Software Optimization: Algorithms and Computational Complexity

Abstract

The speed at which software operates is of paramount importance to both software developers and the eventual users. The key factor in determining the speed at which the software navigates is the algorithm that it depends on. To parse the sheer multitude of algorithms available, one must be aware of the efficiency concerns that each algorithm presents. Algorithms may be blended to utilize the strongest features of both, so it is essential to know the strengths and weaknesses of each. In this article, the various aspects of choosing an algorithm are examined, including the importance of reducing computational time, minimizing instruction latency and instruction throughput for all iterations of the loop, and structuring the algorithm so that its instructions can be executed as independently as possible.

Introduction

Any given operation can be performed in multiple ways, depending on which algorithm is used. A properly designed algorithm can be used to efficiently solve a performance problem, such as the implementation of sets or lists. The algorithms available for performing any task are innumerable, and the efficiency of various algorithms varies greatly. There are a variety of factors that determine whether an algorithm will perform its task efficiently or poorly. By the conclusion of this article, a developer should have an idea of what considerations must be weighed when selecting an algorithm to optimize the software.

Algorithms

Choosing an appropriate algorithm is the most important factor in determining whether software will be slow or fast. A good algorithm solves a problem in a fast and efficient manner, while a poor algorithm, no matter how well implemented and tuned, is never as fast.

Algorithms to solve just about any problem are available from a variety of sources: the Internet, books, coworkers, professional journals. Determining which algorithm will be the best for your purpose can be tricky. Computational complexity, memory usage, data dependencies, and the instructions used to implement the algorithm all play a role in determining whether an algorithm will perform well or poorly. Spending some time up front experimenting with different algorithms often saves time later on.

Computational Complexity

Algorithm performance can be judged by using the computer science O-notation analysis for the typical, best, and worst cases. For example, of the many different algorithms for sorting data, which should you choose? The bubble sort is probably the simplest and slowest sorting algorithm. Its computational

complexity is $O(n^2)$, meaning that if the number of elements to be sorted doubles, the elapsed time to perform the sort quadruples. The quicksort algorithm on the other hand has a significantly better computational time complexity of $O(n \log n)$. Table 1.1 compares the two sorting algorithms for a small, medium, and large numbers of elements to be sorted. It shows the approximate number of operations the algorithms must perform based on their computational complexity, and how many more operations bubble sort must perform compared with quicksort.

Table 1.1 Approximate Number of Operations Performed by Sorting

	256 ELEMENTS	1,000 ELEMENTS	10,000 ELEMENTS
Bubble Sort	65,000	1,000,000	100,000,000
Quicksort	2,048	9,965	133,000

As you can see, the quicksort algorithm uses more than 1,000 times fewer operations than bubble sort for a case involving a large number of elements. No matter how much effort you spend implementing and tuning a bubble sort algorithm, it will never be faster than quicksort because of the nature of these two algorithms and their computational complexities. The two sorting examples in Chapter 2 of *the Software Optimization Cookbook, Second Edition* by Gerber et al. showed a similar performance relationship due to differences in computational complexity.

Different algorithm's computational complexity can be used as an estimate of their relative performance by providing an approximation of how many operations an algorithm will perform. When the computational complexity of two algorithms is similar, other factors need to be taken into account in deciding which algorithm to use. For more discussion of algorithms and their computational complexity see *Introduction to Algorithms* by Cormen, Leiserson and Rivest (Cormen 1990) or *Algorithms* by Robert Sedgewick (Sedgewick 1998).

Choice of Instructions

The instructions needed to implement an algorithm can have a big impact on performance and therefore on determining which algorithm to use. Some instructions, like integer addition, are executed extremely quickly while other instructions, like integer division, are executed slowly. The speed of an instruction is specified by its latency and throughput.

Instruction latency is the number of clocks required to complete one instruction after the instruction's inputs are ready—that is, they are fetched from memory—and execution begins. For example, integer multiplication has a latency of about 9 clocks on a Pentium® 4 processor. So, the answer to a multiplication is available 9 clocks after it begins execution.

