Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 79-95

CODE FACTORING IN GCC ON DIFFERENT INTERMEDIATE LANGUAGES

Cs. Nagy, G. Lóki, Á. Beszédes and T. Gyimóthy

(Szeged, Hungary)

Abstract. Today as handheld devices (smart phones, PDAs, etc.) are becoming increasingly popular, storage capacity becomes more and more important. One way to increase capacity is to optimize static executables on the device. This resulted that code-size optimization gets bigger attention nowadays and new techniques are observed, like code factoring which is still under research.

Although GNU GCC is the most common compiler in the open source community and has many implemented algorithms for code-size optimization, the compiler is still weak in these methods, which can be turned on using the '-Os' flag. In this article we would like to give an overview on implementation of different code factoring algorithms (local factoring, sequence abstraction, interprocedural abstraction) on the IPA, Tree, Tree SSA and RTL passes of GCC.

The correctness of the implementation was checked, and the results were measured on different architectures with GCC's official Code-Size Benchmark Environment (CSiBE) as a real-world system. These results showed that on the ARM architecture we could achieve 61.53% maximum and 2.58% average extra code-size saving compared to the '-Os' flag of GCC.

1. Introduction

GCC (GNU Compiler Collection) [1] is a compiler with a set of front ends and back ends for different languages and architectures. It is part of the GNU project and it is free software distributed by the Free Software Foundation (FSF) under

the GNU General Public License (GNU GPL) and GNU Lesser General Public License (GNU LGPL). As the official compiler of Linux, BSDs, Mac OS X, Symbian OS and many other operating systems, GCC is the most common and most popular compilation tool used by developers. It supports many architectures and it is widely used for mobile phones and other handheld devices, too. When compiling a software for devices like mobile phones, pocket PC's and routers where the storage capacity is limited, a very important feature of the compiler is being able to provide the smallest binary code when possible. GCC already contains code size reducing algorithms, but since in special cases the amount of saved free space may be very important, further optimization techniques may be very useful and may affect our everyday life.

One new technique is code factoring, which is still under research. Developers recognized the power in these methods and nowadays several applications use these algorithms for optimization purposes. One of these real-life applications is called 'The Squeeze Project' maintained by Saumya Debray [2], which was one of the first projects using this technique. Another application is called *aiPop* (Automatic Code Compaction software) which is a commercial program released by the *AbsInt Angewandte Informatik GmbH* with 'Functional abstraction (reverse inlining) for common *basic blocks*' feature [3]. This application is an optimizer suite software with support for C16x/ST10, HC08 and ARM architectures and it is used by SIEMENS as well.

In this paper, we will give an overview of the code factoring algorithms on the different *intermediate representation language*s (IL) of GCC. The main idea of this approach was introduced on the *GCC Summit* in 2004 [4], and since these techniques represented a class of new generation in code size optimization, we thought that the implementation on higher level IL should lead to better results. We tested our implementation and measured the new results on CSiBE [5], which is the official Code-Size Benchmark Environment of GCC containing about 18 projects with roughly 51 MB size of source code.

The paper is organized as follows. Section 2 gives an overview of code factoring algorithms and the optimization structure of GCC. Sections 3 and 4 introduce local code motion and procedural abstraction with subsections for corresponding ILs. Finally, Section 5 contains the results we reached with our implementation, and Section 6 presents our conclusion.

2. Overview

Code factoring is a class of useful optimization techniques developed especially for code size reduction. These approaches aim to reduce size by restructuring the

code. One possible factoring method is to do small changes on local parts of the source. This is called local factoring. One other possible way is to abstract parts of the code and separate them in new blocks or functions. This technique is called sequence abstraction or functional abstraction. Both cases can work on different representations and can be specialized for those languages.

GCC offers several *intermediate languages* for optimization methods (Figure 2). Currently, most of the optimizing transformations operate on the *Register Transfer Language* (RTL) [6], which is a very low level language where the instructions are described one by one. It is so close to assembly that the final assembler output is generated from this level. Before the source code is transformed to this level, it passes higher representation levels, too. First it is transformed to GENERIC form which is a language-independent abstract syntax tree. This tree will be lowered to GIMPLE which is a simplified subset of GENERIC, a restricted form where expressions are broken down into a 3-address form.

