Source Code Compression Using Procedural Abstraction

James ”Jim” 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 extend-able to support 64 bit (as of now code created on a 64 bit machine will not function correctly).

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.

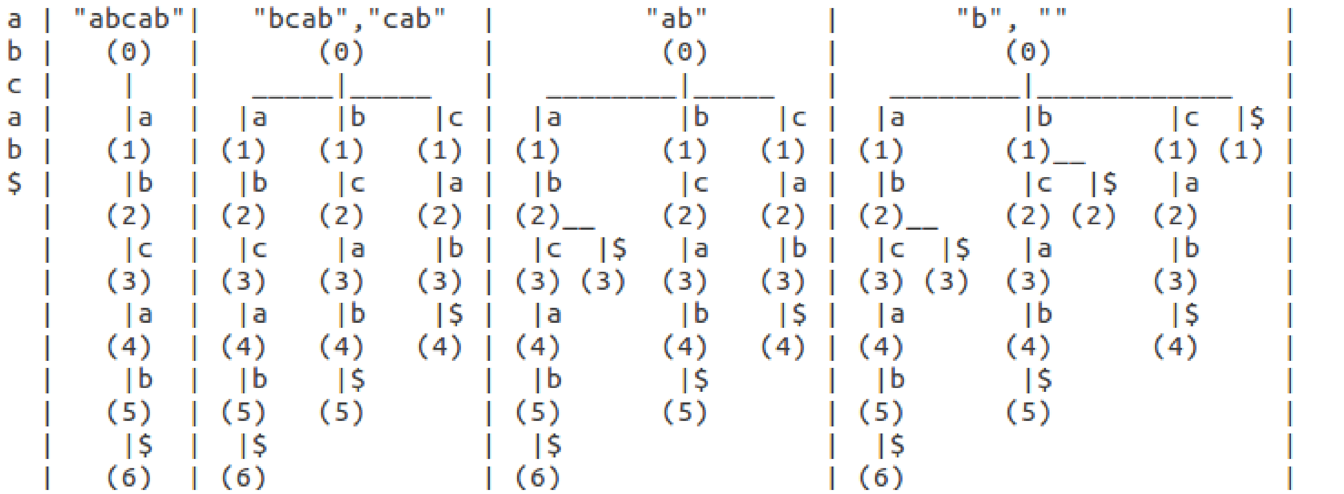
With the .pp.s file created the driver moves on to the procedural abstraction stage.

2.4 Procedural Abstraction

A filename with the .pp.s extension will start in this stage.

This is the stage where the heavy lifting is done to locate a suitable segment, generate a function and replace they occurences of the segment with a call to the generated function. In order to achieve all this the abstractable section of the current code will be isolated and turned into a suffix tree. The resulting structure will then be searched to find a segment with multiple children and of suitable length. If an abstractable segment can be found then each occurence of the segment will be replaced with a call to a function generated by the segment. Each time the driver is run at most one segment will be abstracted, source can be ”fully” abstracted by either running the driver on the .pp.s file until ”No changes made” is indicated or passing the -f flag.

This section will start with a note on what can be successfully abstracted in this simple algorithm described by [1]. A brief description of the suffix tree data structure used to locate viable code segments will then be provided. Then the function used find a good segment will described. Finally the replacement and actual abstract function creation will be touched on.



```
a | "abcab"|   "bcab","cab"  |         "ab"       |       "b", ""             |
b |   (0)  |       (0)       |          (0)       |          (0)              |
c |    |   |   _____|_____   |   ________|_____   |   ________|____________   |
a |    |a  |  |a    |b    |c |  |a       |b    |c |  |a       |b       |c  |$ |
b |   (1)  | (1)   (1)   (1) | (1)      (1)   (1) | (1)      (1)__    (1) (1) |
$ |    |b  |  |b    |c    |a |  |b       |c    |a |  |b       |c  |$   |a     |
  |   (2)  | (2)   (2)   (2) | (2)__    (2)   (2) | (2)__    (2) (2)  (2)     |
  |    |c  |  |c    |a    |b |  |c  |$   |a    |b |  |c  |$   |a       |b     |
  |   (3)  | (3)   (3)   (3) | (3) (3)  (3)   (3) | (3) (3)  (3)      (3)     |
  |    |a  |  |a    |b    |$ |  |a       |b    |$ |  |a       |b       |$     |
  |   (4)  | (4)   (4)   (4) | (4)      (4)   (4) | (4)      (4)      (4)     |
  |    |b  |  |b    |$       |  |b       |$       |  |b       |$              |
  |   (5)  | (5)   (5)       | (5)      (5)       | (5)      (5)              |
  |    |$  |  |$             |  |$                |  |$                       |
  |   (6)  | (6)             | (6)                | (6)                       |
```

