Improving the Linux Test Project with Kernel Code Coverage Analysis

Paul Larson
IBM Linux Technology Center

Nigel Hinds, Rajan Ravindran, Hubertus Franke
IBM Thomas J. Watson Research Center

{ plars, nhinds, rajancr, frankeh } @ us.ibm.com

Appeared in the Proceedings of the 2003 Ottawa Linux Symposium

Abstract

Coverage analysis measures how much of the target code is run during a test and is a useful mechanism for evaluating the effectiveness of a system test or benchmark. In order to improve the quality of the Linux® kernel, we are utilizing GCOV, a test coverage program which is part of GNU CC, to show how much of the kernel code is being exercised by test suites such as the Linux Test Project. This paper will discuss the issues that make code coverage analysis a complicated task and how those issues are being addressed. We will describe tools that have been developed to facilitate analysis and how these tools can be utilized for kernel, as well as application code coverage analysis.

1 Introduction

The Linux Test Project (LTP) is a test suite for testing the Linux kernel. Over the years, the LTP has become a focal point for Linux testing and Linux test tools. Just as the Linux kernel is constantly under development, the LTP test suite must also be constantly updated, modified, and improved upon to keep up with the changes occurring in the kernel. As the Linux Test Project test suite matures, its developers constantly look for ways to improve its usefulness. Some of the considerations currently being addressed are what areas of the Linux kernel test development efforts should focus on, and proving that tests added to the LTP are testing more than what was there before. In order to gather data to support decisions for these questions, analysis should be performed to show what areas of the kernel are being executed by any given test.

The design of a process for doing this analysis took the following features into consideration:

1. Use existing tools as much as possible
2. Take a snapshot of coverage data at any time
3. Clear coverage counters before a test execution so just the coverage from the test could be isolated
4. Provide output that looks nice and is easy to read and compare
5. Show coverage percentages at any level of directory or file
6. Show execution counts for every instrumented line of every file

This paper is outlined as follows: Section 2 provides a general description of code coverage analysis. Sections 3 and 4 discuss GCOV and other code coverage tools. Section 5 describes Linux kernel modifications necessary to provide kernel code coverage. Test results are presented in Section 6. Section 7 presents the LCOV toolkit for processing and displaying GCOV results. Section 8 describes how the results of kernel code coverage analysis can be used to improve the Linux Test Project. Future work planned on this project is discussed in Section 9. Finally, conclusions are presented in Section 10.

2 How code coverage analysis works

Before examining possible methods for the task of kernel code coverage analysis, it is important to first discuss some general concepts of code coverage analysis.

Code coverage analysis shows what percentage of an application has been executed by the test process. The metrics derived from code coverage analysis can be used to measure the effectiveness of the test process [Perry].

Statement coverage analysis [Cornett] breaks the code down into basic blocks that exist between branches. By design, it is clear that if any line in a basic block is executed a given number of times, then every instruction within the basic block would have been executed the same number of times. This provides for a convenient way of showing how many times each line of code in a program has been executed.

There are, however, a number of deficiencies in this approach. The following piece of code exhibits one of the main issues with statement coverage analysis:

```
int *iptr = NULL;
if(conditional)
     iptr = &i;
*iptr = j*10;
```

The above code may show 100% code coverage but will obviously fail miserably if the conditional is ever false.

To deal with situations such as the one demonstrated above, branch coverage analysis [Marick] is required. Rather than focusing on the basic blocks and the lines of code executed, branch coverage looks at possible paths a conditional can take and how often each path through the conditional is taken. This will account for problems such as those described above because it will clearly show that the conditional was never evaluated to be false. Another advantage of branch coverage analysis is that knowledge of how many times each line of code was executed can be derived by knowing how many times each conditional was evaluated for each possible outcome.

Although branch coverage analysis provides a more accurate view of code coverage than statement coverage, it is still not perfect. To gain a better understanding of the path taken through the entire conditional, the outcome of each test in a complex conditional must be determined. For instance:

```
struct somestructure* p = NULL;

if(i == 0 || (j == 1 && p->j == 10))
        printf("got here\n");
```

If i is ever non-zero when j is 1, then the NULL pointer p will be dereferenced in the last part of the conditional. There are paths through this branch in both the positive and negative case that will never expose the potential segmentation fault.

Branch coverage would show how many times the entire conditional was evaluated to true or false, but it would not show the outcome of each test that led to the decision.

There are many other types of coverage for dealing with a variety of corner cases. However, statement coverage and branch coverage are the two types that will be focused on in this paper since they are the only ones supported by GCOV.

3 Methods considered

This section discusses some of the more popular techniques for collecting statement and branch coverage data in a running system.

3.1 Estimation

At a most basic level, estimating techniques could be used to collect code coverage within a given program. Obviously, this method has no redeeming value at all with regards to accuracy or reproducibility. The advantage to this method, and the only reason it is mentioned, is that it requires the least amount of effort. Given a developer with sufficient knowledge of the code, a reasonable estimate of code coverage may be possible to estimate in a short amount of time.