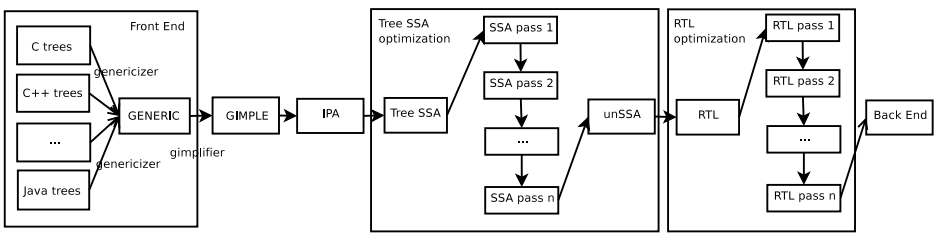

Figure 1. Optimization passes in GCC.

Since many optimizing transformations require higher level information about the source code, that is difficult or in some cases even impossible, to obtain from RTL, GCC developers introduced a new representation level called Tree-SSA [7, 8] based on *Static Single Assignment* form developed by researchers at IBM in the 1980s [9]. This IL is especially suitable for optimization methods that work on the *Control Flow Graph* (CFG), which is a graph representation of the program containing all paths that might be traversed during the execution. The nodes of this graph are *basic blocks* where one block is a straight-line sequence of the code with only one entry point and only one exit. These nodes in the flow graph are connected via directed edges and these edges are used to represent jumps in the control flow.

It may be reasonable to run one algorithm on different representations together (e.g. first on SSA level and later on RTL). Due to the different information stored by various ILs, it is possible that after running the algorithm on a higher level it will still find optimizable cases on a lower level as well. For the same reason, we obtained our best results by combining the mentioned refactoring algorithms on all implemented optimization levels.

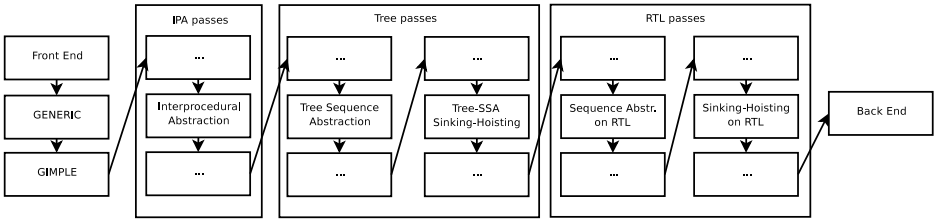

Figure 2. Implemented algorithms introduced in this paper.

We implemented the presented code factoring algorithms on the Tree-SSA and on the RTL levels as well, and for the sequence abstraction we made an interprocedural version, too. Due to the experimental state of interprocedural analysis in GCC, our implementation is also under research. Figure 2 demonstrates the order of our new passes.

3. Sinking-hoisting

The main idea of local factoring (also called local code motion, code hoisting or code sinking) is quite simple. Since it often happens that *basic blocks* with common predecessors or successors contain the same statements, it might be possible to move these statements to the parent or child blocks.

For instance, if the first statements of an `if` node's `then` and `else` cases are identically the same, we can easily move them before the `if` node. With this moving - called code hoisting - we can avoid unnecessary duplication of source code in the CFG. This idea can be extended to other more complicated cases as not only an `if` node, but a `switch` and a source code with strange `goto` statements can contain identical instructions. Furthermore, it is possible to move the statements from the `then` or `else` block after the `if` node, too. This is called code sinking which is only possible when there are no other statements depending on the moved ones in the same block.

For doing this code motion, we must collect *basic blocks* with common predecessors or common successors (also called same parents or same children) (Figure 3). These *basic blocks* represent a sibling set and we have to look for common statements inside them. If a statement appears in all the blocks of the sibling set and it is not depending on any previous statement, it is a hoistable statement, or if it has no dependency inside the *basic blocks*, it is a sinkable statement. When the number of blocks inside the sibling set is bigger than the number of parents (children) it is worth hoisting (sinking) the statement (Figure 4).

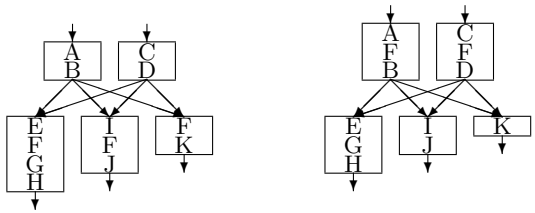

Figure 3. Basic blocks with multiple common predecessors (a) before and (b) after local factoring.

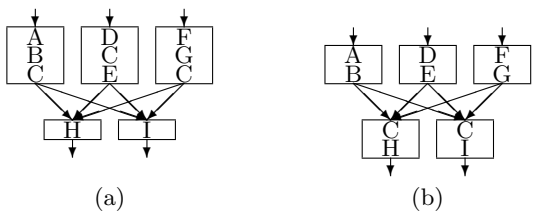

Figure 4. Basic blocks with multiple common successors (a) before and (b) after local factoring.