Instruction throughput is the number of clocks that the processor is required to wait before starting the execution of an identical instruction. Instruction throughput is always less than or equal to instruction latency. Throughput is 4 clocks for multiplication, meaning that a new multiply can begin execution

every 4 clocks even though it takes 9 clocks to get the answer to any specific multiplication. Instruction pipelining causes the number of clocks for throughput and latency to be different.

The latency and throughput for most operations on the Pentium 4 and Pentium M processors is available in the *IA-32 Intel® Architecture Optimization Reference Manual*. See "References" for online availability of this and other Intel product manuals.

Taking latency and throughput into account can have a significant impact on algorithm selection. For example, on a Pentium 4 processor, if one algorithm uses ten additions while a second algorithm uses only one divide, the addition version will be faster because a divide takes forty times longer to execute than an addition.

Finding the greatest common factor of two numbers is a good example of using latency and throughput to select an algorithm. Elementary schools teach children to find the greatest common factor of two numbers by going through the following steps.

1. Factor each number.
2. Find the factors that are common between both numbers.
3. Multiply the common factors together to get the greatest common multiple.

Example 1.1 Find the Greatest Common Factor of 40 and 48 the Elementary School Way

1. Factor each number.
   40 = 2 * 2 * 2 * 5
   48 = 2 * 2 * 2 * 2 * 3
2. Find the common factors.
   2 * 2 * 2
3. Multiply the common factors to get the greatest common multiple.
   Greatest common multiple = 2 * 2 * 2 = 8

The elementary school algorithm is obviously very expensive for a computer; just the first step of factoring the two numbers would take a long time. Lucky for us, long ago Euclid found a much faster algorithm for finding the greatest common factor. Euclid's Algorithm is:

1. Larger number = larger number – smaller number
2. If the numbers are the same, it is the greatest common factor, otherwise go to step 1.

Example 1.2 Find the Greatest Common Factor of 48 and 40 Using Euclid's Algorithm and Repetitive Subtraction

1. 48,40 → 48-40, 40 → 8, 40
2. 8 ≠ 40 so repeat step 1
3. 8, 40 → 40-8, 8 → 32, 8
4. 8 ≠32 so repeat step 1
5. 8, 32 → 32-8, 8 → 24, 8

6. 8 ≠24 so repeat step 1
7. 8, 24 → 24-8, 8 → 16, 8
8. 8 ≠16 so repeat step 1
9. 8, 16 → 16-8, 8 → 8, 8
10. 8 = 8 so 8 is the greatest common factor

Euclid's Algorithm can be written in C as follows:

```
int find_gcf(int a, int b)
{
        /* assumes both a and b are greater than 0 */
        while (1) {
                if (a > b) a = a - b;
                else if (a < b) b = b - a;
                else /* they are equal */ return a;
        }
}
```

The Intel compiler generates the assembly code shown below for this function. The assembly code shows that each iteration of the loop takes one compare, two or three branches, and one subtraction. For the case where a = 48 and b = 40, this implementation executes 5 compares, 14 branches, and 5 subtracts, for a total of 24 instructions.

```
_find_gcf$::
$B2$2:
                cmp             eax, edx
                jle             $B2$4
                sub             eax, edx
                jmp             $B2$2
$B2$4:
                jge             $B2$6
                sub             edx, eax
                jmp             $B2$2
$B2$6:
                Ret
```

A variation on Euclid's Algorithm uses the modulo operation. In C, it can be written as follows:

```
int find_gcf(int a, int b)
{
        /* assumes both a and b are greater than 0 */
        while (1) {
```

```
            a = a % b;
            if (a == 0) return b;
            if (a == 1) return 1;
            b = b % a;
            if (b == 0) return a;
            if (b == 1) return 1;
        }
    }
```

Again, looking at the assembly code that is produced for this function shows exactly what instructions will be executed. For this example, with the values a=48, b=40, the generated code executes 2 divides, 3 compares, 3 branches, 4 moves, and 2 cdq instructions. This implementation uses fewer instructions, a total of only 14. Since 14 instructions is less than 24, the modulo version seems like it should be faster. However, on a Pentium 4 processor, the repetitive subtract algorithm is faster for these input values because modulo uses integer division, which takes about 68 clocks, while subtraction and compares only take 1 clock. In this case, you might choose the repetitive subtraction algorithm, even though it executes more instructions, because the instructions used are much faster.

