Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Code Factoring in GCC

Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes
Department of Software Engineering
Institute of Informatics
University of Szeged, Hungary
{loki,akiss,jasy,beszedes}@inf.u-szeged.hu

Abstract

Though compilers usually focus on optimizing for performance, the size of the generated code has only received attention recently. On general desktop systems the code size is not the biggest concern, but on devices with a limited storage capacity compilers should strive for as small a code as possible. GCC already contains some very useful algorithms for optimizing code size, but code factoring – a very powerful approach to reducing code size – has not been implemented yet in GCC. In this paper we will provide an overview of the possibilities of using code factoring in GCC. Two code factoring algorithms have been implemented so far. These algorithms, using CSiBE as a benchmark, produced a maximum of 27% in code size reduction and an average of 3%.

1 Introduction

In the recent years handheld devices such as PDAs, telephones and smartphones are becoming more important. With these systems the amount of runtime memory and storage capacity is often very limited but at the same time the need for more sophisticated software is increasing. Hence powerful size reducing methods are required to cram new features into the applications.

Although GCC already contains size reducing algorithms, further optimization techniques are needed since GCC is already used for compiling for handheld devices. The official compiler for the increasingly popular Symbian OS-based mobile phones is GCC [8], some PDAs like the iPAQs already have Linux ports [9] (where, needless to say, the default compiler is GCC) and Linux-based mobile phones are also available.

In this paper we will provide an overview on code factoring, a class of powerful optimization techniques for code size reduction, and present a new, enhanced algorithm for procedural abstraction. These algorithms have been implemented in GCC and have resulted in 3% code size reductions on average, while achieving a 27% reduction in the best cases, based on the CSiBE benchmark [5].

The rest of the paper is organized as follows. In Section 2 we introduce code factoring and present a new enhancement for procedural abstraction. In Section 3 we discuss some details of the implementation of the algorithms in GCC, while in Section 4 we give our experimental results. Finally, in Section 5 we present our conclusions and future plans.

2 Code Factoring

Code factoring is the name of a class of useful optimization techniques developed explicitly for code size reduction [1, 2, 3, 4]. These approaches aim to reduce size by restructuring the code. The following subsections will discuss two code factoring algorithms, one of which works with individual instructions, while the other handles longer instructions sequences.

2.1 Local Factoring

The optimization strategy of local factoring (also known as local code motion, code hoisting and code sinking) is to move identical instructions from basic blocks to their common predecessor or successor, if they have any. The semantics of the program have to be preserved of course, thus only those instructions which neither invalidate any existing dependences nor introduce new ones may be moved. Figure 1a shows a control-flow graph (CFG) with basic blocks containing identical instructions. To obtain the best size reduction some of the instructions are moved upwards to the common predecessor, while some are moved downwards to the common successor. Figure 1b shows the result of the transformation.

Let us now consider some more complicated cases. While not frequent, it may occur that multiple basic blocks have more than one predecessors, all of which are common. In this case, if the basic blocks in question have identical instructions and the number of predecessors is less than the number of the examined blocks, then the instructions shall be moved to all the predecessors. Figure 2 depicts this case. A similar situation is when basic blocks have more than one common successors (see Figure 3.) Furthermore, in the case of sinking even those instructions that are not present in all of the blocks may be moved by creating a new

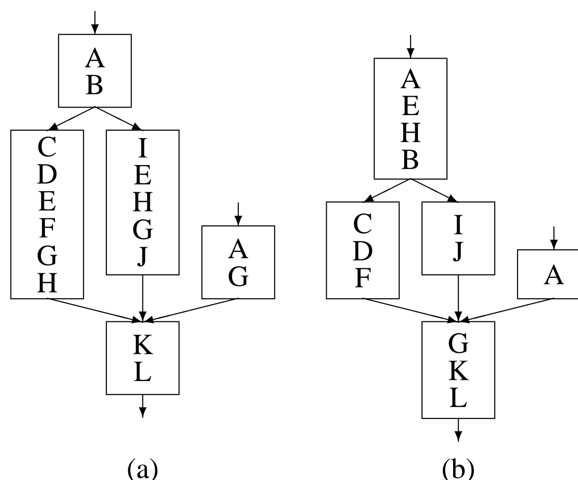

Figure 1: Local code factoring. CFG (a) before and (b) after the transformation. Identical letters denote identical instructions.

successor block for them. Figure 4 shows an example CFG for this case.

