Writing Efficient Programs

Performance Issues in an Undergraduate CS Curriculum *

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721.
debray@cs.arizona.edu

ABSTRACT

Performance is an essential aspect of many software systems, and it is important for programmers to understand performance issues. However, most undergraduate curricula do not explicitly cover performance issues—performance monitoring and profiling tools, performance improvement techniques, and case studies—in their curricula. This paper describes how we address this topic as part of a third-year programming course. We focus on tools and techniques for monitoring and improving performance, as well as the interaction between clean program design and performance tuning.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; D.1 [**Programming Techniques**]: General; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Performance, Design

Keywords

Profiling, Performance tuning

1. MOTIVATION

Performance is an essential aspect of many software systems, and it is vitally important for programmers to understand performance issues: when it matters and when it doesn't; how to systematically identify performance bottlenecks and improve program performance; and how program design interacts with and affects performance tuning. Despite this, the issue of performance and performance tuning—including tools, techniques, and case studies—is not explicitly addressed, as a topic in its own right, in most curricula. Very often, the closest students come to encountering this issue is through a patchwork of topics such as machine organization and/or architecture (focusing on hardware aspects), algorithms (typically focusing only on asymptotic complexity, and ignoring "constant factors"), and possibly compilers (focusing on code generation issues). Our experience has been that such a piecemeal approach does not give students an adequate understanding of performance issues from a programming perspective.

A lack of understanding of performance tuning issues can adversely affect a student's programming even in situations where performance is not an issue. We have observed, for example, that many of our students become interested in computer science through their interest in computer games. Quite often, they have the (deeply held but erroneous) belief that programming is all about speed; in many cases, they also believe that "fast code" is synonymous with "tricky code." Then, when confronted with a programming problem they do not understand fully, they try to write fast code but often end up with code that is neither fast nor correct. In one fourth-year *Compiler Design* course, for example, we found that many students were spending a great deal of time squeezing every ounce of speed out of the hashing routines that manipulated the symbol table—possibly because they had studied hashing in their Data Structures class, this was one part of the assignment that was familiar to them, and "speed is important." This actually did very little to speed up their compilers, because most of the execution time was actually spent doing I/O; the problem was that the time so spent resulted in other parts of the project being neglected. At the other extreme, students whose primary encounter with "efficiency" has been in the context of asymptotic complexity analyses in algorithm design classes may spend inordinate amounts of time working on elaborate data structures, such as AVL trees, in situations where a sequential search through a simple linear list is perfectly adequate.

This paper discusses how we have addressed this topic, as part of a third-year *Systems Programming* course (roughly comparable to CS291s in the IEEE/ACM 2001 Computing Curriculum [3]).

2. BACKGROUND

One option for teaching performance issues would be at the fourth-year level [5], by which point students have encountered topics in computer architecture, operating systems, and compiler design at a reasonable level of detail, and therefore have some appreciation for the various issues affecting performance. We chose not to follow this option: we wanted students learning to program in C to be exposed, at the same time, to topics and tools relating to performance tuning, in order to inculcate good programming habits early. Our approach was to include these topics in a third-year course on *Systems Programming* that covers, in addition, Unix

* This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

and C programming topics. The material on performance tuning covers roughly 20% of the course (about 3 weeks).

In addition to introducing students to the topics and tools for performance tuning early, this has the advantage of allowing them to apply their knowledge in their final-year programming-intensive courses such as *Operating Systems* and *Compiler Design*. However, it imposes significant constraints on the background we can assume for the students. For example, while we can assume that students have had an introductory (at the first- or second-year level) course on machine organization, we cannot assume that they have had a course on data structures (taught in their third year) or algorithm design (final year). This, in turn, affects the sophistication of the data structures they can be expected to use in their assignments. In effect, we are limited to using a small set of straightforward data structures and algorithms covered in their first- and second-year programming classes. It also affects the extent to which we can discuss architectural issues, e.g., cache effects [4], and their impact on performance. We touch on this point in Section 4.1.

3. THE TUNING PROCESS

We begin the discussion on performance by emphasizing that when writing code, not all programs need to be sped up. The point is that what we want is not necessarily the *fastest* program, but one that is most cost-effective in terms of the tradeoff between programmer time, on the one hand, and execution speed (or other performance metric), on the other. In addition to setting the stage for performance tuning issues, this is also helpful in giving students a broader perspective of a range of programming solutions they have been taught—ranging from shell scripts, through *perl* and similar scripting languages, to optimized C code and even assembly code—and how one might go about deciding which of these is most appropriate under different circumstances.