3.2 Profiling

Performance profilers are well known, easy to use, and can be used to track coverage; however these uses are not the purpose for which performance profilers were intended. Performance profilers use statistical profiling rather than exact profiling.

Readprofile is a simple statistical profiler that stores the kernel PC value into a scaled histogram buffer on every timer tick. Readprofile profiles only the kernel image, not user space or kernel modules. It is also incapable of profiling code where interrupts are disabled.

Oprofile, another statistical profiler, leverages the hardware performance counters of the CPU to enable the profiling of a wide variety of interesting statistics which can also be used for basic time-spent profiling. Oprofile is capable of profiling hardware and software interrupt handlers, the kernel, shared libraries, and applications.

All of these statistical profilers can be used indirectly to gather coverage information, but because they approximate event distribution through periodic sampling via an interrupt, they cannot be considered accurate for true coverage analysis.

3.3 User-Mode Linux

User-mode Linux is an implementation of the Linux kernel that runs completely in user mode. Because it runs as a user mode application, the possibility exists to utilize analysis tools that were normally intended to run against applications. These tools would not normally work on kernel code, but User-mode Linux makes it possible. To make use of this feature in User-mode Linux, the kernel needs to be configured to compile the kernel with the necessary flags to turn on coverage tracking. To do this, the "Enable GCOV support" option must be selected under "Kernel Hacking" from the configuration menu.

When GCOV support is turned on for User-mode Linux, the necessary files for using GCOV are created, and coverage is tracked from boot until shutdown. However, intermediate results cannot be gathered, and counters cannot be reset during the run. So the resulting coverage information will represent the boot and shutdown process and everything executed during the UML session. It is not possible to directly gather just the coverage results of a single test on the kernel.

3.4 GCOV-Kernel Patch

In early 2002, Hubertus Franke and Rajan Ravindran published a patch to the Linux kernel that allows the user space GCOV tool to be used on a real, running kernel. This patch meets the requirement to use existing tools. It also provides the ability to view a snapshot of coverage data at any time. One potential issue with this approach is that it requires the use of a kernel patch that must be maintained and updated for the latest Linux kernel before coverage data can be gathered for that kernel.

The GCOV-kernel patch and GCOV user mode tool were chosen to capture and analyze code coverage data. The other methods listed above were considered. However, the GCOV-kernel patch was chosen because it allowed platform specific analysis as well as the ability to easily isolate the coverage provided by a single test.

4 GCOV Coverage Tool

Having decided to use the GCOV tool for LTP, we first describe how GCOV works in user space applications and then derive what modifications need to be made to the kernel to support that functionality.

GCOV works in four phases:

1. Code instrumentation during compilation
2. Data collection during code execution
3. Data extraction at program exit time
4. Coverage analysis and presentation post-mortem (GCOV program)

In order to turn on code coverage information, GCC must be used to compile the program with the flags "`-fprofile-arcs -ftest-coverage`". When a file is compiled with GCC and code coverage is requested, the compiler instruments the code to count program arcs. The GCC compiler begins by constructing a program flow graph for each function. Optimization is performed to minimize the number of arcs that must be counted. Then a subset of program basic blocks are identified for instrumentation. With the counts from this subset GCOV can reconstruct program arcs and line execution counts. Each instrumented basic block is assigned an ID, `blockno`. GCC allocates a basic block counter vector `counts` that is indexed by the basic block ID. Within each basic block the compiler generates code to increment its related `counts`[`blockno`] entry. GCC also allocates a data-object *struct bb* `bbobj` to identify the name of the compiled file, the size of the `counts` vector and a pointer to the vector. GCC further creates a constructor function

`_GLOBAL_.I.FirstfunctionnameGCOV` that invokes a global function `__bb_init_func(struct bb*)` with the `bbobj` passed as an argument. A pointer to the constructor function is placed into the ".ctors" section of the object code. The linker will collocate all constructor function pointers that exist in the various *.o files (including from other programming paradigms (e.g. C++)) into a single ".ctors" section of the final executable. The instrumentation and transformation of code is illustrated in Figure 1.

file1.c

```
→ static ulong counts[numbbs];
→ static struct bbobj =
→         { numbbs, &counts,"file1.c"};
→ static void _GLOBAL_.I.fooBarGCOV()
→         { __bb_init_func(&bbobj); }

void fooBar(void)
{
→   counts[i]++;
    <bb-i>
    if (condition) {
→     counts[j]++;
      <bb-j>
    } else {
      <bb-k>
    }
}

→ SECTION(".ctors"')
→ { &_GLOBAL_.I.fooBarGCOV }
```

"→" indicates the lines of code inserted by the compiler and <*bb-x*> denotes the x-th basic block of code in this file and italic/bold code is added by the GCC compiler. Notice that the arc from the *if* statement into bb-k can be derived by subtracting counts[j] from counts[i].

