Optimizing Your Programs — Chapter 25

Since program optimization is generally one of the last steps in software development, it is only fitting to discuss program optimization in the last chapter of this text. Scanning through other texts that cover this subject, you will find a wide variety of opinions on this subject. Some texts and articles ignore instruction sets altogether and concentrate on finding a better algorithm. Other documents assume you've already found the best algorithm and discuss ways to select the "best" sequence of instructions to accomplish the job. Others consider the CPU architecture and describe how to "count cycles" and pair instructions (especially on superscalar processors or processes with pipelines) to produce faster running code. Others, still, consider the system architecture, not just the CPU architecture, when attempting to decide how to optimize your program. Some authors spend a lot of time explaining that their method is the "one true way" to faster programs. Others still get off on a software engineering tangent and start talking about how time spent optmizing a program isn't worthwhile for a variety of reasons. Well, this chapter is not going to present the "one true way," nor is it going to spend a lot of time bickering about certain optimization techniques. It will simply present you with some examples, options, and suggestions. Since you're on your own after this chapter, it's time for you to start making some of your own decisions. Hopefully, this chapter can provide suitable information so you can make correct decisions.

25.0 Chapter Overview

25.1 When to Optimize, When Not to Optimize

The optimization process is not cheap. If you develop a program and then determine that it is too slow, you may have to redesign and rewrite major portions of that program to get acceptable performance. Based on this point alone, the world often divides itself into two camps – those who optimize early and those who optimize late. Both groups have good arguements; both groups have some bad arguements. Let's take a look at both sides of this arguement.

The "optimize late" (OL) crowd uses the 90/10 arguement: 90% of a program's execution time is spent in 10% of the code[1]. If you try to optimize every piece of code you write (that is, optimize the code before you know that it needs to be optimized), 90% of your effort will go to waste. On the other hand, if you write the code in a normal fashion first and then go in an optimize, you can improve your program's performance with less work. After all, if you *completely removed* the 90% portion of your program, your code would only run about 10% faster. On the other hand, if you completely remove that 10% portion, your program will run about 10 times faster. The math is obviously in favor of attacking the 10%. The OL crowd claims that you should write your code with only the normal attention to performance (i.e., given a choice between an $O(n^2)$ and an O(n lg n) algorithm, you should choose the latter). Once the program is working correctly you can go back and concentrate your efforts on that 10% of the code that takes all the time.

The OL arguements are persuasive. Optimization is a laborious and difficult process. More often that not there is no clear-cut way to speed up a section of code. The only way to determine which of several different options is better is to actually code them all up and compare them. Attempting to do this on the entire program is impractical. However, if you can find that 10% of the code and optimize that, you've reduced your workload by 90%, very inviting indeed. Another good arguement the OL group uses is that few programmers are capable of anticipating where the time will be spent in a program. Therefore, the only real way to determine where a program spends its time is to *instrument it* and measure which functions consume the most time. Obviously, you must have a working program before you can do this. Once

1. Some people prefer to call this the 80/20 rule: 80% of the time is spent in 20% of the code, to be safer in their esitmates. The exact numbers don't matter. What is important is that most of a program's execution time is spent in a small amount of the code.

again, they argue that any time spent optimizing the code beforehand is bound to be wasted since you will probably wind up optimizing that 90% that doesn't need it.

There are, however, some very good counter arguments to the above. First, when most OL types start talking about the 90/10 rule, there is this implicit suggestion that this 10% of the code appears as one big chunk in the middle of the program. A good programmer, like a good surgeon, can locate this malignant mass, cut it out, and replace with with something much faster, thus boosting the speed of your program with only a little effort. Unfortunately, this is not often the case in the real world. In real programs, that 10% of the code that takes up 90% of the execution time is often spread all over your program. You'll get 1% here, 0.5% over there, a "gigantic" 2.5% in one function, and so on. Worse still, optimizing 1% of the code within one function often requires that you modify some of the other code as well. For example, rewriting a function (the 1%) to speed it up quite a bit may require changing the way you pass parameters to that function. This may require rewriting several sections of code outside that slow 10%. So often you wind up rewriting much more than 10% of the code in order to speed up that 10% that takes 90% of the time.

Another problem with the 90/10 rule is that it works on percentages, and the percentages change during optimization. For example, suppose you located a single function that was consuming 90% of the execution time. Let's suppose you're Mr. Super Programmer and you managed to speed this routine up by a factor of two. Your program will now take about 55% of the time to run before it was optimized[2]. If you triple the speed of this routine, your program takes a total of 40% of the original time to execution. If you are really great and you manage to get that function running nine times faster, your program now runs in 20% of the original time, i.e., five times faster.

Suppose you could get that function running nine times faster. Notice that the 90/10 rule no longer applies to your program. 50% of the execution time is spent in 10% of your code, 50% is spent in the other 90% of your code. And if you've managed to speed up that one function by 900%, it is very unlikely you're going to squeeze much more out of it (unless it was *really* bad to begin with). Is it worthwhile messing around with that other 90% of your code? You bet it is. After all, you can improve the performance of your program by 25% if you double the speed of that other code. Note, however, that you only get a 25% performance boost *after* you optimized the 10% as best you could. Had you optimized the 90% of your program first, you would only have gotten a 5% performance improvement; hardly something you'd write home about. Nonetheless, you can see some situations where the 90/10 rule obviously doesn't apply and you can see some cases where optimizing that 90% can produce a good boost in performance. The OL group will smile and say "see, that's the benefit of optimizing late, you can optimize in stages and get just the right amount of optimization you need."

The optimize early (OE) group uses the flaw in percentage arithmetic to point out that you will probably wind up optimizing a large portion of your program anyway. So why not work all this into your design in the first place? A big problem with the OL strategy is that you often wind up designing and writing the program twice – once just to get it functional, the second time to make it practical. After all, if you're going to have to rewrite that 90% anyway, why not write it fast in the first place? The OE people also point out that although programmers are notoriously bad at determining where a program spends most of its time, there are some obvious places where they know there will be performance problems. Why wait to discover the obvious? Why not handle such problem areas early on so there is less time spent measuring and optimizing that code?

Like so many other arguements in Software Engineering, the two camps become quite polarized and swear by a totally pure approach in either direction (either all OE or all OL). Like so many other arguements in Computer Science, the truth actually lies somewhere between these two extremes. Any project where the programmer set out to design the perfect program without worry about performance until the end is doomed. Most programmers in this scenario write *terribly slow* code. Why? Because it's easier to do so and they can always "solve the performance problem during the optimization phase." As a result, the 90% portion of the program is often so slow that even if the time of the other 10% were reduced to zero,

2. Figure the 90% of the code originally took one unit of time to execute and the 10% of the code originally took nine units of time to exeute. If we cut the execution time of the of the 10% in half, we now have 1 unit plus 4.5 units = 5.5 units out of 10 or 55%.

the program would still be way too slow. On the other hand, the OE crowd gets so caught up in writing the best possible code that they miss deadlines and the product may never ship.

There is one undeniable fact that favors the OL arguement – optimized code is difficult to understand and maintain. Furthermore, it often contains bugs that are not present in the unoptimized code. Since incorrect code is unacceptable, even if it does run faster, one very good arguement against optimizing early is the fact that testing, debugging, and quality assurance represent a large portion of the program development cycle. Optimizing early may create so many additional program errors that you lose any time saved by not having to optimize the program later in the development cycle.

The correct time to optimize a program is, well, at the correct time. Unfortunately, the “correct time” varies with the program. However, the first step is to develop program performance requirements along with the other program specifications. The system analyst should develop target response times for all user interactions and computations. During development and testing, programmers have a target to shoot for, so they can’t get lazy and wait for the optimization phase before writing code that performs reasonably well. On the other hand, they also have a target to shoot for and once the code is running fast enough, they don’t have to waste time, or make their code less maintainable; they can go on and work on the rest of the program. Of course, the system analyst could misjudge performance requirements, but this won’t happen often with a good system design.

Another consideration is when to perform *what*. There are several types of optimizations you can perform. For example, you can rearrange instructions to avoid hazards to double the speed of a piece of code. Or you could choose a different algorithm that could run twice as fast. One big problem with optimization is that it is not a single process and many types of optimizations are best done later rather than earlier, or vice versa. For example, choosing a good algorithm is something you should do early on. If you decide to use a better algorithm *after* implementing a poor one, most of the work on the code implementing the old algorithm is lost. Likewise, instruction scheduling is one of the last optimizations you should do. Any changes to the code after rearranging instructions for performance may force you to spend time rearranging them again later. Clearly, the lower level the optimization (i.e., relying upon CPU or system parameters), the later the optimization should be. Conversely, the higher level the optimization (e.g., choice of algorithm), the sooner should be the optimization. In all cases, though, you should have target performance values in mind while developing code.