Figure 1. Step through the creation of a suffix tree for the string ”abcab”. Each node is annotated with its depth and each edge is annotated by the character mapping between the nodes. Note the implicit end of string indicator $.

2.4.1 What can be Abstracted

In order to start this section a basic understanding of the C calling convention and the call/ret instructions is required. The call instruction is an unconditional jump that pushes the return address (Program Counter + 4) onto the stack and then updates the PC to the target of the call. The ret instruction is also an unconditional jump that gets its target by popping a word off the stack, which should be the location pushed by call, and updating the PC to the popped word. Because of the updates to the stack that take place during call, any instructions that use stack relative addressing or which may modify the stack (any instruction where the stack pointer is an operand) cannot be abstracted as the locations accessed would all be off by one word.

It follows from the same logic that segments with a call or ret instruction may not be abstracted because the return address could mistakenly become a parameter to the procedure call and because the wrong return address may be used if the ret instruction occurs too early in the abstracted segment.

As an extension of dissalowing the call or ret instruction to be abstracted, jumps and labels are not allowed because otherwise other code could jump into code and execute a ret instruction with the wrong return address on the stack.

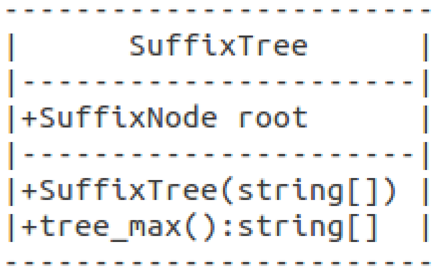
In essence the only code that can be abstracted are basic blocks with no stack based addressing or instructions. A list of the instructions to be avoided can be found in Appendix 1.

2.4.2 Suffix Tree

A suffix tree is a data structure that records all the possible suffixes (strings occurring after the value of a node). It is constructed by iterating over every instruction in the abstractable code and creating an edge between that instruction and the root from the node (which is created if the instruction did not already have an edge, or selected if the edge already exists) all of the following instructions are added treating the node as the root before starting again at the instruction after the original one.

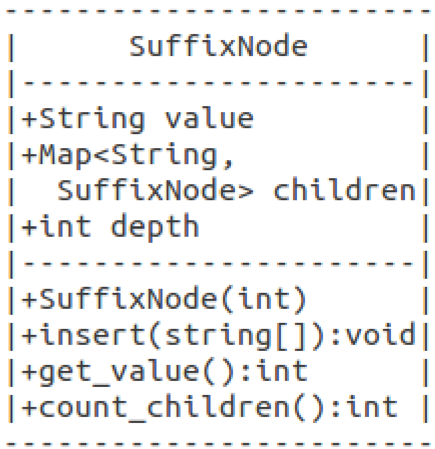
Figure 1 illustrates this procecss.

