# ggcov: A Practical Guide To Making Your Test Suite Suck Less

Greg Banks <*gnb@users.sourceforge.net*>

**Abstract:** *Test coverage is a well known but underused technique for measuring the effectiveness of test suites. This paper provides advice derived from personal experience performing coverage studies using ggcov, the author's graphical coverage browser tool. This includes guidelines for interpreting results, advice on what to expect from coverage studies, issues with multi-threaded programs, and a brief description of issues discovered when coveraging the Linux kernel.*

## Scope

This paper discusses practical test coverage techniques specific to the C and C++ languages, using the GNU Compiler Collection version 4.1.0 on UNIX-like platforms. Similar techniques apply to other platforms, other compilers, and other languages but are not discussed here.

The techniques discussed are (to the author's knowledge) licence-neutral, i.e. they can be applied legally to software which is released under any software licence from BSD to GPL to entirely proprietary. In particular, the gcc code that is linked into an instrumented executable is released under a licence based on the GPL but with a specific exception to allow it's use with any programs, without the usual “infection”[1].

## What Is Test Coverage?

Test coverage is a well-known and standard set of software engineering techniques which measure how much of the Code Under Test is run (i.e. “is covered”) when it's test suite is run.

The justification for using test coverage is based on the following chain of logic:

- code which works today may fail tomorrow, due to changes in the code itself, changes to other code which interacts with it, or changes in the build-time or run-time environments.
- thus, for code to work and to remain working it must be tested regularly.
- this implies the existence of a suite of tests which are run regularly (and hopefully automatically).
- however, the tests are only useful in so far as they actually cause the code under test to be executed.

Test coverage provide metrics which measure the quality of a a test suite, and thus indirectly the quality of the code.

In the author's experience, test coverage is sorely underused in software engineering practice. Too often, only lip service is paid to the testing process, and techniques designed to improve the testing process are entirely ignored. Even when the development team uses methodologies designed to emphasise testing often and early, too much faith may be placed in the existence of large numbers of tests and too little effort may be expended on measuring the quality of those tests.

This paper was written in the hope of achieving wider use of test coverage, thus indirectly better software quality.

## How Does It Work?

A test coverage study proceeds in three phases.

The first phase is at build time. A special path through the build system (e.g. a **make** target) is added, which causes the gcc option --*coverage* to be added to both the compile and link lines[2].

This has several effects. It causes the compiler to insert *instrumentation* at basic block boundaries in the object code; these are extra instructions which count the number of times the basic block is executed. The compiler also adds an array to hold the counters, a data structure describing them, and a global constructor which registers the descriptor before *main* is run.

In addition, the compiler writes a data file in the same directory as the object file, called *foo.gcno*[3]

1 The author is not an lawyer.

2 On versions of gcc before 4.1, use **both** the options *-ftest-coverage -fprofile-arcs*.

3 The file extension has changed several times in gcc's

where *foo.o* is the object file name. This file contains information which later will be used to map the counters back to source file names and line numbers.

The second phase is at run time. Counters are incremented as instrumented code executes. When the program exits[1], a special *atexit* handler writes the counter values into files. Each file is called *foo.gcda* and is located in the same directory as the corresponding *foo.o*.

The third phase is analysis. A post-processor, such as the *gcov* program shipped with gcc [FSF06], or the author's *ggcov* tool [BAN06], to combine the information in the *.gcno, .gcda,* and source files and to present coverage information to the developer.

## How To Interpret Results

When *ggcov* is first run, it shows a window with some stacked bar charts which report overall coverage statistics for the entire program. It's important not to assume too much deep meaning for these numbers. The most use that ought to be made of them is to compare the first digit of the overall line coverage percentage before and after a coverage study.

The procedure I find most useful is as follows. Run the entire test suite to completion once and then run *ggcov*; do not focus on coverage from individual tests[2]. Identify which source files have the lowest line coverage.

*Fig 1. ggcov's File List window*

One way is to use the File List window which shows all the source files and their coverage as percentages, and sort on the Line or Block columns. For example, in Fig. 1, report.C and cov_dwarf.C would be good candidates.

*Fig. 2: ggcov's Lego Diagram window*

Another way is to use the Lego Diagram, which shows block coverage scaled by the number of blocks in each file, and look for large areas of red colour. For example, in Fig. 2, cov_dwarf2.C and cov_file.C would be good candidates.