25.2 How Do You Find the Slow Code in Your Programs?

Although there are problems with the 90/10 rule, the concept behind it is basically solid – programs tend to spend a large amount of their time executing only a small percentage of the code. Clearly, you should optimize the slowest portion of your code first. The only problem is how does one find the slowest code in a program?

There are four common techniques programmers use to find the “hot spots” (the places where programs spend most of their time). The first is by trial and error. The second is to optimize everything. The third is to analyze the program. The fourth is to use a *profiler* or other software monitoring tool to measure the performance of various parts of a program. After locating a hot spot, the programmer can attempt to analyze that section of the program.

The trial and error technique is, unfortunately, the most common strategy. A programmer will speed up various parts of the program by making educated guesses about where it is spending most of its time. If the programmer guesses right, the program will run much faster after optimization. Experienced programmers often use this technique successfully to quickly locate and optimize a program. When the programmer guesses correctly, this technique minimizes the amount of time spent looking for hot spots in a program. Unfortunately, most programmers make fairly poor guesses and wind up optimizing the wrong sections of code. Such effort often goes to waste since optimizing the *wrong* 10% will not improve performance significantly. One of the prime reasons this technique fails so often is that it is often the first choice of inexperienced programmers who cannot easily recognize slow code. Unfotunately, they are probably unaware of other techniques, so rather than try a structured approach, they start making (often) uneducated guesses.

Another way to locate and optimize the slow portion of a program is to optimize everything. Obviously, this technique does not work well for large programs, but for short sections of code it works reasonably well. Later, this text will provide a short example of an optimization problem and will use this technique to optimize the program. Of course, for large programs or routines this may not be a cost effective approach. However, where appropriate it can save you time while optimizing your program (or at least a portion of your program) since you will not need to carefully analyze and measure the performance of your code. By optimizing everything, you are sure to optimize the slow code.

The analysis method is the most difficult of the four. With this method, you study your code and determine where it will spend most of its time based on the data you expect it to process. In theory, this is the best technique. In practice, human beings generally demonstrate a distaste for such analysis work. As such, the analysis is often incorrect or takes too long to complete. Furthermore, few programmers have much experience studying their code to determine where it is spending most of its time, so they are often quite poor at locating hot spots by studying their listings when the need arises.

Despite the problems with program analysis, this is the first technique you should always use when attempting to optimize a program. Almost all programs spend most of their time executing the body of a loop or recursive function calls. Therefore, you should try to locate all recursive function calls and loop bodies (especially nested loops) in your program. Chances are very good that a program will be spending most of its time in one of these two areas of your program. Such spots are the first to consider when optimizing your programs.

Although the analytical method provides a good way to locate the slow code in a program, analyzing program is a slow, tedious, and boring process. It is very easy to completely miss the most time consuming portion of a program, especially in the presence of indirectly recursive function calls. Even locating time consuming nested loops is often difficult. For example, you might not realize, when looking at a loop within a procedure, that it is a nested loop by virtue of the fact that the calling code executes a loop when calling the procedure. In theory, the analytical method should always work. In practice, it is only marginally successful given that fallible humans are doing the analysis. Nevertheless, some hot spots are easy to find through program analysis, so your first step when optimizing a program should be analysis.

Since programmers are notoriously bad at analyzing programs to find their hot spots, it would make since to try an automate this process. This is precisely what a *profiler* can do for you. A profiler is a small program that measures how long your code spends in any one portion of the program. A profiler typically works by interrupting your code periodically and noting the return address. The profiler builds a histogram of interrupt return addresses (generally rounded to some user specified value). By studying this histogram, you can determine where the program spends most of its time. This tells you which sections of the code you need to optimize. Of course, to use this technique, you will need a profiler program. Borland, Microsoft, and several other vendors provide profilers and other optimization tools.

25.3 Is Optimization Necessary?

Except for fun and education, you should never approach a project with the attitude that you are going to get maximal performance out of your code. Years ago, this was an important attitude because that's what it took to get anything decent running on the slow machines of that era. Reducing the run time of a program from ten minutes to ten seconds made many programs commercially viable. On the other hand, speeding up a program that takes 0.1 seconds to the point where it runs in a millisecond is often pointless. You will waste a lot of effort improving the performance, yet few people will notice the difference.

This is not to say that speeding up programs from 0.1 seconds to 0.001 seconds is never worthwhile. If you are writing a data capture program that requires you to take a reading every millisecond, and it can only handle ten readings per second as currently written, you've got your work cut out for you. Furthermore, even if your program runs fast enough already, there are reasons why you would want to make it run twice as fast. For example, suppose someone can use your program in a multitasking environment. If you modify your program to run twice as fast, the user will be able to run another program along side yours and not notice the performance degradation.

However, the thing to always keep in mind is that you need to write software that is *fast enough*. Once a program produces results instantaneously (or so close to instantaneous that the user can't tell), there is little need to make it run any faster. Since optimization is an expensive and error prone process, you want to avoid it as much as possible. Writing programs that run faster than fast enough is a waste of time. However, as is obvious from the set of bloated application programs you'll find today, this really isn't a problem, most programming produce code that is way too slow, not way too fast.

A common reason stated for not producing optimal code is advancing hardware design. Many programmers and managers feel that the high-end machines they develop software on today will be the mid-range machines two years from now when they finally release their software. So if they design their software to run on today's very high-end machines, it will perform okay on midrange machines when they release their software.

There are two problems with the approach above. First, the operating system running on those machines two years from now will gobble a large part of the machine's resources (including CPU cycles). It is interesting to note that today's machines are hundreds of times faster than the original 8088 based PCs, yet many applications actually run *slower* than those that ran on the original PC. True, today's software provides many more features beyond what the original PC provided, but that's the whole point of this arguement – customers will demand features like multiple windows, GUI, pull-down menus, etc., that all consume CPU cycles. You cannot assume that newer machines will provide extra clock cycles so your slow code will run faster. The OS or user interface to your program will wind up eating those extra available clock cycles.

So the first step is to realistically determine the performance requirements of your software. Then write your software to meet that performance goal. If you fail to meet the performance requirements, then it is time to optimize your program. However, you shouldn't waste additional time optimizing your code once your program meets or exceed the performance specifications.

25.4 The Three Types of Optimization

There are three forms of optimization you can use when improving the performance of a program. They are choosing a better algorithm (high level optimization), implementing the algorithm better (a medium level optmization), and "counting cycles" (a low level optimization). Each technique has its place and, generally, you apply them at different points in the development process.

Choosing a better algorithm is the most highly touted optimization technique. Alas it is the technique used least often. It is easy for someone to announce that you should always find a better algorithm if you need more speed; but finding that algorithm is a little more difficult. First, let us define an algorithm change as using a fundamentally different technique to solve the problem. For example, switching from a "bubble sort" algorithm to a "quick sort" algorithm is a good example of an algorithm change. Generally, though certainly not always, changing algorithms means you use a program with a better Big-Oh function[3] For example, when switching from the bubble sort to the quick sort, you are swapping an algorithm with an $O(n^2)$ running time for one with an O(n lg n) expected running time.

You must remember the restrictions on Big-Oh functions when comparing algorithms. The value for *n* must be sufficiently large to mask the effect of hidden constant. Furthermore, Big-Oh analysis is usually *worst-case* and may not apply to your program. For example, if you wish to sort an array that is "nearly" sorted to begin with, the bubble sort algorithm is usually much faster than the quicksort algorithm, regard-

3. Big-Oh function are approximations of the running time of a program.

less of the value for *n*. For data that is almost sorted, the bubble sort runs in almost O(n) time whereas the quicksort algorithm runs in $O(n^2)$ time[4].

The second thing to keep in mind is the constant itself. If two algorithms have the same Big-Oh function, you cannot determine any difference between the two based on the Big-Oh analysis. This does not mean that they will take the same amount of time to run. Don't forget, in Big-Oh analysis we throw out all the low order terms and multiplicative constants. The asymptotic notation is of little help in this case.

To get truly phenomenal performance improvements requires an algorithmic change to your program. However, discovering an O(n lg n) algorithm to replace your $O(n^2)$ algorithm is often difficult if a published solution does not already exist. Presumably, a well-designed program is not going to contain many obvious algorithms you can dramatically improve (if they did, they wouldn't be well-designed, now, would they?). Therefore, attempting to find a better algorithm may not prove successful. Nevertheless, it is always the first step you should take because the following steps operate on the algorithm you have. If you perform the other steps on a bad algorithm and then discover a better algorithm later, you will have to repeat these time-consumings steps all over again on the new algorithm.

