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.*

CSiBE Benchmark: One Year Perspective and Plans

Árpád Beszédes, Rudolf Ferenc, Tamás Gergely,
Tibor Gyimóthy, Gábor Lóki, and László Vidács
Department of Software Engineering
University of Szeged, Hungary
{beszedes,ferenc,gertom,gyimi,loki,lac}@inf.u-szeged.hu

Abstract

In this paper we summarize our experiences in designing and running CSiBE, the new code size benchmark for GCC. Since its introduction in 2003, it has been widely used by GCC developers in their daily work to help them keep the size of the generated code as small as possible. We have been making continuous observations on the latest results and informing GCC developers of any problem when necessary. We overview some concrete "success stories" of where GCC benefited from the benchmark. This paper overviews the measurement methodology, providing some information about the test bed, the measuring method, and the hardware/software infrastructure. The new version of CSiBE, launched in May 2004, has been extended with new features such as code performance measurements and a test bed—four times larger—with even more versatile programs.

1 Introduction

Maintaining a compact code size is important from several aspects, such as reducing the network traffic and the ability to produce software for embedded systems that require little memory space and are energy-efficient. The size of the program code in its executable binary format highly depends on the compiler's ability to produce compact code. Compilers are generally able to optimize for code speed or code size. However, performance has been more extensively investigated and little effort has been made on optimizing for code size. This is true for GCC as well; the majority of the compiler's developers are interested in the performance of the generated code, not its size. Therefore optimizations for space and the (side) effects of modifications regarding code size are often neglected.

At the first GCC summit in 2003, we presented our work related to the measurement of the code size generated by GCC [1]. We compared the size of the generated code to two non-free compilers for the ARM architecture and found that GCC was not too much behind a high-performance ARM compiler, which generated code about 16% smaller than GCC 3.3. However, at the same time we were able to document several problems related to code size as well, and more importantly we have demonstrated examples where incautious modifications to the code base produced code size penalties. At that time we had the idea of creating an automatic benchmark for code size.

To maintain a continuous quality of GCC generated code, several benchmarks have been used for a long time that measure the performance of the generated code on a daily basis [4]. However this new benchmark for

code size (called CSiBE for GCC **C**ode **Si**ze **BE**nchmark) was launched only in 2003 [2]. This benchmark has been developed by and is maintained at the Department of Software Engineering at the University of Szeged in Hungary [3]. Since its original introduction CSiBE has been used by GCC developers in their daily work to help keep the size of the generated code as small as possible. We have been making continuous observations on the latest results and informing GCC developers of any problems when necessary.

The new version of CSiBE, launched in May 2004, has been extended with new features such as code performance measurements and a test bed—four times larger—with even more versatile programs. The benchmark consists of a test bed of several typical C applications, a database which stores daily results and an easy-to-use web interface with sophisticated query mechanisms. GCC source code is automatically checked out daily from the central source code repository, the compiler is built and measurements are performed on the test bed. The results are stored in the database (the data goes back to May 2003), which is accessible via the CSiBE website using several kinds of queries. Code size, compilation time, and performance data are available via raw data tables or using appropriate diagrams generated on demand.

Thanks to the existence of this benchmark, the compiler has been improved a number of times to generate smaller code, either by reverting some fixes with side effects or by using it to fine tune some algorithms. In the period between May 2003 and 2004 an overall improvement of 3.3% in code size of actual GCC mainline snapshots was measured (ARM target with `-Os`) which, we believe, CSiBE also has contributed to.

In this paper we summarize our experiences in designing and running CSiBE. Section 2 overviews the system architecture while in Section 3 we give some examples of our observations and other people's benefits using CSiBE. Finally, we give some ideas for future development in Section 4.

2 The CSiBE system

In this section we overview the measurement methodology. We provide some details about the test bed, the measuring method, and the hardware/software infrastructure. Although the CSiBE benchmark is primarily for measuring code size, it provides two additional measurements: compilation speed, and code speed (for a limited part of the test bed). GCC source code is checked out daily from the CVS, the compilers are built for the supported targets (*arm/thumb*, *x86*, *m68k*, *mips*, and *ppc*) and measurements are performed on the CSiBE test bed. The results are stored in a database, which is accessible via the CSiBE website using several kinds of queries. The test bed and the basic measurement scripts are available for download as well.

