Interesting Results with an Optimizing Compiler when Refactoring Embedded Code

Márcio Afonso Arimura Fialho
DEA - Divisão de Eletrônica Aeroespacial,
INPE - Instituto Nacional de Pesquisas Espaciais
São José dos Campos, SP, Brazil
maaf@dea.inpe.br

Abstract

This paper presents interesting results obtained when refactoring a piece of code compiled with an optimizing compiler. Some of the results were surprising, and at a first glance, contradictory. Most of these results can be explained by the optimizations performed by the compiler. This article concludes with lessons learned about compiler optimizations and some recommendations useful to achieve better code optimization.

1. Introduction

In embedded systems, especially in those with limited memory and computational power, considerations about performance issues and code efficiency cannot be forgotten during the development process. However excessive preoccupation with code efficiency can lead to code that is hard to understand and maintain, and may become unreliable.

This paper presents results obtained during refactoring a piece of embedded code, some of which were surprising, proving that some common assumptions made by developers during code implementation are not always true. The main motivation for refactoring was to improve code clarity and maintainability.

2. Background

In an effort to develop a Brazilian autonomous star sensor, a PC based software for testing algorithms for this star sensor was created [1]. In this software, named PTASE, many versions of two base star identification algorithms were implemented [2]. After various tests were performed and considering the target hardware characteristics, one of the best implementations of these two star identification algorithms was ported from the desktop PC environment to the target hardware, which is an embedded system.

A star sensor is an instrument normally used aboard a spacecraft to gather spacecraft attitude information. In spacecraft terminology, attitude means spatial orientation, and instruments used to gather attitude or information that can be used to calculate the spacecraft attitude are collectively known as attitude sensors. Star sensors are fine attitude sensors, capable of returning very accurate attitude measurements, with uncertainties usually in the order of few arcseconds (micro-radians) or less. Basically a star sensor takes a picture of the sky or space, extracts a list of observed stars from this image, and by comparing this list of observed stars with an internal database of stars (star catalog) is able to derive the relation between the star sensor reference frame to an inertial reference frame. This relation is the star sensor attitude, which can be very easily converted to the spacecraft attitude, since the relation between these two reference frames are known from the spacecraft assembly and alignment processes.

At the heart of the process of computing the star sensor's attitude lies the star identification algorithm (star ID algorithm, for short). This algorithm matches stars from the list of observed stars with stars in the star catalog. When a sufficient number of stars have been matched (identified), the star sensor attitude can be calculated.

The chosen star ID algorithm implementation, ported to the embedded system, had a very large main function, followed by few small helper functions in the same module. This module was ported from PTASE, which was written in C++ to the embedded environment, in plain C language. Since both languages are very similar, this migration was relatively straightforward, with the exception of some modi-

fications required due to naming conventions adopted in the embedded code and for some debugging features present in PTASE but not in the embedded hardware.

For embedded code running in spacecraft hardware, code quality is very important, due to difficulties in performing spacecraft maintenance after launch and associated high costs with a space mission. This prompted a greater care during development of the embedded code to keep the code clean.

Before refactoring, the main function of the star ID module had 628 physical lines of code (including comments and blank lines) and 247 statements. That huge size prompted splitting this function into many smaller functions during a code review, so the code would become easier to understand and maintain. This function's large size can be explained in part by lack of time for better organization during its development, and also by an attempt to prematurely optimize code in order to reduce function call overhead.

Since the overall structure of the algorithm was preserved during the code refactoring, it was expected that after refactoring (mainly "extract function" refactoring), some loss in executable code size and performance would occur, due to the expected increase in function call overhead. To check for losses and gains obtained after every refactoring step the code was instrumented to allow measurement of time spent by the code and the executable code size was closely monitored.

3. Method

Before attempting to refactor the embedded code, a copy of the corresponding star ID module in PTASE was refactored, in order to check if refactoring would be feasible, and also to provide a guideline that could be used when refactoring the embedded code itself. Refactoring was done mainly through the "extract function" technique.

The tests were performed in an embedded system using the ERC-32 single chip processor[3], a processor based on the SPARC-V7 specification, running at 12 MHz. The system had 4 MB of RAM, being accessed by the processor with zero waitstates. The code was compiled with GCC version 4.3.2 cross-compiler for sparc-rtems 4.9 [4], with optimization level -O2.

To measure the time spent by the star ID algorithm, a call to the *rtems_clock_get* function immediately before calling the star ID algorithm main function (*GiIdentifyStars*) and another immediately after, were made [5] (see Figure 1). The time difference between these two calls to *rtems_clock_get* was saved in a debug telemetry created to check the operation of the embedded code.

```
start_time = GlngGetTime();
LiRetCode = GiIdentifyStars(.....);
end_time = GlngGetTime();
```

Figure 1. Instrumenting code to measure execution time by the star ID algorithm. *GlngGetTime* is a wrapper for the RTEMS function *rtems_clock_get*.

