GCC Summit 2003

Optimizing for Space

Measurements and Possibilities for Improvement

Árpád Beszédes, Tamás Gergely,
Tibor Gyimóthy, Gábor Lóki and László Vidács

University of Szeged, Hungary

Motivation

- Command-line option -Os is used to turn on optimization for space rather than (or in addition to?) speed
- However it seems that space optimization was often neglected
  - improvement of GCC wrt. performance is tracked, but benchmarking code size is “not popular”
- A general investigation would be useful on “how this switch works”
- E.g. we achieved ~5% smaller code size only by using some additional -f switches

Benchmarking speed

- Performance of code (speed) is primary for many (general) application areas
- It is constantly benchmarked from release to release, see e.g.:
  - SPEC 95 tests by Diego Novillo http://people.redhat.com/dnovillo/spec95/
  - SPEC 2000 tests by Andreas Jaeger http://www.suse.de/~aj/SPEC/
  - Charles Leggett’s benchmarks http://annwm.lbl.gov/bench/

Benchmarking space

- Space is also important in a certain class of applications
  - embedded systems
  - mobile, control, etc.
  - energy-saving is also important and is related to code size as well
- We are not aware of any continuous benchmarking activity regarding code size
- However it would be useful
  - e.g. we measured 0.016% degradation between GCC 3.3 prerelease snapshots 2003-03-24 and 2003-04-07

Contributions of this work

- Compared GCC with two non-free ARM cross-compiler toolchains
- Measured how GCC evolved from release 3.2.2 to (prerelease) version 3.3
- Compared two runtime libraries for Linux
- Identified some weakpoints regarding code size
- As a side-effect: new combination of command-line options on top of -Os

Measurement method

- We composed a test suite
  - parts from SPEC, MediaBench, GNU applications
- We established an environment that is:
  - automated for each toolchain configuration, producing code sizes and assembly code as well
  - able to execute the test programs
- We used several custom tools to collect the data and convert it into spreadsheet documents

Toolchains assessed

- Source language is C
- Target architecture is ARM (32-bit)
- Two types of target code: standalone and Linux (`arm-elf` and `arm-linux` machines)
- GCC versions
  - ☐ 3.2.2 and 3.3 prerelease snapshot 2003-04-14 (with newlib 1.10.0 and binutils 2.13) as standalone
  - ☐ 3.2.2 and 3.3 prerelease snapshot 2003-04-14 (with glibc 2.2.5) for Linux
- Two non-free compilers for ARM architecture configured for standalone targets

Test suite

- It is composed of programs for different purposes:
  - small toy programs
  - parts of MediaBench (benchmark for multimedia applications)
  - parts of some SPEC benchmarks (2000 cpu, int, …)
  - some GNU programs
- Largest program source code is 1,9MB
- All programs produce one or more executable programs

Environment

- Our environment contains various makefiles and shell scripts that enable automated:
    - ☐ build of GCC from different sources and configurations
    - ☐ measurement
    - ☐ disassembly and assembly generation at function level
    - ☐ execution (for validation)
        - using a simulator for standalone target
        - and a Linux-based handheld device for Linux executables
- We plan to make it publicly available

Method

- We measure binary machine code size for objects and executables (ELF and COFF)
- Assessment method was not trivial; problems:
  - objects vs. executables?
  - standalone vs. Linux programs?
  - which sections of elf (coff) files to consider?
  - tools to extract the relevant data

Objects vs. executables

- When objects are measured the effectiveness of the compiler proper is assessed
  - library implementation still has some impact because of library headers
  - sizes can be measured at function level
- When executables are measured the effectiveness of the whole compiler toolchain is assessed (including libraries and the linker)
  - GCC libraries were compiled with the same flags as the test programs

Standalone vs. Linux

- Although the same compiler toolchain configuration was used,
- The objects are quite different in the two cases:
  - because of the different libraries and other issues
- Executable sizes are not inter-comparable
  - Linux uses shared objects that are linked at runtime
- Regarding Linux we were not able to find competitive compiler toolchains
- Therefore two libraries were compared: glibc and μClibc

Sections

- Obviously the size of the binary file is not relevant
- Sections contain different kinds of data, we examined those that are directly used by the program:
  - executable code, constant or read-write program data
  - others are not counted (such as debug sections, symbol tables, etc.)
- We simply summarized the sizes of these sections
  - different object formats and compilers have different layout and naming conventions
  - the parts can be intermixed (e.g. executable code can contain embedded data)
- We experimented also with “only read-only sections” combination but this is not so fair
  - because of different handling of initialized read-write data sections in the case of ELF and COFF files

