Linux Kernel GCOV - tool analysis

Nicholas Mc Guire

Distributed & Embedded Systems Lab
SISE,Lanzhou University, Lanzhou,P.R.China
mcguire@lzu.edu.cn, http://dslab.lzu.edu.cn

February 8, 2006

Contents

Version	Author	Date	Comment
1.0	Nicholas Mc Guire	Jan 2005	First shot
1.1	Georg Schiesser	18 Jan 2005	converted to TEX document
1.2	Nicholas Mc Guire	Jan 2006	2.6 revision

This gcov intoduction/manual is released under FDL V1.2 [1]. All software used for this session is available under GPL V2 license [2].

1. Kernel gcov support - tool analysis

In the framework of Work Package 5 - Boot-Time Optimization, of ”A Comparative Study on Real-time enhanced Linux Variants” conducted for Siemens CT SE2, Muenchen, research on existing tools to analize boot-times was performed. In this article, derived from analysis notes, we describe the tools basics and usage. The intention of this article is to provide practical guidance for engeneers using these tools and provide concept basics so that thes free-software tools are no long black-boxes. For a general introduction to runtime debugging in embedded systems we refere you to [6].

As one of the well known tools for user-space applications extending gcov into kernel space seems like a quite natural thing to do. In this article we describe the tools analysis, gcov usage, and data acquisition for the 2.4.25 and 2.6.14 kernel.

A brief introduction to the core technology concept and its application in user-space process and libraries is given.

Feedback to `mcguire@lzu.edu.cn` is always appreciated. The latest version of this document is available at `http://dslab.lzu.edu.cn`.

This manual assumes a default installation of Slackware [3] 10.0 or 10.1 - though it should apply to more or less any current distribution.

1.1. Source

`lcov-1.4.tar.gz` (not strictly required) `http://sourceforge.net/projects/ltp -> gcov-2.6.X.patch.gz`

note that some patches use the naming scheme `linux-2.6.X-gcov.patch.gz`.

dependencies: none

1.2. patch file

- /drivers/gcov/gcov-core.c:
  The gcov core functions for initializing logging of code coverage data
- `/drivers/gcov/gcov-proc.c`:
  The proc interface built under /proc/gcov
- /include/linux/gcov.h:
  GCOV related macros and function prototypes, the struct bb is declared here - highly gcc version dependent.

Due to this being quite compiler dependent gcov-core.c is a bit of a mess, basically it is the same function set for three different compiler versions, ifdef'ed .

Note that the actual instrumentation is done by gcc's -fprofile-arcs and -ftest-coverage flags, the kernel patch only needs to make the data accessible (you actually can compile a kernel with `CFLAGS_KERNEL=-fprofile-arcs -ftest-coverage` even without the patch applied, it would only fail in the linking stage with an unresolved symbol to `__bb_init_func` - that is exactly what gcov-core provides.

1.3. Patch analysis

The gcov interface is cleanly encapsulated in `/driver/gcov` - this code is not really architecture dependent. All gcov related parts are cleanly ifdef'ed in the code so turning off gcov support leaves no side effects.

The main changes of the patch are in the configuration (Kconfig for 2.6.X) and in the Makefiles - the patch is not very invasive at code level (though it does off course change the runtime behavior of every function).

Fundamentally `gcov-core.c` is based on a doubly linked list of struct bb (basic blocks) in which all data is collected - see `include/linux/gcov.h` for details on `struct bb`. One such list is initialized for each module. This is also one source of the performance penalty as these lists can become quite large and thus increase the cache misses.

1.4. Architecture dependent changes

The only really architecture specific issue is the sections added for the constructor and destructor functions. Though this is not really that arch dependent, but can't be placed in any arch independent file.

```
/arch/i386/kernel/head.S

.section ".ctors","aw"
.globl  __CTOR_LIST__
.type   __CTOR_LIST__,@object
__CTOR_LIST__:
.section ".dtors","aw"
.globl  __DTOR_LIST__
.type   __DTOR_LIST__,@object
__DTOR_LIST__:
```

Changes to the module interface

Every module has to know where it't gcov related constructor and destructor functions are located - so the struct module has two additional fields:

```
const char *ctors_start; /* Pointer to start of .ctors-section */
const char *ctors_end;   /* Pointer to end of .ctors-section */
```

which are used in the `load_module`, `sys_init_module` and `free_module` function (`kernel/module.c`) to initialize the gcov data acquisition:

- `load_module`:

```
modindex = find_sec(hdr, sechdrs, secstrings, ".ctors");
mod->ctors_start = (char *)sechdrs[modindex].sh_addr;
mod->ctors_end   = (char *)(mod->ctors_start +
                      sechdrs[modindex].sh_size);
```