2.1 System architecture

In Figure 1 the overall architecture of the CSiBE system is shown.

CSiBE is composed of two subsystems. The *Front end servers* are used to download daily GCC snapshots and use them for producing the raw measurement data. The *Back end server* acts as a data server by filling a relational database with the measurement data, and it is also responsible for presenting the data to the user through its web interface. The back end server together with the web client represents a typical three-tier client/server system. It serves as a data server (Postgres), implements various query logics and supplies the HTML presentation. All the servers run Linux.

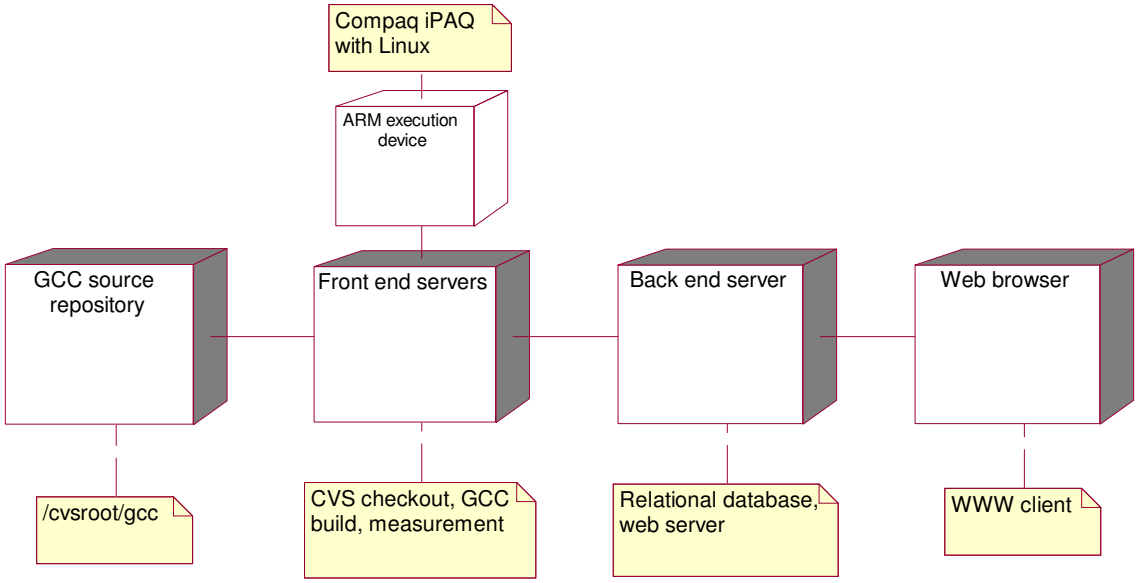


Figure 1: The CSiBE architecture

2.2 Front end servers

The core of CSiBE is the *offline CSiBE benchmark*, which consists of the test bed and required measurement scripts. This package is downloadable from the website, so it can also be used independently of the online system. The front end servers utilize this offline package as well.

The online system is controlled by a so-called *master phase* on the front end servers, which is responsible for the timely CVS checkout, compiler build, measurements using the offline CSiBE, and uploading the data to the relational database.

Hardware and software

The actual setup of the front end servers is flexible. At present, it is composed of three Linux machines, one used for CVS checkout that is shared with other university projects, and two dedicated PCs for the other front end phases. These two PCs are really siblings, having the same hardware and software parameters that are summarized below:

- Asus A7N8x Deluxe
- AMD AthlonXP 2500+ 333FSB @ 1.8GHz
- 2x 512MB DDR (200MHz)
- 2x Seagate 120GB 7200rpm HDD
- Linux kernel version 2.4.26, Debian Linux (woody) 3.0

These two servers are capable of sharing the measurement tasks (like separating them by branches) and, in this way, we also have a backup possibility in case of some unexpected server failure. These two servers are also used for measuring the performance of code generated for the x86 architecture. We are working on adding performance measurements for the ARM architecture as well, which will be made on a Compaq iPAQ device with the following main parameters:

- iPAQ H 3630 with StrongARM-1110 rev 8 (v4l) core
- 16M FLASH, 32M RAM
- Familiar Linux, kernel version 2.4.19-rmk6-pxa1-hh30

Compilers and binaries measured

We measure daily snapshots of the GCC *mainline* development branch (previously the *tree-ssa* too) along with several release versions that serve as baselines for the diagrams. These are the following GCC versions: *2.95.2.1*, *3.2.3*, *3.3.1*, and *3.4*.