Table 1.2 estimates the execution times of these two algorithms for a=48 and b=40 and assuming the simplification that branch execution time is one cycle.

Table 1.2 A Rough Comparison of the Two Different Versions of Euclid's Algorithm

REPETITIVE SUBTRACTION VERSION				**MODULO VERSION**			
Instruction	**Quantity**	**Latency**	**Total clocks**	**Instruction**	**Quantity**	**Latency**	**Total clocks**
Subtractions	5	1	5	Modulo (integer division)	2	68	136
Compares	5	1	5	Compares	3	1	3
Branches	14	1	14	Branches	3	1	3
Other	0	1	0	Other	6	1	6
Totals	24		24	Totals	14		148

Even though the algorithm using modulo operations takes fewer instructions, due to the latency of the divide instruction, it often would take significantly more time than the repetitive subtract version of the algorithm. But, depending on the numbers chosen for a and b, the run-time results can vary widely. This variation occurs because the *O* behavior of the algorithms is different. Take the case where a=1000 and b=1. The GCF is 1, and it would take 999 iterations of the loop in the repetitive subtraction algorithm to figure this out, for a total of roughly 5,000 cycles. The modulo algorithm completes the calculation in a

single iteration of the loop, executing only a modulo operation, a compare, and a branch, and taking only about 74 cycles. Looking at these differences, it seems like it might make sense to combine the two algorithms to take advantage of the best of each. The following code is an example of such a blended-algorithm implementation. It uses the inexpensive subtract operations when it can and uses the more expensive division when it looks like it's going to take many iterations of the subtraction to get a similar result. Table 1.3 compares the performance of these three algorithms when run for all combinations of values a and b in [1..9999] on a 3.6-GHz Pentium 4 processor. As the latency of the divide instruction goes down, the modulo version of the algorithm starts to outperform the blended version of the algorithm. This performance increase occurs because the blended algorithm incurs extra overhead for choosing between subtraction and division.

```
int find_gcf(int a, int b)
{
        /* assumes both a and b are greater than 0 */
        while (1) {
                if (a > (b * 4)) {
                        a = a % b;
                        if (a == 0) return b;
                        if (a == 1) return 1;
                }
                else if (a >= b) {
                        a = a - b;
                        if (a == 0) return b;
                        if (a == 1) return 1;
                }
                if (b > (a * 4)) {
                        b = b % a;
                        if (b == 0) return a;
                        if (b == 1) return 1;
                }
                else if (b >= a) {
                        b = b - a;
                        if (b == 0) return a;
                        if (b == 1) return 1;
                }
        }
}
```

Table 1.3 Run Time of Three Different Implementations of Euclid's Algorithm

REPETITIVE SUBTRACTION VERSION	MODULO VERSION	BLENDED VERSION
14.56s	18.55s	12.14s

Data Dependencies and Instruction Parallelism

In addition to instruction latency and throughput, data dependencies affect the processor's ability to execute instructions simultaneously. When an algorithm is structured so that more of its instructions can be executed simultaneously by the processor, the algorithm generally takes less total execution time. The Pentium 4 processor is capable of executing six instructions during every clock cycle, but due to data dependency issues, the number of instructions that are executed simultaneously is usually lower.

Figure 1.1 is a Gantt chart showing how three multiplies might be executed together.

