factor and the $-N$ term. We can therefore say that the last two algorithms we examined have the same *asymptotic behavior*: $O(N^2)$. We can do this simplification because the term N^2 *dominates* the algorithm's cost; for sufficiently large values of N, this is the term that will decide how this algorithm compares against another. As an example, although an algorithm requiring $1{,}000N$ operations will fare better than an algorithm requiring N^2 operations for $N > 1,000$, our $\frac{N^2-N}{2}$ algorithm will be overtaken by the $1{,}000N$ algorithm only for values of $N > 2,001$. i

We can recognize algorithms that perform in $O(\log N)$ by noting that they divide their set size by two in each iteration, thus requiring $\log_2 N$ operations. *Binary search* is a typical example of this technique and is illustrated in the following code excerpt:[54]

[53] XFree86-3.3/xc/lib/Xmu/Distinct.c:82–85
[54] XFree86-3.3/xc/lib/X11/LRGB.c:1191–1205

```
while (mid != last) {
    last = mid;
    mid  = lo + (((unsigned)(hi - lo) / nKeyPtrSize) / 2) *
           nKeyPtrSize;
    result = (*compar) (key, mid);
    if (result == 0) {
         memcpy(answer, mid, nKeyPtrSize);
         return (XcmsSuccess);
     } else if (result < 0) {
         hi = mid;
     } else {
         lo = mid;
     }
}
```

Note how in each iteration, one of the range's two boundaries, `lo` and `hi`, is set to the range's middle, `mid`, effectively halving the search interval.

There are many other code structures and classes of algorithm complexity that are difficult to trivially recognize from the source code. Typically, these are related to recursive operations or corresponding data structures. Fortunately, nowadays such algorithms are almost never invented by the programmer writing the code. You will therefore be able to look up the algorithm's performance in a textbook or locate a helpful comment giving you the details you require:[55]

```
/** [...]
 * This implementation uses a heap-based callout queue of
 * absolute times.  Therefore, in the average and worst case,
 * scheduling, canceling, and expiring timers is O(\log N)
 * (where N is the total number of timers).  [...]
```

Exercise 4.6 Draw a table listing some typical data set sizes you are dealing with and the time it would take to process them using algorithms of various complexity classes. Assume that each basic operation takes 1μs (a few hundred of instructions).

Exercise 4.7 Sometimes, an algorithm of $O(N^2)$ (or worse) complexity is disguised through the use of function calls. Explain how such a case would appear in the source code.

[55] ace/ace/Timer_Heap_T.h:70–78

4.3 Stand-Alone Code

After we have established that the code we are examining is based on a reasonably efficient algorithm, it may be time to consider the actual instructions executed within the algorithm's body. In the previous section, we hinted that modern processors execute billions of instructions every second. However, seldom will a source code statement correspond to a single processor instruction. In this and the following sections, we examine the distinguishing characteristics of increasingly expensive operations.

The statement[56]

```
i++;
```

indeed compiles into a single processor instruction; on an i386 architecture:[57]

```
incl -2612(%ebp)
```

More complex arithmetic expressions will typically compile into a couple of instructions for every operator. However, keep in mind that in languages supporting operator overloading, such as C++ and C#, the cost of an expression can be deceptive. As an example, the operator `+=`, when applied to `ACE_CString` objects of the ACE framework, maps into an implementation of 48 lines,[58] which also include calls to other functions.

The cost of a call to a function or a method can vary enormously, between 1 ns for a trivial function and many hours for a complex SQL query. As we indicated, the overhead of a function call is minimal and should rarely be considered as a contributing factor to a program's speed. You may keep in mind some rules of thumb regarding the costs of function calls and method invocations.

- Virtual method invocations in C++ often have a larger cost that the invocation of a nonvirtual method, which is typically close to that of a simple function call.
- Compiler optimizations may compile some of a program's functions and methods *inline*, substituting the function's body in the place of the call, effectively removing the overhead of the function call. This optimization is

[56] netbsdsrc/libexec/talkd/announce.c:123

[57] Increment the word located −2,612 bytes away from this function's *frame pointer* `ebp`. The frame pointer is a processor register used for addressing a function's arguments and local variables. See Section 5.6.1.

[58] ace/ace/SString.cpp:391–438

often performed when the body of a function or a method is smaller than the corresponding call sequence.

- In C++ and C99 programs, a programmer can specifically direct the compiler to try to inline a function by defining it with the `inline` function specifier. Many C compilers predating the C99 standard also support this keyword.
i You will find the `inline` keyword typically applied on performance-critical functions that get called only a few times or have a short body, so that the inline expansion will not result in code bloat:[59]

```
static inline struct slist *
this_op(struct slist *s)
{
    while (s != 0 && s->s.code == NOP)
        s = s->next;
    return s;
}
```

- In C and C++, what appears as a function call may actually be a macro that will get expanded before it gets compiled into machine-specific code. The `isalnum` C library function we examined in Section 4.2 is typically also implemented as a macro:[60]

```
#define isalnum(c)  ((int)((_ctype_ + 1)[c] & (_U|_L|_N)))
```

  The function definition with the same name is used in cases in which the function's address is used in an expression or the header file containing the macro definition is not included.

- Intrinsic functions of the C/C++ library, such as `sin`, `strcmp`, and `memcpy`, may be directly compiled in place. For example, the `memcmp` call[61]

```
if (memcmp((char *)&termbuf.sg, (char *)&termbuf2.sg,
           sizeof(termbuf.sg)))
```

  gets compiled in the following i386 instruction sequence:

```
        movl $termbuf,%eax
        movl %eax,%esi
```

[59] netbsdsrc/lib/libpcap/optimize.c:643–650
[60] netbsdsrc/include/ctype.h:92
[61] netbsdsrc/libexec/telnetd/sys_term.c:245–246

```
movl $termbuf2,%edi
movl $6,%ecx
cld
repz cmpsb
je .L18
```

In that code, the `repz cmpsb` instruction will compare `%ecx` (that is, 6 or `sizeof(termbuf.sg)`) bytes located at the memory address `%esi` (i.e., `termbuf`) with `%ecx` bytes located at the memory address `%edi` (i.e., `termbuf2`). As you can see, the sequence does not contain any calls to an external library function.

- In a few cases, you may encounter *inline assembly* instructions intermixed with C code, using compiler and processor-specific extensions. As an example, the following excerpt is an attempt to provide a more efficient implementation of the Internet Protocol (IP) checksum on the ARM-32 architecture:[62]

```
/*
 * Checksum routine for Internet Protocol family headers.
 * This routine is very heavily used in the network
 * code and should be modified for each CPU to be as
 * fast as possible.
 * ARM version.
 */

#define ADD64 __asm __volatile(" \n\
ldmia %2!, {%3, %4, %5, %6} \n\
adds %0,%0,%3; adcs %0,%0,%4 \n\
adcs %0,%0,%5; adcs %0,%0,%6 \n\
ldmia %2!, {%3, %4, %5, %6} \n\
[...]
```

To understand such code sequences, you will need to refer to the specific processor handbook and to the compiler documentation regarding inline symbolic code. Processor-specific optimizations are by definition nonportable. Worse, the "optimizations" may be counterproductive on newer implementations of a given architecture. Before attempting to comprehend processor-specific code, it might be worthwhile to replace the code with its portable counterpart and to measure the corresponding change in performance.