First the whole string ”abcab” is added with each letter getting a new edge. Then ”bcab” and ”cab” are added. As all three of the strings so far began with different letters, each gets its own new edge from the root. For the string ”ab” however the root node already has an edge for ’a’ so the existing node becomes the root.



```
 ----------------------
|      SuffixTree      |
|----------------------|
|+SuffixNode root      |
|----------------------|
|+SuffixTree(string[]) |
|+tree_max():string[]  |
 ----------------------
```

Figure 2. High level overview of SuffixTree class.



```
 ----------------------
|      SuffixNode      |
|----------------------|
|+String value         |
|+Map<String,          |
|  SuffixNode> children|
|+int depth            |
|----------------------|
|+SuffixNode(int)      |
|+insert(string[]):void|
|+get_value():int      |
|+count_children():int |
 ----------------------
```

Figure 3. High level overview of SuffixNode class.

Following this algorithm to completion will generate a complete suffix tree [2].

To implement this functionality two classes were created SuffixTree and SuffixNode each represented by the UML diagrams in Figures 2 and 3.

The SuffixTree constructor behaves similarly to the one described in [2], creating an empty root node, and iterating over every sublist(i) for i from 0 to the length of the list, calling insert on root with the sublist.

The SuffixNode constructor is also fairly basic taking the depth of the node and initializing the children map. SuffixNode insert takes the remaining list of instructions, stores the head as the value of the node, then finds the child node to insert the rest of the code list in. The child node will be found in the children mapping, or a new SuffixNode will be created with (depth+1) and added to the children mapping. The SuffixNode count_children method recursively finds the number of ”branches” made after this node, this provides a rough estimate of the number of times this segment could be replaced in the code which can be used as a metric to compare nodes.

Once the suffix tree has been constructed, the duplicated segments of code can be found.

2.4.3 Find Segment

Inevitably some instructions will be repeated throughout the code, ret or push %ebp for example. The goal of procedural abstraction though is to find repeated segments that are longer and/or appear more frequently in the source. This insight will tailor the way that a segment is deemed to be suitable. This manifests itself in the SuffixNode get_value() method.

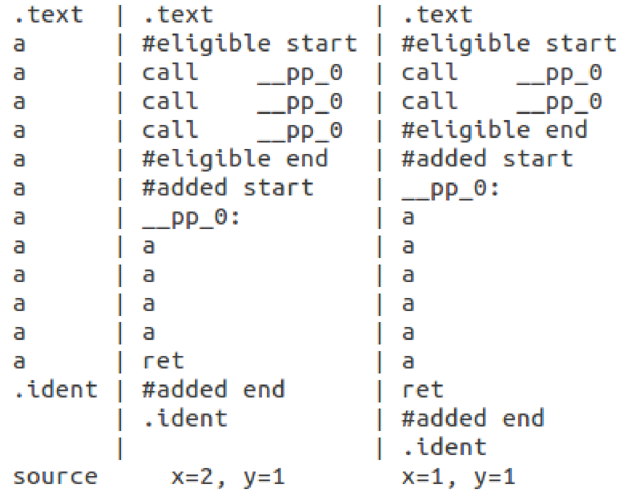
The get_value() method provides a way to compare nodes which is necessary to find a single segment to abstract each time the driver is run. A value function of the form $V(c, d)$ is used where V is the value, c is the number of children this node will have and d is the depth of the node(which is equivalent the number of instructions that will be replaced). Tweaking the value function causes the program to pick different segments to abstract. Below are two general functions which were tried.

The first function used was $V(c, d) = c * (d - 1) - (d + 2)$. This function estimates the number of lines that will be saved by abstracting this segment. The logic is that $\Delta lines = numReplacements * \Delta codeRemoved - sizeOfProcedure$. In other words by calculating the ammount of code removed ($d - 1$ keeping in mind that each segment is replaced with 1 call instruction) by each location where the segment occurs ($*c$) and subtracting the extra space required to replicate the segment in the abstracted section of code ($d + 2$ for the extra label and ret instruction required) the net change in lines can be calculated. Note that this is only an estimate whose flaw can be demonstrated with a trivial example of replacing ”aa” in the string ”aaa” where it may appear to the program as it is traversing that it can be done twice due to the number of children created, but in reality the replacement can only be done once

The second suggestion for calculating a value for the node is the formula $V(c, d) = c^x d^y$. This function is designed to select segments that either appear more frequently in the code or those which are longer. The variables x and y can be changed in order to weight the number of replacements (x) or the number of instructions replaced (y) more heavily. The effects of increasing one can be seen in Figure 4.

