Gcov on an embedded system

Holger Blasum, Frank Görgen, Jürgen Urban
SYSGO AG Klein-Winternheim

(Workshop: GCC for Research in Embedded and Parallel Systems, Brasov 16 Sept 2007)

Abstract

We describe how gcov (based on version gcc 3.4.4) was used in an embedded system (PowerPC architecture). Unlike the gcov kernel analysis of the linux test project, our modifications do not use any file system access during data collection time. It is also shown how a one-line modification may lead towards a conservative estimation of coverage in a multithreaded setting.

1. Coverage analysis and gcov

Coverage analysis tests which lines of code have been run through and can be used for verification of code paths [4, 16]. Coverage is part of industry norms for software development, e.g. for the field of avionics see [14, p. 33-34]. Gcov is the coverage analysis infrastructure provided with the gcc compiler [2, 12], it has been developed since 1990, it also can be used to analyze assembly files [5], and there are even well-maintained derivations to analyze a running linux kernel [8, 9, 13].

One can say that gcov has been quite stable for the last three years (since gcc version 3.4 where the gcov IO framework had been "completely remangled" [15]). However, some of the more extensive descriptions such as [5, 9] refer to earlier versions of gcov; so it seems justified to start with some general introduction:

1.1 Terminology

1.1.1 Arcs and blocks

It is common usage [1, p. 528] that a *basic block* (bb) consists of one or more statements that are called atomically (that is there is no branching between them) during all runs of a program. An *arc* (alternatively: edge; branch) is a pair of basic blocks (source bb, destination bb) that records which bb jumps to which other bbs.

1.1.2 Scope of gcov coverage

The coverage of gcov is based on arcs (branch coverage), from this also the coverage of blocks (line or statement coverage) can be inferred. The expression `(i==0 || (j == 1 && p->j == 10))` is given in [9] to illustrate a limitation of gcov coverage: gcov coverage will show how often the entire expression gets evaluated to true or false but it will not show the outcome of each subexpression that led to that expression. (Gcov's coverage is also classified as "condition coverage" in [4]; the more coarse classification of [16] also excludes gcov from "decision coverage".)

1.2 Gcc source code and versions

In this document we refer to the gcov infrastructure of gcc as of version 3.4.4; apparently since then up to the current gcc 4.3.0 prerelease (checked out on 30 June 2007) the gcov framework has been very stable. In case of ambiguity, file and function names refer to files in `gcc-3.4.4/gcc`.

1.3 The three steps of the gcov user interface

We briefly define three steps in the gcov framework (for a more gentle introduction see [6]):

- compilation phase ("`gcc -O0 -o hello -fprofile-arcs -ftest-coverage hello.c`")
- data collection and extraction phase ("`./hello`" is the collecting binary)
- reporting phase ("`gcov -a hello.c`")

1.4 Code coverage during the three phases

If one wants to modify the framework, it is important first to understand which files one has to tweak for controlling which phase, that is to understand the "coverage" of gcov itself (in an informal sense). To begin with, the gcov framework resides in the files `coverage.c`, `gcov.c`, `gcov-io.c`, `libgcov.c`, `profile.c` (and header files). Instrumenting all functions shows the following division of labor:

1.4.1 Compilation phase

During the compilation phase functions in `toplev.c` call functions in `coverage.c` and `profile.c`, these call functions in `gcov-io.c` (`IN_LIBGCOV` is not set). In particular, the collecting binary's data structure holding counter memory that is made available by `gcov_init` is generated by `build_gcov_info` of `coverage.c` which also for each compilation unit write-opens a `gcno` ("gcov notes") file which will be consulted during the reporting phase. The gcno file is to record the structure of the flow graph containing the arcs and blocks during a compilation (more details on the format in section 1.5.1, this is done in `profile.c`). Correspondingly, a few lines later down the code the meticulous `profile.c` also for each of the arcs defined in the gcno file calls `insert_insn_on_edge` which inserts counter adding instructions. One also can see by the inspection of object files that both on ppc and i386 simply for each instrumented arc a counter of two consecutive 32-bit integers is included within the assembly file, e.g. `addl`, `adcl` on i386 (plus storage overhead; for ppc detail see section 4.1).