- `sys_init_module`:

```
if (mod->ctors_start && mod->ctors_end) {
        do_global_ctors(mod->ctors_start, mod->ctors_end, mod);
}
```

- `free_module`:

```
if (mod->ctors_start && mod->ctors_end)
        remove_bb_link(mod);
```

1.5. Architecture support

i386, ia64, ppc, ppc64, s390. Though it should not be too difficult to extend to further architectures - none of the patch components are actually arch dependent

1.6. Basic technology

gcov is a test coverage program. It operates in conjunction with GCC's `-fprofile-arcs` and `-ftest-coverage`. The main goals are code coverage and hot-spot location. With some limits you can use gcov as a profiling tool to help discover where your optimization efforts will best affect your code, the main limitation being the system load specific results of gcov - thus optimization is for a specified load profile and generally implies decreasing performance in other system load scenarios. The main questions kernel-gcov can answer are:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

The flags used to enable profiling have the following effects:

1.6.1. `-fprofile-arcs`

During execution the program records how many times each branch is executed and how many times it is taken. This data is stored in the `filename.da` file in `/proc/gcov/PATH_IN_KERNEL_TREE/` for every kernel file `filename`. This profiling data is collected by instrumenting the functions. The results from `-fprofile-arcs` is what later can be used to optimize the system by feeding basic-block information back with `-fbranch-probabilities` (`-ftest-coverage` is not needed for this purpose). The instrumentation is not as simple as the one for KFI (Kernel Function Instrumentation) that simply adds entry and exit code to every function (see `-finstrument-functions` in the GCC manual and KFI [?]), `-fprofile-arcs` is more selective, and far more light-weight.

For each function GCC creates a program flow graph, then finds a spanning tree for the graph (that is eliminating loops and redundant paths). Only arcs that are not on the spanning tree have to be instrumented by adding code to count how often these arcs are executed. A arc that is not on the spanning tree is an entry or exit point into the function in question. When an arc is the only exit or only entrance to a block it is directly instrumented, if not, a new basic block is created and instrumented.

Since spanning tree creation starts with block 0, low numbered arcs are more likely to end up on the spanning tree than high numbered arcs. This causes most instrumented arcs to be at the end, which implies a asymmetric distortion of the kernel code - thus timing information gained from gcov instrumented kernels are most likely not reliable. For details on this see the file `gcc/gcov.c` in the gcc source tree.

The actual code change done in the files is to initialize an array the size of the arcs found in the file and then to instrument them with a 64 bit counter for each arc (implemented as two 32 bit values).

```
    addl  $1, .LPBX2+OFFSET
    adcl  $0, .LPBX2+OFFSET+4
```

The LPBX2 array (conforming to the gcov .da file format) can be quite large (i.e. `kernel/sched.c` 13k , `drivers/ide/ide-disk.c` 11k), the overall kernel size is increased by roughly 60%.

The overhead of the instrumentation code it self is thus not to wild - the cache side effects are more dramatic (see section performance below).

1.6.2. `-ftest-coverage`

This flag to gcc tels it to dump the profiling data to files for the gcov code-coverage utility. For an introduction to the gcov utility see the info pages to gcov `info gcov`. Note that `-ftest-coverage` in user-space causes profiling data to be generated on program exit only, which kernel gcov continuously updates the arc counters accessible via `/proc/gcov/`, this has the side effect that you never get a consistent trace state if looking at multiple files. The practical consequence of the non-synchronous file generation via the `proc` files is that you can't trace the effects of short running programs. To see this effect we cleared the counters by writing to `/proc/gcov/vmlinux` and then imediately copied all the files - this copy operation alone creates significant counts throughout the entire kernel tree - thus distorting any application related counts. For long running applications the impact can be concidered negligable, but for short running applications the file copy operation must be considered.

1.7. Building for 2.4.X

GCOV support in the 2.4.X kernel series is still a bit experimental - for the 2.6.X series it looks like a solid tool (see the later section on 2.6.X kernel).