There are two steps to discovering a new algorithms: research and development. The first step is to see if you can find a better solution in the existing literature. Failing that, the second step is to see if you can develop a better algorithm on your own. The key thing is to budget an appropriate amount of time to these two activities. Research is an open-ended process. You can always read one more book or article. So you've got to decide how much time you're going to spend looking for an existing solution. This might be a few hours, days, weeks, or months. Whatever you feel is cost-effective. You then head to the library (or your bookshelf) and begin looking for a better solution. Once your time expires, it is time to abandon the research approach unless you are sure you are on the right track in the material you are studying. If so, budget a little more time and see how it goes. At some point, though, you've got to decide that you probably won't be able to find a better solution and it is time to try to develop a new one on your own.

While searching for a better solution, you should study the papers, texts, articles, etc., exactly as though you were studying for an important test. While it's true that much of what you study will not apply to the problem at hand, you are learning things that will be useful in future projects. Furthermore, while someone may not provide the solution you need, they may have done some work that is headed in the same direction that you are and could provide some good ideas, if not the basis, for your own solution. However, you must always remember that the job of an engineer is to provide a cost-effective solution to a problem. If you waste too much time searching for a solution that may not appear anywhere in the literature, you will cause a cost overrun on your project. So know when it's time to "hang it up" and get on with the rest of the project.

Developing a new algorithm on your own is also open-ended. You could literally spend the rest of your life trying to find an efficient solution to an intractible problem. So once again, you need to budget some time for this process accordingly. Spend the time wisely trying to develop a better solution to your problem, but once the time is exhausted, it's time to try a different approach rather than waste any more time chasing a "holy grail."

Be sure to use all resources at your disposal when trying to find a better algorithm. A local university's library can be a big help. Also, you should network yourself. Attend local computer club meetings, discuss your problems with other engineers, or talk to interested friends, maybe they're read about a solution that you've missed. If you have access to the Internet, BIX, Compuserve, or other technically oriented on-line services or computerized bulletin board systems, by all means post a message asking for help. With literally millions of users out there, if a better solution exists for your problem, someone has probabaly solved it for you already. A few posts may turn up a solution you were unable to find or develop yourself.

At some point or another, you may have to admit failure. Actually, you may have to admit success – you've already found as good an algorithm as you can. If this is still too slow for your requirements, it may be time to try some other technique to improve the speed of your program. The next step is to see if you

4. Yes, $O(n^2)$. The O(n lg n) rating commonly given the quicksort algorithm is actually the *expected* (average case) analysis, not the worst case analysis.

can provide a better implementation for the algorithm you are using. This optimization step, although independent of language, is where most assembly language programmers produce dramatic performance improvements in their code. A better implementation generally involves steps like unrolling loops, using table lookups rather than computations, eliminating computations from a loop whose value does not change within a loop, taking advantage of machine idioms (such as using a shift or shift and add rather than a multiplication), trying to keep variables in registers as long as possible, and so on. It is surprising how much faster a program can run by using simple techniques like those whose descriptions appear thoughout this text.

As a last resort, you can resort to *cycle counting*. At this level you are trying to ensure that an instruction sequence uses as few clock cycles as possible. This is a difficult optimization to perform because you have to be aware of how many clock cycles each instruction consumes, and that depends on the instruction, the addressing mode in use, the instructions around the current instruction (i.e., pipelining and superscalar effects), the speed of the memory system (wait states and cache), and so on. Needless to say, such optimizations are very tedious and require a very careful analysis of the program and the system on which it will run.

The OL crowd always claims you should put off optimization as long as possible. These people are generally talking about this last form of optimization. The reason is simple: any changes you make to your program after such optimizations may change the interaction of the instructions and, therefore, their execution time. If you spend considerable time scheduling a sequence of 50 instructions and then discover you will need to rewrite that code for one reason or another, all the time you spent carefully scheduling those instructions to avoid hazards is lost. On the other hand, if you wait until the last possible moment to make such optimizations to you code, you will only optimize that code once.

Many HLL programmers will tell you that a good compiler can beat a human being at scheduling instructions and optimizing code. This isn't true. A good compiler will beat a mediocre assembly language program a good part of the time. However, a good compiler won't stand a chance against a good assembly language programmer. After all, the worst that could happen is that the good assembly language programmer will look at the output of the compiler and improve on that.

"Counting cycles" can improve the performance of your programs. On the average, you can speed up your programs by a factor of 50% to 200% by making simple changes (like rearranging instructions). That's the difference between an 80486 and a Pentium! So you shouldn't ignore the possibility of using such optimizations in your programs. Just keep in mind, you should do such optimizations last so you don't wind up redoing them as your code changes.

The rest of this chapter will concentrate on the techniques for improving the implementation of an algorithm, rather than designing a better algorithm or using cycle counting techniques. Designing better algorithms is beyond the scope of this manual (see a good text on algorithm design). Cycle counting is one of those processes that differs from processor to processor. That is, the optimization techniques that work well for the 80386 fail on a 486 or Pentium chip, and vice versa. Since Intel is constantly producing new chips, requring different optimization techniques, listing those techniques here would only make that much more material in this book outdated. Intel publishes such optimization hints in their processor programmer reference manuals. Articles on optimizing assembly language programs often appear in technical magazines like Dr. Dobb's Journal, you should read such articles and learn all the current optimization techniques.

25.5 Improving the Implementation of an Algorithm

One easy way to partially demonstrate how to optimize a piece of code is to provide an example of some program and the optimization steps you can apply to that program. This section will present a short program that *blurs* an eight-bit gray scale image. Then, this section will lead though through several optimization steps and show you how to get that program running over 16 times faster.

The following code assumes that you provide it with a file containing a 251x256 gray scale photographic image. The data structure for this file is as follows:

```
Image: array [0..250, 0..255] of byte;
```

Each byte contains a value in the range 0..255 with zero denoting black, 255 representing white, and the other values representing even shades of gray between these two extremes.

The blurring algorithm averages a pixel[5] with its eight closest neighbors. A single blur operation applies this average to all interior pixels of an image (that is, it does not apply to the pixels on the boundary of the image because they do not have the same number of neighbors as the other pixels). The following Pascal program implements the blurring algorithm and lets the user specify the amount of blurring (by looping through the algorithm the number of times the user specifies)[6]:

```
program PhotoFilter(input,output);

(* Here is the raw file data type produced by the Photoshop program *)

type
 image = array [0..250] of array [0..255] of byte;

(* The variables we will use. Note that the “datain” and “dataout” *)
(* variables are pointers because Turbo Pascal will not allow us to *)
(* allocate more than 64K data in the one global data segment it *)
(* supports. *)

var
 h,i,j,k,l,sum,iterations:integer;
 datain, dataout: ^image;
 f,g:file of image;

begin

 (* Open the files and real the input data *)

 assign(f, ‘roller1.raw’);
 assign(g, ‘roller2.raw’);
 reset(f);
 rewrite(g);
 new(datain);
 new(dataout);
 read(f,datain^);

 (* Get the number of iterations from the user *)

 write(‘Enter number of iterations:’);
 readln(iterations);

 writeln(‘Computing result’);

 (* Copy the data from the input array to the output array. *)
 (* This is a really lame way to copy the border from the *)
 (* input array to the output array. *)

 for i := 0 to 250 do
        for j := 0 to 255 do
              dataout^ [i][j] := datain^ [i][j];

 (* Okay, here’s where all the work takes place. The outside     *)
 (* loop repeats this blurring operation the number of           *)
 (* iterations specified by the user.                            *)

 for h := 1 to iterations do begin

        (* For each row except the first and the last, compute  *)
        (* a new value for each element.                        *)

        for i := 1 to 249 do
```

5. Pixel stands for “picture element.” A pixel is an element of the Image array defined above.
6. A comparable C program appears on the diskette accompanying the lab manual.

```
              (* For each column except the first and the last, com-    *)
              (* pute a new value for each element.                      *)

              for j := 1 to 254 do begin

                        (* For each element in the array, compute a new
                           blurred value by adding up the eight cells
                           around an array element along with eight times
                           the current cell's value. Then divide this by
                           sixteen to compute a weighted average of the
                           nine cells forming a square around the current
                           cell. The current cell has a 50% weighting,
                           the other eight cells around the current cel
                           provide the other 50% weighting (6.25% each). *)

                        sum := 0;
                         for k := -1 to 1 do
                            for l := -1 to 1 do
                                       sum := sum + datain^ [i+k][j+l];

                        (* Sum currently contains the sum of the nine     *)
                        (* cells, add in seven times the current cell so  *)
                        (* we get a total of eight times the current cell. *)

                        dataout^ [i][j] := (sum + datain^ [i][j]*7) div 16;

              end;

              (* Copy the output cell values back to the input cells    *)
              (* so we can perform the blurring on this new data on     *)
              (* the next iteration.                                     *)

              for i := 0 to 250 do
               for j := 0 to 255 do
                        datain^ [i][j] := dataout^ [i][j];

 end;

 writeln('Writing result');
 write(g,dataout^);
 close(f);
 close(g);

end.
```