[62] netbsdsrc/sys/arch/arm32/arm32/in_cksum_arm32.c:56–69

The properties we have examined so far apply mainly to languages that compile to native code, such as C, C++, Fortran, and Ada. These languages have a relatively simple performance model, and we therefore can—with some experience—easily predict the cost associated with a statement by hand compiling the statement into the underlying instructions. However, languages that compile to bytecodes, such as Java, the Microsoft .NET language family, Perl, Python, Ruby, and Tcl, exhibit a much higher semantic distance between each language statement and what gets executed underneath. Add to the mix the sophisticated optimizations that many virtual machines perform, and judging about the relative merits of different implementations becomes a futile exercise.

Exercise 4.8 By executing a short sequence of code many times in a loop, you can get an indication of how expensive an operation is. Write a small tool for performing such measurements, and use it to measure the cost of some integer operations, floating-point operations, library functions, and language constructs. Discuss the results you obtained. Note that tight loops, such as the ones you will run, are in most cases not representative of real workloads. Your measurements will probably overrepresent the effects of the processor's cache and underrepresent its pipelined execution performance.

4.4 Interacting with the Operating System

There are some kinds of functions and methods with a fixed, large, and easily predictable cost. The common characteristic of these expensive functions is a trip to another process, typically also involving a visit to the system's operating system kernel. In modern systems, any visit outside the space of a given process involves an expensive *context switch.* A context switch involves saving all the processor-related details of the executing process in memory and loading the processor with the execution details of the other context: for example, the system's kernel. Upon return, this expensive saving and restoring exercise will have to be repeated in the opposite direction. To get an idea of the data transfer involved in a context switch, consider the contents of the NetBSD structure used to save context data on Intel processors:[63]


```
struct  sigcontext {
    int sc_gs;
    /* [ 15 more register value fields omitted ] */
    int sc_ss;
    int sc_onstack;     /* sigstack state to restore */
    int sc_mask;        /* signal mask to restore */
```

[63] netbsdsrc/sys/arch/i386/include/signal.h:56–80

```
    int sc_trapno;
    int sc_err;
};
```

Apart from saving and restoring the 84 bytes of the preceding structure, a context switch also involves expensive CPU instructions to adjust its mode of operation, changing various page and segment tables, verifying the boundaries of the user-specified data, copying data between user and kernel data space, and, often, the invalidation of data held in the CPU caches.

The cost of making calls across the process boundary is so large that it applies to most programming languages. In the following paragraphs, we examine three increasingly expensive types of calls:

1. A system call to an operating system kernel function
2. A call involving another process on the same machine
3. A call involving a process residing on a different machine

For each type of call, we show the context switching involved in a representative transaction by means of a UML sequence diagram. To spare you the agony of waiting for the final results, Table 4.2 contains a summary of the overheads involved.

Please keep in mind that the table does not contain benchmark results of relative performance between the corresponding operating systems. Although we performed

Table 4.2 Overhead Introduced by Context Switching and Interprocess Communication

	Windows XP	Linux (2.4.26)	FreeBSD (5.2)
Function call	1.3 ns	1.3 ns	1.3 ns
System call (`open/close`)	5,125 ns	1,859 ns	2,850 ns
Local IPC (pipe `read/write`)	13 μs	4.3 μs	3.4 μs
Local IPC (socket `send/recv`)	48 μs	21 μs	42 μs
Remote IPC (TCP `send/recv`)	153 μs	165 μs	176 μs
Remote IPC (DNS query over UDP)	7,114 μs	1,176 μs	541 μs

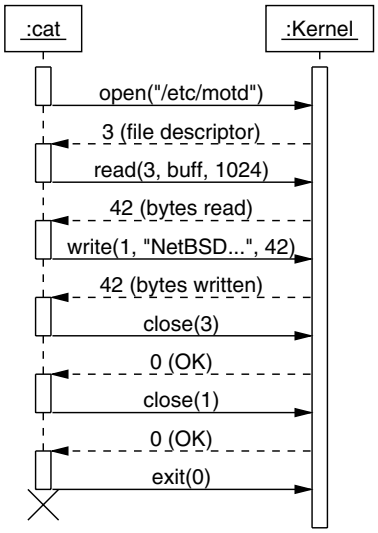

Figure 4.6 System calls of a simple *cat* invocation

the measurements on the same hardware and tried to minimize the influence of irrelevant factors, we never aimed to produce representative performance figures that can be extrapolated to real applications—as a properly designed benchmark is supposed to do. The only thing the table's figures meaningfully represent is a rough picture of the relative costs of the various operations *on the same operating system*.

As an example of the nature and expense of local system calls, consider the calls involved when *cat* [64] is used to print the file `/etc/motd`.[65] The sequence of five system calls illustrated in Figure 4.6, and their corresponding overhead, is unavoidable when printing a file, no matter what language the program is written in. In fact, Figure 4.6 is slightly simplified because it does not include some initial system calls issued for loading dynamically linked libraries and those used to determine the file's type. As you see, at least five system calls and the associated cost of the kernel round-trips are required for copying a file's contents to the standard output (by convention file descriptor 1). The numbers we list in Table 4.2 describe the cost of the `open` and `close` system calls on the system's `null` device; the costs associated with the calls in Figure 4.6 are likely to be higher, as they include the overhead of disk I/O operations.

[64] netbsdsrc/bin/cat/cat.c
[65] netbsdsrc/etc/motd

However, no matter what a system call is doing, each system call incurs the cost of two context switches: one from the process to the kernel and one from the kernel back to the process. This cost is more than two orders of magnitude greater than the cost of a simple function call or method invocation and is something you should take into account when examining code performance.

For this reason, you will often see code going to considerable lengths to avoid the cost of a system call. As an example, the following implementation of the C `perror` function stores the sequence of the four output strings (the user message, a colon, the error message, and a newline) in an array and uses the gather version of the `write` system call, `writev`, to write all four parts with a single call:[66]

```
void
perror(const char *s)
{
    register struct iovec *v;
    struct iovec iov[4];
    static char buf[NL_TEXTMAX];

    v = iov;
    if (s && *s) {
        v->iov_base = (char *)s;
        v->iov_len = strlen(s);
        v++;
        v->iov_base = ": ";
        v->iov_len = 2;
        v++;
    }
    v->iov_base = __strerror(errno, buf, NL_TEXTMAX);
    v->iov_len = strlen(v->iov_base);
    v++;
    v->iov_base = "\n";
    v->iov_len = 1;
    (void)writev(STDERR_FILENO, iov, (v - iov) + 1);
}
```

Now consider a local interprocess communication case: for example, writing a message to the system's log. Processes executing in the background (Unix daemons, Windows services) are not supposed to display warning and error messages on a terminal or a window. Such messages would be annoying and often also displayed to the wrong person. Instead, background processes send all their diagnostic output to

[66] netbsdsrc/lib/libc/stdio/perror.c:60–83

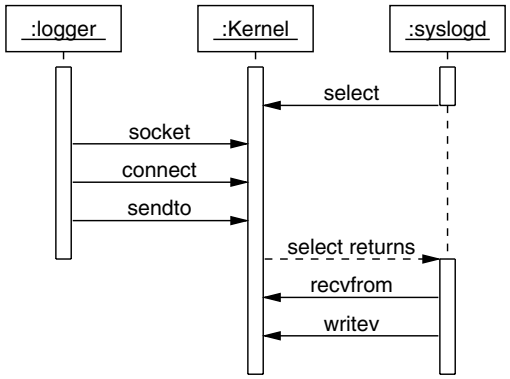

Figure 4.7 System calls for local IPC in a *logger* invocation

a system log. On properly maintained systems, a system administrator periodically audits the log files, looking for problems; in many cases, administrators will also examine the log files, looking for hints that will help them diagnose an existing problem.