1.7.1. patch the kernel

```
  tar -xjf linux-2.4.25.tar.bz
  cd linux-2.4.25
  patch -p1 < ../linux-2.4.25-gcov.patch
  make menuconfig
```

configure as you would usually for a non-gcov kernel and add:

```
   GCOV coverage profiling --->
     [*} GCOV Kernel
     [*] Profile entire Kernel (New)
     <*> Provide GCOV proc file system entry (New)
```

or directly modify the .config and enable the options as follows:

```
...
#
# GCOV coverage profiling
#
CONFIG_GCOV_PROFILE=y
CONFIG_GCOV_ALL=y
CONFIG_GCOV_PROC=y
```

The kernel build procedure is the usual 2.4.X procedure

```
 cp .config config_gcov
 make dep
 make modules
 make modules_install
 make bzImage
```

(note that you MUST recompile the modules as well if you enable profiling)

1.7.2. patch the modultils

Note that this step of patching the modutils is only needed for the 2.4.X kernel series it is not needed for the 2.6.X kernel releases.

```
  tar -xjf modutils-2.4.25.tar.bz2
  patch -p1 <../modutils-2.4.25-gcov.patch
  cd modultils-2.4.25
  ./configure
  make
  make install
```

1.8. building for 2.6.X

Most of the 2.4 notes apply to the 2.6 as well, never the less it seems simpler to present a clean 2.6.X shot rather than plastering the 2.4.X section with 2.6 notes...

1.8.1. Applying the Patch

```
root@rtl17:~# cd /usr/src
root@rtl17:/usr/src# tar -xjf linux-2.6.14.tar.bz2
root@rtl17:/usr/src# gunzip linux-2.6.14-gcov.patch.gz
root@rtl17:/usr/src# cd linux-2.6.14
root@rtl17:/usr/src/linux-2.6.14# patch -p1 --dry-run < \
../linux-2.6.14-gcov.patch
patching file arch/i386/boot/compressed/Makefile
patching file arch/i386/Kconfig
Hunk #1 succeeded at 1004 (offset 20 lines).
...
patching file scripts/Makefile.build
patching file scripts/Makefile.lib
```

Note that there are a few hunks when patching against vanilla linux that show offsets - but no functional problems have been found with this, this is most likely due to the "clean"-tree not being quite unmodified on the patch authors system or due to the use of early kernel versions. As most kernel patches patch clean, you can select a kernel that has no offsets if unsure (i.e. systems that have validation demands should not use any patches that are not clean).

As long as there are no failures we can continue to apply the patch - if offsets are large (as above with 20 lines offset) it is advisable though to check where the patch was applied.

```
root@rtl17:/usr/src/linux-2.6.14# patch -p1 < \
../linux-2.6.14-gcov.patch
```

Note that the patch naming is a bit inconstant as it is sometimes called `linux-2.6.X-gcov.patch.gz` and sometimes `gcov-2.6.X.patch.gz`.

1.9. Configuration

We recommend doing a make distclean before configuration to ensure that all patch related files are cleanly removed - otherwise they tend to clutter up the source management system and nobody knows where they came from...

```
root@rtl17:/usr/src/linux-2.6.14# make distclean
root@rtl17:/usr/src/linux-2.6.14# make menuconfig
```

```
 GCOV coverage profiling  --->
  [*] Include GCOV coverage profiling
  [ ]   Profile entire Kernel (NEW)
  <*>   Provide GCOV proc file system entry
  [ ]   Support for modified GCC version 3.3.x (hammer patch) (NEW)
```

These two are mandatory (proc filesystem support can be a module though). In case one only want to profile specific code parts of the kernel one needs to add the following lines to the respective Makefile

```
 EXTRA_CFLAGS += $(GCOV_FLAGS)
```

where by `$(GCOV_FLAGS` evaluates to the known `-fprofile-arcs -ftest-coverage`, alternative one can simple check:

```
 [*]   Profile entire Kernel (NEW)
```

to profile the entire kernel. Note that we did not check/use the `gcc-3.3.x` extensions so we simply don't know if there are any open issues related to that extension.

The rest is to be configured as usual for the respective platform.

Regarding profiling together with gcov one should note that more or less any kernel debug option will distort the gcov output - so if you do run gcov with oprofile or kernel hacking options enabled you should rerun for the final production kernel settings and cross check. If you locate "hot-spots" in the gcov output the same holds true as well.

```
root@rtl17:/usr/src/linux-2.6.14# cp .config config_gcov
root@rtl17:/usr/src/linux-2.6.14# make bzImage
root@rtl17:/usr/src/linux-2.6.14# make modules
root@rtl17:/usr/src/linux-2.6.14# make modules_install
```

1.9.1. Update Lilo

Add the following lines to /etc/lilo.conf - off course using your settings for root:

```
image = /boot/2614gcov
  root = /dev/hda2
  label = 2614gcov
  read-only
```

and run lilo to update the MBR entries.

```
root@rtl17:/etc# lilo
Added Linux *
Added 2614gcov
```

To reboot into the new kernel you can select it at the Lilo prompt or use lilo's one-time selection like so:

```
root@rtl17:/etc# lilo -R 2614gcov
root@rtl17:/etc# reboot
```