The Pascal program above, compiled with Turbo Pascal v7.0, takes 45 seconds to compute 100 iterations of the blurring algorithm. A comparable program written in C and compiled with Borland C++ v4.02 takes 29 seconds to run. The same source file compiled with Microsoft C++ v8.00 runs in 21 seconds. Obviously the C compilers produce better code than Turbo Pascal. It took about three hours to get the Pascal version running and tested. The C versions took about another hour to code and test. The following two images provide a "before" and "after" example of this program's function:

Before blurring:

After blurring (10 iterations):

The following is a crude translation from Pascal directly into assembly language of the above program. It requires 36 seconds to run. Yes, the C compilers did a better job, but once you see how bad this code is, you'll wonder what it is that Turbo Pascal is doing to run so slow. It took about an hour to translate the Pascal version into this assembly code and debug it to the point it produced the same output as the Pascal version.

```
; IMGPRCS.ASM
;
; An image processing program.
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
```

```
;
;       Performance comparisons (66 MHz 80486 DX/2 system).
;
;       This code-                     36 seconds.
;       Borland Pascal v7.0-           45 seconds.
;       Borland C++ v4.02-             29 seconds.
;       Microsoft C++ v8.00-           21 seconds.

              .xlist
              include    stdlib.a
              includelib stdlib.lib
              .list
              .286

dseg          segment    para public 'data'

; Loop control variables and other variables:

h             word       ?
i             word       ?
j             word       ?
k             word       ?
l             word       ?
sum           word       ?
iterations    word       ?

; File names:

InName        byte       "roller1.raw",0
OutName       byte       "roller2.raw",0

dseg          ends


; Here is the input data that we operate on.

InSeg         segment    para public 'indata'

DataIn        byte       251 dup (256 dup (?))

InSeg         ends


; Here is the output array that holds the result.

OutSeg        segment    para public 'outdata'

DataOut       byte       251 dup (256 dup (?))

OutSeg        ends



cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

Main          proc
              mov        ax, dseg
              mov        ds, ax
              meminit

              mov        ax, 3d00h          ;Open input file for reading.
              lea        dx, InName
              int        21h
              jnc        GoodOpen
              print
              byte       "Could not open input file.",cr,lf,0
              jmp        Quit

GoodOpen:     mov        bx, ax             ;File handle.
              mov        dx, InSeg          ;Where to put the data.
              mov        ds, dx
              lea        dx, DataIn
```

```
                mov     cx, 256*251         ;Size of data file to read.
                mov     ah, 3Fh
                int     21h
                cmp     ax, 256*251         ;See if we read the data.
                je      GoodRead
                print
                byte    "Did not read the file properly",cr,lf,0
                jmp     Quit

GoodRead:       mov     ax, dseg
                mov     ds, ax
                print
                byte    "Enter number of iterations: ",0
                getsm
                atoi
                free
                mov     iterations, ax
                print
                byte    "Computing Result",cr,lf,0

; Copy the input data to the output buffer.

                mov     i, 0
iloop0:         cmp     i, 250
                ja      iDone0
                mov     j, 0
jloop0:         cmp     j, 255
                ja      jDone0

                mov     bx, i               ;Compute index into both
                shl     bx, 8               ; arrays using the formula
                add     bx, j               ; i*256+j (row major).

                mov     cx, InSeg           ;Point at input segment.
                mov     es, cx
                mov     al, es:DataIn[bx] ;Get DataIn[i][j].

                mov     cx, OutSeg          ;Point at output segment.
                mov     es, cx
                mov     es:DataOut[bx], al ;Store into DataOut[i][j]

                inc     j                   ;Next iteration of j loop.
                jmp     jloop0

jDone0:         inc     i                   ;Next iteration of i loop.
                jmp     iloop0

iDone0:

; for h := 1 to iterations-

                mov     h, 1
hloop:          mov     ax, h
                cmp     ax, iterations
                ja      hloopDone



; for i := 1 to 249 -

                mov     i, 1
iloop:          cmp     i, 249
                ja      iloopDone

; for j := 1 to 254 -
                mov     j, 1
jloop:          cmp     j, 254
                ja      jloopDone


; sum := 0;
; for k := -1 to 1 do for l := -1 to 1 do

                mov     ax, InSeg           ;Gain access to InSeg.
```

```
                mov     es, ax

                mov     sum, 0
                mov     k, -1
kloop:          cmp     k, 1
                jg      kloopDone

                mov     l, -1
lloop:          cmp     l, 1
                jg      lloopDone

; sum := sum + datain [i+k][j+l]

                mov     bx, i
                add     bx, k
                shl     bx, 8           ;Multiply by 256.
                add     bx, j
                add     bx, l

                mov     al, es:DataIn[bx]
                mov     ah, 0
                add     Sum, ax

                inc     l
                jmp     lloop

lloopDone:      inc     k
                jmp     kloop


; dataout [i][j] := (sum + datain[i][j]*7) div 16;

kloopDone:      mov     bx, i
                shl     bx, 8           ;*256
                add     bx, j
                mov     al, es:DataIn[bx]
                mov     ah, 0
                imul    ax, 7
                add     ax, sum
                shr     ax, 4           ;div 16

                mov     bx, OutSeg
                mov     es, bx

                mov     bx, i
                shl     bx, 8
                add     bx, j
                mov     es:DataOut[bx], al

                inc     j
                jmp     jloop

jloopDone:      inc     i
                jmp     iloop

iloopDone:
; Copy the output data to the input buffer.

                mov     i, 0
iloop1:         cmp     i, 250
                ja      iDone1
                mov     j, 0
jloop1:         cmp     j, 255
                ja      jDone1

                mov     bx, i           ;Compute index into both
                shl     bx, 8           ; arrays using the formula
                add     bx, j           ; i*256+j (row major).

                mov     cx, OutSeg      ;Point at input segment.
                mov     es, cx
                mov     al, es:DataOut[bx] ;Get DataIn[i][j].

                mov     cx, InSeg       ;Point at output segment.
```

```
                mov     es, cx
                mov     es:DataIn[bx], al ;Store into DataOut[i][j]

                inc     j                 ;Next iteration of j loop.
                jmp     jloop1

jDone1:         inc     i                 ;Next iteration of i loop.
                jmp     iloop1

iDone1:         inc     h
                jmp     hloop

hloopDone:      print
                byte    “Writing result”,cr,lf,0


; Okay, write the data to the output file:

                mov     ah, 3ch           ;Create output file.
                mov     cx, 0             ;Normal file attributes.
                lea     dx, OutName
                int     21h
                jnc     GoodCreate
                print
                byte    “Could not create output file.”,cr,lf,0
                jmp     Quit

GoodCreate:     mov     bx, ax        ;File handle.
                push    bx
                mov     dx, OutSeg        ;Where the data can be found.
                mov     ds, dx
                lea     dx, DataOut
                mov     cx, 256*251       ;Size of data file to write.
                mov     ah, 40h           ;Write operation.
                int     21h
                pop     bx                ;Retrieve handle for close.
                cmp     ax, 256*251       ;See if we wrote the data.
                je      GoodWrite
                print
                byte    “Did not write the file properly”,cr,lf,0
                jmp     Quit

GoodWrite:      mov     ah, 3eh           ;Close operation.
                int     21h


Quit:           ExitPgm                   ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack ‘stack’
stk             byte    1024 dup (“stack “)
sseg            ends

zzzzzzseg       segment para public ‘zzzzzz’
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

This assembly code is a very straight-forward, line by line translation of the previous Pascal code. Even beginning programmers (who've read and understand Chapters Eight and Nine) should easily be able to improve the performance of this code.

While we could run a profiler on this program to determine where the “hot spots” are in this code, a little analysis, particularly of the Pascal version, should make it obvious that there are a lot of nested loops in this code. As Chapter Ten points out, when optimizing code you should always start with the innermost loops. The major change between the code above and the following assembly language version is that we've unrolled the innermost loops and we've replaced the array index computations with some constant computations. These minor changes speed up the execution by a factor of six! The assembly version now runs in six seconds rather than 36. A Microsoft C++ version of the same program with comparable optimzations runs in eight seconds. It required nearly four hours to develop, test, and debug this code. It required an additional hour to apply these same modifications to the C version[7].

```
; IMGPRCS2.ASM
;
; An image processing program (First optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
; Version #2: Three major optimizations. (1) used movsd instruction rather
;          than a loop to copy data from DataOut back to DataIn.
;          (2) Used repeat..until forms for all loops. (3) unrolled
;          the innermost two loops (which is responsible for most of
;          the performance improvement).
;
;
;        Performance comparisons (66 MHz 80486 DX/2 system).
;
;        This code-                    6 seconds.
;        Original ASM code-            36 seconds.
;        Borland Pascal v7.0-          45 seconds.
;        Borland C++ v4.02-            29 seconds.
;        Microsoft C++ v8.00-          21 seconds.