Figure 1: Code modification example

In order to relate the various source code line information with the program flow and the counter vector, GCC also generates two output files for each `sourcefile.c` compiled: (i) `sourcefile.bb`, which contains a list of source files (including headers) and functions within those files and line numbers corresponding to basic blocks in the source file, and (ii) `sourcefile.bbg` contains a list of the program flow arcs for each function which in combination with the `*.bb` file enables GCOV to reconstruct the program flow.

The glibc (C runtime library) linked with the executable provides the glue and initialization invocation and is found in the `libgcc2.c`. More specifically: it provides the `__bb_init_func(struct bb*)` function that links an object passed as an argument to a global `bbobj` list `bb_head`. The runtime library also invokes all function pointers found in the ".ctors" section, which will result in all `bbobj`s being linked to the `bb_head` list, as part of the `_start` wrapper function.

At runtime, the counter vector entries are incremented every time an instrumented basic block is entered. At program exit time, the GCOV enabled main wrapper function walks the `bb_head` list and for each `bbobj` encountered, it creates a file `sourcefile.da` (note the source file name was stored in the structure) and populates the file with the size of the vector and the counters of the vector itself.

In the post mortem phase, the GCOV utility integrates and relates the information of the *.bbg, *.bb, and the *.da to produce the *.gcov files containing per line coverage output as shown in Figure 2. For instrumented lines that are executed at least once, GCOV prefaces the text with the total number of times the line was executed. For instrumented lines that were never executed, the text is prefaced by the string `######`. For any lines of code that were not instrumented, such as comments, nothing is added to the line by GCOV.

example.c

```
          int main() {
     1      int i;

     1      printf("starting example\n");
    11      for(i=0;i<10;i++) {
    10        printf("Counter is at %d\n",i);
    10      }

            /* this is a comment */

     1      if(i==1000) {
######        printf("This line should "
                     "never run\n");
######      }
     1      return 0;
     1    }
```

Figure 2: GCOV output

5 GCOV Kernel Support

The GCOV tool does not work as-is in conjunction with the kernel. There are several reasons for this. Though

the compilation of the kernel files also generates the `bbobj`s constructor, and the ".ctors" section, the resulting kernel has no wrapper function to call the constructors in .ctors. Second, the kernel never exits such that the *.da files can be created. One of the initial overriding requirements was to not modify the GCOV tool and the compiler.

The first problem was solved by explicitly defining a ".ctors" section in the kernel code. The new symbol __CTOR_LIST__ is located at the beginning of this section and is made globally visible to the linker. When the kernel image is linked, __CTOR_LIST__ has the array of constructor function pointers. This entire array is delimited by the ctors_start = &__CTOR_LIST__ ; ctors_end = &__DTOR_LIST__ variables, where __DTOR_LIST__ is the starting address of the ".dtors" section, which in turn contains all destructor functions. By default the counter vectors are placed in zero-filled memory and do not need to be explicitly initialized. The data collection starts immediately with the OS boot. However, the constructors are not called at initialization time but deferred until the data is accessed (see below).

The second problem is the generation of the *.da files. This problem is solved by the introduction of a /proc/gcov file system that can be accessed from "user space" at any time. The initialization, creation, and access of the /proc/gcov filesystem is provided as a loadable module "gcov-proc" in order to minimize modifications to the kernel source code itself. When the module is loaded, the constructor array is called and the `bbobj`s are linked together. After that the `bb_head` list is traversed and a `/proc/gcov` filesystem entry is created for each `bbobj` encountered. It is accomplished as follows: the kernel source tree's basename (e.g. /usr/src/linux-2.5.xx) is stripped from the filename indicated in the `bbobj`. The program extension (e.g. *.c) of the resulting relative file path (e.g. arch/i386/kernel/mm.c) is changed to (*.da) to form the path name of the entry. Next, this pathname is traversed under the /proc/gcov filesystem and, for newly encountered levels (directory), a directory node is created. This process is accomplished by maintaining an internal tree data structure that links `bbobj`s and `proc_entries` together. Because GCOV requires all *.c, *.bb, *.bbg, and *.da files to be located in the same directory, three symbolic links for the *.{c | bb | bbg } files are created in their appropriate kernel source path. This is shown in Figure 3.

Reading from the *.da file gives access to the underlying vector of the corresponding file in the format expected by the GCOV tool. Though the data is accessed on a per file basis, a /proc/gcov/vmlinux file to dump the entire state of all vectors is provided. However, reading the full data from /proc/gcov/vmlinux would require a change of GCOV. Resetting the counter values for the entire kernel is supported through writing to the /proc/gcov/vmlinux file using `'echo "0" > /proc/gcov/vmlinux'`. Individual file counters can be reset as well by writing to its respective *.da file under the /proc/gcov filesystem.