To get comparable measurements every time the test was run after a modification, the star ID algorithm was presented always with the same list of observed stars. During the tests, no piece of code outside the star ID module was modified.

4. Results

Before refactoring, the star ID module (StarIdent.c) had 1051 physical lines of code (LOC) and 5 functions. The largest was *GiIdentifyStars* with 628 LOC (including comments, blank lines, etc) and 247 C language statements. Table 1 (below) and Table 2 show many useful software metrics gathered during module refactoring:

Table 1 – Code size and execution time during refactoring

step	file LOC	number of functions	largest function LOC	executable size (bytes)	execution time*
0**	1051	5	628	143,652	1250 ms
A	1125	8	514	143,508	1233 ms
B	1206	11	253	143,556	1158 ms
B'	1220	12	232	143,556	1158 ms
C	1269	14	160	143,620	1158 ms
D	1349	16	147	143,748	1158 ms

* measured with an 8.333 ms resolution.
** step 0 = code before refactoring.

In the first column of Table 1 there is a label for each step that allows these steps to be further referenced in the text. Step 0 refers to the code before refactoring, while step D refers to the code after refactoring is completed. The second column presents the total number of lines in the module *StarIdent.c* after each refactoring step. The third column lists the number of functions in the module. The fourth shows the count of physical lines for the largest function (including blank and comment lines). The executable

size presented in the fifth column is the size for the whole application software layer binary image. This layer is composed of 24 modules, including *StarIdent.c*, and by the RTEMS operating system. From these 143 kilobytes, *StarIdent.c* accounts for only about 5 kilobytes. The last column lists the time spent in *GiIdentifyStars* and its subroutines when processing a standard list of observed stars. These time measurements were made with an 8.3333 ms resolution.

Table 2 complements Table 1 with additional software metrics, including the number of C statements inside functions and counts of McCabe's cyclomatic complexity.

Table 2 – Software metrics during refactoring

step	max state-ments in function	state-ments inside functions	largest function MVG	module MVG	source code file size (bytes)
0**	247	309	46	75	48,241
A	205	310	33	80	50,856
B	111	322	14	87	50,369
B'	100	323	14	87	51,018
C	64	338	14	89	53,586
D	55	351	12	91	55,944

* measured with an 8.333 ms resolution.
** step 0 = code before refactoring.

The second column of table 2 presents the number of C language statements inside the function with the largest number of statements. The third column presents the summation of statements inside every function in the module. The fourth column shows the highest contribution from a single function to the overall module McCabe's cyclomatic complexity. The fifth column shows the overall module cyclomatic complexity.

The number of C language statements inside a function is a much more meaningful metric than the number of physical lines or even the number of lines of code in a function, since the number of lines of code can vary significantly due to coding style, while the number of statements is practically insensitive to the coding style used. However we also show the number of lines of code in Table 1 for completeness. Values presented in the second and third columns of Table 2 do not count empty statements, those consisting of a single semicolon.

The McCabe's cyclomatic complexity was measured with "CCCC - C and C++ Code Counter" version 3.1.4, a free software for measurement of source code related metrics [9].

4.1. Use of the *static* keyword

Much after this refactoring was performed, it was noticed that the developer had forgotten to declare two functions in this module, that doesn't require external linkage, with the 'static' keyword. In C, when used with a variable/function declared at file scope, the 'static' keyword tells the compiler that this variable or function doesn't need to have external linkage, which means that it will be visible only inside the module where it was declared. This allows further optimizations by the compiler, that would be impossible if these functions/variables had to be visible outside the module in which they were declared. But how much gain can be obtained? Table 3 (below) gives some answers:

Table 3 – Improvement with the use of the *static* keyword

SVN revision	use of the static keyword in file (module) scope	executable size (bytes)	execution time*
150	missing in two 'internal' functions	142,148	1.16 s
151	present in every 'internal' functions	141,572	1.16 s

* Measured with a 10 ms resolution.

In table 3, the column 'SVN revision' refers to the revision number when committing changes made in the software in the revision control system. The difference between revisions 150 and 151 is just the addition of the '*static*' keywords to these two functions where it was missing, an addition of only two words to the code. However this simple modification reduced the code size in 576 bytes, in a module whose total code size (after compilation) was just around 5 kilobytes. This is a huge improvement!