The compilers are configured as cross-compilers for the supported targets. We employ standalone targets for use with the *newlib* runtime library for code size and compilation time measurements, and Linux targets with *glibc* for execution time. At present, *binutils v2.14*, *newlib v1.12.0*, and *glibc v2.3.2* are used.

When we measure code size and compilation time, we do not include linking time and code size of the executable. Furthermore, only those programs that meet certain requirements are used for performance measurements. These are the following:

- The project produces at least one executable program
- The source files are not preprocessed
- The execution environment must not contain any special elements
- The execution time is measurable (i. e. it is not too short and not too long)

CVS checkout

Snapshots of GCC source code are retrieved from the CVS daily at 12:00:00 (UTC). The complete code base is retrieved once a week on Mondays and on the other days only the differences are downloaded.

Configuration

The *Binutils* package is configured with no extra flags, while *newlib* is configured with the only extra flag that enables the optimization for space: `-enable-target-optspace`. We do not build *glibc*, rather we use the stock binaries. Finally, GCC is configured with the following. The common flags are `-enable-languages=c -disable-nls -disable-libgcj -disable-multilib -disable-checking -with-gnu-as -with-gnu-ld`. Furthermore for compilers using the *newlib* library, the additional flags are `-with-newlib -disable-shared -disable-threads` and for *glibc* we also use `-enable-shared`.

Compilation

A simple `make` was used to build *binutils* and the libraries once only, and the same is used for each GCC snapshot as well.

Measurement

The code size is measured using the program `size`. The final result is the sum of the first two columns of the output of the command. This means that only program code and constant and initialized data sizes are incorporated into the final values.

Compilation time and code execution speed are measured three times per object and per test case, respectively. These times are measured with the program `/bin/time` in user mode. For both compilation and execution times all queries through the web will provide a time value that is the median of the three values. While compilation and execution times are being measured only vital processes are running on the machine.

The results of the measurements are stored in simple files in CSV format (comma separated values) for further processing. These files are also the final outputs of the offline CSiBE.

2.3 The test bed

The test bed consists of 18 projects and its source size is roughly 50 MB. When compiled, it is about 3.5 MB binary code in total. The test bed consists of programs of various types such as media (gsm, mpeg), compiler, compressor, editor programs, preprocessed units. Some of the projects are suitable for measuring performance and constitute about 40% of the test bed.

In the latest version of the test bed we added some Linux kernel sources as well. With this aim in mind, we started with the S390 platform and turned it into a so-called "testplatform." On this platform we replaced all assembly code with stubs and left only C code for the important Linux modules (kernel, devices, file systems, etc.)

The test bed is composed of two parts, one for the test programs and measurement scripts, and the other consisting of the test inputs for the executable projects. This separation was carried out so the user would be able to add many different test cases. The test cases were selected to represent one typical execution of the program as our goal was not to attain a good coverage of the program. In some cases the same input is given to a program several times, while in other cases the same program is executed with different inputs. The total size of the test inputs is currently about 60 MB.

In the table in Figure 2 some statistics about the test projects are given. We listed the number of source files, size of the source code in bytes, number of objects, total size of objects as measured using CSiBE for GCC 3.4, i686 and `-O2`, and the number of executable programs for each project.

2.4 Back end server

User queries through the CSiBE website are processed using PHP scripts, from which the necessary SQL queries are composed. The data retrieved from the database is then presented on the HTML output in data tables, bar charts, and timeline diagrams.

The central repository in which the measured data are stored is a relational database (implemented using Postgres). The database stores the measurement results along with the time stamp of the measurement and various entities such as the compiler and library version, compiler flags and measurement type. The version of the test bed is also associated with each result, which allows it to store the results of different test beds consistently. If a query is made that spans different test bed versions this can be easily displayed on the diagrams.