;        « Lots of omitted code goes here, see the previous version»


              print
              byte      “Computing Result”,cr,lf,0

; for h := 1 to iterations-

              mov       h, 1
hloop:

; Copy the input data to the output buffer.
; Optimization step #1: Replace with movs instruction.

              push      ds
              mov       ax, OutSeg
              mov       ds, ax
              mov       ax, InSeg
              mov       es, ax
              lea       si, DataOut
              lea       di, DataIn
              mov       cx, (251*256)/4
        rep   movsd
              pop       ds


; Optimization Step #1: Convert loops to repeat..until form.

; for i := 1 to 249 -

              mov       i, 1
iloop:

; for j := 1 to 254 -
```

7. This does not imply that coding this improved algorithm in C was easier. Most of the time on the assembly version was spent trying out several different modifications to see if they actually improved performance. Many modifications did not, so they were removed from the code. The development of the C version benefited from the past work on the assembly version. It was a straight-forward conversion from assembly to C.

```
                mov     j, 1
jloop:


; Optimization. Unroll the innermost two loops:

                mov     bh, byte ptr i ;i is always less than 256.
                mov     bl, byte ptr j ;Computes i*256+j!

                push    ds
                mov     ax, InSeg      ;Gain access to InSeg.
                mov     ds, ax

                mov     cx, 0             ;Compute sum here.
                mov     ah, ch
                mov     cl, ds:DataIn[bx-257];DataIn[i-1][j-1]
                mov     al, ds:DataIn[bx-256];DataIn[i-1][j]
                add     cx, ax
                mov     al, ds:DataIn[bx-255];DataIn[i-1][j+1]
                add     cx, ax
                mov     al, ds:DataIn[bx-1];DataIn[i][j-1]
                add     cx, ax
                mov     al, ds:DataIn[bx+1];DataIn[i][j+1]
                add     cx, ax
                mov     al, ds:DataIn[bx+255];DataIn[i+1][j-1]
                add     cx, ax
                mov     al, ds:DataIn[bx+256];DataIn[i+1][j]
                add     cx, ax
                mov     al, ds:DataIn[bx+257];DataIn[i+1][j+1]
                add     cx, ax

                mov     al, ds:DataIn[bx];DataIn[i][j]
                shl     ax, 3             ;DataIn[i][j]*8
                add     cx, ax
                shr     cx, 4             ;Divide by 16
                mov     ax, OutSeg
                mov     ds, ax
                mov     ds:DataOut[bx], cl
                pop     ds

                inc     j
                cmp     j, 254
                jbe     jloop

                inc     i
                cmp     i, 249
                jbe     iloop

                inc     h
                mov     ax, h
                cmp     ax, Iterations
                jnbe    Done
                jmp     hloop

Done:           print
                byte    “Writing result”,cr,lf,0

;       «More omitted code goes here, see the previous version»
```

The second version above still uses memory variables for most computations. The optimizations applied to the original code were mainly language-independent optimizations. The next step was to begin applying some assembly language specific optimizations to the code. The first optimization we need to do is to move as many variables as possible into the 80x86's register set. The following code provides this optimization. Although this only improves the running time by 2 seconds, that is a 33% improvement (six seconds down to four)!

```
; IMGPRCS.ASM
;
; An image processing program (Second optimization pass).
```

```
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
; Version #3: Used registers for all variables. Set up segment registers
;         once and for all through the execution of the main loop so
;         the code didn’t have to reload ds each time through. Computed
;         index into each row only once (outside the j loop).
;
;
;        Performance comparisons (66 MHz 80486 DX/2 system).
;
;        This code-                     4 seconds.
;        1st optimization pass-         6 seconds.
;        Original ASM code-             36 seconds.


;        «Lots of delete code goes here»

              print
              byte       “Computing Result”,cr,lf,0


; Copy the input data to the output buffer.


hloop:        mov        ax, InSeg
              mov        es, ax
              mov        ax, OutSeg
              mov        ds, ax
              lea        si, DataOut
              lea        di, DataIn
              mov        cx, (251*256)/4
        rep   movsd

              assume     ds:InSeg, es:OutSeg
              mov        ax, InSeg
              mov        ds, ax
              mov        ax, OutSeg
              mov        es, ax



              mov        cl, 249
iloop:        mov        bh, cl              ;i*256
              mov        bl, 1               ;Start at j=1.
              mov        ch, 254             ;# of times through loop.
jloop:
              mov        dx, 0               ;Compute sum here.
              mov        ah, dh
              mov        dl, DataIn[bx-257]               ;DataIn[i-1][j-1]
              mov        al, DataIn[bx-256]               ;DataIn[i-1][j]
              add        dx, ax
              mov        al, DataIn[bx-255]               ;DataIn[i-1][j+1]
              add        dx, ax
              mov        al, DataIn[bx-1]                 ;DataIn[i][j-1]
              add        dx, ax
              mov        al, DataIn[bx+1]                 ;DataIn[i][j+1]
              add        dx, ax
              mov        al, DataIn[bx+255]               ;DataIn[i+1][j-1]
              add        dx, ax
              mov        al, DataIn[bx+256]               ;DataIn[i+1][j]
              add        dx, ax
              mov        al, DataIn[bx+257]               ;DataIn[i+1][j+1]
```

```
                add     dx, ax

                mov     al, DataIn[bx]              ;DataIn[i][j]
                shl     ax, 3                       ;DataIn[i][j]*8
                add     dx, ax
                shr     dx, 4                       ;Divide by 16
                mov     DataOut[bx], dl

                inc     bx
                dec     ch
                jne     jloop

                dec     cl
                jne     iloop

                dec     bp
                jne     hloop

Done:           print
                byte    “Writing result”,cr,lf,0

;       «More deleted code goes here, see the original version»
```

Note that on each iteration, the code above still copies the output data back to the input data. That's almost six and a half megabytes of data movement for 100 iterations! The following version of the blurring program unrolls the hloop twice. The first occurrence copies the data from DataIn to DataOut while computing the blur, the second instance copies the data from DataOut back to DataIn while blurring the image. By using these two code sequences, the program save copying the data from one point to another. This version also maintains some common computations between two adjacent cells to save a few instructions in the innermost loop. This version arranges instructions in the innermost loop to help avoid data hazards on 80486 and later processors. The end result is almost 40% faster than the previous version (down to 2.5 seconds from four seconds).

```
; IMGPRCS.ASM
;
; An image processing program (Third optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;         once and for all through the execution of the main loop so
;         the code didn’t have to reload ds each time through. Computed
;         index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;         Removed hazards. Maintained common subexpressions. Did some
;         more loop unrolling.
;
;
;       Performance comparisons (66 MHz 80486 DX/2 system, 100 iterations).
;
;       This code-                      2.5 seconds.
;       2nd optimization pass-          4 seconds.
;       1st optimization pass-          6 seconds.
;       Original ASM code-              36 seconds.

;       «Lots of deleted code here, see the original version»
```

```
                print
                byte        "Computing Result",cr,lf,0


                assume      ds:InSeg, es:OutSeg

                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax

; Copy the data once so we get the edges in both arrays.

                mov         cx, (251*256)/4
                lea         si, DataIn
                lea         di, DataOut
        rep     movsd


; "hloop" repeats once for each iteration.

hloop:
                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax

; "iloop" processes the rows in the matrices.

                mov         cl, 249
iloop:          mov         bh, cl              ;i*256
                mov         bl, 1               ;Start at j=1.
                mov         ch, 254/2           ;# of times through loop.
                mov         si, bx
                mov         dh, 0               ;Compute sum here.
                mov         bh, 0
                mov         ah, 0

; "jloop" processes the individual elements of the array.
; This loop has been unrolled once to allow the two portions to share
; some common computations.

jloop:

; The sum of DataIn [i-1][j] + DataIn[i-1][j+1] + DataIn[i+1][j] +
; DataIn [i+1][j+1] will be used in the second half of this computation.
; So save its value in a register (di) until we need it again.

                mov         dl, DataIn[si-256]              ;[i-1,j]
                mov         al, DataIn[si-255]              ;[i-1,j+1]
                mov         bl, DataIn[si+257]              ;[i+1,j+1]
                add         dx, ax
                mov         al, DataIn[si+256]              ;[I+1,j]
                add         dx, bx
                mov         bl, DataIn[si+1]                ;[i,j+1]
                add         dx, ax
                mov         al, DataIn[si+255]              ;[i+1,j-1]

                mov         di, dx                          ;Save partial result.