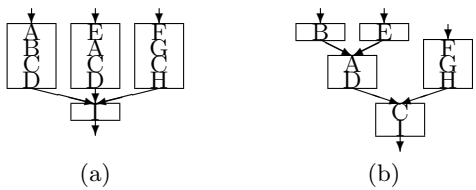

Figure 5. Basic blocks with multiple common successors but only partially common instructions (a) before and (b) after local factoring.

To obtain a better size reduction, we can handle a special case of local factoring when there are identical statements not appearing in all the *basic blocks* of the sibling set. When the number of blocks counting these statements is quite big and we could sink or hoist these statements, it is still possible to simplify the CFG (Figure 5). For instance, we can create a new block and link it before or after the sibling set depending on the direction of the movement. By building correct edges for this new *basic block* we can rerun the algorithm on the new sibling set with the same statements, and move the identically same statements into our new *basic block*. This way the gain may be a bit less because building a new *basic block* needs some extra cost for the new statements. However, this way we can have a more efficient algorithm in code size optimization.

3.1. RTL code motion

We first implemented this algorithm on the RTL level of GCC's optimization phases. This internal representation language is a low level representation of the source code, very close to the assembly. Therefore, when we are thinking of movable statements we should think of assembly kind statements where we are working with registers, memory addresses, etc.

RTL expressions (*RTX*, for short) are small instructions classified by expression codes (*RTX codes*). Each RTX has an expression code and a number of arguments. The number of these small instructions is quite big compared to other representations. Due to this the number of potentially movable instructions is the biggest on this level. Although it has an effect on the compilation time, too, it is not relevant because the instructions must be compared to each other only for local parts of the CFG where we do code factoring. The asymptotic complexity of this algorithm is $O(n^2)$ (where n is the number of instructions). The reason for this complexity is the comparison of all instructions one by one in the *basic block*s of a sibling set in order to find identical statements.

One important flavour of the hoisting-sinking algorithm running on this representation level is the definition for 'identically same' statements. It is evident that equal instructions must have the same RTX code, but because of the low level of RTL, the arguments must be exactly equal, too. To decide if one statement is movable, we have to check its dependencies, too. We simply have to check the statements before the current instruction for hoisting and do the same checking with the instructions after the movable statement until the end of the current *basic block* for sinking.

With this implementation, we could achieve 4.31% code size reduction (compared to '-Os') on a file in CSiBE environment. Compiling *unrarlib-0.4.0* for *i686-elf* target we could reach 0.24% average code-size saving. The average result measured on CSiBE was about 0.19% for the same architecture.

3.2. Tree-SSA code motion

Tree-SSA [7, 8] is a higher level representation than RTL. This representation is similar to GENERIC (a language independent code representation) but it contains modifications to represent the program so that every time a variable is assigned in the code, a new version of the variable is created. Every variable has an SSA_NAME which contains the original name for the variable and an SSA_VERSION to store the version number for it. Usually, in control flow branches it is not possible at compile time (only at run time) to decide for a used variable which of its earlier versions will be taken. To handle these situations in SSA form, *phi nodes* [9] are created to help us follow the lifecycle of the variable.

A *phi node* is a special kind of assignment with operands that indicate which assignments to the given variable reach the current join point of the CFG. For instance, consider the use of variable a (from example Figure 6a) with version number 9 and 8 in the `then` and `else` cases of an `if` node. Where a is referenced after the join point of `if` the corresponding version number is ambigous, and to solve this ambiguity a *phi node* is created as a new artifitial definition of a with version number 1.

The SSA form is not suited for handling non-scalar variable types like structures, unions, arrays and pointers. For instance, for an $M\,[100]\,[100]$ array it would be nearly impossible to keep track of 10000 different version numbers or to decide whether $M\,[i]\,[j]$ and $M\,[l]\,[k]$ referes to the same variable or not. To handle these problematic cases, the compiler stores references to the base object for non-scalar variables in *virtual operands* [7]. For instance, $M\,[i]\,[j]$ and $M\,[l]\,[k]$ are considered references to M in *virtual operands*.

For this optimization level we should redefine 'identically same statements' because we should not use a the strict definition we used before, where we expected from two equal statements for each argument to be equal, too. Due to the strict definition $a_{x1} = b_{y1} + c_{z1}$ differs from $a_{x2} = c_{z2} + b_{y2}$. However, these kinds of assignments are identical because '+' operand is commutable and SSA versions are indifferent for our cases. On the SSA level, we should define two statements to be equal when the TREE_CODE of the statements are equal, and if their arguments are variables, their SSA_NAME operands are equal, too (version number may differ). We require non-variable arguments to be exactly the same, but if the statements are commutable, we check the arguments for different orders as well.