1.4.2 Data collection phase

The instrumented assembly files call `libgcov.c` functions that again use `gcov-io.c` functions (now `IN_LIBGCOV` is set, but `GCOV_LOCKED` is not set). The counters that are incremented reside in libgcov's `struct gcov_info`'s count fields and already have a place in memory before the executable is running. Once the executable is running, the counters are just incremented ($2^{64} \approx 4 \cdot 10^{18}$ which at 1 GHz gives $4 \cdot 10^9$ seconds, i.e. more than 125 years before overflow).

When the program exits (`atexit` on i386), `gcov_exit` is called which writes the collected data into a `gcda` ("gcov data" file).

1.4.3 Reporting phase

During the reporting phase (`gcov.c` and `gcov-io.c`, in `IN_LIBGCOV` is not set, `IN_GCOV` is set), the gcda and gcno files are read out, the basic block data is calculated from arc data and written into a human-readable report (`hello.c.gcov`). To generate more beautiful reports (in color), `lcov` [8] can be used as gcov back-end.

1.5 GCNO and GCDA file formats

Both files are formally described in [15]. To summarize, for each source file gcda and gcno files are created (for readers familiar with the pre-gcc-3.4 gcov utilities: `*.gcno` holds the data of `*.bb` and `*.bbg`; `*.gcda` holds the data of `*.da`). Both consist of a short (12 byte) header (magic, version, crc) followed by a list of information for each function usually in the same order that the functions are in the source file. In a full gcc build a binary named `gcov-dump` for inspection is created but also the binary hexdumps of the files are quite readable, because header tags have been recorded in different endianness than data proper.

1.5.1 Data for each function in the gcno file

(The order of the three list items below has been reversed in order to reflect the priority for the record entries for our purpose.)

- records tagged `0x01450000` (with '`E`' in ASCII showing up for '`45`'): each contains the identity of one bb, and the list of source lines it attached to it. In gcc-3.4 files the record ends with two '0's. When only `-fprofile-arcs` but not `-ftest-coverage` is given, these headers are omitted.
- records tagged `0x01430000` (featuring '`C`' in ASCII): here a list of arcs is kept, format for an arclist: source bb, followed by a list of one or multiple times the pair (destination bb, flag). The flag is 1 for `GCOV_ARC_ON_TREE`, 2 for `GCOV_ARC_FAKE`, 4 for `GCOV_ARC_FALLTHROUGH`.
- records tagged `0x01410000` (featuring '`A`' in ASCII): on inspection of practical examples these are n flags typically initialized to 0 for n records (also from code inspection there is some indication that this part of the files is currently not always used e.g. as of gcc-3.4.4 up to gcc-4.3.0 on reading only the record summary giving its count n is given out by `gcov-dump` and on writing the argument to `gcov_write_unsigned` (in `profile.c:843`) is simply constant 0).

1.5.2 Data for each function in the gcda file

- records tagged `0x01000000` for each function which after an identity (referrring to the gcno file identity) is followed by an `0x01a10000` (`GCOV_TAG_FOR_COUNTER`) record containing the 64-bit counts of some arcs (those not having an odd flag which which indicates that there are not `GCOV_ARC_ON_TREE`, i.e. on the spanning tree). In addition to the nodes listed in the gcno file, there also exist implicit nodes referred to tagged with `GCOV_ARC_FALLTHROUGH` with index 0 (beginning of function) and index $n + 1$ (end of function).

The gcda file ends (`0xa1000000`) with some summary statistics on the object, and, if it is a program, on the program.

2. Compilation for a PPC target without file system and without C standard library

2.1 Entry point: iteration over __CTOR_LIST__

Before we can start the coverage (before `main` is entered), it has to be ensured that memory for counters gets allocated.

2.1.1 Comparison with C++ constructor calls

That task sounds familiar: C++ memory management also has to initialize constructors before calling main, and for ELF files the ELF standard [11] charges section `.init` to allow calls to functions allocating memory before running `main`. Generally, on i386 ELF files compiled by `gcc`, `objdump` shows that the function `_init` in the section `.init` calls `__do_global_ctors_aux` (`crtstuff.c`) which iterates over a list of function pointers (referred to as `__CTOR_LIST__` by the default linker script).

2.1.2 Gcov I386 constructors

When, on i386 `-ftest-coverage` is used, each compilation unit emits a pointer to a corresponding gcov initialization function named after its first global symbol (e.g. `_GLOBAL__I_somesymbol_GCOV` for a compilation unit that contains `somesymbol`) which calls `__gcov_init` with the argument `struct gcov_info *p` is added to the constructor list.