With a completed value method the SuffixTree tree_max method can be implemented. The actual tree_max method is just a wrapper that tracks the best code segment found so far and its value, and calls a recursive search function on the root node.

The search function is a recursive function that accesses every node in the SuffixTree and gets its value, it takes a node and the code list so far as parameters. Each time the search function is called on a non empty node first the length of the code is compared to 2 and the value of the node is compared to the best value so far. If the length of the code is greater than or equal to 2 and the value of the current node is greater than the best value so far then a copy of the code is stored and the best value is updated.



```
.text  | .text             | .text
a      | #eligible start   | #eligible start
a      | call     __pp_0   | call     __pp_0
a      | call     __pp_0   | call     __pp_0
a      | call     __pp_0   | #eligible end
a      | #eligible end     | #added start
a      | #added start      | __pp_0:
a      | __pp_0:           | a
a      | a                 | a
a      | a                 | a
a      | a                 | a
a      | a                 | a
a      | ret               | a
.ident | #added end        | ret
       | .ident            | #added end
       |                   | .ident
source     x=2, y=1          x=1, y=1
```

Figure 4. The effects of weighting the number of replacements more heavily.

Observant readers may wonder why a value of 2 is specified. This number is derived from the way that a code segment is abstracted where each occurence of a segment is replaced by a call instruction, and the segment is surrounded by a label and a ret instruction. If segments of length 1 were replaced then no matter how many occurences of the instruction there were the size of the code would increase because of the 3 line overhead. If segments of length 2 are replaced then each replacement reduces the code size by 1 so 4 replacements are required to break even. Without loss of generality segments longer than 2 instructions can also be replaced with enough matches.

After the optional update stage all of the keys in the current node's children map are iterated over and the search code is called again on the values for each key with the current instruction appended to the list of instructions. There is annother important condition that is checked before calling the search function on each value which is that the value must not contain any of the instructions/symbols which would alter/rely on the stack or cause the segment not to be a basic block. Without this restriction a segment may be selected which should be unabstractable.

When the traversal has finished a single segment remains and is used as the basis for the replacement procedure. If no segment is found then the original program is returned and the driver will terminate.

2.5 Procedure Generation

With an abstractable segment identified the abstract procedure must be generated. This procedure requires 3 parts, a prologue, the code segment, and an epilogue.

The prologue is simply a unique label, it can be generated by taking a prefix and appending to it a number and a colon. The number to append can be found by counting all ocurances of colons (indicating labels) in the previously abstracted portion of the program and adding 1. Because all of the abstracted procedures are basic blocks no other colons will be present in the code.

The code segment is the one found previously.

The epilogue is simply a ret instruction as that will return control to the location where the abstracted procedure was called.

With the procedure generated the segment can then be replaced.

	lines .s	bytes .o	lines .s	bytes .o	lines .s	bytes .o	lines .s	bytes .o
clock.orig.s	354	5343	354	5343	354	5343	354	5343
clock.pp.s	352	5285	352	5261	351	5226	352	5261
double.orig.s	51	694	51	694	51	694	51	694
double.pp.s	51	694	51	694	51	694	51	694
fcyc.orig.s	507	8053	507	8053	507	8053	507	8053
fcyc.pp.s	496	7732	495	7653	494	7627	495	7653
Fibonacci.orig.s	448	7619	448	7619	448	7619	448	7619
Fibonacci.pp.s	442	7469	441	7388	440	7320	442	7442
play.orig.s	254	3530	254	3530	254	3530	254	3530
play.pp.s	254	3530	254	3530	254	3506	254	3530
replicated.orig.s	75	1069	75	1069	75	1069	75	1069
replicated.pp.s	75	1069	70	947	70	947	70	947
rosetta.orig.s	867	13565	867	13565	867	13565	867	13565
rosetta.pp.s	850	13119	849	12979	848	12960	849	12979
Max reduction	17	446	18	586	19	605	18	586
Total reduction	36	975	44	1421	48	1593	43	1367
Mean reduction	5.142	139.285	6.285	203.000	6.857	227.571	6.142	195.285
Value Function	line estimate		d^1c^1		d^2c^1		d^1c^2	

