Source Code Compression Using Procedural Abstraction

James Roberts
UVA
jpr4gc@virginia.edu

Abstract

This is the text of the abstract.

Keywords procedural abstraction, suffix tree

1. Introduction

In an age of relatively cheap memory programmers in most cases are not as burdened by the negative effects of time space tradeoffs and optimizations such as manual memory management and ”lightweight” languages can be forgone in favor of methods that save time writing programs.

A small minority will still write in languages like C where they have more control to an extent. Fields such as embedded systems may do this, but what if simply writing in C doesn't provide a small enough memory footprint between stack, heap, globals, and code? One option that will be explored in this paper is a shrinkage of the code size generated (in the .s and hopefully .o files) by identifying similar segments of code and pulling them into separate procedures. If this segment of code is used in multiple locations and is small enough then by moving it into a separate location with a generated label and replacing each instance of the segment with a call to the label the total size of the code can be reduced.

The remainder of this paper will be devoted to recording the approach, results, and problems associated with the basic approach chosen.

2. Approach

This section will be broken down into four parts that map to the four stages that the driver file executes. All of the code written for this undertaking was done in Python 2.7.2 but is hopefully fairly future-proof, the creation of assembly source was done on a 32 bit machine for ease of compilation but the preprocessor should be extenable to support 64 bit.

The first section deals with the generation of assembly source code from a C file.

The second section deals with preprocessing a plain assembly source code file for use with the shortening algorithm.

The third section deals with creating an original copy of the code for purposes of comparing size and verifying functionality.

The final section deals with the real meat of the procedural abstraction algorithm finding a suitable segment of code to abstract and replacing all occurrences of it.

2.1 Assembly Generation

A filename with the .c extension will start in this stage. GCC is called on the given source filename with the O0 and S flags passed creating an unoptimized assembly source file. The driver exits if GCC returns a non-zero value. Almost any compiler capable of creating assembly similar to GCC should be useable, GCC was chosen over the likes of clang due to the comment free assembly, which makes comparing instructions without parsing the generated file feasible.

With the .s file generated the driver moves onto the preprocessing stage.

2.2 Preprocessing

A filename with the .s extension will start in this stage. The purpose of this stage is to prepare the assembly source for the procedural abstraction function by stripping whitespace from every line, removing empty lines, removing the cfa and cfi directives, and adding comment tags for the start and end of the ”abstractable” code area as well as the previously ”abstracted” code segments.

The majority of this processing is a fairly simple exercise in the use of list filters and mapping. The only things worth noting are the strings ”.text” and ”.ident” which were used to locate the start and end of abstractable and abstracted code areas.

With the preprocessing complete the result is written to a file with a .orig.s extension indicating that it is the original preprocessed source.

The driver then moves on to the copy stage.

2.3 Original Copy

A filename with the .orig.s extension will start in this stage. This stage simply creates a copy of the file with a .pp.s extension which is the file to be updated by the procedural abstraction function. The existence of separate ”original” and ”current” files makes it easier to compare the size and functionality of the completely abstracted source.

With the .pp.s file created the driver moves on to the procedural abstraction stage.

2.4 Procedural Abstraction

3. Results

Did it end up working?

4. Problems

What crippled this approach?

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] Stefan Schaeckeler and Weijia Shang. 2009. Procedural Abstraction with Reverse Prefix Trees. In Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’09). IEEE Computer Society, Washington, DC, USA, 243-253. DOI=10.1109/CGO.2009.25 http://dx.doi.org/10.1109/CGO.2009.25