For dependency checking, Tree-SSA stores the immediate uses of every statement in a list that we can walk through using an iterator macro. Thanks to this representation, the dependency check is the same as before in the RTL phase.

When moving statements from one *basic block* to another one, we have to pay attention to the *phi nodes*, the *virtual operands* and the immediate uses carefully. Usually, a variable of a sinkable assign statement's left hand appears inside a *phi node* (example Figure 6). In these situations, after copying the statement to the children or parent blocks we must recalculate the *phi nodes*. After this, in both sinking and hoisting cases, we must walk over the immediate uses of the moved statements and rename the old defined variables to the new definitions.

The current implementation has a weakness in moving statements with temporary variables. This problem occurs when an assignment with more than one argument is transformed to the SSA form. The problem is that in the SSA form one assignment statement can contain one operand on the right hand, and when GCC splits a statement with two or more operands, it creates temporary variables which have unique SSA_NAME containing a creation id. As we have defined, two variables are equal when their SSA names are equal. Consequently

```
  int D.1770;
<bb 0>:
  if (a_2 > 100) goto <L0>;
  else goto <L1>;
<L0>:;
  a_9 = b_5 + a_2;
  c_10 = a_9 * 10;
  a_11 = a_9 - c_10;
  goto <bb 3> (<L2>);
<L1>:;
  a_6 = a_2 + b_5;
  c_7 = a_6 * 12;
  a_8 = a_6 - c_7;
  # a_1 = PHI <a_11(1), a_8(2)>;
<L2>:;
  D.1770_3 = a_1;
  return D.1770_3;
```

(a) Before code motion

```
  int D.1770;
<bb 0>:
  a_16 = a_2 + b_5;
  if (a_2 > 100) goto <L0>;
  else goto <L1>;
<L0>:;
  c_10 = a_16 * 10;
  goto <bb 3> (<L2>);
<L1>:;
  c_7 = a_16 * 12;
  # a_14 = PHI <a_16(1), a_16(2)>;
  # c_15 = PHI <c_10(1), c_7(2)>;
<L2>:;
  a_1 = a_14 - c_15;
  return a_1;
```

(b) After code motion

Figure 6. An example code for Tree-SSA form with movable statements.

these statements will not be recovered as movable statements. Since these kinds of expressions are often used by developers, by solving this problem in the implementation we may get better results in size reduction.

On Tree SSA, with this algorithm the best result we could reach was 10.34% code saving on a file compiled to ARM architecture. By compiling *unrarlib-0.4.0* project in CSiBE for *i686-elf* target, we could achieve 0.87% extra code saving compared to '-Os' optimizations and the average code size reduction measured for the same target was about 0.1%.

4. Sequence abstraction

Sequence abstraction (also known as procedural abstraction) as opposed to local factoring works with whole single-entry single-exit (SESE) code fragments, not only with single instructions. This technique is based on finding identical regions of code which can be turned into procedures. After creating the new procedure we can simply replace the identical regions with calls to the newly created subroutine.

There are well-known existing solutions [10, 11] already, but these approaches can only deal with such code fragments that are either identical or equivalent in some sense or can be transformed with register renaming to an equivalent form. However, these methods fail to find an optimal solution for these cases where an instruction sequence is equivalent to another one, while a third one is only

identical with its suffix (Figure 7a). The current solutions can solve this situation in two possible ways. One way is to abstract the longest possible sequence into a function and leave the shorter one unabstracted (Figure 7b). The second way is to turn the common instructions in all sequences into a function and create another new function from the remaining common part of the longer sequences (Figure 8c). This way, we should deal with the overhead of the inserted extra call/return code as well.

Our approach was to create multiple-entry single-exit (MESE) functions in the cases described above. This way, we allow the abstraction of instruction sequences of differing lengths. The longest possible sequence shall be chosen as the body of the new function, and according to the length of the matching sequences we define the entry points as well. The matching sequence will be replaced with a call to the appropriate entry point of the new function. Figure 8d shows the optimal solution for the problem depicted in Figure 7a.