On both Unix and Windows systems, the system log is maintained by a separate program. The program receives logging requests from processes and writes them to the system log in an orderly, timestamped fashion and in an appropriate format. Thus, program interactions with the logging facility have to cross not only the kernel boundary of the process creating a log entry but also the boundary of the system's logging process. The numbers we list in Table 4.2 illustrate the cost of such a local interprocess communication (IPC) operation and were calculated by measuring the amortized cost of a single small `send`/`recv` transaction.

The overhead of the IPC operation is almost an order of magnitude larger than a simple system call, and this can be easily explained by examining the system calls involved in a typical IPC operation. Figure 4.7 is a sequence diagram depicting the system calls made when the *logger* Unix command is run to register a system log message. Initially, the system's logger *syslogd* is waiting idly for a `select` system call to return, indicating that *syslogd* can read data from the system-logging socket; *logger* will indeed establish a socket for communicating with *syslogd* (`socket`), `connect` that socket to the *syslogd* endpoint, and send (`sendto`) the message to *syslogd*. At that point, the *syslogd*'s `select` call returns, indicating that data is available; *syslogd* will then read (`recvfrom`) and assemble the data from the socket into a properly formatted log entry, and write it (`writev`) to the system log file. As you can see, this

sequence involves 6 system calls and 12 context switches. It is therefore natural for an IPC exchange to be a lot more expensive than a system call. (Note that the number listed in Figure 4.7 reflects a different setup whereby the cost of an initial `socket` and `connect` operation is amortized over many `send` calls.)

The communication with a logger process we examined is only one example of an expensive local IPC operation. Other examples of the IPC cost being incurred include the communication between filter processes in a pipeline, the interaction with a locally running RDBMS, and the execution of I/O operations through a local X Window System server. (This last case applies to all X client GUI programs.) It is also worth noting that there are cases in which some of the data copies we described can be eliminated. As an example, in the FreeBSD system, when a sending process writes data to a pipe through a sufficiently large buffer (`PIPE_MINDIRECT`—8,192 bytes long), the write buffer will be fully mapped to the memory space of the kernel, and the receiving process will be able to copy the data directly from the memory space of the sending process. We examine some more cases when we discuss file-mapping operations in Section 5.4.2. Eliminating data copies across different layers of a network stack is also a favorite pastime of network researchers.

Finally, consider a remote interprocess communication case, such as contacting a remote DNS server to obtain a host's address. Such an exchange is, for example, performed every time a workstation's user visits a web page on a different host. The last set of numbers listed in Table 4.2 corresponds to such an operation. Note how the time cost of the remote IPC is three more orders of magnitude larger than the (already large) cost of the local IPC. We can appreciate this cost by examining the corresponding interactions depicted in Figure 4.8. The figure represents the calls taking place when the *ping* command queries a remote DNS server to obtain a host's address. The initial sequence of system calls—`socket`, `connect`, and `sendto`—is the same as the one we examined in the local IPC case. Because, however, the DNS query packet is not addressed to a local process, the kernel will queue the packet for remote delivery. When the kernel's networking subsystem is ready to send the packet (immediately in our case: We assume an unloaded system and network), it will assemble the packet appropriately for the local network, put it in a buffer of the network interface hardware, and instruct the hardware to send the packet over the network. At some point, *ping* issues a `recvfrom` system call to obtain the query's answer. This call, however, remains blocked until the corresponding packet has been received from the remote end.

At the remote end, the arrival of the packet over the network will probably trigger an interrupt, causing the kernel's networking subsystem to collect the packet from

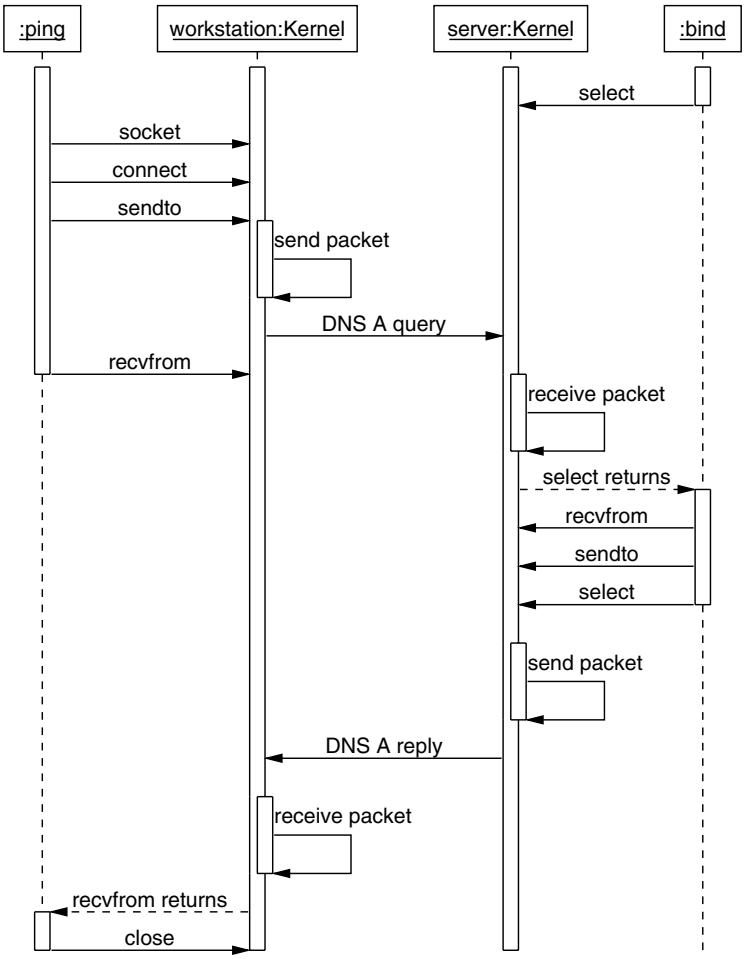

Figure 4.8 System calls in remote DNS IPC for a *ping* name query

the networking hardware buffer. The domain name server process (*bind*), which was blocked waiting for input with a `select` system call, resumes its operation, retrieving the packet with a `recvfrom` system call. After possibly consulting other servers or local files (our tests did not involve any of these expensive operations), it will send the query's reply with a `sendto` call and block again, waiting for input on a `select`. The sent packet is again queued on the remote end, transmitted over the network, received at the local end, and delivered to *ping* as the result data of the `recvfrom`

call. At that point, *ping* can `close` the socket and continue its operation. Note that the exchange we described involved UDP network packets, which are exchanged without any formalities; an IPC operation relying on TCP packets would be even more expensive, easily tripling the number of packets that would have to be exchanged for a simple operation. The moral of this example is simple: Remote IPC, including protocols such as RMI and SOAP, is expensive.