Measurement tools

- The default makefile target in the environment is to produce raw measurement data:
  - ☐ using “objdump” and “size” (from binutils)
  - ☐ the so obtained raw data is given to small tools to separate functions
    - if data is shared this may not be precise (but we can use “one function-per-section” option)
  - ☐ csv files are produced that can be further processed using e.g. spreadsheet editors

Best options

- One “side-result” of our work is a combination of GCC options that improve the “only- Os” results
- Generally, -Os enables or disables certain optimization passes/algorithms in toplev
  - any other part can also check for its status
- The “smallest code size” objective can be influenced by combining other (mainly -f) options
- We determined a set of such options by experimentation
  - we started with a base set of options (from GCC manual)
  - all other -f options were individually tried and added if made gain
  - some options were later removed because its combined effect with the other options was worse
- We achieved 4.78% overall improvement wrt. “only- Os”

Best options (cont.)

- The “winning set” is the following:

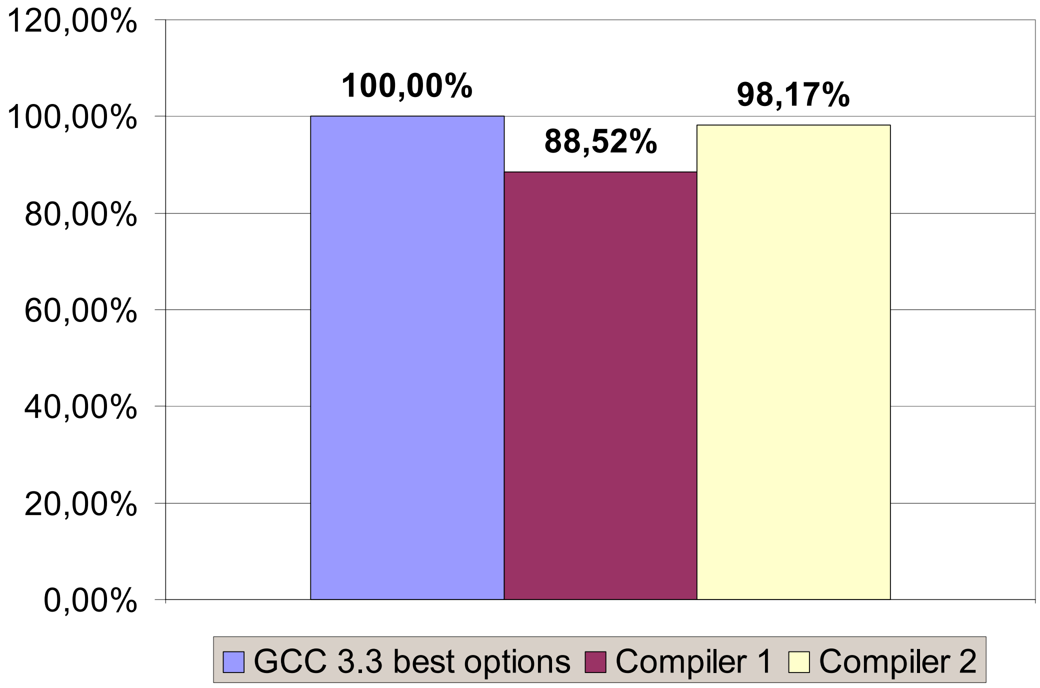
-Os	-mno-apcs-frame
-fomit-frame-pointer	-ffunction-sections
-fdata-sections	-fno-force-mem
-fno-force-addr	-fno-inline-functions
-fnew-ra	-fbranch-probabilities
-finline-limit=1	-fno-schedule-insns
-fno-optimize-sibling-calls	-fno-if-conversion
-fno-thread-jumps	-fno-hosted

Best options (cont.)

- Notes:
  - -mno-apcs-frame is ARM specific (-mno-thumb-interwork was also used)
  - -fnew-ra and -fno-if-conversion are not present in GCC 3.2
- There are 170+ options starting in- f
- Many of them had some problems (details in paper):
  - combined use was worse
  - parameterized options could not be tried with all possible values
  - invalid generated code
  - irrelevant options
- Options for the linker: -O2 --gc-sections –relax --no-whole-archive