1.9.2. Update GRUB

To boot into the new kernel with grub add the folowing line to your `menu.lst` on many distributions it will be found in `/boot/grub`, though this is not mandatory, thus don't be supprised if you don't find it there.

```
title  2614gcov
kernel (hd0,1)/boot/2614gcov root=/dev/hda2 read-only
```

Note that grub starts counting partitions at 0 thus `/dev/hda2` maps to `hd0,1`. As grub knows how to read filesystems you don't need to reinstall grub, but simply reboot after adding the above entry and select it at the boot-prompt - grub does not have a `lilo -R target` alike command, once you find your nwe kenrel is ok you can set the default boot to the target to boot, for details we refer you to the man pages of grub.

1.10. Runtime Configuration

gcov's instrumentation is statically compiled into the kernel, the following parameters (either as kernel command-line parameters or as module parameters to gcov-proc) allow some tuning of its behavior:

- gcov_link=#
  (0/1) default setting is 1. If set to 1, symbolic links to source and GCOV graph files (`.bbg`) are created in `/proc/gcov` along with the data files (`.da`). This requires that the source tree of the compiled kernel is available on the system (or all links will be broken) - for embedded systems this is not really needed as the analysis is generally done off-line any way. The symbolic link creation does not have any runtime impact though so it is not critical if set or not.
- gcov_persist=#
  (0/1)(default setting is 0. If set to 1, GCOV data for dynamically loaded kernel modules are kept after module unload, so that coverage measurements can be extended all the way to module cleanup code. To clear persistent data, write to /proc/gcov/vmlinux (i.e. echo 0 ¿ /proc/gcov/vmlinux should do).

1.11. Data acquisition

After the system reboot you should see a message in the kernel log messages.

```
root@rtl17:~ # dmesg | grep gcov
gcov-core: initializing core module: format=pre-gcc 3.4
gcov-core: init done
gcov-proc: initializing proc module: persist=0 link=1 format=gcc 3.3
gcov-proc: init done
```

If you don't see this or a similar message then gcov is not properly patched or configured.

1.11.1. File content

For every source file in the kernel tree the following files are created in `/proc/gcov`:

- `sched.bb`:
  Symbolic link to the basic block file created during compilation, this profiles a mapping from basic blocks to line numbers. When running gcov on a kernel file the content of the .bb file is uses to reverse map the runtime recorded basic block execution counts with line numbers.
- `sched.bbg`:
  Symbolic link to the arc list of the program flow graph created during compilation. gcov uses this to map arc execution counts back to basic blocks and reconstruct the program flow graph from the runtime data.
- `sched.c`:
  Symbolic link to the source file
- `sched.da`:
  Proc pseudo file giving access to the runtime arc execution counts

For more details on this see `info gcov` or the source file for gcov in gcc-3.4.4/gcc/gcov.c.

1.12. Extracting profiling data

As with the 2.4 patches the code coverage data is accessible via `/proc/gcov`, To actually use the data for code coverage analysis and compile time feedback optimization copy all the data files (.da extension) to the kernel source tree with the most natural UNIX command sequence:

```
root@rtl17:/etc # cd /proc/gcov
root@rtl17:/proc/gcov # for F in ‘find . -name *.da‘ ; do \
cp -p $F /usr/src/linux-2.6.14/$F ; done
root@rtl17:/proc/gcov # cd /usr/src/linux-2.6.14/
root@rtl17:/usr/src/linux-2.6.14 #
```

This now copied all the profiling data files to the kernel tree, note that this is only one "snapshot" of the systems code coverage behavior - so you most likely want to run your application/system specific load scenarios for considerable time to get reliable data. Now to get the branch statistics for the scheduler do

```
root@rtl17:/usr/src/linux-2.6.14 # cd kernel
root@rtl17:/usr/src/linux-2.6.14/kernel # gcov -b -c sched.c | \
tee sched_branch_profile
```

```
0.00% of 3 lines executed in file /usr/src/linux-2.6.14/include/asm/div64.h
No branches in file /usr/src/linux-2.6.14/include/asm/div64.h
No calls in file /usr/src/linux-2.6.14/include/asm/div64.h
Creating div64.h.gcov.
0.00% of 3 lines executed in file /usr/src/linux-2.6.14/include/linux/jiffies.h
...
54.55% of 1001 lines executed in file /usr/src/linux-2.6.14/kernel/sched.c
47.83% of 299 branches executed in file /usr/src/linux-2.6.14/kernel/sched.c
40.13% of 299 branches taken at least once in file /usr/src/linux-2.6.14/kernel/sched.c
31.11% of 135 calls executed in file /usr/src/linux-2.6.14/kernel/sched.c
Creating sched.c.gcov.
root@rtl17:/usr/src/linux-2.6.14/kernel #
```

As we tee'ed it into a file - the branch information is now available in sched_branch_profile - note that this is the summary information staring from boot onward - so this does not reflect a specific load profile.