Up to now, we have twice encountered the use of *blocking* operations to interact with the operating system, other processes, and peripherals. These operations, like the system calls to `select` and `poll` with a nonzero timeout value or `read` on a file descriptor opened in blocking mode (the default), represent the correct and efficient way for a process to interact with its environment. If the remote end is not ready to complete the operation, our calling process will relinquish control, and the operating system will be able to schedule other tasks to execute until the remote end becomes ready. Other, equally efficient methods are the calls to `GetMessage` under Windows or the setup of a *callback* function that will get called when a specific event occurs. In most cases, this strategy of waiting for an operation's completion will not affect the wall clock time our process takes to execute but will drastically improve our process's processor time requirements. The alternative approach, involving *polling* our data source at periodic time intervals to see whether the operation has completed—also called a *busy-wait*—typically wastes computing resources. On a system executing multiple processes or threads, the busy-wait approach will result in a drop in the overall system's performance. Therefore, whenever you find code in a loop polling to determine an operation's completion, look at the API for a corresponding blocking operation or callback that will achieve the same effect. If you are unable to find such an operation, you've probably encountered either a design problem or a hardware limitation (a few low-end hardware devices are unable to signal their hosts that they have completed a task). Repeated calls, such as `select` and `poll` with a zero timeout, `read` on nonblocking file descriptors, and `PeekMessage`, are prime examples of inefficient code. When timeouts and busy loops *are* used, they should represent an exceptional scenario rather than the normal operation. Furthermore, the timeout values should be many orders of magnitude larger than the time required to complete the typical case.

Exercise 4.9 Measure the typical expected execution time of some representative operating system API calls. For one of these calls (for example, `open` or `CreateFile`), differentiate between the time required for successful completion and for each possible different error. If your operating environment supports it, include in the error cases you examine parameters containing invalid memory addresses.

Exercise 4.10 The cost of operating system calls can in some cases be minimized by using more specialized calls that group a number of operations with a single call. As an example, some versions of Unix provide the `readv` and `writev` calls to perform scatter/gather I/O operations, `pwrite` to combine a `seek` with a `write`, and `sendfile` to send a file directly to a socket. Outline the factors that lead to increased performance under this approach. Discuss the problems associated with it.

4.5 Interacting with Peripherals

One other source of expensive operations is a program's interaction with slower peripheral devices. We got a taste of this cost when we examined remote IPC operations, which have to go through the host's network interface. Programs also often resort to the use of secondary storage, in the form of magnetic hard disks, either to provide long-term storage to their data or to handle data that cannot reasonably fit in the system's main memory. When possible we should try to avoid or minimize a program's interactions with slow peripherals. One reason that embedded databases, such as HSQLDB,[67] are in same cases blazingly fast is that they can keep all their data in the system's main memory.

We can get a rough idea of the relative costs by examining the numbers in Table 4.3. We measured the table's top four figures on the system on which this book was written, and they represent ideal conditions: copying of large aligned memory blocks, sequential writes to the disk with a block size that allows the operating system to interleave write operations, and flooding an 100Mb/s ethernet with UDP packets. Even in this case, we see that writing to the system's main memory is one or two orders of magnitude faster than writing to the hard disk or a network interface. The differences in practice can be a lot more pronounced.

When accessing a data element at a random location of a disk drive, the operation may involve having the disk head seek to that location and waiting for the disk platter to rotate to bring the data under the head. As we can see in Table 4.3, both of these figures are four orders of magnitude larger than the time required for sending a byte to the disk interface. Of course, programs typically write data to disk in larger units, so the seek and rotational overhead is amortized over many more bytes. In addition, both the disk controller and the operating system maintain large memory caches of disk data; in many situations, the program's data will reside in such a cache, avoiding the expense of the mechanical operations. Also keep in mind that on typical network

[67] hsqldb

Table 4.3 Overhead Introduced by Slower Peripherals

Operation	Time
Copy a byte from memory to memory (cached)	0.5 ns
Copy a byte from memory to memory (uncached)	2.15 ns
Copy a byte from memory to the disk	68 ns
Copy a byte from memory to the network interface	88 ns
Hard disk seek time (average)	12 ms
Hard disk rotational latency (average)	7.1 ms

interfaces and load scenarios, it is seldom possible to saturate an ethernet above 50% of its rated capacity.

Exercise 4.11 Technology advances often render useless some laboriously implemented peripheral-specific optimizations. In some cases, the optimizations may even be counterproductive, imposing a higher CPU load and antagonizing a new peripheral's optimization methods. Examples of nowadays useless optimizations include operating system-based sector interleaving and head scheduling for hard disks,[68] ordering of line segments to minimize travel distance in pen plotter output, and minimizing the number and perceived cost of terminal control sequences in character displays.[69] On the other hand, each one of these optimizations would in its time, make a difference between a responsive and an unusable application. How can you recognize such legacy code? What should you do with it? Discuss design and implementation strategies that will minimize the deleterious effects of peripheral-support code when it becomes outdated in the future.

4.6 Involuntary Interactions

Sometimes, the costly interaction with the slow disk subsystem and the operating system may come as a surprise. Figure 4.9 shows the time the *make*[70] program takes to read files containing a (very) large number of targets. When parsing files with up to 46,000 targets, the relationship between the number of targets and the execution time appears to be linear, providing a reasonable performance for the types of files *make* typically processes. After that point, however, things begin to go horribly wrong. Whereas *make* will process a file with 46,000 targets in 13.7 s, it takes 30.71 s for 47,000 targets and 368 s for 51,000 targets. The reason behind the spectacular perfor-

[68] netbsdsrc/sys/arch/vax/vsa/hdc9224.c:122–124
[69] netbsdsrc/lib/libcurses/refresh.c:452–704
[70] netbsdsrc/usr.bin/make