5.1 Dynamic Module Support

Dynamic modules are an important feature that reduces the size of a running kernel by compiling many device drivers separately and only loading code into the kernel when a specific driver is needed. Like the kernel, the module compile process inserts the coverage code making counter collection automatic through code execution. In order to access the GCOV results through /proc/, the ".ctors" section of the module code needs to be executed during the dynamic load. As stated above, the GCOV enabled kernel locates the array of constructor routine function pointers starting from `ctors_start` to `ctors_end`. Due to the dynamic nature of a module, its constructors are not present in this section. Instead, to obtain the constructor address of the dynamic module, two more variables were added to "struct module" (ie. ctors start and end) the data structure which is passed along by the modutils to the kernel each time a module is loaded. When the dynamic module is loaded, the kernel invokes the constructor which results in the linkage of the `bbobj` to the tail of the `bb_head`. If the gcov-proc module has already been loaded, then the /proc/gcov filesystem entries are added. If the gcov-proc module is not loaded, their creation is deferred until the gcov-proc is loaded. Once when gcov-proc is loaded, the /proc/gcov filesystem is created by traversing the `bb_head` list.

Dynamic modules will create their proc file system under /proc/gcov/module/{path to the dynamic module source}. A module's .da and symbolic links *.{c | bb | bbg} will be automatically removed at the time of module unloading.

Up to Linux 2.5.48, all the section address manipulation and module loading part are performed by modutils. Accordingly we modified the modutils source to store the constructor address of the module which is loaded and pass that to the kernel as an argument. Starting with Linux 2.5.48, Rusty Russell implemented an in-kernel module loader, which allows the kernel to obtain

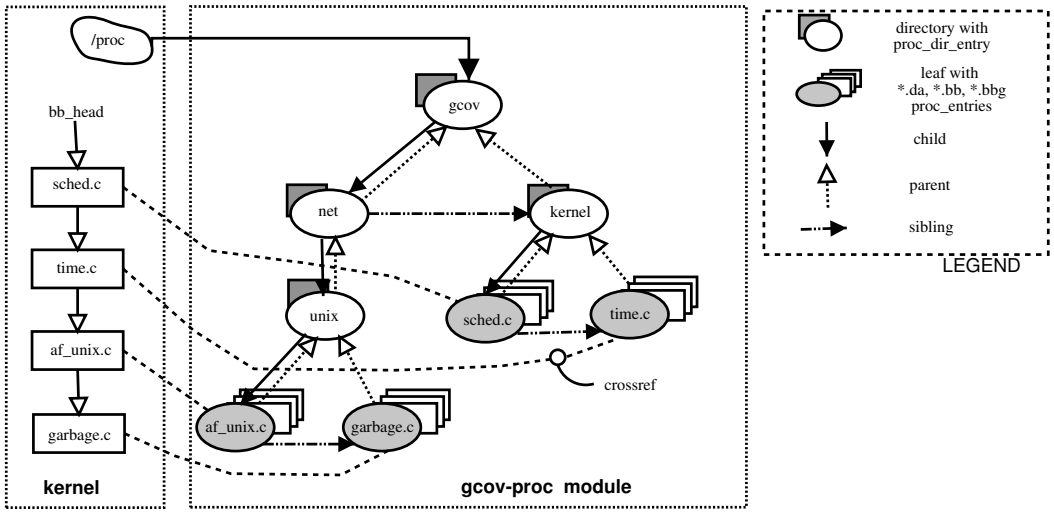


Figure 3: Data structures maintained by the gcov-proc loadable module

the constructor instead of requiring modifications to the modutils.

5.2 SMP Issues

One of the first concerns with this approach was data accuracy on symmetric multiprocessors (SMPs). SMP environments introduce the possibility of update races leading to inaccurate GCOV counter data. The machine level increment operation used to update the GCOV counter is not atomic. In CISC and RISC architectures such as Intel® and PowerPC®, memory increment is effectively implemented as load, add, & store. In SMP environments, this can result in race conditions where multiple readers will load the same original value, then each will perform the addition and store the same new value. Even if the operation is atomic on a uniprocessor, caching on SMPs can lead to unpredictable results. For example, in the case of three readers, each would read the value n, increment and store the value n+1. After all three readers were complete the value would be n+1. The correct value, after serializing the readers, should be n+3. This problem exists on SMPs and multi-threaded environments using GCOV in kernel or user space. Newer versions of GCC are experimenting with solutions to this issue for user space by duplicating the counter array for each thread. The viability of this approach is questionable in the kernel where GCOV would consume about 2 megabytes per process. We discuss this issue further in the Future Work section (10).

A test was constructed to observe the anomaly. A normally unused system call was identified and was used so the exact expected value of the coverage count would be known. Two user space threads running on different CPUs made a total of 100K calls to the `sethostname` system call. This test was performed 35 times and the coverage count was captured. On average, the coverage count for the lines in the `sethostname` kernel code was 3% less than the total number of times the system call was made.

The simplest solution was to ensure that counter addition was implemented as atomic operations across all CPUs. This approach was tested on the x86 architecture where a `LOCK` instruction prefix was available. The LOCK prefix uses the cache coherency mechanism, or locks the system bus, to ensure that the original operation is atomic system-wide. An Assembler pre-processor was constructed to search the assembly code for GCOV counter operands (LPBX2 labels). When found, the LOCK prefix would be added to these instructions. A GCC machine dependent *addition* operation was also constructed for x86 to accomplish the same task.