2.1.3 Manual generation of constructor infrastructure

On the embedded target, when we compile with gcc with gcov but where we do not have glibc infrastructure, we still generate the `_GLOBAL__I_somesymbol_GCOV` hook, but we do not want to install the whole `crtstuff` infrastructure on the target, so by adaptation of the linker script we carry the contents of `.ctors` over to `.text` and call `gcov_entry` during the boot phase:

```
void gcov_entry() {
    ctor const *ptr = __CTOR_LIST__;
    ctor const *end = __CTOR_END__;
while(ptr != end) {
     if(*ptr)
          (*ptr)();
     ptr++;
}
```

The contents of struct `gcov_info` are addressed via a hardcoded memory pointer.

2.2 Replacing C standard library parts of glibc

2.2.1 Storage-related calls

On i386, gcov would like to write data to files and use dynamic memory allocation (most of this is in `gcov_exit`). In particular, `gcov_exit` calls the three file operation functions `gcov_open`, `gcov_close`, `gcov_write_block`. (It also calls the intermediate-level functions `gcov_write_tag_length`, `gcov_write_counter`, `gcov_write_summary`, `gcov_write_unsigned`, `gcov_write_words`, but all these intermediate-level functions end up calling `gcov_write_block`.) In particular, standard library function that need to be replaced are `fseek`, `fopen`, `fclose`, `fwrite`, `malloc`, `realloc`.

2.2.2 Exit point

Exit point: On i386, during data extraction, GCDA files are written from the instrumented files via a call to `atexit` (part of glibc) via `__gcov_exit`. On the embedded system we will halt the system manually and make a call to `gcov_exit` to write the data into memory (e.g. in the a Lauterbach hardware JTAG debugger call set the instruction pointer to the exit routine via `register.set ip gcov_exit; go`).

2.2.3 A datastructure for holding file system data

In `gcdamem.h`, for each gcda file define (when `gcov_exit` is called the number of files is known, thus giving a single-directory memory-based "file system" where filenames may contain forward slashes):

```
struct gcda_record {
   /** Name of file. */
    char        gr_name[FILENAMELEN];
   /** Number of 32bit words in file. */
    uint32_t    gr_size;
}  __attribute__ ((packed));
```

Hence the equivalent of opening a gcda file on i386 is incrementing a counter pointing an instance of `gcda_record`.

2.2.4 Replacing dynamic memory allocation

We have replaced dynamic memory allocation by static allocation.

```
#define GCOV_MEMBASE 0xD0A00000
writePosition = (uint8_t*)GCOV_MEMBASE;
```

Then increments of `writePosition` replace memory allocation calls from `gcov-io.c` functions (this can be done so simply, because `realloc` calls always in `gcov-io.c` refer to the most recently assigned memory).

2.3 Compilation flags

Compile with `-DIN_LIBGOV`.

3. Running on the target, post-mortem memory analysis on the I386 host

3.1 Memory boundaries

In a first attempt, the beginning and end of the the dump was indicated by messages from libgcov such as

```
libgcov: size of mem 27752 (0x6c68)
libgcov: memory begins at 0xc082b1c8
libgcov: memory ends at 0xc0831e30
```

from these the desired memory segments can be read off the serial interface (e.g. via `kermit`). However, this way of debugging had influenced the counts (calls to `printf` infrastructure) and negative solutions to flow graphs were to be avoided. Hence from the next iteration on, `printf`s have been omitted, instead the memory of interest is well-known and the end is given by the file header.

3.2 Reading in the memory dump

We assume that a memory dump has been generated.

3.2.1 Generation of gcda files

PPC endianness is big-endian (our target's memory dump) whereas i386 (our host) is little-endian. However, this is a conversion we get for free... the `gcov` analysis tool automatically is aware whether a file starts with "`gcda`" or "`adcg`" and even does not require gcda and gcno files to be of the same endianness.

3.2.2 Gcda + gcno + C sources → coverage (how to report it)

Recall from section 1.4.1 that the `*.gcno` files already had been generated during compilation of the sources (parallel to the instrumentation). During the reporting phase, the standard gcov tool (`gcov-3.4` on our host), uses our gcda generated by the previous step and the gcno files to make coverage reports. For annotating the sources, it is needed that the source is also located in the directory. Adding up results of several runs (taken care of by `gcov_merge_add` if we were in a file-system setting) can be done on the host by parsing the text file output of the `gcov` tool and adding up the results. Similarly, for convenience during development, simple scripts have been written to check a memory dump for negative counts (which would result from `solve_flow_graph` when it has been fed garbage).