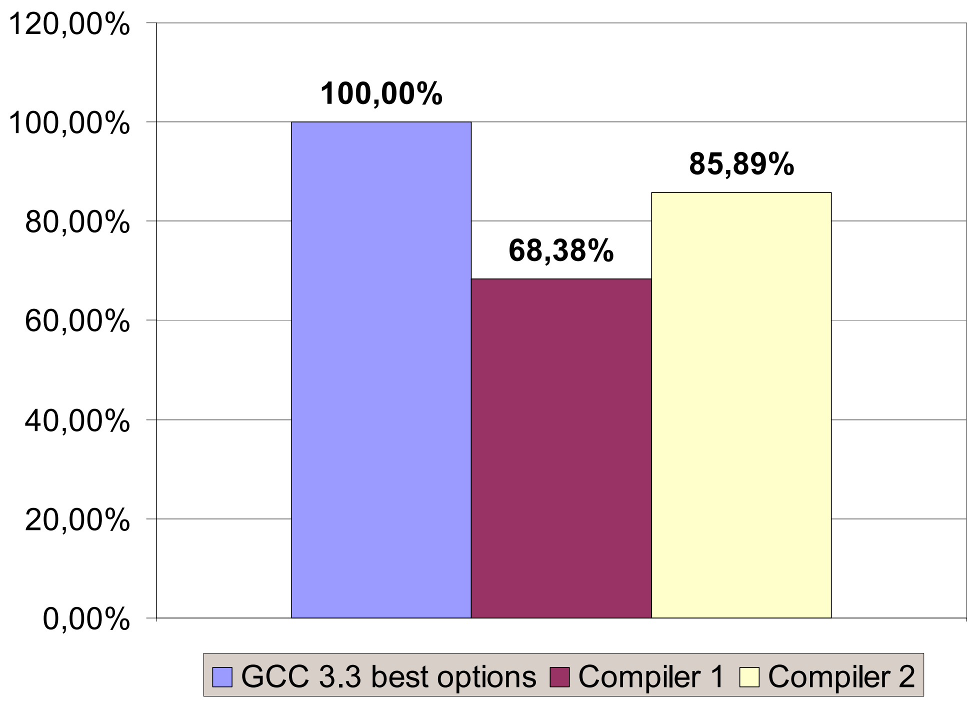
Results for standalone

- Compiler results on objects
  - sum of the sizes of all objects of the test programs, relative to GCC with best options



Results for standalone (cont.)

- Toolchain results on executables
  - ☐ sum of the program section sizes in executables, relative to GCC with best options



Linux

- GCC for `arm-linux-elf` target with best options (w/o -ffunction-sections)
- Linux executables are not comparable with standalone configuration's executables
  - (and with the non-free compilers)
- We were not aware of any other compiler toolchain for Linux target
- Objects are comparable but this result is not very important:
  - objects for Linux are 8.35% smaller with GCC 3.2.2

glibc vs. µClibc

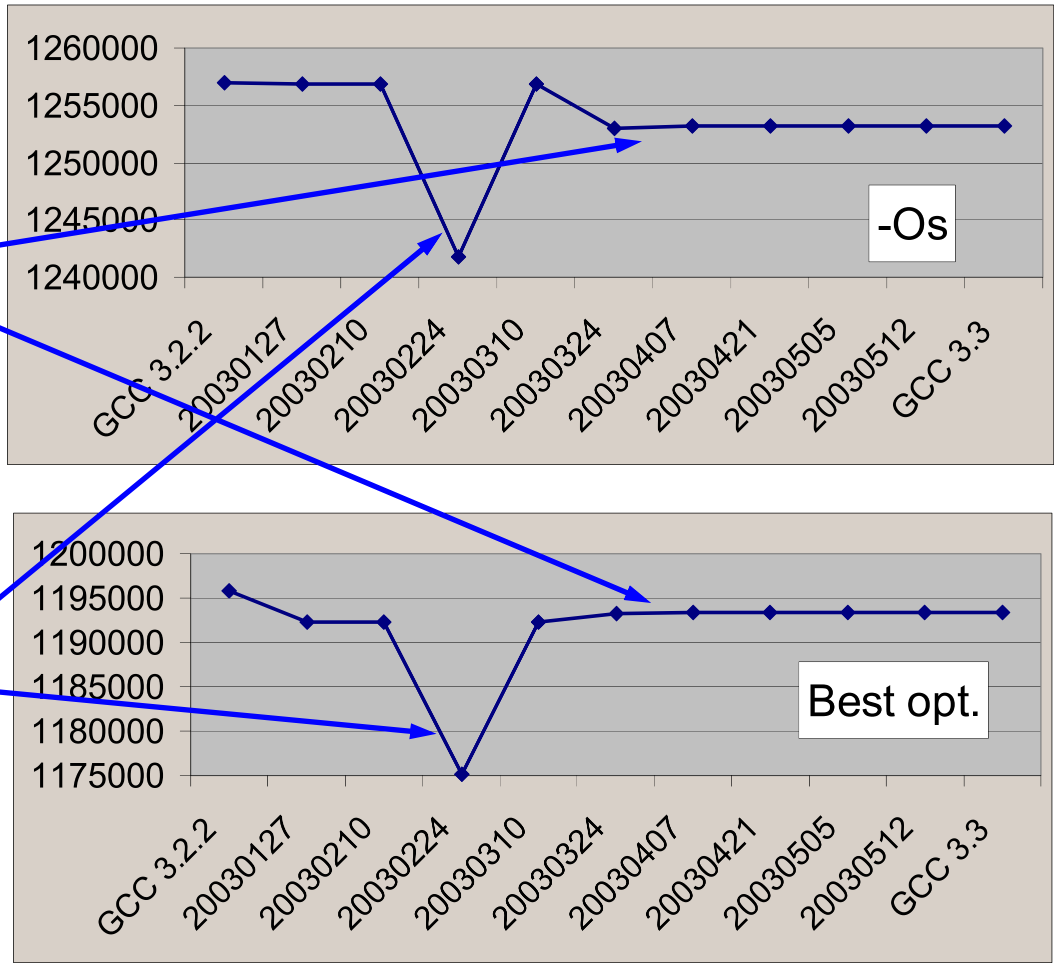
- We used 3.2.2 version because later versions are not supported by µClibc
- We used the same options with best switches
  - they brought 3.22% on glibc and 2.04% on µClibc for shared objects version binaries
- Interesting: µClibc generates 1-2% larger objects
- µClibc library (section sizes in all generated library files) is only a fraction of glibc:
  - 19.42% (0.38MB vs. 1.94MB) for shared object binaries
  - 40.51% (0.64MB vs. 1.59MB) for static libraries