The results of the sethostname test on the new locking gcov kernel (2.5.50-lock-gcov) showed no difference between measured coverage and expected coverage.

5.3 Data Capture Accuracy

Another effect on data accuracy is that capturing GCOV data is not instantaneous. When we start capturing the data (first item on bb_head) it may have value $\overline{v}_{orig}$. By

the time the last item on bb_head is read, the first item may now have value $\overline{v}_{new}$. Even if no other work is being performed, reminiscent of the Heisenberg effect, the act of capturing the GCOV data modifies the data itself. In Section 8 we observed that 14% of the kernel is covered just from running the data capture. So 14% coverage is the lowest result we would expect to see from running any test.

5.4 GCOV-Kernel Change Summary

In summary, the following changes were made to the linux kernel:

(a) modified head.S to include symbols which identify the start and end of the .ctors section

(b) introduced a new file kernel/gcov.c where the `__bb_init_func(void)` function is defined

(c) added two options to the configure file, (i) to enable GCOV support and (ii) enable the GCOV compilation for the entire kernel. If the latter is not desired, then it is the responsibility of the developer to explicitly modify the Makefile to specify the file(s). For this purpose we introduced GCOV_FLAGS as a global macro.

(d) Modified kernel/module.c to deal with dynamic modules.

(e) An assembler pre-processor was written to address issues with inaccurate counts on SMP systems

Everything else is kept in the gcov-proc module.

6 GCOV patch Evaluation

This section summarizes several experiments we conducted to evaluate the GCOV-kernel implementation. Unless otherwise stated all tests were conducted on a generic hardware node with a two-way SMP, 400 MHz Pentium® II, 256 MB RAM. The Red Hat 7.1 distribution (with 2.5.50 Linux kernel) for a workstation was running the typical software (e.g. Inetd, Postfix), but remained lightly loaded. The GCC 2.96 compiler was used.

The total number of counters in all the .da files produced by GCOV was calculated for 2.5.50. 336,660 counters * 4 bytes/counter $\Rightarrow$ 1.3 Mbytes of counter space in the kernel[1]. The average number of counters per file is 718. *drivers/scsi/sym53c8xx.c* had the most lines instrumented with 4,115. And *include/linux/prefetch.h* was one of 28 files that had only one line instrumented.

Test kernel	runtime (sec)
2.5.50	596.5
2.5.50-gcov	613.0
2.5.50-gcov-lock	614.0

Table 1: Results: System and User times (in seconds) for 2.5.50 kernel compiles using three test kernels.

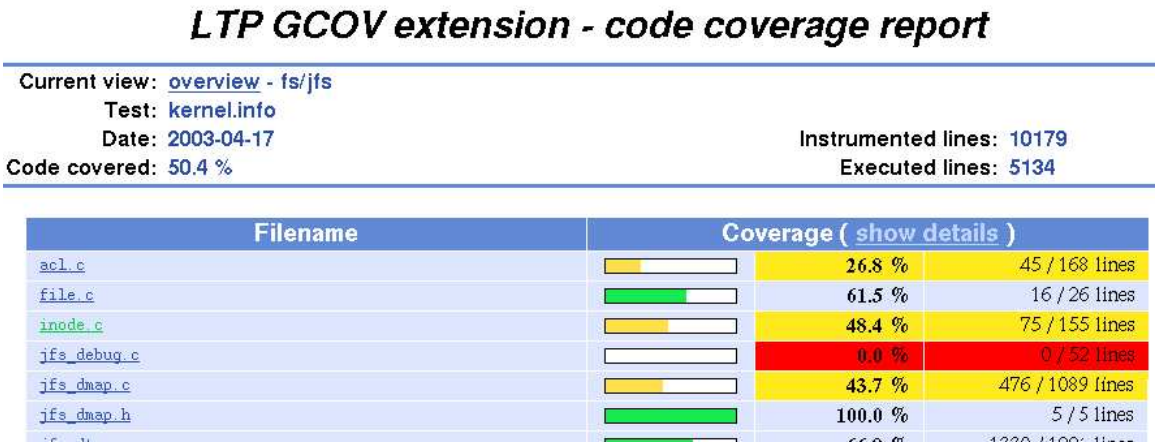
6.1 LOCK Overhead

A micro-benchmark was performed to estimate the overhead of using the LOCK prefix. From two separate loops, multiple increment and locked increment instructions were called. The runtime for the loops were measured and an estimate of the average time per instruction was calculated. The increment instruction took $0.015\mu s$, and the locked increment instruction took $0.103\mu s$. This represents a 586% instruction runtime increase when using the LOCK prefix. At first this might seem like a frightening and unacceptable overhead. However, if the actual effect on sethostname runtime is measured, it is $19.5\mu s$ per system call with locking and $13.9\mu s$ per call without locking. This results in a more acceptable 40% increase. However, further testing is required to ensure results do not vary depending on lock contention.