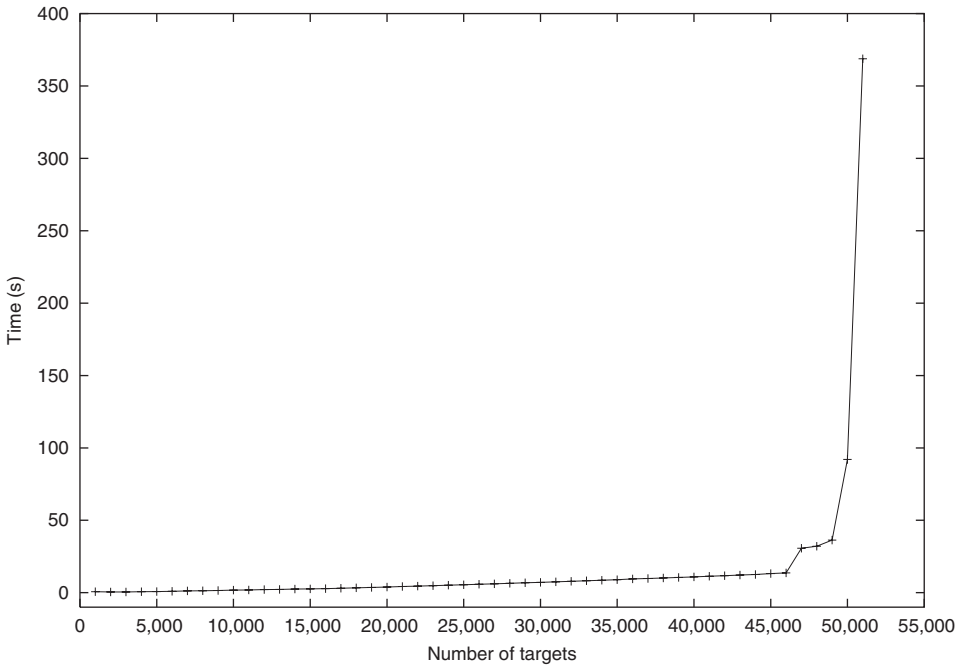

Figure 4.9 The effect of thrashing on runtime performance

mance drop can be traced to an (involuntary) interaction with the slow peripherals we examined in the previous section. When it reads a file, *make* stores its entire contents in memory. This is a reasonable decision for normally sized files. However, after some point, the files we used to stress-test the performance of *make*, required more memory than the miserly 32MB of RAM the machine we run the tests on was equipped with. At that point, the operating system employed *paging* to create memory space:[71] moving less used memory pages to secondary storage, freeing them for the insatiable appetite of *make*. Each time one of the pages moved to secondary storage was needed again, the processor's memory management unit caused a *page fault*, and the operating system replaced another less used page with the needed page.

This strategy for implementing virtual memory is in most cases worthwhile because typical programs and workloads exhibit a *locality of reference*: Only a small proportion of the memory they occupy is used at different times. When, however, a

[71] netbsdsrc/sys/vm/vm_page.c

program wanders all over the memory space allocated to it or the amount of virtual memory in use is out of proportion to the physical memory available, the system's performance will rapidly degrade, owing to the very large number of page faults. A page fault brings together all the performance-degrading elements we have examined so far. It involves a context switch to the kernel where the handling code resides, access to the slow secondary storage, and, in some cases, such as diskless systems, network access. Instructions interrupted by frequent page faults will therefore execute many orders of magnitude slower than their normal pace. When the excessive number of page faults essentially prevents the system from getting any useful work done, as is the case in the last execution illustrated in Figure 4.9, the system is said to be *thrashing*.

Thrashing differs from the other elements affecting performance we have examined so far, both in its relationship to code and in its behavior. Algorithms and expensive operations are typically associated with specific areas of the code, which can be located and improved using appropriate profiling tools. Thrashing, on the other hand, is related to a program's memory footprint and the program's locality of reference properties. Both aspects are difficult to isolate in the code. However, the characteristic behavior of a system moving into thrashing can help us identify it as a cause of a performance problem. When a system's performance over workload shape abruptly changes once the workload reaches a given point, the most probable cause is thrashing. The change in shape is typically dramatic and visible irrespective of the complexity characteristics of the underlying algorithm. Thrashing problems are typically handled in three different ways:

1. Reducing the system's memory footprint
2. Using a system with a larger amount of physical memory
3. Improving the system's locality of reference

The last approach is the most difficult to employ, but in some cases, you may indeed encounter code that makes assumptions regarding its locality of reference, as illustrated in the following comment:[72]

```
/*
 * XXX
 * A seat of the pants calculation: try to keep the file in
 * 15 pages or less.  Don't use a page size larger than 10K
 * (vi should have good locality) or smaller than 1K.
 */
```

[72] netbsdsrc/usr.bin/vi/common/exf.c:204–209

Another form of involuntary interaction occurs when interrupts interfere with the execution of code. Peripherals and timers often use an *interrupt* to notify the operating system or a program that an event has occurred. Execution then is temporarily transferred to an *interrupt service routine*, which is responsible for handling the specific event. If on a given system, interrupts occur too frequently or the interrupt service routine takes too long to execute, performance can degrade to the point of making the system unusable. Interrupts will also mess up our profiling data: If their execution time is not recorded, the profile results will not match our subjective experience or the wall clock times; if their execution time is tallied together with our code, our code will appear to be mysteriously slow. When dealing with interrupts, we should try to minimize the number of interrupts that can occur and the processing required for each interrupt. To minimize the number of interrupts, our (low-level) code must interact with the underlying hardware, using the hardware's most efficient mechanisms, such as buffers and direct memory access (DMA). To minimize an interrupt service routine's execution time, we can either queue an expensive request for later synchronous processing, or we can carefully optimize its code:[73]


```
* This routine could be expanded in-line in the receiver
* interrupt routine to make it run as fast as possible.
```

Exercise 4.12 The performance of some algorithms degrades to an abysmal level once the available main memory is exhausted and thrashing sets in. Yet for many operations, there exist other algorithms tuned for operating on data stored in secondary storage. Examine the implementation of GNU sort,[74] and explain how it can efficiently sort multigigabyte files, using only a small fraction of that space of physical memory.

Exercise 4.13 In a number of cases, a memory-hungry program could adjust its operational profile at the onset of thrashing. As an example, a Java VM implementation could continuously expand its memory pool and garbage collect only once the available physical memory is exhausted. Discuss which operating system calls of your environment could be used to provide a reliable indication of physical memory exhaustion.

4.7 Caching

A *cache* was originally a part of a memory hierarchy that was used to couple the speed difference between the fast CPU and the slower main or peripheral memory.

[73] netbsdsrc/sys/kern/tty_tb.c:184–185

[74] http://www.gnu.org/software/coreutils/

Nowadays, the term is used pervasively to denote a temporary place where a result of (a typically expensive) operation is stored to facilitate faster access to it. In the computer architecture field, we can envisage a continuum moving from fast, transient, small, and expensive CPU registers toward slow, permanent, huge, and cheap offline media. In-between lie the level 1, level 2, and, sometimes, level 3 caches associated with the processor, the main memory, disk-based virtual memory, and disk-based files. Because each level of this caching hierarchy is smaller than the one below it, a large number of different mechanisms are used for creating a map between the small set of elements that are available at one level and the larger set available at the lower level. Moving downward in the hierarchy, we will encounter register allocators, set associative cache blocks, page tables and translation look-aside buffers, filesystems, and offline media management systems. All caches capitalize on the *locality of reference* principle. Once an element is accessed, it is likely to be accessed again soon; elements near to that element are also likely to be accessed soon.