5. Discussion

Looking at the second column from table 1 (column 'file LOC') and the third column from table 2 (column 'max statements in function') it can be seen that as the refactoring progressed the overall source code number of lines and statements increased as the large *GiIdentifyStar* function was split into smaller functions, and even though the size of individual functions on average decreased, the total number of functions increased. At first glance, this might suggest that we have simply traded off complexity inside this large function for complexity outside functions and in the

function call hierarchy, without too much gain. However this is not the case. As that large function was split into many smaller functions, each important segment of that function became a function with clear interface. In a sense, the code became more self documenting. Added to that, comments explaining every parameter passed to these new functions were written, as required by the automatic documentation system. These comments were responsible for much of the line count increase while the file was being refactored.

As explained in the section II, one of the reasons the star ID algorithm was implemented with a very large function was to avoid function call overhead, which can be very costly in some platforms. However looking at the fifth column of table 1 we see that the executable code size fluctuates around the size it had before refactoring, sometimes increasing a bit, but at other times decreasing a little. Also, contrary to expectations, we can see in the last column, that the processing time has actually decreased after refactoring. These results suggests that somehow the compiler is avoiding these function call overheads, probably by merging functions with internal linkage, that are small or are called only once, into the caller function. This suggestion is confirmed when we look more carefully at what happened between step B and step B' during code refactoring.

The only difference between those two versions, is that when going from step B to step B' a function was extracted from the largest function in B, which had 253 lines and 111 statements. The extracted function, having 25 lines and 12 statements overall, is used by the caller function to compute an attitude estimate that is used to identify the remaining stars selected for identification. Performing a binary comparison between the executable code generated from step B with the code generated from step B', no difference was detected, which means that the object code generated by these two versions were identical. This has happened despite the fact that the extracted function had 12 statements, a function call and a local variable, and is crucial for the stellar identification algorithm.

When going from step A to step B, it was seen a big improvement in the processing time. The algorithm became around 6.5% faster. Between those two steps the code inside two nested loops was extracted as a new function. It happens that is precisely in those two nested loops that the algorithm spends most of it's time. When the code inside these loops was extracted, some variables that had scope greater than these loops, but were used only inside this loop have been moved to the new extracted function, effectively changing their scope to a smaller scope that does not involve these loops. Probably this is what allowed the compiler to perform a better code optimization. A similar gain was observed in PTASE, when the same refactoring was done in PTASE, using another compiler.

On the other hand, one should not take this refactoring technique to extremes. Having too many small functions with only one or two statements also reduces code clarity. From our experience, it seems that good code clarity is better achieved when functions have between 5 to 200 lines of code and the number of functions per module is between 5 and 20, excluding special cases.

5.1. The *static* keyword case

The huge improvement seen in section 4.1 can be explained by the fact that the two functions are very similar. One of them increments an index, while the other decrements the same index, however their structure is practically the same, to the point that the object code in one function may be essentially duplicated in the other. Thus, it seems that during the optimization allowed by the addition of the *static* keyword, the compiler noticed the strong similarity between those two functions, finding a better implementation where a single code could perform the function of both functions, provided that some variables where set up correctly at the beginning, depending on the case. With this optimization, we believe that roughly the code of one of these functions could be removed from object code. To prove this explanation, an analysis of the generated assembly code would be required. This will be left for the future.

Regarding execution time, there was no noticeable difference before and after the addition of the *static* keyword. This is due to the fact that the affected code is not in a critical section, so that any timing differences, if any, are smaller than the sensibility of our experiment.

In face of the reduced risk of name clashes that the addition of the *static* keyword brings to variables and functions declared at file scope that don't need external linkage, it's use is mandated or strongly recommended by most of the coding standards used in the aerospace and high reliability industries [6] [7].

5.2. The compiler documentation

After discovering that the code produced from step B and step B' where identical, we decided to check in the compiler documentation [10] what compiler optimization switch was responsible for merging the extracted function in B' with its caller. This extracted function is

called only in one place, and was declared with internal linkage.

The command line `-O2` optimization switch acts as a master switch that enables many optimization switches in GCC. One of these, in GCC 4.3, is the `-funit-at-a-time` switch which in turn turns the `-finline-functions-called-once`. This last switch considers for inlining every function with internal linkage that is called only once. If the call to that function is inlined by the compiler, no separate code is generated for that function.

The compiler documentation [10] warns that some optimizations may introduce compatibility issues with code that relies in assumptions that may become invalid after optimization (such as a particular ordering of variables, etc). Hence it is strongly advisable that the developer read carefully the chapter about compiler optimizations in the compiler manual if he/she is compiling code with optimizations turned on.

6. Conclusion

Compiler technology and compiler optimization techniques have improved significantly in the last decades, to the point that in many situations it has become hard to surpass code generated by a good optimizing compiler with handwritten assembly code [8].