Address all the source files, starting with those files with low line coverage. Use the Source Window and scroll through the code looking for fragments of code with multiple consecutive lines coloured red (i.e. untested). For example, in Fig 3 the second function is a good candidate.

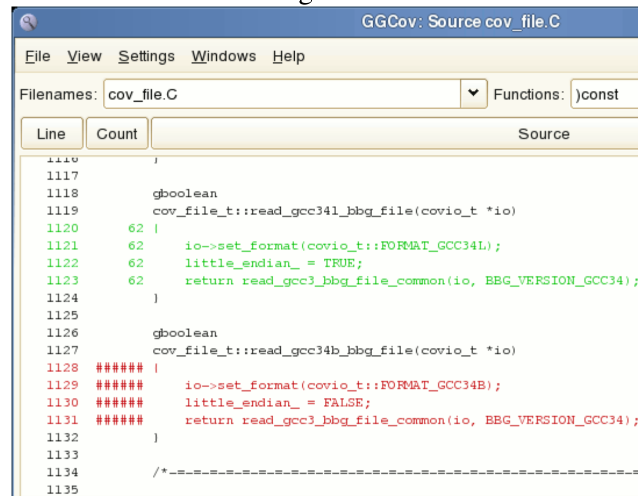


*Fig. 3: ggcov's Source window*

Use your understanding of the code, or clues in function names or comments, to map that code back to a high-level feature (e.g. some particular commandline option) that has not been tested. Keep track of which features are untested, and how many times you see untested code which implements that feature. After a complete run through the source, that list of tests can be used as a list of tests to be written, and a measure of how important each test is.

history.

1 To preserve counter semantics, this also happens at other times, like *fork* and *execve* system calls.

2 One reason for this recommendation is that there is no easy way to merge coverage results from multiple *.gcda* files.

## What To Expect

You should be prepared to spend some time and effort setting up the coverage study and interpreting the results, mostly due to imperfections in the toolchain. However the results will be worth it, so persevere.

The first time you run a coverage study, be prepared to be surprised by how poorly your test suite actually covers the code. This is because manual test suite design is a poor substitute for coverage-driven testing (unless you have an enormous army of testers). Your initial reaction is likely to be dismay.

For example, when members of the Samba team first used *ggcov* to measure the coverage of the Samba4 test suite they found that only 17% of the source lines were covered.

Even if your test suite does achieve respectable numbers in your first coverage study, it's still likely that entire features of the code are not tested.

For example, the XFS QA team in SGI recently did a coverage study of the XFS test suite. Nearly 70% of the XFS code was covered, but entire features such as real-time volumes were never tested.

It is highly recommended that you use the latest release of ggcov to get all the relevant bug fixes. In particular, be aware that the Debian package may lag behind to the point of not being usable.

## What Not To Expect

Do not expect to get perfect numbers from the coverage tool. There are enough bugs in the compiler and post-processors, and enough room for differences in interpretation, that you will see the occasional strange number[1]. Concentrate on finding code which is never executed at all, rather than trying to work out why a particular line is reported as executed 4 times instead of 3. In *ggcov*, the colour-coding helps you think this way.

Do not aim to achieve 100% coverage of your code. This is a very tempting target, but is only realistic for tightly constrained artificial examples or projects with extremely high quality requirements (and costs). Your code contains many arcs and even whole lines, the testing of which lies beyond the point of diminishing returns, e.g. code which handles memory allocation failures, or C++ exception paths.

There is also the problem of the *assert* macro. This macro emits code which will never be executed in a correctly functioning program, regardless of how many test cases you add. Because *ggcov* detects and reports the difference between a source line which is entirely executed and a line which is only partly executed, lines containing *assert* will spuriously reduce the overall line coverage result. Likewise, the branch which is never executed will reduce the overall branch coverage result.

Do not expect a coverage tool to tell you when to stop testing. You need to decide for yourself where the point of diminishing returns lies. I recommend continuing until every value of every user input option has been tested, and every source fragment of 3 or more consecutive lines which is not an error case has been tested.

Do not expect a coverage tool to write tests for you. Coverage tools are simple and do not understand your code well enough to do that.

Do not expect a coverage tool to be able to reduce (or "optimise") your test suite automatically. Some coverage tools include a utility which claim to be able to do this but this claim is based on false assumptions. The first false assumption is that you have too many tests; the second is that simple line or branch coverage can be a useful indicator that tests are redundant.