In programs, you will find data caches (also often termed *buffers*) used to combat all the different factors that affect a program's execution speed: inefficient algorithms, expensive instructions, interactions with the operating system and other processes, and access to slow peripherals. The caching of code is typically performed under the control of the CPU and the operating system. In some cases, you may be able to force the linker to collocate critical code sections close together to maximize your code's locality of reference.

4.7.1 A Simple System Call Cache

The `pwcache` library function `user_from_uid`,[75] illustrated in Figure 4.10, is a typical example of modestly complex caching code. The purpose of this function is to speed up operations that perform many lookups of a numerical user ID to obtain the user's name. The corresponding library function `getpwuid` will retrieve the name either by reading the local `/etc/passwd` file or (in a distributed environment) by interacting with the Network Information Service (NIS) maps. Both operations can be costly, involving an `open`, `read`, `close` system call sequence. A typical example, whereby a cache for storing the results of `getpwuid` will help, is the invocation of the *ls* `-l` command. The command will list the files in a directory in a "long" format, which includes, among other details, the file's owner and group. In most directories, files belong to the same owner; caching the mapping between the owner's *uid*, stored as

[75] netbsdsrc/lib/libc/gen/pwcache.c:60–94

```
/* power of 2 */
#define    NCACHE    64
/* bits to store with */
#define    MASK    (NCACHE - 1)

char *
user_from_uid(uid_t uid, int nouser)
{
    static struct ncache {
        uid_t    uid;
        char    name[MAXLOGNAME + 1];
    } c_uid[NCACHE];
    static char nbuf[15];       /* 32 bits == 10 digits */
    register struct passwd *pw;
    register struct ncache *cp;

    cp = c_uid + (uid & MASK);
    if (cp->uid != uid || !*cp->name) {
        [...]
        if ((pw = getpwuid(uid)) == NULL) {
            if (nouser)
                return (NULL);
            (void)snprintf(nbuf, sizeof(nbuf), "%u", uid);
            return (nbuf);
        }
        cp->uid = uid;
        (void)strncpy(cp->name, pw->pw_name, MAXLOGNAME);
        cp->name[MAXLOGNAME] = '\0';
    }
    return (cp->name);
}
```

Entries in the cache

Map from many uids to a cache entry

1 Uid to name cache

2 Cache location for uid

3 Is the uid stored in that location?

4 No, obtain the name through the expensive way

5 Store the uid/name pair in the cache

6 Return the result from the cache

Figure 4.10 The user ID to name cache code

part of the file metadata, and the owner's name (normally obtained with a call to `getpwuid`) can save hundreds of system calls in a directory containing 50 files.

The function `user_from_uid` defines a 64-element cache (Figure 4.10:1) for storing the last encountered lookups. You will often encounter specialized local caches defined as a `static` variable within the body of a C/C++ function. The number and range of numeric user identifiers on a system will in most cases be larger than the 64 elements provided by the cache. A simple map function (Figure 4.10:2) truncates the user identifier to the cache's element range, 0–63. This truncation will result in many `uid` values mapped to the same cache position; for example, `uid` values 0, 64, 128, and 192 will be mapped to the cache position 0. To ensure that the value in the cache position does indeed correspond to the `uid` passed to the function, a further check against the `uid` value stored in the cache is needed (Figure 4.10:3). If the stored `uid` does not match the `uid` searched, or if no value has yet been stored in the cache, the name for the corresponding `uid` will be retrieved using the `getpwuid` function (Figure 4.10:4). In that case, the cache is updated with the new result, automatically erasing the old entry (Figure 4.10:5). At the function's end, the value returned from the cache (Figure 4.10:6) will be either the newly updated entry or an entry already existing in the cache.

4.7.2 Replacement Strategies

In the example we examined in the previous subsection, each new entry will replace the previous entry cached in the same position. This strategy is certainly far from optimal and can even behave pathologically when two alternating values map to the same position. A better approach you will often encounter maintains a pool of cache entries, providing a more flexible mechanism for placing and locating an entry in the cache. Such a method will employ a strategy for removing from the cache the "least useful" elements. Consider the caching code of the HSQL database engine, used for optimizing the access time to a table's disk-based backing store by storing part of a table in the main memory. To prevent cache data loss in the case of key collisions, the code stores in each cache position a linked list of rows that map to that position. The following code excerpt illustrates the linked list traversal:[76]

```
Row getRow(int pos, Table t) throws SQLException {
    int k = pos & MASK;
    Row r = rData[k];

    while (r != null) {
        int p = r.iPos;
        if (p == pos) {
            return r;
        }
        r = r.rNext;
    }
```

More important, to keep the cache in a manageable size, a `cleanUp` method is periodically called to flush from the cache less useful data.[77] Figure 4.11 illustrates the algorithm's salient features. To minimize the algorithm's amortized cost, the cache cleanup is performed in batches. A high-water mark (`MAX_CACHE_SIZE`) contains the number of cache entries above which the cache will be cleared. This limit is set to 75% of the cache size:[78]

```
private final static int LENGTH = 1 << 14;
private final static int MAX_CACHE_SIZE = LENGTH * 3 / 4;
```

[76] hsqldb/src/org/hsqldb/Cache.java:258–281
[77] hsqldb/src/org/hsqldb/Cache.java:324–369
[78] hsqldb/src/org/hsqldb/Cache.java:51–52

```
void cleanUp() throws SQLException {
    if (iCacheSize < MAX_CACHE_SIZE) {
        return;
    }
    int count = 0, j = 0;
    while (j++ < LENGTH && [...]
           && (count * 16) < LENGTH) {
        Row r = getWorst();
        if (r.bChanged) {
            rWriter[count++] = r;
        } else {
            remove(r);
        }
    }
    if (count != 0) {
        saveSorted(count);
    }
    for (int i = 0; i < count; i++) {
        Row r = rWriter[i];
        remove(r);
        rWriter[i] = null;
    }
}
```

If the cache has enough free entries there is no need to clean it

Remove entries until cache is sufficiently empty

Select an underperforming entry

Dirty, mark it for deletion

Clean, delete it now

Save marked dirty entries

Remove marked dirty entries from cache

Figure 4.11 Caching database row entries

Once the high-water mark is reached, the cache cleanup loop will prune the cache, removing `LENGTH / 16` elements.[79] The way cache entries are removed is interesting for two reasons. First of all, a separate method, `getWorst()`, is used to obtain an underperforming entry. Every row contains a member named `iLastAccess`, and every time the row is accessed, that member gets the next value from a monotonically increasing counter:[80]

```
r.iLastAccess = iCurrentAccess++;
```

Every time the `getWorst` method is called, it goes through the next six rows and returns the one that was the *least recently used*:[81]

```
private Row getWorst() throws SQLException { [...]
    Row candidate = r;
    int worst = Row.iCurrentAccess;
    // algorithm: check the next rows and take the worst
    for (int i = 0; i < 6; i++) {
        int w = r.iLastAccess;
        if (w < worst) {
            candidate = r;
            worst = w;
```

[79] The original code contains an additional condition limiting the total number of elements in the cache, but the corresponding expression is coded incorrectly.