Except for this last case, which involves the creation of a new basic block, local factoring has an additional benefit of being good for performance also.

2.2 Procedural Abstraction

Procedural abstraction is a size optimization method which, unlike local factoring, works with whole single-entry single-exit code fragments (instruction sequences smaller than a basic block, whole blocks or even larger units) instead of single instructions. The main idea of this technique is to find identical regions of code, which can be turned into procedures, and then replace all occurrences with calls to the newly created subroutine.

The existing solutions [2, 4] can only deal with such code fragments that are either identical or equivalent in some sense or can be transformed somehow (e.g. by means of register renaming) to an equivalent form. However, these approaches fail to find an optimal solution for

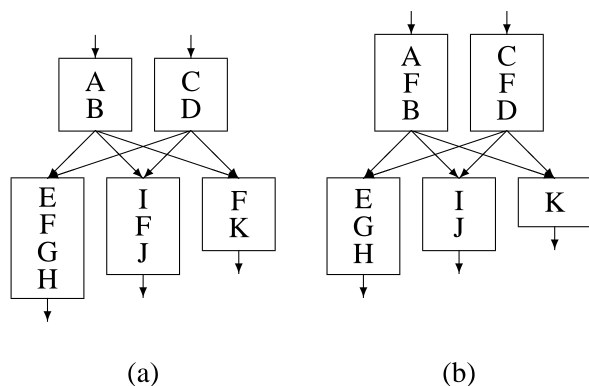

Figure 2: Basic blocks with multiple common predecessors (a) before and (b) after local factoring.

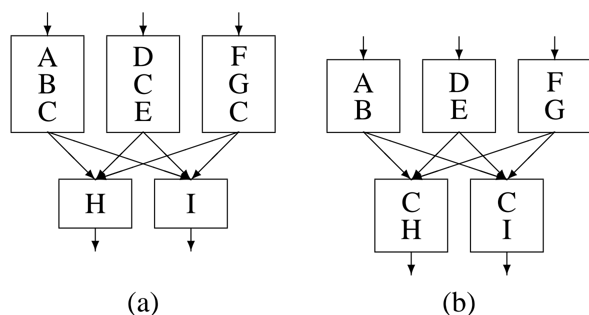

Figure 3: Basic blocks with multiple common successors (a) before and (b) after local factoring.

those cases where an instruction sequence is equivalent to another one, while a third one is only identical with its suffix (as shown in Figure 5a). The current solutions either choose to abstract the longest possible sequence into a function and leave the shorter one unabstracted (Figure 5b) or turn the instructions common in all sequences into a function and create another new function from the remaining common part of the longer sequences, thus introducing the overhead of the inserted extra call/return code (Figure 5c).

In this paper we propose to create multiple-entry functions in the cases described above to allow the abstraction of instruction sequences

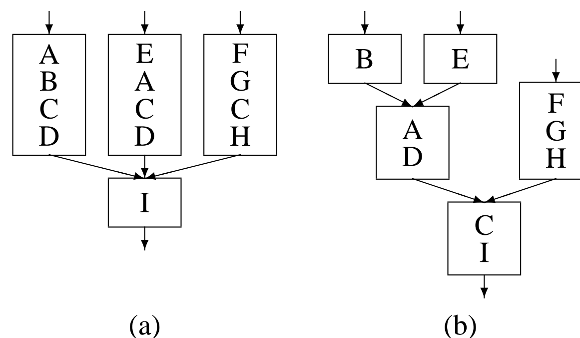

Figure 4: Basic blocks with multiple common successors but only partially common instructions (a) before and (b) after local factoring.

of differing lengths without the overhead of superfluous call/return code. The longest possible sequence shall be chosen as the body of the new function and entry points need to be defined according to the length of the matching sequences. Each matching sequence has to be replaced with a call to the appropriate entry point of the new function. Figure 5d shows the optimal solution for the problem depicted in Figure 5a.