We observe, also, that it is often difficult to determine, ahead of time, whether a particular piece of software will have "adequate" performance or whether it will need to be tuned further—the more so if it is being implemented by a team, where each team member has only a relatively limited view of the code. This raises three issues for discussion:

1. How can we identify *where* we should focus efforts to improve performance?
2. How might we go about figuring out how we should transform the program so as to improve its performance?
3. How can we design the program so as to simplify subsequent performance tuning?

The classroom discussions on performance tuning address each of these issues by examining specific programming examples in detail. In this paper, the first point is discussed in Section 3.1, the second in Sections 3.2 and 3.3, and the third in Section 5.

3.1 Identifying Performance Bottlenecks

We focus on two tools for identifying possible candidates for performance tuning: the function-level profiling tool *gprof* [2]; and basic block profiling within a function using the *gcc* compiler.

The profiles obtained from *gprof* typically provide high-level information about the program's behavior. This includes statistics about how many times each function in the program is called; the execution time spent in that function, both by itself and taking into account all of the functions it calls (transitively); and the average execution time per call. This is illustrated in Figure 1(a). We use these data to identify the functions that account for a significant percentage of the program's execution time. These functions are candidates for performance-improving transformations. The other function-level profile data—the number of times it is called and average execution time per call—can then be used to guide us to the next step in the tuning process: either the application of code transformations at the function level, or further profiling to obtain more fine-grained basic block level profile data.

While students don't actually encounter the notion of a "basic block" officially until a fourth-year Compiler Design class, the intuitive idea—that of a straight-line sequence of code with no branches into or out of its middle—can be explained easily enough. A basic block profile (generated, for example, by compiling a program using '`gcc -g -a`' or '`gcc -g -ax`' and running the resulting executable) gives execution counts for the basic blocks in a program, together with information about their location (file name, function, and line number), as shown in Figure 1(b). Basic block profiles are used primarily to identify the lines within the body of a function that are executed the largest number of times; they are a crude but generally effective way to identify expensive program fragments. In our discussions, we resort to basic block profiles once function-level optimizations have been carried out, and further performance improvements require profile information at a finer granularity than the function level.

3.2 Program Transformations

Obviously, the particular program transformations that might be appropriate for improving the performance of a program will depend on specifics of that program, and are therefore likely be different for different programs. We can, nevertheless, identify different classes of transformations that can be used to improve a program's performance, such as the following (this is not intended as an exhaustive list):

Cacheing : computing some data once, ahead of time, and then looking them up, instead of recomputing them repeatedly. The speed improvement comes from avoiding recomputation; the tradeoff is the additional space needed to store the precomputed values.

Buffering : Instead of invoking some action repeatedly on a large number of small objects, we may choose to "buffer" them into larger collections, and invoke the actions on the larger data that result. The speed improvement comes from reducing the number of times the action is invoked. The tradeoff is that more space is needed for the buffered objects.

Filtering : When examining large quantities of data to select those items that satisfy some specific property, if it may sometimes be possible to devise a "filter" that allows us to avoid examining (some) useless data. Examples of filtering include hashing, and also the use of binary search, which uses a simple test to localize the search and thereby avoid searching through irrelevant data.

We also discuss low-level program transformations that can be useful in improving performance, but in a lot less detail. Specific examples include function inlining (and the conceptually related transformation of replacing function calls with macros), as well as transformations aimed at reducing the amount of unnecessary computation, such as invariant code motion out of loops and common subexpression elimination. The points made in this context are that such transformations are typically applied by a compiler, but there may be situations where a compiler does not carry out the optimization. In such cases, the ability to understand and apply the transformations manually can be useful.

