Code Optimization Techniques

Glen I. Magee
Northwestern University

Jim Klarkowski, Technical Advisor

Eric Disch, Manager

Sandia National Laboratories

Department 5721, Space Processor Engineering

August 10, 2000

Abstract:

Faced with optimizing the Reed-Solomon error correction code, it became necessary to synthesize years of code optimization theory and practice. In order to optimize the code sufficiently, the types of optimizations available were examined and ordered into a multi-stage process. As not all optimizations provide the same level of gain, and the order was decided to maximize the effectiveness of the optimizations. Because the wealth of optimization knowledge is relatively old (considering the incredible rate at which technology is advancing), many of the lessons discovered needed to be reassessed in light of current technology and the particular application. Optimization includes a variety of tasks from modifying algorithms to examining compiler switches. Proper ordering of the optimization tasks eliminates redundant optimizations and moves the greatest gains to the beginning of the optimization process. By starting with the most effective optimizations, some projects can stop early if the code is fast enough after only a couple stages of the optimization process. Optimization is tailored to individual projects, but the general optimization process is applicable to a wide range of applications and project constraints.

DISCLAIMER

Introduction:

Computers transfer data in a number of different ways. Whether through a serial port, a parallel port, over a modem, over an ethernet cable, or internally from a hard disk to memory, some data will be lost. To compensate for that loss, numerous error detection and correction algorithms have been developed. One of the most common error correction codes is the Reed-Solomon code, which is a special subset of BCH (Bose-Chaudhuri-Hocquenghem) linear cyclic block codes. In the AURA project, an unmanned aircraft sends the data it collects back to earth so it can be analyzed during flight and possible flight modifications made. To counter possible data corruption during transmission, the data is encoded using a multi-block Reed-Solomon implementation with a possibly shortened final block. In order to maximize the amount of data transmitted, it was necessary to reduce the computation time of a Reed-Solomon encoding to three percent of the processor's time. To achieve such a reduction, many code optimization techniques were employed. This paper outlines the steps taken to reduce the processing time of a Reed-Solomon encoding and the insight into modern optimization techniques gained from the experience.

Process:

The first task was to research code optimization and get a feel for the general consensus on optimization do's and don'ts. Optimization techniques have been around for years, and not all of the long held beliefs are still applicable today. Because of the boom in hardware speeds and the drop in hardware prices, many developers have let code optimization slip to the back of their minds. As a result, the rigorously developed techniques from years ago have not been updated to account for modern compiler optimizations or hardware features. Synthesizing a sizable volume of data from opposing viewpoints led to the development of a general outline to follow when optimizing code. This is the process that was followed during the optimization of the Reed-Solomon code.

Before beginning any project, it is imperative to choose a programming language. Many programmers will choose a language they are familiar with, even if it is not the best language for the project. Speed, flexibility, and ease of coding are a few of the major factors in deciding which language to use. Whatever the language, knowing it well will help reduce number of awkward constructs and otherwise improve the quality of the code. When inheriting code, the programming language is already chosen, but it may be more efficient to rewrite the code in another language than try to optimize the existing code. With all of the choices available, choosing the right language can be difficult, yet is an important part of the optimization process. The Reed-Solomon implementation for this project was written in C. This was a good choice because the GNU compiler collection (GCC) has a very good optimizing C compiler, allowing the compiler to do much of work. The availability of a good optimizing compiler (or fast command interpreter for an interpreted language) should always be considered when deciding which language to use.

Assuming the programming language is already chosen and there is little need to choose a new one, the first step towards optimization is to look at algorithm choice.

Choosing a good algorithm can be a difficult task and there is no definitive process for choosing the right one. The gains of a good algorithm, however, include a major speed increase and improved code readability and stability. Reducing an $O(N^2)$ algorithm to an O(N) algorithm will speed up the code (especially for large amounts of data) and enable later optimizations to work with more efficient code to produce even greater returns. Choosing an algorithm early in the process prevents optimizations from being performed, and then being performed again after an algorithm change. Any changes to the algorithm should be performed as early in the optimization process as possible.

After settling on an algorithm, the compiler optimization options should be enabled. This provides an idea about the final speed and size of the code. The compiler will also perform many optimizations faster and just as well as, if not better than, the human programmer will. Optimizations like moving constant expressions outside of loops, storing variables in registers, moving functions inline, and unrolling loops should be performed by the compiler in most cases. Occasionally the programmer can perform an optimization better than the compiler because he has additional knowledge about the code that the compiler lacks. For most code, however, the compiler has enough information to make good decisions and perform the proper optimizations. There are some cases where certain optimizations will hinder performance or unacceptably enlarge the code. To prevent that hindrance the programmer can specify which optimizations to include or omit by using compiler flags. There is little point in performing an optimization by hand when a compiler can perform the same optimization faster and more accurately.