The last phase in the online CSiBE benchmark is the presentation on the website. The CSiBE pages provide quick and easy access to the most important measurements like the latest results in a timeline diagram or more elaborate query possibilities. Extensive help is provided for each function, making CSiBE simple to use. In Figure 3 the opening page can be seen.

There are several ways of retrieving the re-

Project	*# Src.*	*Src. bytes*	*# Obj.*	*Bin. bytes*	*# Exec.*
`bzip2-1.0.2`	11	242,034	9	80,112	2
`cg_compiler_opensrc`	42	813,343	22	148,838	—
`compiler`	9	202,938	6	27,928	1
`flex-2.5.31`	33	658,799	22	240,206	1
`jikespg-1.3`	29	978,833	17	267,712	1
`jpeg-6b`	81	1,119,991	66	156,078	3
`libmspack`	40	319,611	25	76,506	—
`libpng-1.2.5`	21	859,762	18	128,941	2
`linux-2.4.23-pre3-testpl...`	2,430	34,238,976	271	993,815	—
`lwip-0.5.3.preproc`	30	928,538	30	86,486	—
`mpeg2dec-0.3.1`	43	461,047	29	62,873	1
`mpgcut-1.1`	1	28,889	1	29,845	—
`OpenTCP-1.0.4`	40	545,358	22	38,221	—
`replaypc-0.4.0.preproc`	39	1,692,413	39	64,221	—
`teem-1.6.0-src`	370	2,786,644	293	1,210,365	2
`ttt-0.10.1.preproc`	6	311,311	6	19,049	—
`unrarlib-0.4.0`	4	93,894	3	16,339	—
`zlib-1.1.4`	27	305,136	14	42,422	1
Total	**3,256**	**46,587,517**	**893**	**3,689,957**	**14**

Figure 2: CSiBE test bed statistics

sults. One is *Summarized queries*, which provides instant access with a click of a button to all kinds of results (code size, compilation time, and code performance) for a selected target architecture. On the *Latest results* pages the last few days or weeks can be observed in several ways: timeline, normalized timeline (the various kinds of data are shown as normalized to the last value), a comparison of different targets, and raw number data. The *Advanced queries* pages provide the possibility of retrieving the data in any desired combination; one can compare any branch and target with any other combination and timeline diagrams for arbitrary intervals. Baseline values of major GCC releases are also available for most queries, which can be optionally selected for the diagrams.

All queries can be performed by a series of selections from drop-down lists like the selection of targets, branches, and optimization switches. The results can be displayed in a diagram (Figure 4a), in a bar chart (Figure 4b), or as raw data tables. The resulting latest timeline diagrams are supplied with two automatically generated links that can be copied for further reference. The *Static URL* link will always give the same diagram since all query parameters are converted to absolute time stamp values, while the *Reference URL* link supplies the actual query parameters at the time of usage, which gives values relative to the actual time.

3 Experiences

CSiBE has been quickly accepted by the community. Patches with references to its usage started to appear only after 2 months. At present we have 47 hits per day on average and a total of 193 downloads of the offline benchmark. A good thing about its introduction is that more and more GCC developers