6.2 Performance Results

Finally, a larger scale GCOV test was performed by comparing compile times for the Linux kernel. Specifically each test kernel was booted and the time required to compile (execute *make bzImage*) the Linux 2.5.50 kernel was measured. Table 1 shows runtimes (in seconds) as measured by */bin/time* for the three kernels tested. These results show a 3% overhead for using the GCOV kernel and a very small additional overhead from using the locking GCOV kernel.

[1] in GCC 3.1 and higher counters are 8 bytes.

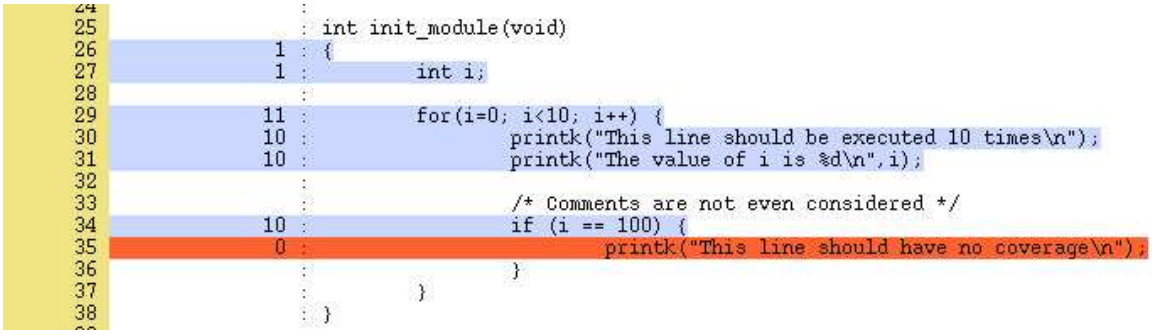


LTP GCOV extension - code coverage report

Current view: overview - fs/jfs
Test: kernel.info
Date: 2003-04-17
Code covered: 50.4 %
Instrumented lines: 10179
Executed lines: 5134

Filename	Coverage (show details)		
acl.c		26.8 %	45 / 168 lines
file.c		61.5 %	16 / 26 lines
inode.c		48.4 %	75 / 155 lines
jfs_debug.c		0.0 %	0 / 52 lines
jfs_dmap.c		43.7 %	476 / 1089 lines
jfs_dmap.h		100.0 %	5 / 5 lines

Figure 4: Partial LCOV output for fs/jfs



```
                :
                : int init_module(void)
          1     : {
          1     :         int i;
                :
         11     :         for(i=0; i<10; i++) {
         10     :                 printk("This line should be executed 10 times\n");
         10     :                 printk("The value of i is %d\n",i);
                :
                :                 /* Comments are not even considered */
         10     :                 if (i == 100) {
          0     :                         printk("This line should have no coverage\n");
                :                 }
                :         }
                : }
```

Figure 5: Example LCOV output for a file

7 Using GCOV and LCOV to generate output

Once the .da files exist, GCOV is capable of analyzing the files and producing coverage output. To see the coverage data for a user space program, one may simply run 'gcov program.c' where program.c is the source file. The .da file must also exist in the same directory.

The raw output from GCOV is useful, but not very readable. In order to produce nicer looking output, a utility called LCOV was written. LCOV automates the process of extracting the coverage data using GCOV and producing HTML results based on that data [Lcovsite].

The LCOV tool first calls GCOV to generate the coverage data, then calls a script called geninfo to collect that data and produce a .info file to represent the coverage data. The .info file is a plaintext file with the following format:

```
TN: [test name]
SF: [path and filename of the source file]
FN: [function start line number],[function name]
DA: [line number],[execution count]
LH: [number of lines with count > 0]
LF: [number of instrumented lines]
end_of_record
```

Once the .info file has been created, the genhtml script may be used to create html output for all of the coverage data collected. The genhtml script will generate both output at a directory level as illustrated in Figure 4 and output on a file level as illustrated in Figure 5. The basic process flow of the LCOV tool is illustrated in Figure 6. The commands used to generate kernel results are as follows:

1. Load the gcov-proc kernel module.
2. `lcov -zerocounters`
   This resets the counters, essentially calling 'echo "0" > /proc/gcov/vmlinux'
3. Run the test.

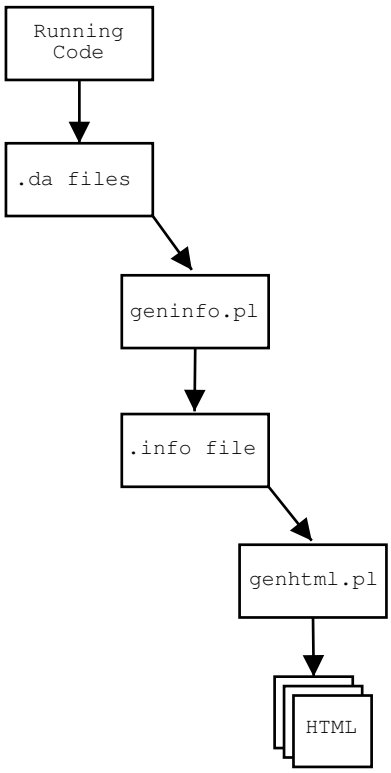


Figure 6: lcov flow

4. `lcov -c -o kerneltest.info`
   This produces the .info file described above.
5. `genhtml -o [outputdir] kernel-test.info`
   The final step produces HTML output based on the data collected in step 3.