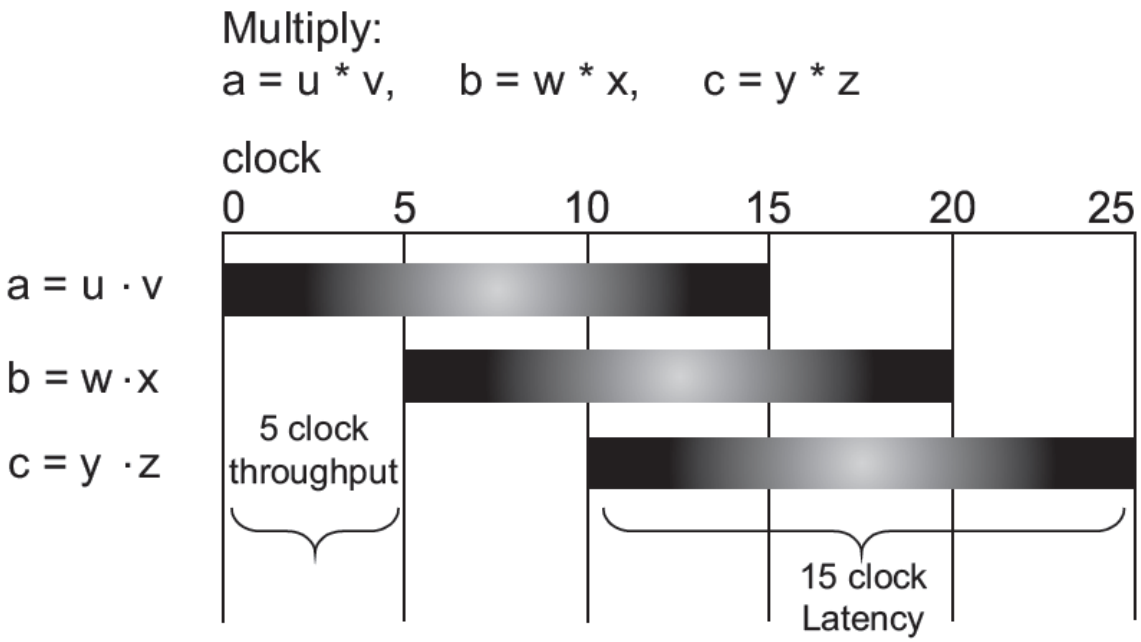

Figure 1.1 Sample Gantt Chart of Instruction Execution without Data Dependencies

The Gantt chart in Figure 1.1 assumes no data dependencies exist among any of the instructions, allowing them to execute at the same time limited only by instruction throughput. However, in the real world, data dependencies do exist, and they can make a big difference. For example, if the three multiplies were data dependent, as in the statement `a = w * x * y * z`, the graph would look very

different because the result of w * x would not be ready to be multiplied by the result of y * z for fifteen clocks. The graph shown in Figure 1.2 is much longer because parallelism is not possible between the multiply operations. It is worth noting that if the throughput and latency are the same for an instruction, the processor can only run a single instruction of that type at a time. Enabling parallelism for such instructions doesn't help performance because the processor can't execute them in parallel anyway.

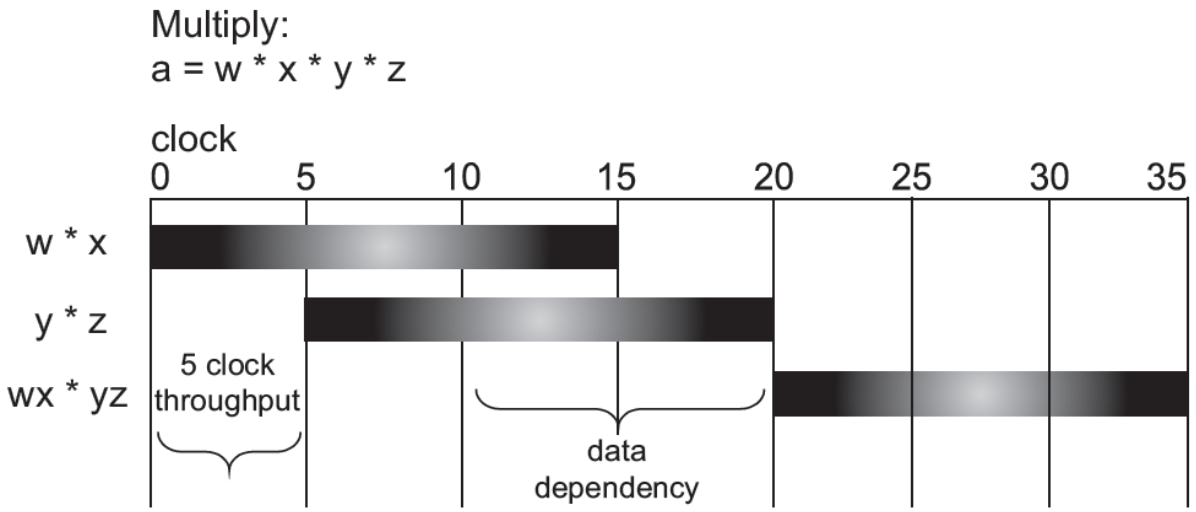

Figure 1.2 Gantt Chart of Instruction Execution with Data Dependencies

Instruction parallelism limited by data dependencies, latencies, and throughputs is a key limiting factor to algorithm performance. Additional instruction parallelism rules exist that are specific to each processor family. However, using only the data dependencies, instruction latencies, and instruction throughputs makes for a good approximation of instruction execution, and you generally can ignore any additional unique parallelism rules.