                add         dx, bx
                mov         bl, DataIn[si-1]                ;[i,j-1]
                add         dx, ax
                mov         al, DataIn[si]                  ;[i,j]
                add         dx, bx
                mov         bl, DataIn[si-257]              ;[i-1,j-1]
                shl         ax, 3                           ;DataIn[i,j] * 8.
                add         dx, bx
                add         dx, ax
                shr         ax, 3                           ;Restore DataIn[i,j].
                shr         dx, 4                           ;Divide by 16.
                add         di, ax
                mov         DataOut[si], dl
```

```
; Okay, process the next cell over. Note that we've got a partial sum
; sitting in DI already. Don't forget, we haven't bumped SI at this point,
; so the offsets are off by one. (This is the second half of the unrolled
; loop.)

                mov     dx, di                          ;Partial sum.
                mov     bl, DataIn[si-254]              ;[i-1,j+1]
                mov     al, DataIn[si+2]                ;[i,j+1]
                add     dx, bx
                mov     bl, DataIn[si+258]              ;[i+1,j+1];
                add     dx, ax
                mov     al, DataIn[si+1]                ;[i,j]
                add     dx, bx
                shl     ax, 3                           ;DataIn[i][j]*8
                add     si, 2                           ;Bump array index.
                add     dx, ax
                mov     ah, 0                           ;Clear for next iter.
                shr     dx, 4                           ;Divide by 16
                dec     ch
                mov     DataOut[si-1], dl
                jne     jloop

                dec     cl
                jne     iloop

                dec     bp
                je      Done


; Special case so we don't have to move the data between the two arrays.
; This is an unrolled version of the hloop that swaps the input and output
; arrays so we don't have to move data around in memory.

                mov     ax, OutSeg
                mov     ds, ax
                mov     ax, InSeg
                mov     es, ax
                assume  es:InSeg, ds:OutSeg

hloop2:

                mov     cl, 249
iloop2:         mov     bh, cl
                mov     bl, 1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0
                mov     bh, 0
                mov     ah, 0
jloop2:
                mov     dl, DataOut[si-256]
                mov     al, DataOut[si-255]
                mov     bl, DataOut[si+257]
                add     dx, ax
                mov     al, DataOut[si+256]
                add     dx, bx
                mov     bl, DataOut[si+1]
                add     dx, ax
                mov     al, DataOut[si+255]

                mov     di, dx

                add     dx, bx
                mov     bl, DataOut[si-1]
                add     dx, ax
                mov     al, DataOut[si]
                add     dx, bx
                mov     bl, DataOut[si-257]
                shl     ax, 3
                add     dx, bx
                add     dx, ax
                shr     ax, 3
                shr     dx, 4
                mov     DataIn[si], dl
```

```
                mov     dx, di
                mov     bl, DataOut[si-254]
                add     dx, ax
                mov     al, DataOut[si+2]
                add     dx, bx
                mov     bl, DataOut[si+258]
                add     dx, ax
                mov     al, DataOut[si+1]
                add     dx, bx
                shl     ax, 3
                add     si, 2
                add     dx, ax
                mov     ah, 0
                shr     dx, 4
                dec     ch
                mov     DataIn[si-1], dl
                jne     jloop2

                dec     cl
                jne     iloop2

                dec     bp
                je      Done2
                jmp     hloop

; Kludge to guarantee that the data always resides in the output segment.

Done2:
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax
                mov     cx, (251*256)/4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd

Done:           print
                byte    "Writing result",cr,lf,0

;       «Lots of deleted code here, see the original program»
```

This code provides a good example of the kind of optimization that scares a lot of people. There is a lot of cycle counting, instruction scheduling, and other crazy stuff that makes program very difficult to read and understand. This is the kind of optimization for which assembly language programmers are famous; the stuff that spawned the phrase "never optimize early." You should never try this type of optimization until you feel you've exhausted all other possibilities. Once you write your code in this fashion, it is going to be very difficult to make further changes to it. By the way, the above code took about 15 hours to develop and debug (debugging took the most time). That works out to a 0.1 second improvement (for 100 iterations) for each hour of work. Although this code certainly isn't optimal yet, it is difficult to justify more time attempting to improve this code by mechanical means (e.g., moving instructions around, etc.) because the performance gains would be so little.

In the four steps above, we've reduced the running time of the assembly code from 36 seconds down to 2.5 seconds. Quite an impressive feat. However, you shouldn't get the idea that this was easy or even that there were only four steps involved. During the actual development of this example, there were many attempts that did not improve performance (in fact, some modifications wound up reducing performance) and others did not improve performance enough to justify their inclusion. Just to demonstrate this last point, the following code included a major change in the way the program organized data. The main loop operates on 16 bit objects in memory rather than eight bit objects. On some machines with large external caches (256K or better) this algorithm provides a slight improvement in performance (2.4 seconds, down from 2.5). However, on other machines it runs slower. Therefore, this code was not chosen as the final implementation:

```
; IMGPRCS.ASM
;
; An image processing program (Fourth optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;         once and for all through the execution of the main loop so
;         the code didn't have to reload ds each time through. Computed
;         index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;         Removed hazards. Maintained common subexpressions. Did some
;         more loop unrolling.
;
; Version #5: Converted data arrays to words rather than bytes and operated
;         on 16-bit values. Yielded minimal speedup.
;
;        Performance comparisons (66 MHz 80486 DX/2 system).
;
;        This code-                    2.4 seconds.
;        3rd optimization pass-        2.5 seconds.
;        2nd optimization pass-        4 seconds.
;        1st optimization pass-        6 seconds.
;        Original ASM code-            36 seconds.

              .xlist
              include    stdlib.a
              includelib stdlib.lib
              .list
              .386
              option        segment:use16



dseg          segment    para public 'data'

ImgData       byte       251 dup (256 dup (?))

InName        byte       "roller1.raw",0
OutName       byte       "roller2.raw",0
Iterations    word       0

dseg          ends


; This code makes the naughty assumption that the following
; segments are loaded contiguously in memory! Also, because these
; segments are paragraph aligned, this code assumes that these segments
; will contain a full 65,536 bytes. You cannot declare a segment with
; exactly 65,536 bytes in MASM. However, the paragraph alignment option
; ensures that the extra byte of padding is added to the end of each
; segment.

DataSeg1      segment    para public 'ds1'
Data1a        byte       65535 dup (?)
DataSeg1      ends

DataSeg2      segment    para public 'ds2'
Data1b        byte       65535 dup (?)
DataSeg2      ends
```

```
DataSeg3        segment     para public 'ds3'
Data2a          byte        65535 dup (?)
DataSeg3        ends

DataSeg4        segment     para public 'ds4'
Data2b          byte        65535 dup (?)
DataSeg4        ends



cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg
                mov         ds, ax
                meminit

                mov         ax, 3d00h     ;Open input file for reading.
                lea         dx, InName
                int         21h
                jnc         GoodOpen
                print
                byte        "Could not open input file.",cr,lf,0
                jmp         Quit

GoodOpen:       mov         bx, ax            ;File handle.
                lea         dx, ImgData
                mov         cx, 256*251       ;Size of data file to read.
                mov         ah, 3Fh
                int         21h
                cmp         ax, 256*251       ;See if we read the data.
                je          GoodRead
                print
                byte        "Did not read the file properly",cr,lf,0
                jmp         Quit

GoodRead:       print
                byte        "Enter number of iterations: ",0
                getsm
                atoi
                free
                mov         Iterations, ax
                cmp         ax, 0
                jle         Quit

                printf
                byte        "Computing Result for %d iterations",cr,lf,0
                dword       Iterations



; Copy the data and expand it from eight bits to sixteen bits.
; The first loop handles the first 32,768 bytes, the second loop
; handles the remaining bytes.

                mov         ax, DataSeg1
                mov         es, ax
                mov         ax, DataSeg3
                mov         fs, ax

                mov         ah, 0
                mov         cx, 32768
                lea         si, ImgData
                xor         di, di            ;Output data is at ofs zero.
CopyLoop:       lodsb                         ;Read a byte
                mov         fs:[di], ax       ;Store a word in DataSeg3
                stosw                         ;Store a word in DataSeg1
                dec         cx
                jne         CopyLoop

                mov         di, DataSeg2
```

```
                mov         es, di
                mov         di, DataSeg4
                mov         fs, di
                mov         cx, (251*256) - 32768
                xor         di, di
CopyLoop1:      lodsb                               ;Read a byte
                mov         fs:[di], ax             ;Store a word in DataSeg4
                stosw                               ;Store a word in DataSeg2
                dec         cx
                jne         CopyLoop1

; hloop completes one iteration on the data moving it from Data1a/Data1b
; to Data2a/Data2b

hloop:          mov         ax, DataSeg1
                mov         ds, ax
                mov         ax, DataSeg3
                mov         es, ax