This experiment showed some remarkable results, from where some lessons could be learned:

- The optimizing compiler used is capable of performing many intra and interprocedural optimizations, including the ability to merge functions in order to avoid function call overheads.
- These and many other optimizations performed by the compiler allow a very high performance to be achieved without the need to hand optimize code.
- Many optimizations are only possible when variables and functions are declared with the 'static' keyword. Hence, every function or variable (declared at file scope) that doesn't need external linkage should be declared with the 'static' keyword.

These results provides another argument to the recommendation that programmers should avoid optimizing code prematurely when implementing code, since this may reduce code clarity, and many optimizations that the programmer tries to perform by hand can be better performed by a good optimizing compiler. However, this doesn't mean that the programmer should completely forget about code efficiency, only that code efficiency and performance should be set as secondary goals, with safety and clarity set as primary goals [7]. Another important conclusion is that the programmer should never forget the 'static' keyword, as this oversight may significantly impair optimization, aside from increasing the risk of name clashing in the linking process.

When implementing a system where performance is critical, it's advisable first to check if the compiler is indeed able to perform these optimizations before relying on them. We have used GCC 4.3.2 which is fairly recent. Older versions of GCC and older compilers might not be so good in code optimization. Also, when compiler optimizations are being used, it is strongly recommended that the development team reads the compiler manual carefully, in order to know the implications of the optimizations performed by the compiler. This is specially true for projects with some safety criticality aspect, as ours.

In some very safety critical applications compiler optimizations are severely restricted or even completely forbidden by requirements. The results and conclusions of this study do not apply to these cases.

This work has led to many new ideas that could be better explored in future works. For example, one interesting test would be to perform the same comparison done here, but with optimizations turned off to see how the employed refactorings would affect code efficiency in this case.

As additional suggestions for future works, this experiment could be repeated with more precise time measurements (using resolution of microseconds or better), and using additional software metrics besides those we have used.

7. Acknowledgments

First, we would like to thank Omnisys Engenharia Ltda. and Wisersoft Informática companies, who have written a significant portion of the star sensor embedded software, which has served as the basis for this work.

We are also grateful for the OAR Corporation for freely making a great open source real time operating system such as RTEMS available for download, and for RTEMS contributors, for their efforts in improving RTEMS quality and reliability.

Many thanks to FINEP, who has sponsored the development of the aforementioned star sensor and also of a significant fraction of the embedded software.

We would like also to thank for everyone who in a manner or in another contributed to this work.

8. References

[1] FIALHO, Márcio Afonso Arimura. *"Ambiente de simulações e testes de algoritmos para sensores de estrelas autônomos"*. 2003. 120p. Undergraduate Thesis - Instituto Tecnológico de Aeronáutica, São José dos Campos.

[2] FIALHO, Márcio Afonso Arimura. "Estudo comparativo entre dois algoritmos de identificação de estrelas para um sensor de estrelas autônomo de campo largo". 2007. 237p. Master Thesis – Instituto Tecnológico de Aeronáutica, São José dos Campos.

[3] ATMEL Corporation. *"Low-voltage rad-hard 32-bit SPARC embedded processor TSC695FL"*. product datasheet. May 2005. (Doc. Rev. 4204C–AERO–05/05). Available online at: <http://www.atmel.com/dyn/products/product_card.asp?part_id=3187>.

[4] OAR Corporation. *GCC Cross compiler system for the SPARC-RTEMS target*. Available at: <http://www.rtems.com/ftp/pub/rtems/linux/4.9/fedora/9/i386/> retrieved in December 2009.

[5] OAR Corporation. *"RTEMS C User's Guide"* Edition 4.9.0, September 2008. Available at: <http://www.rtems.org/onlinedocs/releases/rtemsdocs-4.9.0/share/rtems/html/> retrieved in February 2011.

[6] MIRA Limited. *"MISRA-C: 2004 Guidelines for the use of the C language in critical systems."* Edition 2. Warwickshire, UK: MIRA Limited, July 2008 (ISBN 978-0-9524156-4-0)

[7] Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle C++ coding standards for the system development and demonstration program." Document Number 2RDU00001 Rev D. June 2007

[8] Byte Craft Limited. "*Proof that C can match or beat assembly.*" 2006. <http://www.phaedsys.com/principals/bytecraft/bytecraftdata/bcCversusAssemblyProof.pdf> retrieved on February 16th 2011.

[9] Littlefair, T. et al. "*CCCC - C and C++ Code Counter. A free software tool for measurement of source code related metrics by Tim Littlefair.*" 2006. <http://cccc.sourceforge.net/> and <http://sourceforge.net/projects/cccc/> retrieved on February 14th, 2011.

[10] Free Software Foundation, Inc., 'Options That Control Optimizations' in "*Using the GNU Compiler Collection. For GCC version 4.3.5*", GNU Press, Boston, 2010. pp. 77-113. Available online at: <http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc.pdf> retrieved on April 2nd, 2011.