The original intent of the LCOV tool was for kernel coverage analysis, so it contains some features, such as resetting the counters through /proc, that are only for use with the gcov-proc module. However, the LCOV tool can also be used for analyzing the coverage of user space applications, and producing HTML output based on the data collected. The process for using LCOV with applications is mostly the same. In order for any coverage analysis to take place, the application must be compiled with GCC using the `-fprofile-arcs` and `-ftest-coverage` flags. The application should be executed from the directory it was compiled in, and for GCOV to work on user space programs, the application must exit normally. The process for using LCOV with an application is as follows:

1. Run the program following your test procedure, and exit.
2. `lcov -directory [dirname] -c -o application.info`
3. `genhtml -o [outputdir] application.info`

LCOV can also be used to manipulate .info files. Information describing coverage data about files in a .info file may be extracted or removed. Running `lcov -extract file.info PATTERN` will show just the data in the .info file matching the given pattern. Running `lcov -remove file.info PATTERN` will show all data in the given .info file except for the records matching the pattern, thus removing that record. Data contained in .info files may also be combined by running `lcov -a file1.info -a file2.info -a file3.info... -o output.info` producing a single file (output.info) containing the data from all files passed to lcov with the `-a` option.

8 Applying results to test development

The real benefit of code coverage analysis is that it can be used to analyze and improve the coverage provided by a test suite. In this case, the test suite being improved is the Linux Test Project. 100% code coverage is necessary (but not sufficient) for complete functional testing. It is an important goal of the LTP to improve coverage by identifying untested regions of kernel code. This section provides a measure of LTP coverage and how the GCOV-kernel can improve future versions of the LTP suite.

Results were gathered by running the LTP under a GCOV enabled kernel. There are a few things that must be taken into consideration.

It is obviously not possible to cover all of the code. This is especially true for architecture and driver specific code. Careful consideration and planning must be done to create a kernel configuration that will accurately describe which kernel features are to be tested. This usually means turning on all or most things and turning off drivers that do not exist on the system under test.

In order to take an accurate measurement for the whole test suite, a script should be written to perform the following steps:

ID	Test	KC	DC	TC
LTP	LTP	35.1	0.0	100.0
CP	Capture	14.4	0.3	97.7
MK	Mkbench	17.0	0.0	99.9
LM	Lmbench	22.0	0.3	98.5
SJ	SPECjAppServer	23.6	1.2	95.1
X	Xserver	24.7	1.7	93.0
XMLS	X+MK+LM+SJ	29.4	3.1	89.5
B	Boot	40.1	16.2	59.5

Table 2: Benchmarks: KC: kernel coverage; DC: delta coverage; TC: test coverage

1. Load the gcov-proc kernel module
2. 'echo "0" > /proc/gcov/vmlinux' to reset the counters
3. Execute a script to run the test suite with desired options
4. Gather the results using LCOV

Using a script to do this will minimize variance caused by unnecessary user interaction. Resetting the counters is important to isolating just the coverage that was caused by running the test suite.

Alternatively, a single test program can be analyzed using the same method. This is mostly useful when validating that a new test has added coverage in an area of the kernel that was previously uncovered.

Coverage tests were performed on a IBM® Netfinity 8500R 8-way x 700 MHz SMP running the 2.5.50 Linux kernel. LTP is a test suite for validating the Linux kernel. The version of LTP used in this test was ltp-20030404. LTP hits 47,172 of the 134,396 lines of kernel code instrumented by GCOV-kernel (35%)[2]. This figure alone suggests significant room for LTP improvement. However, to get an accurate picture of LTP usefulness it is also important to make some attempt to categorize the kernel code. This way we can determine how much of the *important* kernel code is being covered by LTP. For the purposes of this work we have collected coverage information from a set of benchmarks. This set is intended to represent one reasonable set of important kernel code. The code is important because it is used by our set of popular benchmarks. Table 2 shows the coverage for LTP and our suite of tests. *Boot* captures the kernel coverage during the boot process. *Xserver* captures the kernel coverage for starting the Xserver. *Capture* is the coverage resulting from a counter reset immediately followed by a counter capture. The column *KC (kernel coverage)* specifies the percentage of kernel code covered by the test. *DC (delta coverage)* is the percent of kernel code covered by the test, but not covered by LTP.