Needless to say, procedural abstraction introduces some performance overhead with the execution of the inserted call and return code. Moreover, the size overhead of the inserted code must also be taken into account. The abstraction shall only be carried out if the gain resulting from the elimination of duplicates exceeds the loss arising from the insertion of extra instructions.

3 Implementation details

GCC already contains some algorithms similar to those discussed in Section 2, but they usually reduce code size only if the transformation does not introduce a (significant) performance overhead. Furthermore, they are usually of less potential than the previously described ones. The *cross-jumping* algorithm merges identical

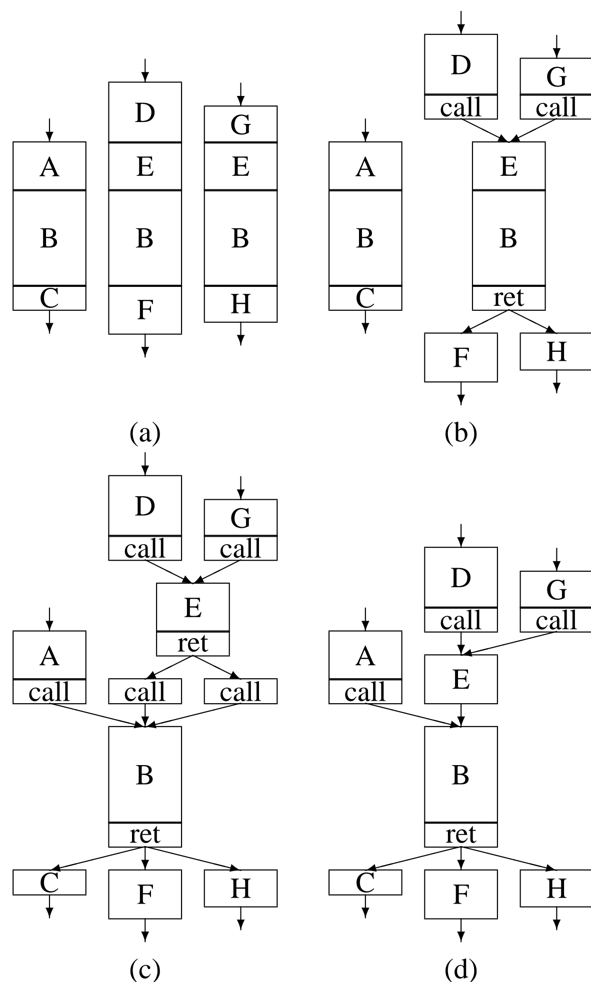

Figure 5: Abstraction of (a) instruction sequences of differing lengths to procedures using different strategies (b,c,d). Identical letters denote identical sequences.

tails of basic blocks, but this approach can only deal with a very limited subset of the generic problems of procedural abstraction. Another algorithm, called *if conversion*, has a similar effect on the code as local factoring when followed by a *combine* phase. As contrast to local factoring, *if conversion* is bound to conditional jumps.

Both of the new algorithms have been implemented as new RTL optimization phases in GCC (a snapshot taken from mainline on 2004-03-10 12:00:00 UTC). Using the RTL representation algorithms can optimize only one function at a time. Although procedural abstraction is inherently an interprocedural optimization technique, it can be adapted to intraprocedural operation. Instead of creating a new function from the identical code fragments, one representative instance of them has to be retained in the body of the processed function and all the other occurrences will be replaced by code transferring control to the retained instance. To preserve the semantics of the original program, however, the point where control has to be returned after the execution of the retained instance must be remembered somehow, thus the subroutine call/return mechanism has to be mimed. In the current implementation we use labels to mark the return addresses, registers to store references to them and jumps on registers to transfer control back to the "callers."

Unfortunately, the current implementation of the enhanced procedural abstraction algorithm suffers from the problem of increasing the compilation time by a factor of 2–4 on average. This stems from the complex problem of finding the optimal candidates for abstraction. However, we hope that by applying more efficient algorithms we will be able to bring down the compilation time factor to a manageable level.