; Process the first 127 rows (65,024 bytes) of the array):

                mov         cl, 127
                lea         si, Data1a+202h         ;Start at [1,1]
iloop0:         mov         ch, 254/2               ;# of times through loop.
jloop0:         mov         dx, [si]                ;[i,j]
                mov         bx, [si-200h]           ;[i-1,j]
                mov         ax, dx
                shl         dx, 3                   ;[i,j] * 8
                add         bx, [si-1feh]           ;[i-1,j+1]
                mov         bp, [si+2]              ;[i,j+1]
                add         bx, [si+200h]           ;[i+1,j]
                add         dx, bp
                add         bx, [si+202h]           ;[i+1,j+1]
                add         dx, [si-202h]           ;[i-1,j-1]
                mov         di, [si-1fch]           ;[i-1,j+2]
                add         dx, [si-2]              ;[i,j-1]
                add         di, [si+4]              ;[i,j+2]
                add         dx, [si+1feh]           ;[i+1,j-1]
                add         di, [si+204h]           ;[i+1,j+2]
                shl         bp, 3                   ;[i,j+1] * 8
                add         dx, bx
                add         bp, ax
                shr         dx, 4                   ;Divide by 16.
                add         bp, bx
                mov         es:[si], dx             ;Store [i,j] entry.
                add         bp, di
                add         si, 4                   ;Affects next store operation!
                shr         bp, 4                   ;Divide by 16.
                dec         ch
                mov         es:[si-2], bp           ;Store [i,j+1] entry.
                jne         jloop0

                add         si, 4                   ;Skip to start of next row.

                dec         cl
                jne         iloop0

; Process the last 124 rows of the array). This requires that we switch from
; one segment to the next. Note that the segments overlap.

                mov         ax, DataSeg2
                sub         ax, 40h                 ;Back up to last 2 rows in DS2
                mov         ds, ax
                mov         ax, DataSeg4
                sub         ax, 40h                 ;Back up to last 2 rows in DS4
                mov         es, ax

                mov         cl, 251-127-1           ;Remaining rows to process.
                mov         si, 202h                ;Continue with next row.
iloop1:         mov         ch, 254/2               ;# of times through loop.
jloop1:         mov         dx, [si]                ;[i,j]
                mov         bx, [si-200h]           ;[i-1,j]
                mov         ax, dx
                shl         dx, 3                   ;[i,j] * 8
```

```
                add     bx, [si-1feh]       ;[i-1,j+1]
                mov     bp, [si+2]          ;[i,j+1]
                add     bx, [si+200h]       ;[i+1,j]
                add     dx, bp
                add     bx, [si+202h]       ;[i+1,j+1]
                add     dx, [si-202h]       ;[i-1,j-1]
                mov     di, [si-1fch]       ;[i-1,j+2]
                add     dx, [si-2]          ;[i,j-1]
                add     di, [si+4]          ;[i,j+2]
                add     dx, [si+1feh]       ;[i+1,j-1]
                add     di, [si+204h]       ;[i+1,j+2]
                shl     bp, 3               ;[i,j+1] * 8
                add     dx, bx
                add     bp, ax
                shr     dx, 4               ;Divide by 16
                add     bp, bx
                mov     es:[si], dx         ;Store [i,j] entry.
                add     bp, di
                add     si, 4               ;Affects next store operation!
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp       ;Store [i,j+1] entry.
                jne     jloop1

                add     si, 4               ;Skip to start of next row.

                dec     cl
                jne     iloop1

                mov     ax, dseg
                mov     ds, ax
                assume  ds:dseg

                dec     Iterations
                je      Done0

; Unroll the iterations loop so we can move the data from DataSeg2/4 back
; to DataSeg1/3 without wasting extra time. Other than the direction of the
; data movement, this code is virtually identical to the above.

                mov     ax, DataSeg3
                mov     ds, ax
                mov     ax, DataSeg1
                mov     es, ax

                mov     cl, 127
                lea     si, Data1a+202h
iloop2:         mov     ch, 254/2
jloop2:         mov     dx, [si]
                mov     bx, [si-200h]
                mov     ax, dx
                shl     dx, 3
                add     bx, [si-1feh]
                mov     bp, [si+2]
                add     bx, [si+200h]
                add     dx, bp
                add     bx, [si+202h]
                add     dx, [si-202h]
                mov     di, [si-1fch]
                add     dx, [si-2]
                add     di, [si+4]
                add     dx, [si+1feh]
                add     di, [si+204h]
                shl     bp, 3
                add     dx, bx
                add     bp, ax
                shr     dx, 4
                add     bp, bx
                mov     es:[si], dx
                add     bp, di
                add     si, 4
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp
```

```
                jne         jloop2

                add         si, 4

                dec         cl
                jne         iloop2


                mov         ax, DataSeg4
                sub         ax, 40h
                mov         ds, ax
                mov         ax, DataSeg2
                sub         ax, 40h
                mov         es, ax

                mov         cl, 251-127-1
                mov         si, 202h
iloop3:         mov         ch, 254/2
jloop3:         mov         dx, [si]
                mov         bx, [si-200h]
                mov         ax, dx
                shl         dx, 3
                add         bx, [si-1feh]
                mov         bp, [si+2]
                add         bx, [si+200h]
                add         dx, bp
                add         bx, [si+202h]
                add         dx, [si-202h]
                mov         di, [si-1fch]
                add         dx, [si-2]
                add         di, [si+4]
                add         dx, [si+1feh]
                add         di, [si+204h]
                shl         bp, 3
                add         dx, bx
                add         bp, ax
                shr         dx, 4
                add         bp, bx
                mov         es:[si], dx
                add         bp, di
                add         si, 4
                shr         bp, 4
                dec         ch
                mov         es:[si-2], bp
                jne         jloop3

                add         si, 4

                dec         cl
                jne         iloop3

                mov         ax, dseg
                mov         ds, ax
                assume      ds:dseg

                dec         Iterations
                je          Done2
                jmp         hloop

Done2:          mov         ax, DataSeg1
                mov         bx, DataSeg2
                jmp         Finish

Done0:          mov         ax, DataSeg3
                mov         bx, DataSeg4
Finish:         mov         ds, ax
                print
                byte        “Writing result”,cr,lf,0

; Convert data back to byte form and write to the output file:

                mov         ax, dseg
                mov         es, ax
```

```
                mov       cx, 32768
                lea       di, ImgData
                xor       si, si            ;Output data is at offset zero.
CopyLoop3:      lodsw                       ;Read a word from final array.
                stosb                       ;Write a byte to output array.
                dec       cx
                jne       CopyLoop3

                mov       ds, bx
                mov       cx, (251*256) - 32768
                xor       si, si
CopyLoop4:      lodsw                       ;Read final data word.
                stosb                       ;Write data byte to output array.
                dec       cx
                jne       CopyLoop4


; Okay, write the data to the output file:

                mov       ah, 3ch           ;Create output file.
                mov       cx, 0             ;Normal file attributes.
                mov       dx, dseg
                mov       ds, dx
                lea       dx, OutName
                int       21h
                jnc       GoodCreate
                print
                byte      “Could not create output file.”,cr,lf,0
                jmp       Quit

GoodCreate:     mov       bx, ax            ;File handle.
                push      bx
                mov       dx, dseg          ;Where the data can be found.
                mov       ds, dx
                lea       dx, ImgData
                mov       cx, 256*251       ;Size of data file to write.
                mov       ah, 40h           ;Write operation.
                int       21h
                pop       bx                ;Retrieve handle for close.
                cmp       ax, 256*251       ;See if we wrote the data.
                je        GoodWrite
                print
                byte      “Did not write the file properly”,cr,lf,0
                jmp       Quit

GoodWrite:      mov       ah, 3eh           ;Close operation.
                int       21h


Quit:           ExitPgm                     ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment   para stack ‘stack’
stk             byte      1024 dup (“stack “)
sseg            ends

zzzzzzseg       segment   para public ‘zzzzzz’
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       Main
```

Of course, the absolute best way to improve the performance of any piece of code is with a better algorithm. All of the above assembly language versions were limited by a single requirement – they all must produce the same output file as the original Pascal program. Often, programmers lose sight of what it is that they are trying to accomplish and get so caught up in the computations they are performing that they fail to see other possibilities. The optimization example above is a perfect example. The assembly code faithfully preserves the semantics of the original Pascal program; it computes the weighted average of all interior pixels as the sum of the eight neighbors around a pixel plus eight times the current pixel's value, with the entire sum divided by 16. Now this is a *good* blurring function, but it is not the *only* blurring function. A Photoshop (or other image processing program) user doesn't care about algorithms or such. When that user selects "blur image" they want it to go out of focus. Exactly how much out of focus is generally immaterial. In fact, the less the better because the user can always run the blur algorithm again (or specify some number of iterations). The following assembly language program shows how to get better performance by modifying the blurring algorithm to reduce the number of instructions it needs to execute in the innermost loops. It computes blurring by averaging a pixel with the four neighbors above, below, to the left, and to the right of the current pixel. This modification yields a program that runs 100 iterations in 2.2 seconds, a 12% improvement over the previous version:

```
; IMGPRCS.ASM
;
; An image processing program (Fifth optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;         once and for all through the execution of the main loop so
;         the code didn't have to reload ds each time through. Computed
;         index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;         Removed hazards. Maintained common subexpressions. Did some
;         more loop unrolling.
;
; Version #6: Changed the blurring algorithm to use fewer computations.
;         This version does *NOT* produce the same data as the other
;         programs.
;
;
;
;        Performance comparisons (66 MHz 80486 DX/2 system, 100 iterations).
;
;        This code-                     2.2 seconds.
;        3rd optmization pass-          2.5 seconds.
;        2nd optimization pass-         4 seconds.
;        1st optimization pass-         6 seconds.
;        Original ASM code-             36 seconds.