[80] hsqldb/src/org/hsqldb/Row.java:140

[81] hsqldb/src/org/hsqldb/Cache.java:433–462

```
            }
            r = r.rNext;
        } [...]
        return candidate;
}
```

Dropping from the cache the least recently used (LRU) elements is a common element-replacement policy;[82,83] other policies you may encounter include first-in first-out, random selection,[84] not recently used,[85,86] unused,[87] not frequently used, and the use of explicit expiration limits.[88] i

The second interesting aspect of `cleanUp` is that it contains special code for handling "dirty" cache entries. The HSQLDB row cache can be used for both reading and writing entries. As a result, when an entry is removed, it must first be committed to disk, if it is a result of a write operation or if it has been modified while it was in the cache. For this reason, `cleanUp` will first scan the cache, removing entries whose disk copy is still valid, while saving "dirty" entries in a separate structure (`rWriter`). In the end, all the modified entries are saved and then removed from the cache. Remember, when the cache allows both read and write operations, special code must be used to maintain the coherence between the cached data and the primary copy of the data. i

4.7.3 Precomputing Results

When a computation is expensive, two approaches you will encounter involve either caching the results of each calculation performed[89] or precomputing the results offline and incorporating a *lookup table* of them in the program's source code. A representative example of this approach is the method used for detecting errors in Point to Point Protocol (PPP) connections.[90] Communication over a PPP link is performed by sending data in separate frames. Each frame contains a 16-bit frame check sequence (FCS) field, whose value is calculated using a cyclic redundancy check (CRC) algorithm. The sender and the receiver can separately apply the same algorithm to the data to detect many types of bit corruption. The algorithm implementation used in the PPP

[82] XFree86-3.3/xc/programs/Xserver/hw/xfree86/accel/cache/xf86bcache.c:379–387
[83] netbsdsrc/sys/kern/vfs_cache.c
[84] XFree86-3.3/xc/lib/font/util/patcache.c:155–163
[85] XFree86-3.3/xc/programs/xfs/difs/cache.c:215–260
[86] netbsdsrc/sys/vm/vm_pageout.c:164–172
[87] ace/ace/Filecache.h:73–74
[88] apache/src/modules/proxy/proxy_cache.c
[89] XFree86-3.3/xc/util/memleak/getretmips.c:78–89
[90] ftp://ftp.internic.net/rfc/rfc1171.txt

specification calculates the modulo-2 division remainder of the frame bits, divided by the polynomial $x^{16} + x^{12} + x^5 + x^0$. (When calculating the modulo-2 division result, the corresponding subtractions do not propagate a carry bit; they are equivalent to an exclusive-or operation). The PPP CRC algorithm can detect all single- and double-bit errors, all errors involving an odd number of bits, and all burst errors of up to 16 bits. A naive implementation of the algorithm for calculating the CRC value of `s` would be the following code:[91]

```
unsigned short
crc_ccitt(const unsigned char *s)
{
    unsigned short p = 0x8408;  // Bits 0 5 12
    unsigned short v = 0xffff;  // Initial value

    for (s = string; *s; s++) {
        v ^= *s;
        for (int i = 0; i < 8; i++)
            v = v & 1 ? (v >> 1) ^ p : v >> 1;
    }
    return v;
```

Note that the code, when processing N bytes, will evaluate the innermost expression $8 \times N$ times. The overhead of this operation for code that processes packets arriving over a high-speed network interface can be considerable; therefore, all implementations you will find in the book's source code collection contain a precomputed table with the 256 different 16-bit CRC values corresponding to each possible 8-bit input value:[92,93]

```
/*
 * FCS lookup table as calculated by genfcstab.
 */
static u_int16_t fcstab[256] = {
    0x0000, 0x1189, 0x2312, 0x329b,
    0x4624, 0x57ad, 0x6536, 0x74bf,
    [...] [30 more lines] [...]
    0x7bc7, 0x6a4e, 0x58d5, 0x495c,
    0x3de3, 0x2c6a, 0x1ef1, 0x0f78
};
```

[91] This code *does not* appear in the book's source code collection.
[92] netbsdsrc/usr.sbin/pppd/pppd/demand.c:173–206
[93] netbsdsrc/sys/net/ppp_tty.c:448–481

The loop we listed earlier is then reduced to a loop over the frame's bytes with a single lookup operation:[94,95]

```
#define PPP_FCS(fcs, c) (((fcs) >> 8) ^ fcstab[((fcs) ^ (c)) & 0xff])

    while (len--)
        fcs = PPP_FCS(fcs, *cp++);
```

Lookup tables do not neccessarily contain complex data and are sometimes even computed by hand. The following code excerpt is using a lookup table to calculate the number of bytes required to pad a message to a 32-bit word boundary:[96]

```
/* lookup table for adding padding bytes to data that is
   read from or written to the X socket.  */
static int padlength[4] = {0, 3, 2, 1};
padBytes = padlength[count & 3];
```

We end this section by noting that the caching of data that does not exhibit locality of reference properties can degrade to the system's performance. The operations associated with caching—searching elements in the cache before retrieving them from secondary storage, storing the fetched elements in the cache, and organizing the cache's space—have a small but not insignificant cost. If this cache-maintenance cost is not offset by the savings of operations performed using the cache's data, the cache ends up being a drain on the system's performance and memory use. The following comment describes such a case:[97]

```
* Lookups seem to not exhibit any locality at all (files in
* the database are rarely looked up more than once...).
* So caching is just a waste of memory.
```

Exercise 4.14 The C `stdio` library uses an in-process memory buffer to store intermediate results of I/O operations before committing them to disk as a block. For example, reading a character is defined in terms of the following macro:[98]

```
#define __sgetc(p) (--(p)->_r < 0 ? __srget(p) : (int)(*(p)->_p++))
```

[94] netbsdsrc/sys/net/ppp_defs.h:90
[95] netbsdsrc/sys/net/ppp_tty.c:492–493
[96] XFree86-3.3/xc/programs/Xserver/os/io.c:767–769, 805
[97] netbsdsrc/bin/pax/tables.c:343–345
[98] netbsdsrc/include/stdio.h:329

Discuss whether in practice this scheme benefits the program by minimizing operating system interactions or by speeding the secondary storage accesses. Take into account the operating system's buffer cache and potential differences between the behavior of read and write operations.

Exercise 4.15 Locate code containing a cache algorithm implementing an LRU replacement strategy, and measure its performance on a realistic data set. Change the strategy to first-in first-out and least frequently used, and measure the corresponding performance.

Exercise 4.16 Time the lookup table implementation of the PPP CRC calculation against the bit-processing loop appearing on page 200. Implement the algorithm with lookup tables for 4, 12, 16, 20, and 24 bits, measuring the corresponding throughput. Discuss the results you obtained.

Advice to Take Home

- It is easier to improve bandwidth (by throwing more resources at the problem) than latency (*p. 152*).
- Don't optimize (*p. 154*).
- Measure before optimizing (*p. 154*).
- Humans are notoriously bad at guessing why a system is exhibiting a particular time-related behavior (*p. 156*).
- The only reliable and objective way to diagnose and fix time inefficiencies and problems is to use appropriate measurement tools (*p. 156*).
- The relationship among the real, kernel, and user time in a program's (or complete system's) execution is an important indicator of its workload type, the relevant diagnostic analysis tools, and the applicable problem-resolution options (*p. 157*).
- When analyzing a process's behavior, carefully choose its execution environment: Execute the process either in a realistic setting that reflects the actual intended use or on an unloaded system that will not introduce spurious noise in your measurements (*p. 157*).
- To locate the bottleneck of I/O-bound tasks, use system-monitoring tools (*p. 158*).
- To locate bottlenecks of kernel-bound tasks, use system call tracing tools (*p. 162*).
- To locate bottlenecks of CPU-bound tasks, use program-profiling tools (*p. 163*).
- Make sure that the data and operations you are profiling can be automatically and effortlessly repeated (*p. 164*).
- Make it a habit to instrument performance-critical code with permanent, reliable, and easily accessible time-measurement functionality (*p. 172*).

- A loop executed N times expresses an $O(N)$ algorithm (*p. 175*).
- Any operation on N elements that does not involve a loop, recursion, or calls to other operations depending on N expresses an $O(1)$ algorithm (*p. 176*).
- K nested loops over N elements express an $O(N^K)$ algorithm (*p. 177*).
- We can recognize algorithms that perform in $O(\log N)$ by noting that they divide their set size by two in each iteration (*p. 177*).
- The cost of a call to a function or a method can vary enormously, between 1 ns for a trivial function and many hours for a complex SQL query (*p. 179*).
- Processor-specific optimizations are by definition nonportable. Worse, the "optimizations" may be counterproductive on newer implementations of a given architecture. Before attempting to comprehend processor-specific code, it might be worthwhile to replace the code with its portable counterpart and measure the corresponding change in performance (*p. 181*).
- In modern systems, any visit outside the space of a given process involves an expensive *context switch* (*p. 182*).
- No matter what a system call is doing, each system call incurs the cost of two context switches: one from the process to the kernel and one from the kernel back to the process (*p. 185*).
- Whenever you find code in a loop polling to determine an operation's completion, look at the API for a corresponding blocking operation or callback that will achieve the same effect (*p. 189*).
- Try to avoid or minimize a program's interactions with slow peripherals (*p. 190*).
- When a system's performance over workload shape abruptly changes once the workload reaches a given point, the most probable cause is thrashing (*p. 193*).
- Try to minimize the number of interrupts that can occur and the processing required for each interrupt (*p. 194*).
- All caches capitalize on the *locality of reference* principle. Once an element is accessed, it is likely to be accessed again soon; elements near to that element are also likely to be accessed soon (*p. 195*).
- When the cache allows both read and write operations, special code must be used to maintain the coherence between the cached data and the primary copy of the data (*p. 199*).
- Caching of data that does not exhibit locality-of-reference properties can be detrimental to the system's performance (*p. 201*).

Further Reading

The book by Hennessy and Patterson [HP02] is a must-read item for anyone interested in quantitatively analyzing computer operations. A wonderful article by Patterson [Pat04] examines the historical relationship between latency and bandwidth we described in the chapter's introduction and provides a number of explanations for the bountiful bandwidth but lagging latency we typically face. Bentley's guide on writing efficient programs [Ben82] is more than 20 years old but still provides a well-structured introduction to the subject. Two other works by the same author also contain highly pertinent advice: [Ben88, Chapter 1] and [Ben00, Chapters 6–9]. Insightful discussions on code performance can also be found in the works by McConnell [McC93, Chapters 28–29] and [McC04, Chapters 25–26], and Kernighan and Pike [KP99, Chapter 7]. Apple's documentation on application and hardware performance is worth examining, even if you are not coding on the Mac OS X platform.[99] The tension between portable protocols and performance is lucidly presented in two different conference papers [CGB02, vE03].

The aphorisms on premature optimization appearing in Figure 4.1 come from a number of sources: [Jac75] (Jackson), [Knu87, p. 41] (Knuth), [Wei98, p. 130] (Weinberg), [McC93, p. 682] (McConnell), [BM93] (Bentley and McIlroy), [Wal91] (Wall), [Wul72] (Wulf), and [Blo01, p. 164] (Bloch); a recent article argues, however, that avoiding dealing with optimization issues in the undergraduate curriculum has created its own set of problems [Dug04].

The field of software performance engineering is defined mostly by the work of Connie Smith and Lloyd Williams. A couple of articles [Smi97, SW03] summarize some of the details you will find in their book [SW02b]. While on the subject, you should also read their excellent descriptions of software performance antipatterns: well-known and often-repeated design and implementation mistakes that can bring applications to their knees [SW00, SW02a].

A clear presentation of the origins of the Pareto Principle and its application in modern management can be found in Magretta's excellent primer on management [Mag02]. The validity of the Pareto Principle in the domain of the computer science applications has been known since at least 1971, when Knuth, as part of an empirical study of Fortran programs, found that 4% of a program contributed more than 50% of its execution time [Knu71]. The 80/20 relationship can be found in a paper by Boehm

[99] http://developer.apple.com/documentation/Performance/

[Boe87]; Bentley describes an instance in which a square root routine consumed 82% of a program's execution time [Ben88, p. 148].

The theory behind the implementation of *gprof* is described in the article [GKM83]. The *dtrace* profiling tool supports the configurable dynamic tracing of operating system performance. It is described in a Usenix paper [CSL04]. Detailed guidelines for improving the code's locality of reference and thereby its performance are contained in Apple's document [App05].

For the study of algorithm performance, Sedgewick's five-part work [Sed98, Sed02] provides an excellent reference. You will gain additional insights from Harel's book [HF04], the venerable work by Aho et al. [AHU74], Tarjan's classic [Tar83], and, of course, from Knuth's magnum opus [Knu97a, Knu97b, Knu98].

Modern advances in processor and memory architectures have made the accurate modeling of code performance almost impossible [Kus98]. The paper [Fri99] describes an approach whereby the code for a time-critical application (FFTs—Fast Fourier Transforms) is generated automatically to adapt the computation to the underlying hardware. Modern *just-in-time* bytecode compilers [Ayc03] will often specialize code for a specific architecture [CLCG00, HS04].

The cost of operations over the network is very important in the context of enterprise information systems. Books dealing with enterprise application patterns [Fow02, AMC03] devote considerable space presenting ways to minimize network calls and data transfers.

You will find the concepts of context switching, paging, and thrashing explained in any operating systems textbook, such as Tanenbaum's [Tan97]. Two perennial classics on the subject of virtual memory, thrashing, and the locality of reference are Denning's papers [Den70, Den80]; see also his recent historical recollection [Den05]. A comprehensive survey of cache-replacement strategies can be found in [PB03]; a more concise overview together with an intriguingly efficiently improvement of the venerable LRU replacement strategy is contained in the recent article by Megiddo and Modha [MM04]. More recently, Fonseca and his colleagues discussed the concept of caching as an integrated feature of the modern web environment [FAC05].