For the sake of simplicity, local factoring has been split into two parts and implemented in GCC as two individual algorithms. One of the algorithms implements the hoisting of instructions, i.e. moving them upwards to their predecessor blocks, while the other one is responsible for the sinking of the instructions, that is move them downwards to their successor basic blocks. A central problem for both algorithms is to decide whether an instruction may be moved freely out from its block. An instruction cannot be moved across instructions, which use parameters defined by the instruc-

tion or define parameters used or defined by the instruction. GCC provides methods for gathering the required definition/use information for the whole processed function. However, from a local factoring point of view, these methods are too expensive since only a small portion of the computed information is used. Therefore the implementation contains a "slim" version of the definition/use calculation code. Being sensitive to the compilation time in the implementation, we also made it possible to parameterize the maximum number of instructions the algorithms should analyze starting from the top or bottom of the basic blocks when looking for candidates of motion.

The implementation of the two algorithms are publicly available. They have been sent in form of patches to the appropriate mailing list [6, 7].

4 Results

On examining code size we found the code factoring algorithms had impressive effects. We evaluated the discussed algorithms with the help of CSiBE, the GCC Code Size Benchmark Environment, version 1.1.1, and found that a 3% code-size reduction can be achieved on average, but in some cases they are able to produce reduction ratios as high as 27%. Table 1 details the average code size reduction achieved by each algorithms on some relevant targets. The table also shows the combined effect of the techniques. The figures are relative to the unmodified GCC optimizing for size, i.e. optimizing with `-Os`. Table 2 shows the best figures for each algorithm.

5 Conclusion and future plans

In this paper we gave an overview of two code factoring algorithms and provided an enhancement to procedural abstraction, which provides superior results compared to the existing solutions. We implemented the discussed algorithms in GCC and achieved a 3% code-size reduction on average, based on the CSiBE benchmark. In the best cases the optimizations yielded reduction ratios as high as 27%.

From the nature of procedural abstraction it follows that it can optimize larger inputs better than small ones. To be able to utilize the full potential of the algorithm the current implementation has to be modified so that it can work interprocedurally, which means a unit-at-a-time in GCC terminology instead of working intraprocedurally, i.e. transforming only one function at a time. This may necessitate rewriting the implementation so it can work on the GIMPLE representation, as some feedback already suggested. We are also aware of the algorithm complexity problem and have been striving to improve the implementation in order to reduce the compilation time by applying more efficient algorithms.

We are already investigating the possibility of making the local factoring implementation

Target	Local Factoring	Procedural Abstraction	Combined
arm-elf	0.148%	2.785%	3.120%
i386-elf	0.701%	1.356%	2.052%
i686-linux	0.696%	1.448%	2.143%
m68k-elf	0.092%	2.312%	2.401%

Table 1: Average code size reduction achieved by code factoring algorithms.

Target	Local Factoring	Procedural Abstraction	Combined
arm-elf	3.794%	27.230%	27.342%
i386-elf	14.621%	13.210%	16.795%
i686-linux	11.592%	13.261%	17.389%
m68k-elf	1.468%	23.174%	23.174%

Table 2: Maximum code size reduction achieved by code factoring algorithms.

work on GIMPLE also, even if the algorithm cannot be extended to work interprocedurally, since GIMPLE is now preferred over RTL. Our preliminary results are very promising.

When we have finished with our ongoing research, we also plan to consider the adaptation and implementation of other algorithms in GCC such as the procedural abstraction of single-entry single-exit regions larger than a basic block or the compaction of matching single-entry multiple-exit regions.

References

[1] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. Code compaction of matching single-entry multiple-exit regions. In *Proc. 10th Annual International Static Analysis Symposium*, pages 401–417, June 2003.

[2] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[3] Bjorn de Sutter, Bruno de Bus, Koen de Bosschere, and Saumya Debray. Combining global code and data compaction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.

[4] Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[5] Department of Software Engineering, University of Szeged. GCC code-size benchmark environment (CSiBE). `http://www.inf.u-szeged.hu/CSiBE`.

[6] Department of Software Engineering, University of Szeged. [patch] Local factoring algorithms. `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg01907.html`.

[7] Department of Software Engineering, University of Szeged. [patch] Sequence abstraction. `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg01921.html`.

[8] Symbian Ltd. Symbian OS. `http://www.symbian.com`.

[9] The Familiar Project. Familiar distribution. `http://familiar.handhelds.org`.