% time	cum. seconds	self seconds	no. of calls	self ms/call	total ms/call	name
39.9	1.39	1.39				internal_mcount [4]
25.0	2.26	0.87	7450087	0.00	0.00	strcmp [5]
21.8	3.02	0.76	45404	0.02	0.04	wordLookup [3]
1.4	3.07	0.05	88364	0.00	0.00	_doprnt [15]
1.1	3.11	0.04	499876	0.00	0.00	_getc_unlocked [17]
0.9	3.21	0.03	176728	0.00	0.00	_smalloc [18]
0.9	3.24	0.03	45404	0.00	0.00	sort_string [21]
0.9	3.27	0.03	45404	0.00	0.00	string [16]
0.6	3.34	0.02	133768	0.00	0.00	strlen [23]
0.6	3.36	0.02	88364	0.00	0.00	hashval [24]
0.3	3.43	0.01	176730	0.00	0.00	malloc [8]

(a) (Partial) function-level profile obtained using *gprof*

Block No.	no. of times executed	Address	Function name	Line no.	File name
1	88364	0x10c70	hashval	37	hash.c
2	712740	0x10c98	hashval	40	hash.c
3	88364	0x10cc8	hashval	44	hash.c
4	1	0x10d04	hashInit	51	hash.c
5	0	0x10d28	hashInit	51	hash.c
6	0	0x10d48	hashInit	52	hash.c
7	45404	0x10d80	wordLookup	61	anagram.c
8	0	0x10d9c	wordLookup	65	anagram.c
9	45404	0x10dc4	wordLookup	67	anagram.c
10	45276	0x10df8	wordLookup	70	anagram.c
11	7447073	0x10e0c	wordLookup	70	anagram.c

(b) (Partial) basic block profile obtained using `gcc -a`

Figure 1: Examples of Function-level and Basic block-level execution profiles

3.3 Applying the Transformations

The execution time profiles for different functions is illuminating, not just for pointing out potential optimization candidates, but also for suggesting how we might go about optimizing them. For example, if we find a "heavyweight" function—one called perhaps only a few times, but where each call incurs a significant execution cost—then the natural course of action is to use basic block profiles to identify which portions of its body are the most frequently executed, then focus on these portions of the code. On the other hand, we may have a situation where a function has been coded quite efficiently, so that the time taken for a single execution through its body is very small, but where it is called so many times that overall, it incurs a large cost. In this case, we want to reduce the number of times the function is called. While the details of how this might be done depends on specifics of the underlying application, we can draw some general guidelines. For example, if the commonly-called function is being used to search for objects satisfying some property, we might consider some form of hashing (or, more generally, filtering); if it is carrying out actions that can be "bundled up together," we can consider buffering.

The utility of basic block profiles comes from the fact that—unlike the function-level profiles obtained from tools like *gprof*—they offer a detailed look at the execution behavior within a function. This is especially useful if, as a result of other program transformations and/or due to specific coding decisions, a programmer is confronted with a "heavyweight" function whose performance needs to be improved. For example, suppose that a basic block profile indicates that a function contains a doubly nested loop where the inner loop has a high execution count, and an examination of the inner loop indicates that it is traversing a data structure searching for objects satisfying some property. In this case, a plausible approach to reducing the performance overhead for this piece of code might be to try and use some sort of filter, or a more intelligent data structure, to reduce the number of times the inner loop is executed. Basic block profiles can also be useful for determining a good ordering for tests. For example, given code of the form

if ($test_1$ && $test_2$) ...

If $test_1$ and $test_2$ are independent tests, and basic block profiles indicate that $test_1$ is false a smaller fraction of the time than $test_2$—indicating that $test_1$ is not as effective filter as $test_2$—then a plausible transformation would be to reorder them so that $test_2$ is done first.

The larger point made in these discussions on program transformation strategies is that the problem of performance tuning can be approached systematically, using general principles and guidelines.

4. ASSIGNMENTS

Hands-on experimentation is crucial for getting a good understanding of profiling and performance tuning, and we accordingly attach a great deal of importance to programming assignments on this topic. This section discusses various aspects of the design and administration of these assignments.

4.1 Desiderata

The problem of designing appropriate performance tuning assignments that are consistent with the student background we can assume raises some interesting challenges. We attempt to construct programs with the following characteristics:

- The program should be of reasonable size: small enough that a student can understand the code without undue effort; yet large enough that the performance bottlenecks are not obvious simply from inspecting the code.
- The program should have a number of different performance bottlenecks, in order to avoid an "all-or-nothing" situation with a bipolar distribution of assignment scores.
- The different sources of overhead should preferably be amenable to different types of program transformations, in order to force students think about performance issues rather than reflexively apply a stock solution (e.g., hashing).
- The performance bottlenecks should require different levels of sophistication to identify and address, so as to provide "something for everyone," ranging from average and relatively inexperienced students to the top students and experienced programmers.
- The solutions should not require knowledge of advanced data structures or algorithms, e.g., AVL-trees or sophisticated graph algorithms, since such topics are not part of the student background that can be assumed given the course prerequisites.

The following is an example problem satisfying some of these criteria. The problem, taken from Bentley's *Programming Pearls* [1], is to count the number of '`1`' bits in a file. The unoptimized program given to the students is a straightforward piece of code that repeatedly reads a single byte from the input file, counts the number of '`1`' bits in it using a simple repeated shift-and-mask scheme to examine each bit in isolation, and adds the count for each byte to an accumulator. The code can be improved in a number of ways:

Simple : The shift-and-mask code to count the number of '`1`' bits iterates over 32 bits, even though the input file is read in a byte, i.e., 8 bits, at a time. The iteration count can be reduced to 8.

Moderate : The system call used to read a byte, `fgetc()`, is not the most efficient one for the purpose. A more efficient system call that is equivalent for our purposes in this case, namely, `getc()`, can be used instead.

Moderate : The input can be read a page at a time, instead of a single byte at a time, and buffered.

Sophisticated : The number of '`1`' bits in each byte can be precomputed into an array of size 256, such that the entry in element *k* of this array is the number of '`1`' bits in the byte whose (unsigned) value is *k*. Instead of a shift-and-mask loop to count the number of '`1`' bits in each byte, we can then simply index into this table with any particular byte to obtain the count of the number of '`1`' bits in it.

Referring back to the classes of transformations mentioned in Section 3.2, these can be seen to include Cacheing and Buffering. The interested reader may find additional performance tuning examples and assignments in
`http://www.cs.arizona.edu/~debray/PerfTuning/`.

4.2 Administration

For each performance tuning assignment, students are given the following:

1. A source program whose performance has to be improved.
2. Sample inputs, including one that will be used as the "official" input for determining the overall speed improvement achieved.
3. An executable for a "fast" version of the program, which indicates the extent of improvement realizable, and provides students a concrete performance target.

Students take the program given, modify it as appropriate, and eventually turn in the modified source program. They are also required to turn in a `README` file that describes in detail the particular performance bottlenecks they observed (consisting of raw profile data together with the conclusions they drew from these data) and the specific changes they made to the program code to ameliorate the bottlenecks. The modifications to the program are required to address the specific performance bottlenecks observed (ruling out submissions of the form "*the program was observed to be quite slow, and I found this much faster program on the Internet, which I decided to turn in*").

The rules that have to be followed when modifying the program are as follows:

1. The modified program must be semantically equivalent to the original, i.e., produce the same output on all inputs. This includes both the sample inputs provided as part of the assignment handout, and others we may choose.
2. The modified program must obtain its performance improvements using "general-purpose" techniques: approaches of the form

   > "*if the input file is named* '`infile.big`' *print out the value* 1234567"

   are not allowed.
3. There is a "free allowance" of 4 Kbytes of additional space (compared to the original program) that programs can use. Students must get special permission if their program uses more space than this.

The last rule serves to preclude programs that "bludgeon the problem to death" using space-time tradeoffs that might work for the small programs used for the assignments but would be wildly impractical for larger programs that students would encounter in reality. It also makes students think carefully about how much space their solutions use (all else being same, one that uses less space is better).

4.3 Grading

Programs are graded based on how much performance improvement they are able to achieve. This rewards students who work harder and do a better job of improving the program, but raises the question of exactly how "the amount of performance improvement" realized by a program is to be measured. Since the execution time of a program can be affected by the overall system load, a fair comparison between the runtimes of the original program given out, and the optimized version turned in by a student, requires that they be run under as similar conditions as possible. We therefore run each student's program alternately with the base program in an effort to mitigate the effects of system load fluctations, and compute some sort of average for the resulting run times for each executable.[1] The ratio of the original program's average run time to the optimized program's average run time constitutes the extent of performance improvement achieved; this number is then used to determine a score for the student.

To offer motivated students an additional challenge, students whose programs are found to be faster than the "fast executable" supplied as part of the assignment are given a small bonus score (typically around 15%–20%).

5. PROGRAM DESIGN ISSUES

Students sometimes have the perception that software design principles such as abstraction and modularization incur a performance penalty, and that fast software must, *ipso facto*, abandon such niceties. An important aspect of our discussion of performance considerations is the focus on the relationship between program design and performance tuning. In particular, we emphasize that not only does clean software design not preclude good performance, but that it can actually be quite helpful in attaining it.