Although LTP coverage is not yet complete, we see that it exercises all but 3.1% of the kernel code used by the combined test XMLS. Another useful view of the data is the code covered by LTP divided by the code covered by the test, *percent of test coverage covered by LTP*. The column *TC (test coverage)* shows these results. For example, LTP covered (hit) 99.9% of the kernel code covered (used) by mkbench. This figure is particularly useful because it does not depend on the amount of kernel code instrumented. LTP provides almost 90% coverage of the important code used by our combined test, XMLS. This result shows very good LTP coverage for the given test suite. *Boot* should be considered separately because much of the code used during boot is never used again once the operating system is running. But even here, LTP covers 60% of the code used during the boot process.

Another benefit of code coverage analysis in the Linux Test Project is that it provides a method of measuring the improvement of the LTP over time. Coverage data can be generated for each version of LTP released and compared against previous versions. Great care must be taken to ensure stable test conditions for comparison. The same machine and same kernel config options are used each time, however the most recent Linux kernel and the most recent version of LTP is used. Changing both is necessary to prove that the code coverage provided by LTP is increasing with respect to the changes that are occurring in the kernel itself. Since features, and thus code, are more often added to the kernel than removed, this means that tests must constantly be added to the LTP in order to improve the effectiveness of LTP as a test suite.

9 Future Work

Obviously, the tools and methods described in this paper are useful in their current state. However, there is some functionality that is still required. LCOV should be modified to process and present branch coverage data currently produced by GCOV. This information would

[2]Results are based on one run of the LTP test suite. Although the number of times any one line is hit can vary randomly, we would expect much smaller variance when considering hits greater than zero.

give a better picture of kernel coverage. Many other enhancements are currently being designed or developed against the existing set of tools.

It would be very useful to know kernel coverage for a specific process or group of processes. This could be done by introducing a level of indirection, so process-specific counters can be accessed without changing the code. This approach could utilize Thread Local Store (TLS) which is gaining popularity[Drepper]. TLS uses an entry in the Global Descriptor Table and a free segment register as the selector to provide separate initialized and uninitialized global data for each thread. GCC would be modified to reference the TLS segment register when accessing global variables. For a GCOV kernel, the storage would have to be in kernel space (Thread Kernel Storage) The challenge here is to efficiently provide the indirection. Even if the cost of accessing TLS/TKS is low, it is probably infeasible to gather coverage data for all processes running on a system. The storage required would be considerable. So, a mechanism for controlling counter replication would be necessary.

Some parts of the kernel code are rarely used. For example, *init* code is used once and removed from the running image after the kernel boots. Also, a percentage of the kernel code is devoted to fault management and is often not tested. The result of this rarely used code is to reduce the practically achievable coverage. Categorizing the kernel code could isolate this code, provide a more realistic view of LTP results, and help focus test development on important areas in the kernel. The kernel code could be divided into at least three categories. Init and error path code being two separate categories. The rest of the kernel code could be considered mainline code. Coverage would then be relative to the three categories. A challenge to some of this categorization work would be to automate code identification, so new kernels could be quickly processed. In Section 8 we attempted to define *important* kernel code based on lines hit by some popular benchmarks. This approach could also be pursued as a technique for categorizing kernel code.

Finally, a nice feature to have with the system would be the ability to quickly identify the coverage specific to a particular patch. This could be accomplished with a modified version of the *diff* program that is GCOV output format aware. Such a system would perform normal code coverage on a patched piece of code, then do a reversal of the patch against GCOV output to see the number of times each line of code in the patch was hit. This would allow a patch developer to see how much of a patch was being tested by the current test tools and, if necessary, develop additional tests to prove the validity of the patch.

10 Conclusions

This paper described the design and implementation of a new toolkit for analyzing Linux kernel code coverage. The GCOV-kernel patch implements GCOV functionality in the kernel and modules, and provides access to results data through a /proc/gcov filesystem. LCOV processes the results and creates user friendly HTML output. The test results show minimal overhead when using the basic GCOV-kernel. The overhead of an SMP-safe extension can be avoided when precise counter data is not necessary. This paper also described how this toolkit is being used to measure and improve the effectiveness of a test suite such as the Linux Test Project.

This system is already sufficient for use in producing data that will expose the areas of the kernel that must be targeted when writing new tests. It will then help to ensure that the new tests written for the Linux Test Project are successful in providing coverage for those sections of code that were not covered before. The actual implementation of the methods and tools described above as a means of improving the Linux Test Project is just beginning as of the time this paper was written. However, several tests such as mmap09, as well as LVM and device mapper test suites have been written based solely on coverage analysis. Many other tests are currently in development.

References

[Cornett] *Code Coverage Analysis* `http://www.bullseye.com/coverage.html` (2002)

[Drepper] Urlich Drepper, *ELF Handling For Thread Local Storage* `http://people.redhat.com/idrepper/tls.pdf` (2003)

[Lcovsite] *LTP Gcov Extension* `http://ltp.sourceforge.net/coverage/lcov.php` (2002)

[Marick] Brian Marick, *The Craft of Software Testing*, Prentice Hall, 1995.

[Perry] William E. Perry, *Effective Methods for Software Testing, Second Edition*, Wiley Computer Publishing, 2000.

Legal Statement