Example 1.3 Find the Greatest Common Factor of 40 and 48 the Elementary School Way

Sometimes data dependencies are hard to spot because they are hiding in loop constructs or among multiple functions.

Problem

Reduce the hidden data dependencies in the following code to improve performance.

```
a = 0;
for (x=0; x<1000; x++)
        a += buffer[x];
```

Solution

Looking only at data dependencies, the increment of the variable x and the addition of a + buffer[x] can occur at the same time because the code appears to have no data dependencies. But that misses the data dependencies that span loop iterations. A dependence exists between the computation of a in iteration *i* and the computation of a in iteration *i+1*. A better way to write the loop is to use four accumulators so that more arithmetic can occur on each clock due to fewer data dependencies.

```
a = b = c = d = 0;
for (x=0; x<1000; x+=4)
{
        a += buffer[x];
        b += buffer[x+1];
        c += buffer[x+2];
        d += buffer[x+3];
}
a = a + b + c + d;
```

Even though each iteration in this "unrolled" loop executes more instructions, the data dependencies are lower, and the total number of iterations has been reduced by a factor of four. These factors in combination allow this loop to run faster. This example shows a technique that is exactly what the Intel compiler's vectorization optimization does using the SSE2 instructions that were introduced in the Pentium 4 processor (see Chapters 12 and 13). And it does so without the need to recode the loop. A good goal is to lower data dependencies to the point where the processor is able to execute at least four or more operations at the same time.

Conclusion

The speed and effectiveness of software is determined in large part by which algorithm is employed to perform its task. The elements that determine the suitability of an algorithm include computational complexity, implementation instructions, and data dependencies. An algorithm with a faster computational time, such as a quicksort, should be chosen in lieu of one with a slower computational time, such as a bubble sort. The instructions needed to implement an algorithm should minimize both instruction latency and instruction throughput. As long as these conditions are met, one type of algorithm may make sense for a limited number of iterations of a loop, whereas another type may make sense for more iterations. Algorithms may be combined to harness the best features of each. It is also essential that an algorithm should be structured so that as many of its instructions as possible can be executed independently of one another and thus executed simultaneously. The book from which this article was sampled contains additional information on the topic, and provides a strong foundation for developers interested in learning more about multi-core processing and software optimization.

This article is based on material found in the book *The Software Optimization Cookbook, Second Edition*, by Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian. Visit the Intel Press website to learn more about this book:

http://noggin.intel.com/intelpress/categories/books/software-optimization-cookbook-second-edition

Also see our Recommended Reading List for similar topics:

http://noggin.intel.com/rr

About the Authors

Richard Gerber has worked on numerous multimedia projects, 3D libraries, and computer games for Intel. As a software engineer, he worked on the Intel VTune™ Performance Analyzer and led training sessions on optimization techniques. He is the author of The Software Optimization Cookbook.

Aart J.C. Bik holds a PhD in computer science and is a Principal Engineer at Intel Corporation, working on the development of high performance Intel® C++ and Fortran compilers. Aart received an Intel Achievement Award, the company's highest award, for making the Intel Streaming SIMD Extensions easier to use through automatic vectorization. Aart is the author of The Software Vectorization Handbook.

Kevin B. Smith is a software architect for Intel's C and FORTRAN compilers. Since 1981 he has worked on optimizing compilers for Intel 8086, 80186, i960®, Pentium®, Pentium Pro, Pentium III, Pentium 4, and Pentium M processors.

Xinmin Tian holds a PhD in computer science and leads an Intel development group working on exploiting thread-level parallelism in high-performance Intel® C++ and Fortran compilers for Intel Itanium®, IA-32, Intel® EM64T, and multi-core architectures.