Do not expect a coverage tool to guide you when writing your first test. At that time you already know that none of your code is tested, so coverage will not tell you anything new.

## Separate Test Machines

On program exit, the instrumented code writes counter values to *.gcda* files in the original source directory, using absolute pathnames. At analysis time, the post-processor needs to see those *.gcda* files.

If the tests run in an environment where that directory doesn't exist or isn't writeable, such as a separate test machine or under a different user id, the counter values will be silently lost.

Currently, there are two approaches to solving this problem. The first is to make that directory visible to the test program, e.g. by remotely mounting it using NFS. The second approach is to create a writeable dummy directory at the expected pathname, and after testing copy the *.gcda* files back to the build machine, e.g. using *tar*.

1 Examples of code which cause problems include static inline functions in headers, for(;;) loops, and source lines containing multiple function calls.

The *.gcda* files are in general not cross-platform, so the build, run and analysis phases should all proceed on the same machine architecture, e.g. i386.

## Performance Impact

The performance impact of the *gcc* coverage instrumentation is quite light. This is partly because the instrumentation comprises relatively inexpensive increments of global counters, and partly because *gcc* takes care to instrument only a minimal subset of the branches in each function.

However, coverage can interact badly with optimisation, so the usual recommendation is to remove all optimisation options when building for coverage. This will affect the performance of your program.

## Build System Integration

How you add coverage support to your build system depends on what build system you use, but a few general recommendations may be useful.

For small to medium sized projects, a single target in the top level makefile is the most convenient. Larger projects may wish to only coverage specific subdirectories at a time.

Provide a single target which does all the necessary build-time steps. In particular, turn off optimisation options (-O), enable debug symbols (-g) and disable executable stripping.

## Multi-process/Multi-threaded

Coveraging programs which comprise multiple single-threaded processes is known to work, even when the programs share source code. The *libgcov* code which writes *.gcda* files is careful to handle this case; it uses POSIX file locking calls and accumulates counters in the file rather than writing them anew.

Unfortunately, programs with multiple threads of control in a single address space are a different matter. The instrumented code does not explicitly handle this case; the counter increments are not atomic and not protected by locks. If such programs are run on multi-CPU hardware where actual parallelism is possible, the counters may become corrupted. An unfortunate side-effect of minimal instrumentation in *gcc* is that a single incorrect counter can render impossible calculation of any coverage data for the entire function.

A *gcc* patch to enable atomic incrementing of counters [GNU06] has been written to solve this problem, and may be available in *gcc* 4.1.2.

## Coveraging the Linux Kernel

Coveraging the kernel poses unique challenges. A patch from IBM [FRA06] needs to be applied; this patch allows the counters to be exported to userspace via a */proc* interface. The patch also grants the ability to reset counters, e.g. between tests.

Multi-processor machines will experience the problem of non-atomic counter increments. Possible solutions are to use the *gcc* patch, or run the machine in a uni-processor mode (e.g. booting a uni-processor kernel or disabling all but one CPU on the kernel commandline).

The kernel also suffers from a problem not seen in userspace. Because the counters are extracted while the program is still running, any functions which have not completed (because tasks are still executing them) will have counters in an inconsistent state. This can lead to the same counter corruption problem seen for MT programs.

Because the counters are extracted using a filesystem interface, this means there are several functions in the VFS layer and generic filesystem code for which coverage data can never be gathered.

Indeed, because the coverage data needs to be written to some filesystem as it is copied from the kernel, the same applies to functions in the write path for that filesystem. Thus, when coveraging filesystem code it is advisable to set up the machine with different types of filesystems for root (/) and the filesystem under test.

## References

[BAN06] Greg Banks, ggcov coverage browser, *http://ggcov.sf.net/*

[FRA06] Hubertus Franke, Nigel Hinds, Peter Oberparleiter, Rajan Ravindran, IBM Linux kernel coverage patch, *http://ltp.sf.net/coverage/gcov-kernel.readme.php*

[FSF06] Free Software Foundation, GCC Manual, *http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Gcov.html*

[GNU06] Free Software Foundation, Need atomic increment of gcov counters for MP programs, *http://gcc.gnu.org/bugzilla/show_bug.cgi?id=28441.*

[LTP06] Linux Test Project, *http://ltp.sf.net/*