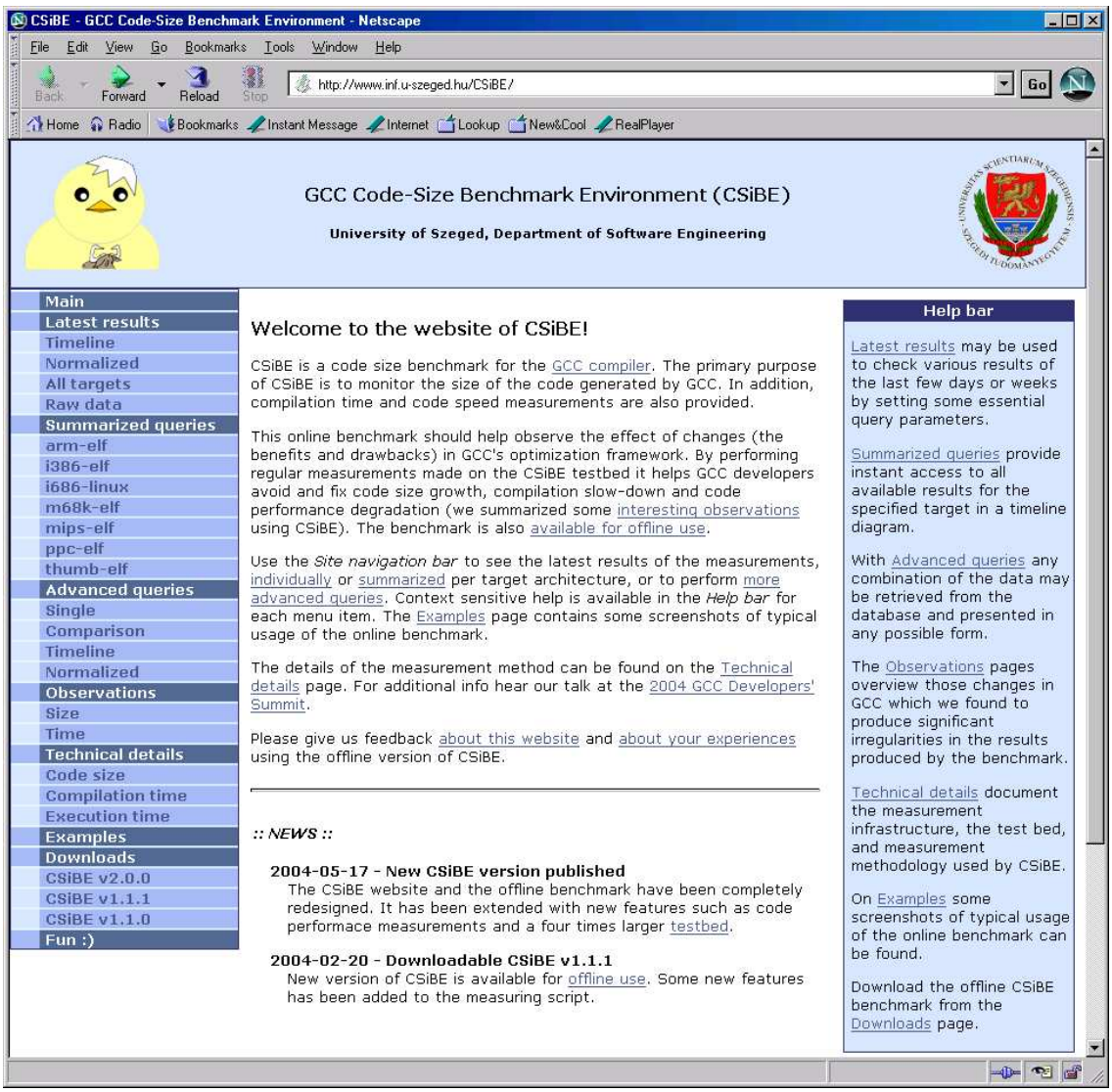
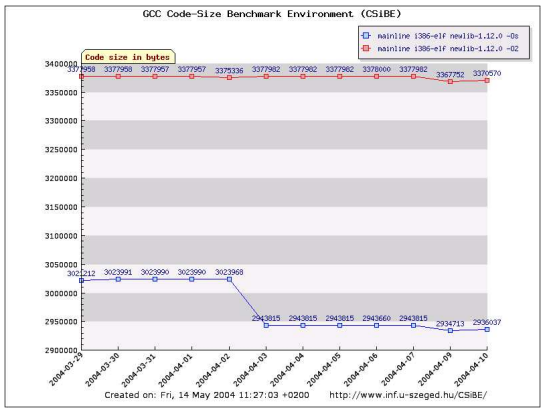