If the code is still not fast enough after the compiler optimizations, there are a number of hand optimizations that can be performed. Before optimizing all the code, it is a good idea to profile the code and get a sense of where the bottlenecks are. In general, most of the code in a program will only run once, and most of the processing time is spent in an inner loop. Optimizing just that loop will reap the greatest benefits, as a single optimization will save on each run through the loop. Any good optimization book will outline basic optimization techniques, but it is good to keep in mind the capabilities of the compiler. The programmer knows many aspects of the code better than the compiler and can therefore perform some optimizations the compiler cannot. Like any other tools, compilers are not perfect so it is important to understand the specific compiler being used. As good as the compiler may be, it is foolish to rely on it to do all of the work. When done properly, utilizing a compiler's features is quicker, easier, and more effective than doing all the work by hand.

For code that needs to be extremely streamlined, assembly language is a good choice. Some programming languages, like C, allow assembly statements to be inserted directly into the code. It is also possible to write an entire section of code in assembly. For many programmers, modifying compiler-generated assembly will produce the best results in the least amount of time. Skilled assembly programmers, however, may be able to write entire blocks of assembly that will outperform compiler-generated assembly. Even so, using the compiler-generated assembly is a good way to start out and it is always possible to write the assembly from scratch if modifying the compiler-generated assembly does not yield great enough gains. It is not always a good idea to write an entire program in assembly. For code that only runs once, or for which the compiler produces good assembly, it will often be faster to use a high-level

language than to hand-code assembly. The loss in code performance may not offset the time saved by using a high-level language.

If all optimizations fail to make the code run fast enough it may be necessary to explore hardware options. Implementing the code in hardware allows faster processing than that attainable by software. Because there is a minimum number of cycles required to perform any given task, it may be necessary to use faster hardware. Ultimately there will be some project constraint imposing a limit on the speed of the code, and the solution may be difficult to find or accept.

Equipment:

The target system was a PowerPC™ 604e RISC microprocessor running VxWorks 5.4 by Wind River Systems. The code was developed on a Windows NT 4.0 (SP6?) PC using Wind River Systems' Tornado 2.0 development environment powered by version 2.7.2 of the GNU C compiler. The Reed-Solomon implementation used was by Phil Karn, September 1996. The initial optimizations to the code were made by Tad Ashlock as a proof-of-concept that a software implementation could be fast enough for the project.

Application:

The modifications to Phil Karn's implementation by Tad Ashlock increased the speed of the code by just under a factor of three. Those modifications transformed a nested loop into a single loop with a constant external array. Using two pointers, the array could be iterated upon much more efficiently than the nested loop could. Also, by moving data that remained constant outside of the loop, calculations could be performed once and referenced by a pointer instead of being calculated over and over again. After careful examination of the remaining loop, it was decided that there was no way to move any more data out of the loop or to reduce the number of calculations needed to perform the encoding.

With the algorithm comfortably decided, there was freedom to look at the compiler optimizations. The GNU C compiler has three basic levels of optimization (enabled by the `-O` flag) and several flags that can be used for fine-tuning. Turning on optimization (using the lowest level, `-O1`) improved the speed of the code dramatically. With the lowest level of optimization, the code was approximately a three and a half times faster than the code using just the new algorithm. The higher levels of optimization (`-O2` and `-O3`) slimmed down the code, but did not produce as great a change as did the first level of optimization. At this point, the code was just about as fast as it was going to get, and was ten times as fast as the original code. With the highest level of optimization, the GNU C compiler enables most of the optimization flags, and disabling individual flags did not increase the speed any. The two major flags that are not enabled by the highest level of optimization are the `-funroll-loops` and `-funroll-all-loops` flags. The first flag tells the compiler to unroll any loop (example 1) that it believes will create a speed increase. Because branching operations can be more costly then arithmetic and other basic operations, a loop may run faster in

an unrolled form. Also, if the compiler can determine how many times the loop will execute, it is unnecessary to perform comparisons during run-time to decide if the loop should stop executing. Unlike many of the other optimizations, loop unrolling always increases the size of the program file and may not be a good idea when storage space is at a premium. The size increase is why the option is not enabled with the highest optimization flag. The compiler optimizations aim to increase the speed of the code while decreasing the size of the application file, and loop unrolling increases file size. The `-funroll-all-loops` flag tells the compiler to unroll any loop it can, and will often slow down the code in addition to increasing file size. When optimizing by hand, reducing the strength of a loop (example 2) might be a reasonable alternative to complete loop unrolling. The Reed-Solomon code, with the new algorithm, benefited from `-funroll-loops`, but not `-funroll-all-loops`.