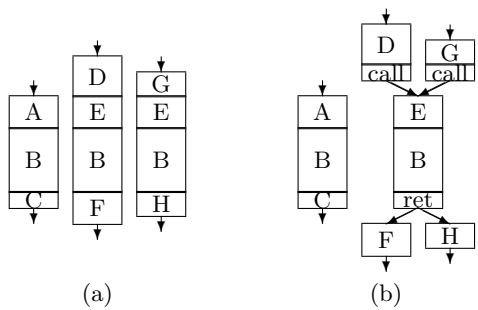

Figure 7. Abstraction of (a) instruction sequences of differing lengths to procedures using the strategy for abstracting only the longest sequence (b). Identical letters denote identical sequences.

Sequence abstraction has some performance overhead with the execution of the inserted call and the 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.

4.1. Sequence abstraction on RTL

Using the RTL representation algorithms one can optimize only one function at a time. Although sequence 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

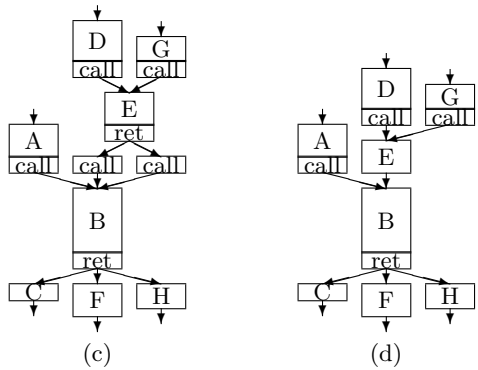

Figure 8. Abstraction of instruction sequences from Figure 7 of differing lengths to procedures using different strategies (c,d). Identical letters denote identical sequences.

instance of them has to be retained in the body of the processed function, and all the other occurrences will be replaced by the code transferring control to the retained instance. However, to preserve the semantics of the original program, the point where the 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 the control back to the "callers."

Implementing sequence abstraction on the RTL phase has the quite significant benefit of reducing the code size instead of implementing the abstraction on a higher intermediate language. Most of the optimization algorithms are finished while the abstraction is started in the compilation queue. Those very few algorithms which are executed after sequence abstraction do not have or only have a very little impact on code size. So our algorithm still has an observable effect on the output before it is generated by GCC.

The current implementation can only deal with identical statements where the registers and the instructions are exactly the same. For further improvements, with some extra cost it might be possible to abstract identical sequences where the registers may differ.

This implementation has approximately $O(n^2)$ cost of running time. The reason for it is the comparison of possible sequences inside *basic blocks* with n instructions on the current IL. This cost can be optimized to $O(n \log n)$ using hashtables with fingerprints. We already have a version for this optimization as well but further tests are required to validate this implementation.

This algorithm brought us maximum 45.69% code saving on a source file

compiled in CSiBE. Compiling *libmspack* project of CSiBE for *arm-elf* target, we achieved 2.39% extra code saving compared to '-Os' optimizations, and our average result in size reduction measured for the same target was about 1.01%.

4.2. Sequence abstraction on tree

As a general rule in compilers the commands in a higher intermediate language representation could describe more architecture dependent instructions than in a lower IL. In our view if the sequence abstraction algorithm can merge similar sequences in a higher IL, it could lead to better code size reduction.

In Tree IL, there are less restrictions than in RTL. For instance, we do not have to care about register representations. So, the algorithm is able to find more sequences as good candidates to abstraction, while in RTL we must be sure that all references to registers are the same in every subsequence.

Unfortunately, the results do not achieve our expectations. The main problem is that the sequence abstraction algorithm is in a very early stage of compilation passes. Other algorithms followed by abstraction could simply mess up the results. This is supported by the fact that after our pass there, we could achieve even 9.25% (2.5% in average) code size reduction counted in Tree unit. In addition, there are some cases when the abstraction does the merge, but it is not really wanted because one or more sequences are dropped (for example a dead code), or there is a better solution for code optimization. These cases mostly occur when the algorithm tries to merge short sequences.

However, for a better performance there are still additional improvements on the current implementation. One of them is to extend the current implementation with the ability to abstract approximately equal sequences as well. The current implementation realizes abstractable sequences where the statements are equal, but for several cases it might be possible to deal with not exactly the same sequences as well. Another possible improvement is to compare temporary variables as well. It is exactly the same problem as the one described before for code motion on the Tree-SSA level.

This implementation, similarly to the RTL, has a cost of running time about $O(n^2)$ for the same reason. We also have an optimized version for $O(n \log n)$ using hashtables with fingerprints, but further tests are required for validation.

With this optimization method we could reach maximum 41.60% code-size saving on a source file of CSiBE, and by compiling *flex-2.5.31* project of the environment for *arm-elf* target, we achieved 3.33% extra code saving compared to '-Os' optimizations. The average result in size reduction measured for the same target was about 0.73%.

4.3. Procedural abstraction with IPA

The main idea of *interprocedural analysis* (IPA) optimizations is to produce algorithms which work on the entire program, across procedure or even file boundaries. For a long time, the open source GCC had no powerful interprocedural methods because its structure was optimized for compiling functions as units. Nowadays, new IPA framework and passes are introduced by the *IPA branch* and it is still under heavy development [12].

We have also implemented the interprocedural version of our algorithm. This implementation is very similar to the one we use on Tree with one big difference: we can merge sequences from any functions into a new real function.

With this approach we can handle more code size overhead coming from function call API than the other two cases described above. In addition, we are also able to merge the types of sequences which use different variables or addresses, because it is possible to pass the variables as parameters for our newly created function. The procedural abstraction identifies and compares the structure of the sequences to find good candidates to the abstraction, and with this method we are able to merge sequences more efficiently than in the previously discussed implementations.

In spite of these advantages, there are disadvantages as well. The IPA is also in a very early stage in compilation passes, so there is a risk that other algorithms are able to optimize the candidates better than procedural abstraction (the same cases as in sequence abstraction on Tree). The other disadvantage is that the function call requires many instructions and estimating its cost is difficult. This means that we can only guess the gain we can reach by a given abstraction, because determining how many assembly instructions are in a higher level statement is not possible. This cost computation problem is our difficulty on the Tree level, too. The guessing will not be as accurate as the calculation of the gain on RTL level, and for several cases where we could save code size this may cause that the algorithm does no abstraction.

Our implementation of this algorithm has a cost of running time about $O(n \log n)$ already, as the comparision of possible sequences is realized using hashtables. It can be compared to the slower $O(n^2)$ implementations for other representations on Table 3.

With IPA our best result on a source file of CSiBE was 59.29% code saving compared to '-Os'. By compiling *zlib-1.1.4* project of the environment for *arm-elf* target, we achieved 4.29% average code saving and the measured average result on CSiBE was about 1.03%.

5. Experimental evaluation

For the implementation, we used the GCC source from the repository of the *cfo-branch* [13]. This source is a snapshot taken from GCC version 4.1.0 on 2005-11-17. Our algorithms are publicly available in the same repository as well.

We used CSiBE v2.1.1 [14] as our testbed, which is an environment developed especially for code-size optimization purposes in GCC. It contains small and commonly used projects like zlib, bzip, parts of Linux kernel, parts of compilers, graphic libraries, etc. CSiBE is a very good testbed for compilers and not only for measuring code size results, but even for validating compilation.

Our testing system was a dual Xeon 3.0 Ghz PC with 3 Gbyte memory and a Debian GNU/Linux v3.1 operating system. For doing tests on different architectures we crosscompiled GCC for *elf* binary targets on common architectures like *ARM* and *SH*. For crosscompiling, we used binutils v2.17 and newlib v1.15.0.

The results show (on Table 1 and Table 2) that these algorithms are really efficient methods in code size optimization, but on higher level intermediate representation languages further improvements may still be required to get better performance due to the already mentioned problems. For *i686-elf* target, by running all the implemented algorithms as extension to the '-Os' flag, we could achieve maximum 57.05% and 2.13% average code-size saving. This code-size saving percentage is calculated by dividing the size of the object (compiled with given flags) with the size of the same object compiled with '-Os' flag. This value is substraced from 1 and converted to percentage (by multiplying it with 100).

flags	i686-elf		arm-elf		sh-elf	
	size (byte)	saving (%)	size (byte)	saving (%)	size (byte)	saving (%)
-Os	2900177		3636462		3184258	
-Os -ftree-lfact -frtl-lfact	2892432	0.27	3627070	0.26	3176494	0.24
-Os -frtl-lfact	2894531	0.19	3632454	0.11	3180186	0.13
-Os -ftree-lfact	2897382	0.10	3630378	0.17	3179622	0.15
-Os -ftree-seqabstr -frtl-seqabstr	2855823	1.53	3580846	1.53	3149822	1.08
-Os -frtl-seqabstr	2856816	1.50	3599862	1.01	3162678	0.68
-Os -ftree-seqabstr	2888833	0.39	3610002	0.73	3166054	0.57
-Os -fipa-procabstr	2886632	0.47	3599042	1.03	3160626	0.74
all	2838348	2.13	3542506	2.58	3123398	1.91

Table 1. Average code-size saving results. (*size* is in byte and *saving* is the size saving correlated to '-Os' in percentage (%))

Here we have to mention that by compiling CSiBE with only '-Os' flag using the same version of GCC we used for implementation, we get an optimized code which size is 37.19% smaller than compiling it without optimization methods.

flags	i686-elf	arm-elf	sh-elf
	max. saving (%)	max. saving (%)	max. saving (%)
-Os -ftree-lfact -frtl-lfact	6.13	10.98	10.29
-Os -frtl-lfact	4.31	3.51	4.35
-Os -ftree-lfact	5.75	10.34	8.78
-Os -ftree-seqabstr -frtl-seqabstr	36.81	56.92	43.89
-Os -frtl-seqabstr	30.67	45.69	42.45
-Os -ftree-seqabstr	30.60	41.60	44.72
-Os -fipa-procabstr	38.21	56.32	59.29
all	57.05	61.53	60.17

Table 2. Maximum code-size saving results for CSiBE objects. (*saving* is the size saving correlated to '-Os' in percentage (%))

By running all the implemented algorithms together we get a smaller code saving percentage compared to the sum of percentages for individual algorithms. This difference occurs because the algorithms work on the same source tree and the earlier passes may optimize the same cases which would be realized by later methods, too. This is the reason why for *i686-elf* target, by running local factoring on RTL level and Tree-SSA, we could achieve 0.19% and 0.10% average code saving on CSiBE, while by running both of these algorithms the result was only 0.27%. This difference also proved that the same optimization method on different ILs may realize different optimizable cases and running the same algorithm on more than one IL will lead to better performance.

The answer for negative values in the code-saving percentages column (Figure 9) is exactly the same. Unfortunately, even if an algorithm optimizes the source tree, it might be possible that it messes up the input for another one which could do better optimization for the same tree. These differences do not mean that the corresponding algorithms are not effective methods, but these usually mean that later passes could optimize the same input source better.

In the table of compilation times (Table 3), two algorithms surpass the others. These are the RTL and Tree sequence abstractions because the current implementation is realized with a running time of about $O(n^2)$. Nevertheless, these algorithms can be implemented with $O(n \log n)$ length of duration and this can result in a relative growth of compilation time similar to the interprocedural abstraction, which is already developed for $O(n \log n)$ running time.

We have to note that the sequence abstraction algorithms have an effect on the running time as well, because these methods may add new calls to the CFG. As local factoring does not change the number of executed instructions, these optimizations do not change the execution time of the optimized binaries. We measured on our testing system with CSiBE that the average growth of running time compared to '-Os' flag was about 0.26% for tree abstraction, 0.18% for interprocedural abstraction and 2.49% for the execution of all the algorithms together.

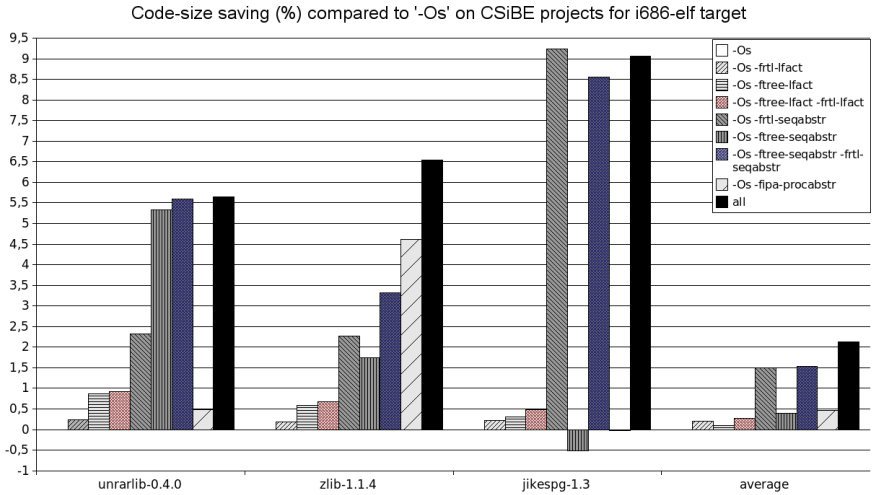

Figure 9. Detailed results for selected CSiBE projects.

flags	i686-elf		arm-elf		sh-elf	
	absolute	relative	absolute	relative	absolute	relative
-Os	259.17	1.0000	285.97	1.0000	311.73	1.0000
-Os -ftree-lfact -frtl-lfact	262.73	1.0137	288.93	1.0104	333.64	1.0703
-Os -frtl-lfact	264.33	1.0199	303.27	1.0605	321.45	1.0312
-Os -ftree-lfact	259.90	1.0028	343.06	1.1996	321.97	1.0328
-Os -ftree-seqabstr -frtl-seqabstr	455.53	1.7576	464.00	1.6225	609.11	1.9540
-Os -frtl-seqabstr	315.75	1.2183	521.80	1.8247	651.13	2.0888
-Os -ftree-seqabstr	303.25	1.1701	325.76	1.1391	360.41	1.1562
-Os -fipa-procabstr	284.31	1.0970	298.79	1.0448	337.98	1.0842
all	342.69	1.3223	393.36	1.3755	489.69	1.5709

Table 3. Compilation time results. (*absolute* is in second (s) and *relative* means the multiplication factor of the compilation time with '-Os')

6. Conclusion

As a conclusion of the results above, the sequence abstraction algorithms produced the best results for us. However, the percentages show that these algorithms did not work as well on higher level abstraction languages as we expected. The reason for this is that in earlier passes our methods could easily mess up the code for later passes which could better deal with the same optimization cases. Perhaps, these situations should be eliminated by compiling in two passes, where

in the first pass we recover these problematic situations, and in the second one we optimize only the cases which would not set back later passes.

About local code motion we have to notice that these algorithms yielded smaller percentages, but collaborated better with later passes. The reason for this is that these methods do transformations on local parts of the source tree and do not do global changes on it. Another benefit is that local factoring should deal with very small or usually no overhead, and the accuracy of cost computation does not affect the optimization too much.

Another conclusion is about the running time of the algorithms. The sequence abstraction brought us the best results, but with a quite slow running time. Since this time affects the compilation, we evolve the deduction that if the duration of the compilation is really important, we suggest using local code motion as a fast and effective algorithm. Otherwise, when compilation time does not matter, we can use all the algorithms together for better results.

Finally, as an acknowledgment, this work was realized thanks to the *cfo-branch* of GCC. Developers interested in our work are always welcome to help us improve the code factoring algorithms introduced in this paper.

References

[1] **Free Software Foundation,** *GCC, GNU Compiler Collection,* `http://gcc.gnu.org/`; accessed March, 2007.

[2] **Debray S., Evans W., Bosschere K. D. and Sutter B. D.,** The Squeeze Project: Executable Code Compression, `http://www.cs.arizona.edu/projects/squeeze/`, accessed March, 2007.

[3] **AbsInt (Angevandte Informatik),** *aiPop - Automatic Code Compaction,* `http://www.absint.com/aipop/`, accessed March, 2007.

[4] **Lóki G., Kiss A., Jász J. and Beszédes A.,** Code factoring in GCC, *Proceedings of the 2004 GCC Developers' Summit,* 2004, 79–84.

[5] **Beszédes A., Ferenc R., Gergely T., Gyimóthy T., Lóki G. and Vidács L.,** CSiBE benchmark: One year perspective and plans, *Proceedings of the 2004 GCC Developers' Summit, June 2004*, 7–15.

[6] **Free Software Foundation,** *GNU Compiler Collection (GCC) internals,* `http://gcc.gnu.org/onlinedocs/gccint`, accessed March, 2007.

[7] **Novillo D.,** Tree SSA a new optimization infrastructure for GCC, *Proceedings of the 2003 GCC Developers' Summit,* 2003, 181–193.

[8] **Novillo D.,** Design and implementation of Tree SSA, *Proceedings of the 2004 GCC Developers' Summit,* 2004, 119–130.

[9] **Cytron R., Ferrante J., Rosen B. K., Wegman M. N. and Zadeck F. K.,** Efficiently computing static single assignment form and the control dependence graph, *ACM Transactions on Programming Languages and Systems*, **13** (4) (1991), 451–490.

[10] **Cooper K.D. and McIntosh N.,** Enhanced code compression for embedded RISC processors, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999, 139–149.

[11] **Debray S. K., Evans W., Muth R. and de Sutter B.,** Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, **22** (2) (2000), 378–415.

[12] **Hubička J.,** The GCC call graph module, a framework for interprocedural optimization, *Proceedings of the 2004 GCC Developers' Summit,* 2004, 65–75.

[13] **Lóki G.,** *cfo-branch - GCC development branch,* `http://gcc.gnu.org/projects/cfo.html`, accessed March, 2007.

[14] **Department of Software Engineering, University of Szeged,** *GCC Code-Size Benchmark Environment (CSiBE),* `http://www.csibe.org/`, accessed March, 2007.

Cs. Nagy, G. Lóki, Á. Beszédes and T. Gyimóthy
Department of Software Engineering
University of Szeged
Dugonics tér 13.
H-6720 Szeged, Hungary
{ncsaba,loki,beszedes,gyimi}@inf.u-szeged.hu