3.2.3 Testing of the tools

To test the above-mentioned modifications to libgcov and the subsequent extraction, we have run tests on C flow control and function call statements, recursion to a depth allowed by the embedded target and target scheduler tests. Comparison of gcov output (w.r.t. function coverage, statement coverage, preprocessor behavior) showed correct results for the modified gcov variant as far semantically relevant code is concerned (for all practical applications).

As side result, these tests also revealed that the standard gcc version 3.4.4 (hence including e.g. the i386 platform) shows false coverage for some semantically empty code when in a `switch` compound statement the default case only contains a `break` statement. Closer investigation of the `switch` behavior in the gcc code showed that a patch would have to fix code in the jump optimization pass 4 in gcc-3.4.4 (as apparently one also would have to block optimization at other places, forcing this into the gcc 3 branch seems not advisable). The behavior does *not* occur when gcc 4 is used where with `passes.c` replacing part of `toplev.c` the optimization pipeline has been cleaned up.

4. One-line modification of gcc to cover reentrant code

4.1 Problem: possible race condition in a multithreaded run

If one looks at generated code (compilation on ppc architecture via `gcc-3.4 -O0 -ftest-coverage -fprofile-args hello.c`)

```
lis r9,-16252
addi r11,r9,21112
lwz r9,0(r11)
lwz r10,4(r11)
addic r10,r10,1
addze r9,r9
stw r9,0(r11)
stw r10,4(r11)
```

it looks that there may be race conditions in case of code reentrance, e.g. that one looses a count when one thread reads in the counter register `r10` via `lwz` and then the other reads in the register `r10` via `lwz` too (leading to a loss of 1 in the count on `stw r10` write).

In such a situation it may happen that `solve_flow_graph` reconstructing block counts (where subtractions occur) from arc counts may give out coverage>0 for blocks where coverage actually has been 0 (this is the critical case [3], also [7] explicitly does not claim thread-safety). E.g. empirically, in a threaded context, negative values for positive coverage had been obtained during `pthread/i386` experimentation.

4.2 Evaluation by visual inspection

In a not too complicated system, sometimes only a few functions are reentrant, so it is sometimes feasible to do an automated analysis of non-reentrant functions combined with a manual stepping through reentrant functions.

4.3 Modification of instrumentation

In our case we were only interested in showing that there is no dead code, that is each statement has been covered at least once. In this situation a conservative estimate (lower bounds) for block coverage is feasible, this can be achieved by a simple (one-line) change to the gcc sources (recall that, up to now, all of the analyses had been done with a customized libgcov but with a standard gcc compile-time framework): for gcc-3.4.4 it was sufficient to disable the call to `find_spanning_tree` in `profile.c:branch_prob`, for gcc-4.3 from withing `find_spanning_tree` return after having finished

Figure 1. Gcno file structure changes by disabling spanning tree optimization (`kdiff3` bird's eye view)

the block inserting all critical edges (before "And now the rest." comment). This leads to an increase of the binary size of ca. 10%, a size increase of typically $10 - 50\%$ in gcda files, and actually a small decrease in gcno size file, because a fake edge exit to entry is no longer added (Figure 1 shows the comparison of gcno files of a compilation unit with several functions with and without use of spanning trees). During preliminary tests, a running binary was produced, and also testing results generated by the standard `gcov` inspection tool (counts of function where the behavior was well-known) were consistent with the non-modified instrumentation, so one can confirm that `gcov.c:solve_flow_graph` also works fine on an empty spanning tree.

Note that because the gcov analysis is strictly per compilation unit for a system it is also possible to simply compile the reentrant functions with spanning trees disabled, and to compile other functions with the standard (unmodified) gcc, and then link and analyze the result together.

To simply remove an optimization of course is a bit "backwards", but it is useful to know that even after its rich evolution the gcov framework can be cut down quickly when needed. An alternative to our reductionist approach might be to assign each thread its own memory for coverage or do only tree optimizations that will not require subtraction of counts.

5. Discussion and outlook

5.1 Development pathway taken

As a first step we had instrumented gcc 3.4.4's `*gcov*` sources to get a clear understanding of the different phases of the gcov framework. In a next step, still on the i386, we removed file system dependence and then general libc dependence. Then, the framework was ported to ppc. Finally, first steps towards thread-safety were explored.