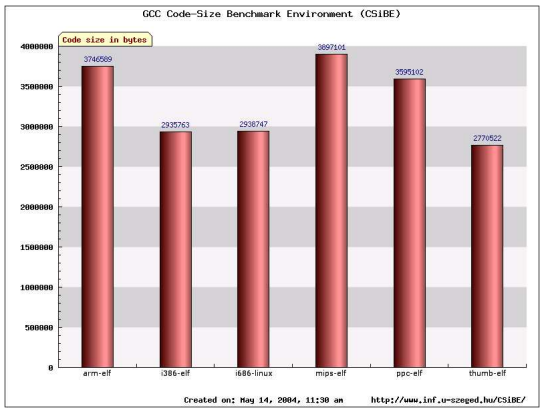
Figure 3: CSiBE website

seem to be using CSiBE in their daily work to check how their modifications affect the code size. Some people are developing patches to decrease code size, and the effect is measured with CSiBE, while others verify whether other modifications affect code size or not. Thanks to CSiBE, in 4 cases a patch was reverted or improved because of its negative effect on code size. These statistics suggest that the developers are starting to focus not only on code efficiency, but its size as well. We have been following the activity on the `gcc-patches` mailing list and found that more and more people are referring to CSiBE as a reference benchmark for code size (54 e-mails).

Our group has also contributed to the overall improvement of code optimization for size, because we are carrying out continuous observations of the results produced by CSiBE, of which the important ones are documented on the website. Where possible we also suggest a possible cause of any anomalies seen in the latest diagrams, and take steps to draw the attention of the community to the problem. In



(a) Timeline



(b) All targets

Figure 4: Diagram examples

the following we offer some examples of our observations and successful participations:

- On August 31 in 2003 a patch was applied to improve the condition for generating jump tables from switch statements by including the case when optimizing for size. This caused a code size reduction on all targets. The threshold value was determined based on the CSiBE statistics.
- In September 2003 unit-at-a-time compilation was enabled in mainline, which resulted in major code size improvement for most targets.
- A patch related to constant folding done in October 2003 increased the code size for all targets. Several days later another patch was used to disable some features when optimizing for size.
- A significant code size increase was measured on October 21, 2003 on ARM architecture when optimizing for size due to a patch that allows factorization of constants into addressing instructions when optimizing for space. One week later the patch was reverted.
- In January 2004 a patch saved code size on ARM with `-Os` but introduced a new bootstrap failure.
- A patch on April 3, 2004 saved about 1% of code size for most targets. The patch inlines very small functions that usually decrease the code size when optimizing for size.

4 Conclusion and future plans

In this paper we overviewed GCC's code size benchmark, CSiBE. We presented the overall architecture, the test bed and the measuring method. Although it primarily serves as a benchmark for measuring code size, other parameters such as compilation time and code execution performance are also part of the regular measurements. We offered some examples of where GCC benefited from using the benchmark, and pointed out that, in recent years, a general interest towards code size has increased among GCC developers. As a result of this, GCC mainline improved about 3.3% in terms of generated code size between May 2003 and May 2004 (measured with CSiBE test bed version 1.1.1 for the ARM target and `-Os`).

We plan to continue our work with CSiBE and

hence we welcome users' comments and suggestions. Some of the targets were added after user requests, and the bigger test bed in the latest CSiBE version is also composed of programs based on the demands of those who contacted our team. In the future we will try to follow the real needs of the GCC community, those of the developers and users.

One of the straightforward enhancements of CSiBE might be to introduce new targets and development branches, should there be an interest in it by the community. As long as the available hardware capacity permits (the measurement of one day's data currently takes about 5 hours), we may extend the test bed with new programs, should it prove necessary.

Another idea of ours for enhancing the online benchmark is to allow users to upload, via the web interface, measurement data they produced offline into the central database. This would be interesting in cases where a developer makes use of the offline benchmark to measure a custom target or examine code performance with different inputs.

5 Availability

The online CSiBE benchmark can be accessed at

`http://www.inf.u-szeged.hu/CSiBE/`

From here the offline version can also be downloaded.

Acknowledgements

The CSiBE team would like to thank all those GCC developers who helped us develop the benchmark with their useful comments and constructive criticisms.

References

[1] Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Optimizing for space: Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers' Summit*, pages 7–20, May 2003.

[2] Department of Software Engineering, University of Szeged. GCC Code-Size Benchmark Environment (CSiBE). `http://www.inf.u-szeged.hu/CSiBE`.

[3] Department of Software Engineering, University of Szeged. Homepage. `http://www.inf.u-szeged.hu/tanszekek/szoftverfejlesztes/starten.xml`.

[4] The GNU Compiler Collection. GCC benchmarks homepage. `http://gcc.gnu.org/benchmarks`.