Example 1, unrolling a simple loop

```
/* original code */

unsigned int a[5];

for (int i = 0; i < 5; i++) {
    a[i] = (i + 2);
}
```

```
/* unrolled code */

unsigned int a[5];

a[0] = 2;
a[1] = 3;
a[2] = 4;
a[3] = 5;
a[4] = 6;
```

Example 2, strength reduction of a simple loop

```
/* original code */

unsigned int a[200];

for (int i = 0; i < 200; i++) {
    a[i] = (i + 2);
}
```

```
/* strength reduced code */

unsigned int a[200];

for (int i = 0; i < 200; i++) {
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2); i++;
    a[i] = (i + 2);
}
```

The hand optimizations to the C code were limited, and did not produce an appreciable gain in speed. A few of the calculations eliminated were related to the constants that define the Reed-Solomon block (specifically, `KK` and `NN` where `KK` is the amount of data and `NN` is the total block size). Instead of leaving `(KK - 1)` in the code, that calculation can be replaced by its final value (the values of `KK` and `NN` are fixed for a given implementation). While the `(KK - 1)` is more readable, its numeric equivalent executes faster. To make up for the lost readability of the code, comments were inserted showing the equivalent code fragments. The code had to be able to handle shortened blocks, which means it has to pad the existing data with zeros to make it the right size (`KK`). Instead of copying the data and adding zeros to the end, it was faster to split the main loop into two loops. One loop performed calculations on the data passed in, and the second loop performed the calculations as if the data was zero. It was possible to know where to split the loop because the length of the data was also passed into the function. The first loop would execute `DataLength` times (where `DataLength` is the amount of data passed in and `DataLength` $\leq$ `KK`) and the second loop would execute `(KK - DataLength)` times.

The final stage of optimization was to modify the compiler-generated assembly. It was while working with the assembly code that it was discovered that one of the loops performed better in loop form instead of the unrolled formed. The reason for the performance difference was that it was a tight loop in which the microprocessor could accurately predict when the branch was going to be taken. The main speed improvement gained from working with the assembly was freeing up two additional registers to store temporary values. That increase was small compared to some of the previous gains; the speed improvement was only one percent. The register holding the counter variable could have been freed by using the special purpose counter register, but loading and storing its values for comparison would have overshadowed any gains by doing so. The first freed register was gained by redefining one pointer in terms of the other. This also saved one calculation per pass in the main loop. In the original code, the second pointer was incremented and referenced each cycle; in the new code, the first pointer is referenced with an offset in place of referencing the second pointer. The other register freed is only available for part of the function. The register is the one holding the length of the data and after the last use of that value, the register can be reused. Since that coincides with the second loop derived from the main loop (to enable shortened block processing as discussed above), those calculations can be preformed slightly faster than those in the first loop. The last major change to the assembly code was eliminating a number of lines of repetitive code. While each line of code was necessary, a whole portion of code was repeated for use in a separate branch. Eliminating the extra code and rewriting the branch reduced the file size by almost ten percent, but did not improve performance.

Conclusions:

After going through each stage of optimization for the Reed-Solomon code, it is clear that there is very little correlation between the difficulty and quality of optimizations. The easiest optimization (i.e., turning on the compiler flags) had one of

the greatest benefits while the hardest (i.e., working with the assembly code) produced only a small improvement in speed and file size. The final code, however, ran ten times faster then the original, and the file was ten percent smaller. Though not quite as fast as originally hoped, the code now performs at an acceptable level. Unfortunately, when optimizing code there is a limit that makes each additional increase in speed harder to attain and less pronounced than the previous one.

When optimizing code, it is best to start with the optimizations that produce the greatest returns in the shortest amount of time and with the least amount of effort. It is inefficient and time-consuming to start with the most difficult and least rewarding optimizations. The order used in optimizing the Reed-Solomon code was effective and limited the amount of unnecessary work done. Picking a good algorithm has the greatest potential for improvement and should be done first. Even if it does not increase the speed of the code, a good algorithm will enhance the stability and readability of the code. Compiler optimizations are one of the easiest most effective optimization methods and should be performed before any hand optimizations. After letting the compiler perform its optimizations is a good time to perform hand optimizations and possibly modify the compiler-generated assembly code. If software still proves too slow, it may be necessary to examine various hardware options.

Acknowledgements:

I would like to thank Tad Ashlock for the initial optimizations to the Reed-Solomon code and getting me up and running with the equipment I needed. Thank you to Jim Klarkowski, my Project Lead and Technical Advisor, for overseeing my work and keeping me informed. Thanks also to my Manager, Eric Disch, for introducing me to the world of microprocessors and embedding computing.

Reference:

Rorabaugh, C. Britton, *Error Coding Cookbook: Practical C/C++ Routines and Recipes for Error Detection and Correction*, McGraw-Hill, New York, 1996.

PowerPC™ Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Motorola Inc., 1997.

PowerPC™ 604e: RISC Microprocessor User's Manual with Supplement for PowerPC 604 Microprocessor, Motorola Inc., 1998.

GCC Home Page: <http://www.gnu.org/software/gcc/gcc.html>

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.