5.2 Results

We have shown how to do the necessary modifications to libgcov to work on an embedded target without file system and without memory management. To the best of our knowledge, this is the first description of a modification of gcov to for memory-only systems. Also little has been so far published on handling threading in a gcov environment via cutting off everything that can be discarded.

5.3 Limitations

Some hard-coded assumptions such as `FILENAMELEN` need to be respected. So far, on the target, we have not imported checksums (because with the memory-based approach accidental in-between recompilation the checksums were intended to prevent is not likely). One could do an implementation of `fseek` (if one wants the `write_program_summary` feature).

6. Hardware and OS details

Freescale MPC5554 processor, e200z6 CPU, PikeOS 2.2 development toolchain. The custom-built operating system has thread abstraction to provide pseudoparallel execution of program units on a single physical CPU.

We have used a Lauterbach LA-7753-DEBUG-MPC5500 debugger controlled from in-circuit `trace32` [10] to transfer memory from the embedded node to the host (depending on the memory mapping applied one has to take care that the CPU L1-cache is properly flushed before reading out).

Thanks to Ben Fischer, Ingo Hornberger, Daniel Junglas, Christian Körner, Bertrand Marquis for suggestions, advice and test design.

References

[1] AV Aho, R Sethi, JD Ullman, 1986, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, Mass.

[2] JH Andrews 2004, Coverage-checked random testing of data structures: The sourceforge case study, Technical Report No. DCS-285-IR, Department of Computer Science, University of Victoria `http://www.csd.uwo.ca/faculty/andrews/papers/sourceforge-tr.ps`, accesssed 15 May 2007.

[3] H Blasum 2007, Gcov and thread-safety, posting to gcc-help at gcc.gnu.org 05 June `http://gcc.gnu.org/ml/gcc-help/2007-06/msg00056.html`.

[4] S Cornett, Code coverage analysis, `http://www.bullseye.com/coverage.html`, accessed 04 June 2007.

[5] X Fei, L Luo 2004, Using the GNU tools to measure assembly program coverage (Liyong GNU gongju shixian huibian chengxu fugai ceshi), Journal of Computer Applications (Jisuanji Yingyong, ISSN 1001-9081), Vol 24 No. 12 pp. 95-98.

[6] Free Software Foundation, gcov - a Test Coverage Program, in: Using the GNU Compiler Collection (GCC), `http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Gcov.html`, accessed 19 July 2007.

[7] J Hubicka 2005, Profile driven optimisations in GCC, in: Proceedings of the GCC Developers' Summit, June 21-24, 2005, Ottawa, Canada, pp. 107-124, `http://www.gccsummit.org/2005/2005-GCC-Summit-Proceedings.pdf`, accessed 06 Aug 2007.

[8] M Iyer et al., The LTP GCOV extension (lcov), `http://ltp.sourceforge.net/coverage/lcov.php`, accessed 03 June 2007.

[9] P Larson et al. 2003, Improving the linux test project with kernel coverage analysis, Proceedings of the 2003 Ottawa Linux Symposion, `http://ltp.sourceforge.net/docs/ols2003/gcov-ols2003.pdf`, accessed 15 May 2007.

[10] Lauterbach Datentechnik GmbH 2007, Command list, as trace32/pdf/commandlist.pdf in the Lauterbach PowerView install `http://www.lauterbach.com/powerview.html`, accessed 06 June 2007.

[11] H Lu 1995, ELF: From The Programmer's Perspective, `http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/elf.html`, accessed 17 July 2007.

[12] N McGuire 2006, Linux Kernel GCOV - tool analysis, Lanzhou Univ, `http://linuxdevices.com/files/article062/der_herr_gcov.pdf`, accessed 15 May 2007.

[13] P Oberparleiter, New gcov-kernel patches, 22 May 2007, `http://www.nabble.com/New-gcov-kernel-patches-t3795953.html`.

[14] RTCA SC-167, EUROCAE WG-12, 1992, DO-178B, Software considerations in airborne system and equipment certification, RTCA Secretariat, Washington DC.

[15] N Sidwell, `gcov-io.h` and `gcov-io.c`, GCC, the GNU compiler collection, `http://gcc.gnu.org/`.

[16] Q Yang, JJ Li, D Weiss, 2006, A survey of coverage based testing tools, in: Proceedings of the 2006 International Workshop on Automation, AST' 06 Shanghai 2006, pp. 99-103.