;        «Lots of deleted code here, see the original program»


              print
              byte       "Computing Result",cr,lf,0


              assume     ds:InSeg, es:OutSeg

              mov        ax, InSeg
              mov        ds, ax
              mov        ax, OutSeg
              mov        es, ax

; Copy the data once so we get the edges in both arrays.

              mov        cx, (251*256)/4
              lea        si, DataIn
```

```
                lea         di, DataOut
        rep     movsd


; "hloop" repeats once for each iteration.

hloop:
                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax

; "iloop" processes the rows in the matrices.

                mov         cl, 249
iloop:          mov         bh, cl              ;i*256
                mov         bl, 1               ;Start at j=1.
                mov         ch, 254/2           ;# of times through loop.
                mov         si, bx
                mov         dh, 0               ;Compute sum here.
                mov         bh, 0
                mov         ah, 0

; "jloop" processes the individual elements of the array.
; This loop has been unrolled once to allow the two portions to share
; some common computations.

jloop:

; The sum of DataIn [i-1][j] + DataIn[i-1][j+1] + DataIn[i+1][j] +
; DataIn [i+1][j+1] will be used in the second half of this computation.
; So save its value in a register (di) until we need it again.

                mov         dl, DataIn[si]                  ;[i,j]
                mov         al, DataIn[si-256]              ;[I-1,j]
                shl         dx, 2                           ;[i,j]*4
                mov         bl, DataIn[si-1]                ;[i,j-1]
                add         dx, ax
                mov         al, DataIn[si+1]                ;[i,j+1]
                add         dx, bx
                mov         bl, DataIn[si+256]              ;[i+1,j]
                add         dx, ax
                shl         ax, 2                           ;[i,j+1]*4
                add         dx, bx
                mov         bl, DataIn[si-255]              ;[i-1,j+1]
                shr         dx, 3                           ;Divide by 8.
                add         ax, bx
                mov         DataOut[si], dl
                mov         bl, DataIn[si+2]                ;[i,j+2]
                mov         dl, DataIn[si+257]              ;[i+1,j+1]
                add         ax, bx
                mov         bl, DataIn[si]                  ;[i,j]
                add         ax, dx
                add         ax, bx
                shr         ax, 3
                dec         ch
                mov         DataOut[si+1], al
                jne         jloop

                dec         cl
                jne         iloop

                dec         bp
                je          Done


; Special case so we don't have to move the data between the two arrays.
; This is an unrolled version of the hloop that swaps the input and output
; arrays so we don't have to move data around in memory.

                mov         ax, OutSeg
                mov         ds, ax
                mov         ax, InSeg
                mov         es, ax
```

```
                assume      es:InSeg, ds:OutSeg

hloop2:

                mov         cl, 249
iloop2:         mov         bh, cl
                mov         bl, 1
                mov         ch, 254/2
                mov         si, bx
                mov         dh, 0
                mov         bh, 0
                mov         ah, 0
jloop2:
                mov         dl, DataOut[si-256]
                mov         al, DataOut[si-255]
                mov         bl, DataOut[si+257]
                add         dx, ax
                mov         al, DataOut[si+256]
                add         dx, bx
                mov         bl, DataOut[si+1]
                add         dx, ax
                mov         al, DataOut[si+255]

                mov         di, dx

                add         dx, bx
                mov         bl, DataOut[si-1]
                add         dx, ax
                mov         al, DataOut[si]
                add         dx, bx
                mov         bl, DataOut[si-257]
                shl         ax, 3
                add         dx, bx
                add         dx, ax
                shr         ax, 3
                shr         dx, 4
                mov         DataIn[si], dl

                mov         dx, di
                mov         bl, DataOut[si-254]
                add         dx, ax
                mov         al, DataOut[si+2]
                add         dx, bx
                mov         bl, DataOut[si+258]
                add         dx, ax
                mov         al, DataOut[si+1]
                add         dx, bx
                shl         ax, 3
                add         si, 2
                add         dx, ax
                mov         ah, 0
                shr         dx, 4
                dec         ch
                mov         DataIn[si-1], dl
                jne         jloop2

                dec         cl
                jne         iloop2

                dec         bp
                je          Done2
                jmp         hloop


; Kludge to guarantee that the data always resides in the output segment.

Done2:
                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax
                mov         cx, (251*256)/4
                lea         si, DataIn
                lea         di, DataOut
```

```
        rep   movsd

Done:         print
              byte      "Writing result",cr,lf,0


;       «Lots of delete code here, see the original program»
```

One very important thing to keep in mind about the codein this section is that we've optimized it for 100 iterations. While it turns out that these optimizations apply equally well to more iterations, this isn't necessarily true for fewer iterations. In particular, if we run only one iteration, any copying of data at the end of the operation will easily consume a large part of the time we save by the optimizations. Since it is very rare for a user to blur an image 100 times in a row, our optimizations may not be as good as we could make them. However, this section does provide a good example of the steps you must go through in order to optimize a given program. One hundred iterations was a good choice for this example because it was easy to measure the running time of all versions of the program. However, you must keep in mind that you should optimize your programs for the expected case, not an arbitrary case.

25.6 Summary

Computer software often runs significantly slower than the task requires. The process of increasing the speed of a program is known as *optimization*. Unfortunately, optimization is a difficult and time-consuming task, something not to be taken lightly. Many programmers often optimize their programs before they've determined that there is a need to do so, or (worse yet) they optimize a portion of a program only to find that they have to rewrite that code after they've optimized it. Others, out of ignorance, often wind up optimizing the wrong sections of their programs. Since optimization is a slow and difficult process, you want to try and make sure you only optimize your code *once*. This suggests that optimization should be your last task when writing a program.

One school of thought that completely embraces this philosophy is the *Optimize Late* group. Their arguement is that program optimization often destroys the readability and maintanability of a program. Therefore, one should only take this step when absolutely necessary and only at the end of the program development stage.

The *Optimize Early* crowd knows, from experience, that programs that are not written to be fast often need to be completely rewritten to make them fast. Therefore, they often take the attitude that optimization should take place along with normal program development. Generally, the optimize early group's view of optimization is typically far different from the optimize late group. The optimize early group claims that the extra time spent optimizing a program during development requires less time than developing a program and then optimizing it. For all the details on this *religious* battle, see

• "When to Optimize, When Not to Optimize" on page 1311

After you've written a program and determine that it runs too slowly, the next step is to locate the code that runs too slow. After identifying the slow sections of your program, you can work on speeding up your programs. Locating that 10% of the code that requires 90% of the execution time is not always an easy task. The four common techniques people use are trial and error, optimize everything, program analysis, and experimental analysis (i.e., use a profiler). Finding the "hot spots" in a program is the first optimization step. To learn about these four techniques, see

• "How Do You Find the Slow Code in Your Programs?" on page 1313

A convincing arguement the optimize late folks use is that machines are so fast that optimization is rarely necessary. While this arguement is often overstated, it is often true that many unoptimized programs run fast enough and do not require any optimization for satisfactory performance. On the other hand, programs that run fine by themselves may be too slow when running concurrently with other software. To see the strengths and weaknesses of this arguement, see

• "Is Optimization Necessary?" on page 1314

There are three forms of optimization you can use to improve the performance of a program: choose a better algorithm, choose a better implementation of an algorithm, or "count cycles." Many people (especially the optimize late crowd) only consider this last case "optimization." This is a shame, because the last case often produces the smallest incremental improvement in performance. To understand these three forms of optimization, see

• "The Three Types of Optimization" on page 1315

Optimization is not something you can learn from a book. It takes lots of experience and practice. Unfortunately, those with little practical experience find that their efforts rarely pay off well and generally assume that optimization is not worth the trouble. The truth is, they do not have sufficient experience to write truly optimal code and their frustration prevents them from gaining such experience. The latter part of this chapter devotes itself to demonstrating what one can achieve when optimizing a program. Always keep this example in mind when you feel frustrated and are beginning to believe you cannot improve the performance of your program. For details on this example, see

• "Improving the Implementation of an Algorithm" on page 1317