Table 1. Resulting line counts and size in bytes of the original and abstracted files. The files were run through the driver until no segments of code could be abstracted (using the -f flag). Note that the absence of a decrease in line count does not mean that no procedures were abstracted or that the generated object file was not smaller.

2.5.1 Segment Replacement

With a suitable segment found the program shifts to replacing all possible occurrences of the segment in the abstractable code list.

A simple loop is used iterating over each index in the abstractable code list stopping when there are more instructions in the segment then are remaining in the abstractable code list. At each instruction in this loop the next instructions are compared to the segment and if the next d instructions match then the d instructions are replaced with a single call to the temporary function whose label has been calculated already.

Once this loop has terminated the abstractable code will have had all occurences of the segment removed and replaced with a call to an abstracted procedure.

The driver can then write the new .pp.s file to disk.

3. Results

The algorithm successfully shortens code although not necessarily by large amounts.

Some example results are shown above in Table 1 for seven files abstracted to completion. As shown the largest decrease in absolute size was by about 19 lines/605 bytes and the largest relative decrease in size in 6.66% in lines/11.41% in bytes for a file designed to perform replicated operations. One out of the seven files had no reduction in size at all, and three out of the six files with a reduction were reduced by less than 10 lines.

For the files tested using the d^2c^1 value function yielded the best results across the board in terms of both line count and object file size with an average line reduction of 6 and an average byte reduction of 227. It also had an interesting result of reducing the object file size while maintaining the same line count in the play.c source. This highlights another problem with the value functions where only line counts were considered and not the actual size of instructions.

While the algorithm was successful in the sense that it does what it was designed to do for a majority of the files, its usefulness is limited by the small decreases in code size that it is able to obtain.

4. Problems

At this point a reader might be wondering where everything went wrong, the algorithm seems so simple, everything makes sense, etc. but it turns out that some fairly cripleing assumptions were made along the way in order to achieve simplicity.

The first and probably most limiting assumption made was the list of instructions and symbols that could not be abstracted. As mentioned earlier this limitation was imposed in order to ensure that all abstracted functions were essentially basic blocks with no instructions that used the stack pointer.

5. Possible Improvements

6. Conclusion

A. Appendix Title

A.1 Instructions/Symbols to Avoid Abstracting

”:”, ”jo”, ”js”, ”je”, ”jz”, ”jb”, ”jc”, ”ja”, ”jl”, ”jg”, ”jp”, ”jno”, ”jne”, ”jnz”, ”jnb”, ”jae”, ”jnc”, ”jbe”, ”jna”, ”jge”, ”jnl”, ”jle”, ”jng”, ”jpe”, ”jnp”, ”jpo”, ”pop”, ”ret”, ”esp”, ”rsp”, ”jnae”, ”jnbe”, ”jnge”, ”jnle”, ”jcxz”, ”call”, ”push”, ”jecxz” List of jump instructions from [3]

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

[2] Gayle Laakmann McDowell. 2013. Cracking the Coding Interview 5th ed. CareerCup , LLC, Palo Alto, CA, USA

[3] Intel. 2014. Intel 64 and IA-32 Architectures Software Developer's Manual Colume 2(2A, 2B, & 2C): Instruction set Reference, A-Z. Intel, Santa Clara, CA, USA http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf