Optimizing for Space : Measurements and Possibilities for Improvement

Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki and László Vidács
Research Group on Artificial Intelligence
University of Szeged
Aradi vértanúk tere 1., H-6720 Szeged, Hungary, +36 62 544126
{beszedes,gertom,gyimi,alga,lac}@cc.u-szeged.hu, http://gcc.rgai.hu/

Abstract

GCC's optimization for space seems to have been often neglected, in favor of performance tuning. With this work we aim at determining the weakpoints of GCC concerning its optimization capability for space. We compare (1) GCC with two non-free ARM cross-compiler toolchains, (2) how GCC evolved from release 3.2.2 to version 3.3, and (3) two runtime libraries for the Linux kernel. All tests were performed using the C front end and for the ARM target both as standalone and as Linux executables. The test suite is comprised of applications from well-known benchmark suites such as SPEC and Mediabench. An optimal combination of compiler (and linker) options with respect to minimal code size is elaborated as well. We conclude that GCC 3.3 steadily improves with respect to version 3.2.2 and that it is only about 11% behind a high-performance non-free compiler. At the same time, we were able to document a number of issues that deserve further investigation in order to improve code generation for space.

1 Introduction

GCC is increasingly used as a cross-compiler to produce programs for embedded systems. Although performance in terms of speed is also important, in many cases the amount of consumed resources (memory, energy, etc.) plays an even greater role in the case of devices with limited resources. So, when GCC is used to build these software, the code produced should be as small as possible. Indeed, GCC is able to optimize for space but, alas, it seems that this objective was often neglected when designing and implementing various code generation and optimization algorithms [1, 5]. We may conclude the same when we consider the fact that beside the vital regression testing methods and the results of several benchmark suites available on GCC web pages [9, 8, 3], no word is spoken about benchmarking *code size*. In fact, were unable to find any related publication at all which deals with the assessment of compilers' capabilities for space optimization.

With this work we attempted to determine the weakpoints of GCC concerning its optimization capability for space. We present the results of our assessments where we compared:

- GCC for standalone executable with two non-free ARM cross-compiler toolchains,
- How GCC evolved from release 3.2.2 to version 3.3, and
- Two runtime libraries for GNU/Linux, glibc [2] and μClibc [7].

All tests were performed using the C front end and for the ARM target (both for standalone and Linux executables) as this combination is one of the most frequently used nowadays for embedded applications. A testbed was utilized with applications from various well known benchmark suites.

We did our best to discover the optimal combination of compiler (and linker) options with respect to minimal code size; we elaborate on the relevant ones for GCC and propose a set of options to extend the default settings for code size. With this option set an improvement of nearly 5% was achieved.

In the investigation we included both the object sizes produced by the compiler and the linked executable sizes to see what effect the runtime libraries had on the overall linked code size. Comparing only object sizes, one non-free compiler is about 11% better than GCC, but in the case of executables this ratio rises to 32%.

We investigated the generated code by GCC more thoroughly and finally we document several issues that deserve further investigation in order to improve code generation for space. These include the lack of interprocedural optimizations, the required unit at a time compilation, more intelligent handling of `-Os`, etc.

In Section 2 we describe our measurement environment and methodology. Section 3 deals with GCC's different compiler options and there also we give our proposal for the best combination. Sections 4 and 5 present the actual results for standalone executables and Linux libraries, respectively. Finally, in Section 6 we summarize our conclusions and give our view on the possibilities for improving GCC.

2 Measurement Environment

For all three objectives of our investigation presented in the previous section, we have set up a common measurement environment. It consists of a collection of test programs that are suitable for compiling and measuring code size for all compilers and configurations under investigation. The environment is able to perform these measurements and present the data in a simple form ready for further processing. In addition, it also facilitates the execution of the executable programs.

2.1 Compiler Toolchains

In each experiment we employed C as the source language and the chosen target architecture was ARM (32-bit ARM instruction set). Two types of target code were used: standalone programs (that run on the hardware without an operating system) and Linux target for the ARM architecture (for GCC compiler `arm-elf` and `arm-linux` machines, respectively). The following toolchains were used for the measurements:

- GCC 3.2.2 version with newlib version 1.10.0 [6] for standalone target (with binutils version 2.13)
- GCC 3.3 prerelease snapshot (2003-04-14) with the same newlib and binutils
- GCC version 3.2.2 with glibc version 2.2.5 [2] for Linux target
- GCC 3.3 prerelease snapshot (2003-04-14) with glibc version 2.2.5
- GCC version 3.2.2 with μClibc version 0.9.15 [7]
- Two non-free compilers for ARM architecture configured as standalone targets. These will be denoted by *Compiler 1* and *Compiler 2* in the following discussions. The former uses `elf` output format, while the latter produces `coff` files.

The switches that control optimization for space were turned on for all toolchains. In addition, several further options (both compiler and linker) that enable or disable certain code optimization and/or generation algorithms were also set that resulted the most compact code size. The combination of these extra options was determined by trial and error, and for GCC toolchains we elaborate on these in Section 3.

For each GCC toolchain the runtime libraries were compiled using the same options as for the test programs. (Neither of the two non-free compilers libraries were prepared in such way.) The use of such libraries has an effect where the executables are compared, because the overall code size incorporates library code as well.

2.2 Testbed

The testbed used in the experiments consists of two parts: small example programs and real applications from several well-known benchmark suites (GNU applications, SPEC CPU2000 [10], MediaBench [4]). In the following table some information is given about the sizes of the test programs:

Test project	*files*	*lines*	*bytes*	*exec.*
bzip2	1	4,250	121,279	1
catdvi	6	770	24,332	1
flex	21	19,571	530,312	1
g721	8	1,725	46,980	2
gsm	29	5,982	182,809	1
jpeg	84	34,181	1,150,110	6
mcf	25	2,414	53,310	1
mpeg2enc	22	7,608	217,864	1
osdemo	147	68,434	1,925,141	1
parser	18	11,391	356,526	1
sed	20	12,393	365,886	1
P3szogr	1	48	1,568	1
_3szog	1	48	1,419	1
abc	1	17	443	1
arg	1	25	390	1
datum	1	48	870	1
eltelt	1	32	939	1
endian	1	18	258	1
geometry	1	435	11,869	1
lnkoszt	1	52	1,121	1
minimax	1	52	1,444	1
static	1	35	460	1
szinusz	1	52	1,372	1

The first column shows the number of files that constitute the test project, the second one gives the total number of program lines, and the third column gives the size of the source code in bytes. In the last column the number of executables that are built from the test project is shown.

All test programs were compiled to produce the object files and the given executable programs were prepared by linking. These objects and the linked executables for each of the toolchain under investigation were used for measurement.

In the following for each measurement the small programs (the last 12) are treated jointly and are denoted by "small."

2.3 Measurement Method

The way to measure the size of the generated code (i.e. its compactness) is not always trivial. As obvious, we chose to investigate the final binary machine code (instead of, for example, the assembly code).

Objects and executables. The granularity of the code was a further aspect: should we measure the function sizes individually, the object code for a complete compilation unit, or investigate the size of the linked executable? In this paper we present the results for the latter two because in certain environments both can be interesting. When we compare the object sizes the effectiveness of the compiler proper is actually compared,[1] while in the second case the whole compiler toolchain is assessed including the compiler, the linker and libraries as well. This is because the size of a linked program depends on the size of the libraries and also how they are processed by the linker. Hence, in this paper we mostly rely on comparing objects which is more informative with regard to a compiler's optimization capability for space.

In order to get the best possible results when measuring executables, we also built the libraries of GCC toolchains with the same flags as the test sources. With the libraries of the two non-free compilers we were not able to do the same.

Standalone and Linux programs. Another dimension of the categorization we investigated was both kinds of targets: standalone executables (i.e. for without an operating system) and executables built for a specific operating system (in our case GNU/Linux). Although the same compiler is used with the same settings, the resulted binaries generally contain several notable differences: a few in the case of objects and a significant difference with executables. These are mostly due to different executable production and to the fact that different runtime libraries are used for the two cases (i.e. in the case of GCC, newlib and glibc).

One would expect that with objects there should be no difference at all. However, some minor impact of the library is still noticeable. The library headers should contain the same standard prototypes (e.g. standard functions), but the difference comes from the different implementation of some features. For example, some standard names can be implemented using macros and function calls as well.

Clearly, then, measuring the size of the executables incorporates a much greater impact of the library code. It is apparently measurable on standalone executables. However, the situation becomes more complicated when we investigate executables built

[1] Note, that the library implementation still has a minimal impact on the object sizes because of the library headers, which are also translated by the compiler. Consider for example, that macros can be used to implement function-like behavior.

for Linux. The reason for this is that Linux executables do not embed the library code, but they maintain only references to the so-called shared objects, which are linked at runtime. (Even if static linking is used some functionality will still be implemented in the operating system rather than the executable.) We present some results for Linux executables in Section 5.

Sections. Another problem was deciding which parts of the generated files we should take into account (obviously the size of the binary file is not relevant because of various headers, etc.). The generated program code consists of many parts; instructions, data and so on, which are generally separate in a binary file (in the *sections*). However, in many cases these parts can be intermixed (e.g. executable code can contain embedded data). In addition, several other sections are generally also put into the binary file, which are of no interest with respect to the size of the code. These include the debug sections, symbol tables, etc.

The different types of object files (`elf` and `coff`) can have different kinds of sections and, what is more, the different compilers may use various strategies for laying out code and data into sections. More specifically, different compilers may split some code into several sections, or put other things together in one section. For example, `elf` files contain one (or more) initialized read-write data section(s), while `coff` files contain program code that will initialize the data at runtime. So no common handling could be used and the combination of the sections to be incorporated in the measurements needed to be determined separately for each toolchain.

In each case we summarized the size of only those sections that contains generated code that is directly used by the program. These are the sections that contain executable code and constant- or initialized read-write program data. Note, however, that executable code and constant data cannot always be clearly separated (there are constant data items which are "hidden" in the executable code) so we handle them together during the comparison.

We experimented with two kinds of section combinations: (1) the size of sections containing program code or constant data (referred to as "read-only sections") and (2) the size of sections that contain any kind of program data, which also includes read-write data (referred to as "all sections"). We decided to follow the second approach because it seemed to be the most reasonable because of the above-mentioned various types of handling of initialized read-write data.

Measurement tools. When assessing both the object and executable sizes the `elf` and `coff` files needed to be investigated. To this end different methods for extracting the section sizes were employed because of the different binary formats. The program `size` (part of *binutils*) is a suitable tool for extracting the size of the mentioned sections from `elf` files. We were unaware of any similar tool for `coff` files. The program `coffdump` extracts the sizes of the sections from `coff` files, but not in a summarized form. Fortunately, all `coff` files contain almost the same sections and have the same names. We examined what kind of data was contained in the sections, and counted the required sizes by hand. (Fortunately, only one of the non-free compilers uses this format, with all other toolchains including GCC we were able to extract code sizes automatically.)

Execution. The measurement environment is capable for executing the built programs using a simulator for standalone programs and an ARM-based hardware device with Linux system for Linux binaries. We ran the programs and checked their outputs for validating the compiler toolchain with components of different versions, and for verifying the correctness of various compiler option combinations. Throughout our measurements only those configurations were used that produced correct and running programs.

3 Compiler and Linker Options

With each toolchain investigated we sought to find the best possible combination of options with respect to code size. In general, compilers provide a special optimization option that instructs them to optimize for space rather than for speed. With GCC, this option is the switch called `-Os`.

3.1 Best Options for Space in GCC

Commonly, `-Os` is used internally in GCC to enable or disable certain optimization algorithms, but generally any part of the compiler proper can depend on this option and perform differently when space is the concern. However, there are a number of other compiler options (mostly related to optimization) which have a notable effect on the size of the generated code. By experimenting with these options we found that `-Os` alone does not produce the minimal code for our testbed. Hence we determined the combination of options on top of `-Os`, which proved to be the best on our testbed.[2]

The following table summarizes the final choice of options, which we used in all our trials (except where mentioned otherwise). (Note, that some of these are implicitly enabled or disabled by `-Os`,[3] therefore we supply the options later in the command-line so that they will be overridden.)

Compiler Option	*3.2*	*3.3*
`-Os`	yes	yes
`-mno-apcs-frame`	yes	yes
`-fomit-frame-pointer`	yes	yes
`-ffunction-sections`	yes	yes
`-fdata-sections`	yes	yes
`-fno-force-mem`	yes	yes
`-fno-force-addr`	yes	yes
`-fno-inline-functions`	yes	yes
`-fnew-ra`	no	yes
`-fbranch-probabilities`	yes	yes
`-finline-limit=1`	yes	yes
`-fno-schedule-insns`	yes	yes
`-fno-optimize-sibling-calls`	yes	yes
`-fno-if-conversion`	no	yes
`-fno-thread-jumps`	yes	yes
`-fno-hosted`	yes	yes

Some options were not available in GCC 3.2 releases, evidently they were left out in the cases when this release was measured. We will use the notation *opt-1* for the best options for 3.2.2 and *opt-2* for the best options for 3.3.

The option `-mno-apcs-frame` is specific to the ARM target. We also used another ARM-specific option `-mno-thumb-interwork` to tell the compiler that we were generating for just 32-bit ARM instruction set.

Two interesting options are `-ffunction-sections` and `-fdata-sections` which generate only one function/data per section and this helps the linker to omit the unused functions/data from the executable. Generally speaking, they do not influence the object sizes, but the executables may become smaller.

Another notable option is `-fno-inline-functions` which disables the automatic inlining of GCC. In general, automatic inlining performs very badly with respect to code size and it could be made more intelligent.

The linker also has a number of options that were worth experimenting with. We determined the following combination which produced an overall smaller code than the default:

Linker Option
`-O 2`
`--gc-sections`
`--relax`
`--no-whole-archive`

The options listed above produced, on our testbed, an overall improvement in code size of 4.78% with respect to using only `-Os`. Figure 1 shows the results separately for each program. To obtain the relevant data we used the GCC 3.3 snapshot with only `-Os` turned on and compared it to the same compiler with additional options from the table above (average object sizes of test projects in standalone target). The total sizes of the test projects is given with the project's name in bytes.

We can see from the above plot that every test program has benefited from these options, especially the bigger ones (except `flex`, which is probably due to the fact that it contains uncommonly large amount of data).

In Section 5 we present some data which shows that

[2]One option belongs to this set if it produces an overall gain with respect to the default `-Os`, so it may happen that in some cases it performs worse. It may also happen that one option combined with another one degrades the overall result, but of course, we could not try every combination of the options available.

[3]This is the list taken from the GCC 3.3 sources:

`-falign-functions -falign-jumps -falign-labels -falign-loops -fbranch-probabilities -fcaller-saves -fcprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdefer-pop -fdelete-null-pointer-checks -fexpensive-optimizations -fforce-mem -fgcse -fif-conversion -fif-conversion2 -floop-optimize -fno-merge-constants -fno-reorder-blocks -foptimize-sibling-calls -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -frerun-loop-opt -fstrength-reduce -fstrict-aliasing -fthread-jumps` options that depend on a define: `-fdelayed-branch -fomit-frame-pointer -fschedule-insns -fschedule-insns-after-reload`.

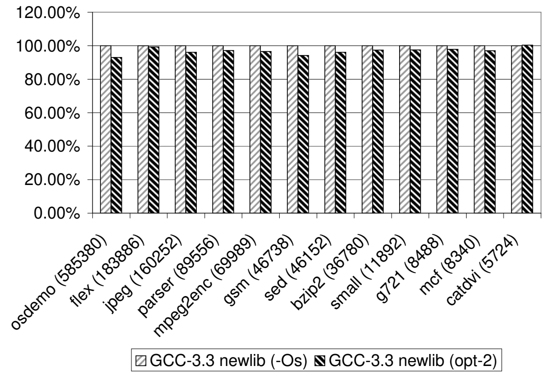

Figure 1: The effect of additional compiler options

a marked improvement in library size can also be achieved using this options set.

Due to the above results we propose to add these to the default operation of `-Os` in future releases of GCC (at least for the ARM target).

3.2 Other Optimization Options

There are high number of optimization options (starting in `-f`) in GCC that can be given on command-line (170+). Most of them have a binary state and so a corresponding `-fno-`*`XXX`* is also normally present. We examined all available options in GCC 3.3 but of course, we could not try all of the possible combinations, so we followed a simple approach in that an option (both the enabling and disabling versions) was added to the list of good options if it brought improvement over the default `-Os`. An individual option was tried separately from the others rather than by cumulating them. The final result is given in the previous section.

Many of the investigated options had some problems or did not yield improvements and hence they were ignored. In the following we categorize these options rather than listing them all (they can be found in the GCC manual). Those options that are not mentioned here did not improve the code (the correctness of the output was not verified either).

Combined use. The following options separately produced certain improvements, but their combined effect was not better on average: `-ffast-math` `-ffreestanding` `-fno-builtin` `-fno-inline` `-fno-sched-interblock` `-fno-sched-spec` `-fsched-spec-load` `-fvolatile-static`.

Parameterized options. For this work we were not able to include the investigation of those options that accept some parameters (i.e. not a binary). This parameter is generally a number but in some cases it can be a string. We only investigated `-finline-limit=`*`number`* which showed a minor improvement. The following options were left with their default settings: `-falign-functions=`*`number`* `-falign-labels=`*`number`* `-falign-loops=`*`number`* `-falign-jumps=`*`number`* `-fcall-used-`*`number`* `-fcall-saved-`*`number`* `-fdiagnostics-show-location=`*`string`* `-ffixed-`*`number`* `-fmessage-length=`*`number`* `-fsched-verbose=`*`number`* `-fstack-limit-register=`*`number`* `-fstack-limit-symbol=`*`string`* `-ftls-model=`*`number`*.

Invalid generated code. The options listed here always produced smaller code, but these codes could not be correctly executed on GCC 3.3: `-fshort-double` `-fsingle-precision-constant` `-funsafe-math-optimizations`. These should be investigated for possible bugs in GCC.

Irrelevant option. Some options are either not implemented in GCC 3.3 or they did produce some extremely small code. These are the following: `-fallow-single-precision` `-fcall-saved` `-fcall-used` `-fconstant-string-class` `-fdiagnostics-show-location` `-fdump-tree` `-ffixed` `-finline-functions` `-finline-limit` `-finstrument-functions` `-fleading-underscore` `-fmessage-length` `-fno-allow-single-precision` `-fno-call-saved` `-fno-call-used` `-fno-constant-string-class` `-fno-diagnostics-show-location` `-fno-dump-class-hierarchy` `-fno-dump-translation-unit` `-fno-dump-tree` `-fno-fixed` `-fno-function-sections` `-fno-inline-limit` `-fno-message-length` `-fno-pretend-float` `-fno-sched-verbose` `-fno-stack-limit-register` `-fno-stack-limit-symbol` `-fno-tabstop` `-fno-template-depth` `-fpreprocessed` `-fpretend-float` `-fprofile` `-fprofile-arcs` `-fsched-verbose` `-fshort-enums` `-fssa` `-fstack-limit` `-fstack-limit` `-fstack-limit-register` `-fstack-limit-symbol` `-fsyntax-only` `-ftabstop` `-ftemplate-depth`.

4 Compiler and Toolchain Comparisons

In this section we present the results of a comparison of the sizes of objects and executables of GCC configured for a standalone target with two non-free compilers. The two compilers shall remain anonymous, which will be referred to as *Compiler 1* and *Compiler 2*. In both cases the best configuration of compiler options was used for code size. In the diagrams *opt-1* denotes the best options for GCC 3.2.2 and *opt-2* the best options for 3.3.

A comparison of objects is more informative with regard to a compiler's optimization capability for space, because in this case no pre-generated code of libraries or startup routines are included.

All sizes comprise of the program section sizes (as described in Section 2.3), and we present these in a relative form: with respect to GCC 3.3 snapshot with our option-set (elaborated in Section 3).

4.1 Compiler Results on Objects

In Figure 2 the average achievement of the C compilers is shown in terms of object size. The values are computed as the sum of the sizes of all objects of the test programs, and are shown as relative to GCC.

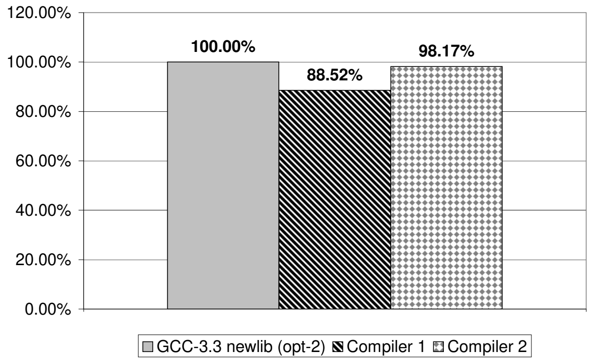

Figure 2: Average compiler results for objects

As can be seen, *Compiler 1* provides the best results and *Compiler 2* is still better than GCC. The gain in size achieved by *Compiler 1* is 11.48% and 1.83% by *Compiler 2* relative to the size of the objects compiled with GCC.

The same measurement is shown in more detail in Figure 3. It shows the effect of the C compilers separately for the different test programs. The sizes of the objects are summarized per test project (which is shown in parentheses after the project name at the bottom of the diagram in bytes).

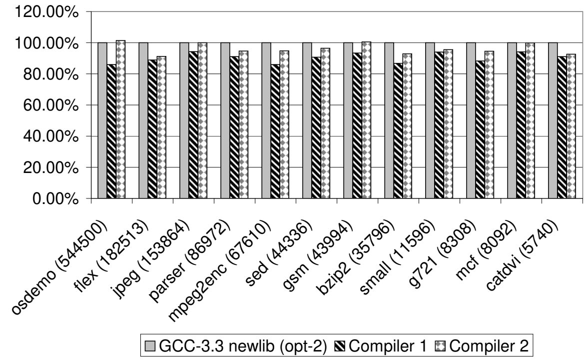

Figure 3: Individual compiler results for objects

The optimization capabilities of the compilers seems to be similar for each test project: *Compiler 1* produces the smallest code; the sizes of the result of *Compiler 2* are between the sizes of the output of *Compiler 1* and GCC.

4.2 Toolchain Results on Executables

We also investigated the difference in the generated code size of the executable files using the same environment and options as for the objects. We performed this comparison for standalone executable images, which means that apart from the application objects, the library code and the effectiveness of the linker is also incorporated in these number.

In Figure 4 the average result of executable sizes is shown. We computed the average values in the same way as for the objects, so they are simple sums of the program section sizes in the executables. Relative values are shown as well with respect to GCC.

We can observe that the ranking of the toolchains regarding code size in this comparison has not changed with respect to investigating only the compilers. The differences are, at the same time, more significant than in the case of objects comparison (about twice as much). Apparently, the reason for this is twofold: the tools use different implementations of standard C runtime libraries and the linkers may also behave differently. It is an open ques-

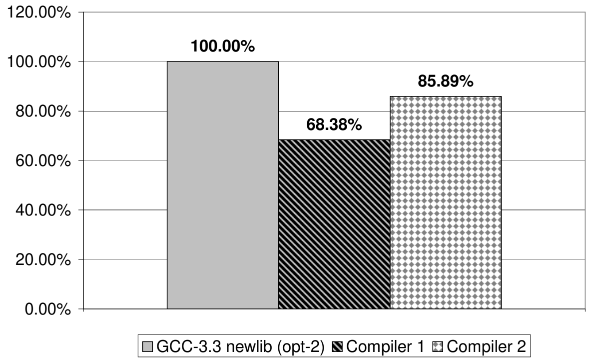

Figure 4: Average toolchain results for executables

tion whether the difference in the libraries causes a bigger difference or it is the linker that is responsible (e.g. by performing different optimizations at link time). Whatever the case, the comparison of the executables is not as a good measure of the toolchains as a comparison of the objects is a measure of the compilers, because the implementation of the libraries is also an important factor, which is included in the result.

In Figure 5 the same measurement is shown in more detail individually for the various executables. The sizes of the executables are summarized per test project (which is shown in parentheses after the executable name at the bottom of the diagram in bytes).

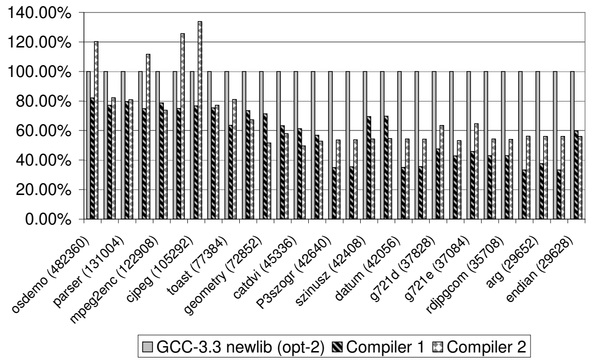

Figure 5: Individual toolchain results for executables

As can be seen, the ranking of the three toolchains does not always show the same order as in the average case, but we can see that *Compiler 1* is still in all but one cases much better than GCC. *Compiler 2* produced both the worse and the best results: there are cases when this tool had the largest code, but there are also cases where it seems to be the best tool.

5 Results for Linux Libraries

Apart from using as a cross-compiler generating standalone executable images, GCC is also widely used to generate programs for GNU/Linux. Hence we thought that it would be a good idea to investigate the sizes of the generated objects and executables in this case as well. In these experiments we used a GCC compiler configured for the `arm-linux-elf` target with the same environment and compiler options as for the standalone target (the only exception being that we needed to omit the `-ffunction-sections` option of GCC because it caused some problems when executing the programs on a Linux system). In this case we employed the commonly used GNU library *glibc* [2].

The Linux executables are not comparable with a standalone configuration (namely, with the GCC `arm-unknown-elf` target or with the two non-free compilers). This is because Linux uses shared objects that are linked at runtime to the executable (see Section 2.3). Nevertheless, objects should be comparable. Our results showed that the objects for Linux target have a smaller code size than objects for standalone target (by 8.35% with GCC 3.2.2 on our testbed). By examining the compiled objects we found that the size differences were primarily due to the different implementation of the library headers.

5.1 glibc vs. μClibc

Alas we could not find any other compiler toolchain (either free or non-free) that was able to generate for Linux target. Only the μClibc toolchain [7] could serve as a comparison basis. However it also uses the GCC compiler, so it really compares two implementations of the standard C runtime libraries.

We performed all measurements on the testbed and investigated the sizes of the objects and executables as well. We used GCC version 3.2.2 because the later versions (3.3 snapshots and the active development 3.4) are not supported by μClibc. With glibc- and μClibc-based toolchains we used the same compiler options that we found to be best for size with the standalone target (as described in Section 3). It is interesting to note that compiling the libraries using our combination of options brought a significant improvement in library size with respect to the default settings: 3.22% for glibc and 2.04%

for μClibc (computed for shared object binaries and not for static libraries).

An interesting observation was that the μClibc toolchain generally produces a slightly larger code size (1 or 2% at most) than GCC with glibc. We do not present the actual results here. Rather it is more interesting to look at the difference in the sizes of the actual libraries.

We measured the total code section sizes for all the generated library files. On average the μClibc library was smaller by 80.58% (1.94MB vs. 0.38MB) for the shared object binaries, and was smaller by 59.49% (1.59MB vs. 0.64MB) for static libraries counting simply the sum of all sections in all of the library files.

6 Conclusion : Improvements and Limitations

Assessing a compiler's effectiveness in optimizing for space poses a number of difficulties. Based on our measurement results presented in previous sections, we can say that the most reliable way is to compare the section sizes containing program code and data in objects rather than executables. This is because the implementation of the libraries is also an important factor: all tools work with their own implementation, and this difference is also included in the result.

We managed to narrow the gap between a high-performance non-free compiler and GCC 3.3 using our own set of compiler options from 15.71% to 11.48% measured on objects for a standalone target. However, this number is nearly double when we consider executables. This suggests that not only GCC needs improvement, but the associated libraries as well (in this case newlib).

Things get more complicated if we wish to compare toolchains configured for Linux target and not for standalone. This is because Linux uses shared objects that are linked at runtime. In this case the only reasonable thing is to measure the size of the corresponding libraries. For example, we found that the total size of μClibc,—an alternative library to glibc—is far less than glibc (only one fifth).

6.1 Improvement of Prerelease 3.3

In the previous sections we presented the results of measurements with the latest snapshot of GCC 3.3 version. We performed the same experiments with version 3.2.2 as well (which is the last official release at the time of writing) and found that prerelease 3.3 has improved slightly in terms of optimizing for space. In this section we summarize the results of our measurements of what are the exact improvements.

The average difference between object sizes generated by GCC 3.2.2 and the 3.3 snapshot configured for standalone (with newlib) is only 0.31%. With both configurations we used the best compiler options, where some options are new to 3.3 and therefore not present in measurements with 3.2.2 (see Section 3). Figure 6 shows the same separately for each program of the testbed.

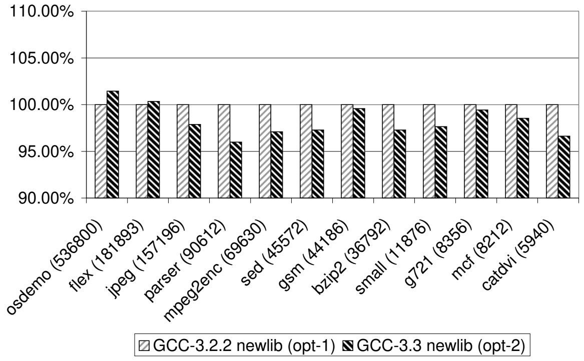

Figure 6: Improvement of GCC 3.3

Overall, no extraordinary improvement can be seen from this diagram and, in fact, the biggest program even shows that the older GCC generates smaller code. The difference is slightly larger in the case of executables; (it is 1.86% on average measured under the same conditions as for objects), which can also be attributed to the library code which is incorporated into the executable.

We also investigated the amount of improvement that can be achieved with Linux libraries. We prepared the glibc binaries using GCC 3.2.2 and 3.3 snapshot using the best options and found that with the new version the library was 0.95% smaller, which is similar to what we got for object sizes above. Figure 7 shows this improvement for each library component.

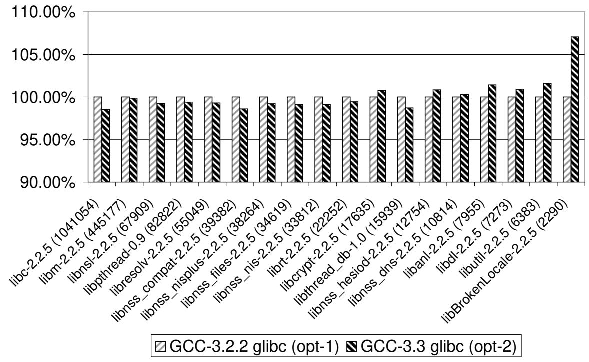

Figure 7: Improvement of GCC 3.3 measured on glibc

We made some investigations to found out what enhancements in GCC 3.3 caused this improvement in code size. There are a number of minor issues that could probably account for this, like some smaller optimizer improvements and target specific optimizations. However, we think that the major factor was the introduction of the new register allocation algorithm. In fact, by disabling `-fnew-ra` in GCC 3.3, the difference of 0.31% between 3.2.2 and 3.3 using the best options disappears and GCC 3.3 becomes to produce larger code by 0.29% on average!

6.2 Remaining Problems

By looking at the generated code in more depth, we managed to identify several weakpoints of GCC that could be improved in order to generate a more compact code. Another group of issues addresses GCC's limitations that are due to its architecture and logic of compilation. Some of them may not be solved or at least with very high effort. In the following we summarize the main issues for providing some starting point to future improvements.

Unit at a time compilation. GCC generally translates one function at a time and therefore it misses the opportunity of performing such optimizations that rely on seeing all functions of a compilation unit at the same time. With version 3.4 there was recently added the possibility for unit at a time compilation, but its utilization in optimization has not yet been fully achieved. If this feature is fully implemented in GCC, it would enable, for example, the sharing of global variables, the elimination of unused static functions, and the sharing of common data among functions (when the function-per-section option is not used).

More intelligent `-Os`. Generally, when `-Os` is turned on it means `-O2` with some additional optimization algorithms being implicitly enabled. In addition, any part of GCC can check for the state of this option. However, the semantics of this option could be further improved. First, a more careful selection of algorithms that need to be enabled could be implemented, similar to those proposed in Section 3. This could be further enhanced using the possibility for target-specific configuration of this switch. Furthermore, if `-Os` could act as an orthogonal option to other levels of optimization, it would offer for an even more flexible configuration.

Interprocedural optimizations. Due to the above-mentioned missing unit at a time compilation, no interprocedural optimization algorithms could be used. A number of existing algorithms could be extended to interprocedural operation, which would undoubtedly produce significant improvement, e.g. interprocedural dead-code elimination and redundant code elimination [1, 5]. Even some evidently redundant code constructs are currently generated by GCC. Consider, for example, the following code and notice that the call to function `foo` will be superfluously generated:

```
int a,b;
int foo(int x) { return x; }
void bar() {
  a = 1;
  b = foo(a);
}
```

Minor optimization issues. Here we list several minor issues that are related to some optimization algorithm (or are possibly specific for ARM target).

- The organization of loops is sometimes too complicated with redundant condition checking at higher optimization levels.
- The organization of the generated code for the `switch` statement can be made more optimal, especially when jump tables are used.
- RTL code generation from trees can be made more optimal than that for the current naïve preorder walk.
- Automatic function inlining does not seem to take into account when code size is the ob-

jective rather than speed. In this case only those functions should be inlined, which produce smaller code than calling the function.

- In ARM target, multiple variable load and save instruction are generated only for simple cases.

Library issues. Although the inadequacies of library implementations are not the subject of this article, we would like to remind the reader of the fact that the library headers indeed have some impact on the size of the generated code, which we elaborate in Section 2.3. Another interesting observation of ours was that a lot of space could be saved if some operators could be implemented by a library function call. For example, if integer division and modulo operators (`/` and `%`) would have a corresponding library function then for targets where these operations are not part of the instruction set, a simple call would be generated instead of the inline implementation of the division. Naturally, this would require that all library implementations provide such builtin functions for certain commonly-used operators.

6.3 Conclusion

We have seen that GCC is getting better and better with regard to code size. The latest version 3.3 (using an optimal combination of options) is only 11.48% worse than a high-performance non-free compiler. In Figure 8 we summarize the results of our measurements.

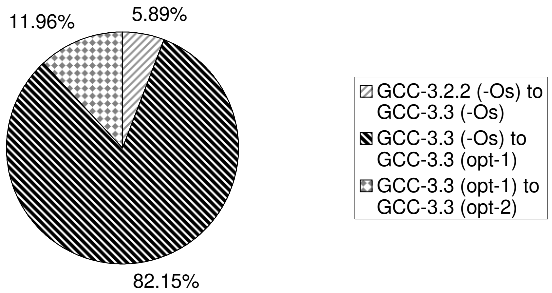

Figure 8: Summary of improvements

In this diagram we can observe (1) how much improvement version 3.3 brings with `-Os` only (0.3%), (2) the effect of a combination of options that we suggest over `-Os` measured on GCC 3.3 (4.15%) and (3) the effect of some new algorithms in GCC 3.3 (0.61%). These three constitute the total difference of 5.06% between GCC 3.2.2 with `-Os` and GCC 3.3 with *opt-2*.

Nevertheless there still are a number of issues—which we summarized in the previous section—that could make GCC's capabilities of optimization for space even better and this way shift its mainly academic use nowdays towards industry environments to become a serious competition to non-free commercial compilers.

7 Availability

The present document and related information including complete measurement data are available at

`http://gcc.rgai.hu/docs.php`

The homepage `http://gcc.rgai.hu/` aims to collect and maintain references to official GCC pages in connection with the ARM port.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers : Principles, Techniques, and Tools. Addison-Wesley Pub Co, 1985.

[2] Homepage of glibc. `http://www.gnu.org/software/libc/`

[3] Charles Leggett's benchmarks. `http://annwm.lbl.gov/bench/`

[4] Homepage of MediaBench. `http://www.cs.ucla.edu/~leec/mediabench/`

[5] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[6] Homepage of newlib. `http://sources.redhat.com/newlib/`

[7] Homepage of μClibc. `http://www.uclibc.org/`

[8] SPEC 2000 tests by Andreas Jaeger. `http://www.suse.de/~aj/SPEC/`

[9] SPEC 95 tests by Diego Novillo. `http://people.redhat.com/dnovillo/spec95/`

[10] Standard Performance Evaluation Corporation – spec. `http://www.spec.org/`