A fundamentally important principle in the course is that correctness cannot be compromised for speed: an incorrect result, no matter how quickly it is computed, is not very useful. We attempt to discuss the meaning of the word "correctness" in context. E.g., reordering arithmetic operations on floating point numbers can sometimes result in small changes to the value computed: whether this compromises correctness or not depends on the error bounds on the result promised by the computation and/or expected by the user.

As repeatedly emphasized in class (and demonstrated via case studies of programs), the actual performance bottlenecks in a program are often difficult to predict ahead of time, even for relatively moderate sized programs. This is, in fact, the rationale for using profiling tools such as *gprof*. Meanwhile, programming projects—whether in the classroom or in the workplace—usually have deadlines by which they must be completed. Suppose that our priorities, in descending order, are as follows:

1. the code should work, i.e., be as free of bugs as possible; and

2. the code should be "fast enough."

Given these goals, how should a programmer proceed? Since we don't know, ahead of time, where the bottlenecks are and how much time it'll take to tune the code, a sensible approach would be to leave as much time as possible towards the end for performance tuning: this means that we want to get the code written, tested, and debugged quickly. Furthermore, during the performance tuning process, we want each profile-run-transform iteration over the program to be as quick as possible, so that we can get rid of as many performance bottlenecks as we can. Both these considerations suggest that the program should be designed and structured in such a way that changes to data structures and algorithms induce only local changes to the code—i.e., they do not require making (potentially error-prone) edits across large portions of the program. The inevitable conclusion is that the programmer should design the code in a clean and modular way, using appropriate abstractions, so that changes to the algorithms and data structures can be made quickly via edits of limited and localized scope.

Students often feel that abstraction necessarily involves indirection, which in turn involves a performance penalty. For example, code abstraction is viewed as necessarily involving functions or procedures, which incur call/return overheads at runtime. An alternative, discussed in class, is to use macros instead of functions where appropriate, i.e., in situations where (part of) the performance bottleneck is due to a function call in a frequently executed portion of the program. Macros are also very valuable as accessors for components of complex data structures. The reason for this is that when we start writing a program we may not know what our data structures will look like eventually—they may change during development, e.g., due to performance tuning. Ideally, any such data structure changes—which require changes to the code for accessing and updating their components as well—should result in only localized edits to a program. This can be done by systematically using macros to access data structure components. Changes to data structures are then simply accompanied by the corresponding changes to the macro definitions; the remainder of the program does not have to change.

The bottom line is that, not only are cleanly designed and structured programs easier to debug and maintain, but—once performance bottlenecks have been identified—such programs are usually also much easier to optimize than tricky spaghetti code. This gives even the "macho programmers" among the students a very real incentive to pay attention to clean design.

6. CONCLUSIONS

While performance considerations are an important aspect of many software systems, most undergraduate curricula do not explicitly teach students about tools and techniques for improving sortware performance. Instead, the topic seems to be addressed almost incidentally, via a patchwork of disparate topics, ranging from architectural issues in Machine Organization courses, to asymptotic complexity in Algorithm Design courses.

Our experience has been that such a piecemeal approach does not give students a good understanding of performance issues from a programming perspective. As a result, students often have deeply held but erroneous beliefs about programming for performance. The problem can be addressed, to some extent, by making performance issues an integral part of the programming curriculum. This can help teach students how design programs with performance issues in mind, and identify and rectify performance bottlenecks, in a systematic way.

Acknowledgements

The author is grateful to Suzanne Westbrook for many very helpful comments on the manuscript.

7. REFERENCES

[1] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999.

[2] S. L. Graham, P. B. Kessler, and M. K. McKusick. `gprof`: A call graph execution profiler. In *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[3] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*. IEEE Computer Society and ACM, December 2001.

[4] A. L. Lebeck. Cache conscious programming in undergraduate computer science. In *Proc. 30th SIGCSE technical symposium on Computer science education*, pages 247–251, March 1999.

[5] C. M. Shub. Performance experiments for the performance course. In *Proc. 20th SIGCSE technical symposium on Computer science education*, pages 222–225, February 1989.

[1] In our class, each of the two executables is run five times and timed; the best and worst of the five run times are discarded; and the remaining three run times averaged to obtain a single number.