Improvements in GCC 3.3

- We performed all experiments with GCC version 3.2.2 and GCC 3.3 prerelease snapshot (at the time of writing)
  - some new switches could not be used
- 3.3 has improved slightly in terms of optimizing for space
  - only 0.31% on object sizes for standalone target; 1.86% on executables; 0.95% on glibc
- The main factor is the introduction of the new RA algorithm
  - by disabling new-ra in GCC 3.3, it will even produce larger code by 0.29% on average!

Timeline of 3.3 prereleases

- Total size of objects on our test suite
- Notice the degradation at the beginning of April
- copy_loop_headers patch (introduced on 2002-10-06 by R. Henderson, PR optimization/2960) should not be dropped!



Existing problems in GCC

- We looked at the generated code in more depth
  - individually at functions
- Some weakpoints could be improved, while others are due to basic GCC architecture/compilation policy

1. More intelligent -Os
   - the semantics of this option could be improved:
     - more careful selection of algorithms (see our switches!)
     - target-specific configuration of the switch
   - should act as an orthogonal option to other levels (e.g. -O2 -Os)

Existing problems (cont.)

2. Unit at a time compilation
   - ☐ one-function-at-a-time compilation policy was recently extended in GCC 3.4 (-funit-at-a-time)
   - ☐ when fully implemented, it will enable further optimizations for space, e.g.:
     - sharing of global variables
     - elimination of unused static functions
     - sharing of common data among functions

Existing problems (cont.)

3. Interprocedural optimizations
   - if unit at a time is accessible, many existing algorithms could be extended:
     - interprocedural dead code elimination
     - interprocedural redundant code elimination, etc.
   - in some cases GCC is really “stupid”:

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

call to `foo` could be optimized out

Existing problems (cont.)

4. Minor issues
   - ☐ organization of loops at higher optimization levels
   - ☐ organization of the switch statement (jump tables!)
   - ☐ RTL generation from tree could be more optimal (currently simple preorder)
   - ☐ automatic function inlining when optimizing for space

Library issues

- The inadequacies of library implementations are not the current subject
- However library header implementations have some impact on code size of objects
  - ☐ e.g. if newlib is used `stderr` macro expands to a pointer to struct member which is a pointer
  - ☐ when compiled, this takes several instructions
- lib1funcs problem
  - ☐ in the paper we were not aware of lib1funcs for the implementation of some operators (e.g. / and %)
  - ☐ GCC still generates inline implementation when the divisor is a constant, why?

Summary

- Lessons learnt:
  - GCC is only 11.48% worse than a high-performance non-free compiler
  - switches matter a lot (~5%)
  - library implementation also counts
  - speed ↔ size tradeoff exists, but not everywhere
  - GCC 3.3+ improves, but the lack of size benchmarks can cause that effect on code size will degrade
- Planned enhancements:
  - adding more test projects
  - contributing the environment to the community
- http://gcc.rgai.hu
  - in our group four researchers and several students are working on the possibilities to improve GCC regarding code size optimization (we have some patches)