The second method you can use is to directly run gcov in /proc/gcov/WHATEVER/ but the problem with this is that every run will change the results so you don't get a snapshot that you can really analyze - it would be a cool feature to add to gcov for linux to allow to halt gcov with some simple cat 0¿ /proc/something (unfortunately this would probably require changing the now quite light weight instrumentation code - at least we could not think of a simple way how to do this).

For completeness - to get the profiling data in the running system per file.

```
root@rtl17:~# cd /proc/gcov/kernel
root@rtl17:/proc/gcov/kernel# gcov sched.c
...
100.00% of 21 lines executed in file /usr/src/linux-2.6.14/include/asm/bitops.h
Could not open output file bitops.h.gcov.
90.48% of 21 lines executed in file /usr/src/linux-2.6.14/include/linux/list.h
Could not open output file list.h.gcov.
49.45% of 1001 lines executed in file kernel/sched.c
Could not open output file sched.c.gcov.
```

The message "Could not open output file `FILENAME.c.gcov`" is due to `/proc/gcov` obviously not being writable - the output is correct though.

1.13. Checking Code Coverage

Collecting code coverage for a given load profile requires to reset the data files first and then rerun the copy operation for the file of interest.

```
root@rtl17:/proc/gcov/kernel # echo 0 > sched.da
root@rtl17:/proc/gcov/kernel # cd ..
root@rtl17:/proc/gcov # for F in ‘find . -name *.da‘ ; do \
cp -p $F /usr/src/linux-2.6.14/$F ; done
```

this resets the `sched.c` data file - we then rerun the cp operation, and rerun the gcov:

```
root@rtl17:/proc/gcov# cd /usr/src/linux-2.6.14/kernel/
root@rtl17:/usr/src/linux-2.6.14/kernel# gcov -b -c sched.c
...
33.37% of 1001 lines executed in file /usr/src/linux-2.6.14/kernel/sched.c
24.75% of 299 branches executed in file /usr/src/linux-2.6.14/kernel/sched.c
16.39% of 299 branches taken at least once in file /usr/src/linux-2.6.14/kernel/sched.c
11.85% of 135 calls executed in file /usr/src/linux-2.6.14/kernel/sched.c
Creating sched.c.gcov.
```

1.14. Kernel Optimization

Feedback of profile data to the kernel for optimization purposes is done by providing the code coverage information at compile time, this allows gcc to improve branch prediction. Note also that hard-coded `__builtin_expect`, known in the kernel sources as `likely()/unlikely()` are overruled by the gcov data when feeding back profiling data to the compiler - we are currently not aware of any compiler flag to reverse this behavior.

The procedure shown here is for 2.4.25 it applies to all gcov patched kernels without any version specifics (if you do find any let us know). Note though that the exact location of some variables (i.e. `GCOV_FLAGS`) may be different but the settings shown are correct never the less.

To now feedback profiling data to the kernel during recompilation you must clean the sources (make clean/distclean does not matter), then copy the `.da` files to the kernel tree (see above) and modify the makefile a bit

```
#
# when you got the profiling data - turn this on to optimize the hell
# out of the kernel :)
CFLAGS_KERNEL = -fbranch-probabilities
#CFLAGS_KERNEL =
GCOV_FLAGS =
#GCOV_FLAGS = -fprofile-arcs -ftest-coverage
```

Note that you must rerun make menuconfig and disable GCOV this time. The `-fbranch-probabilities` compiler flag is what tells gcc to read the `.da` files and use this data to adjust placement of code (inline or at an appended tag), to set branches according to the probabilities found. This obviously is `load/profile` dependent. Note this included overriding the builtin_expec() feature of gcc - which can be undesired if one wants to optimize for a particular load profile but not modify some highly optimized code for the general case - due to the structure of the Linux kernel source tree the `GCOV_FLAGS` apply at directory granularity only.

1.14.1. X86 Note

Due to a stupidity of the BTB design in X86 hardware feedback of the profiling data on x86 can worsen the systems behavior - in fact rerunning lmbench on x86 (ia32) with profiling feedback showed a slight degradation of almost all benchmarks.

1.15. encountered problems

gcov kernel patch 0.5 (2.4.X kernel) usage problem description, note that these seem to be eliminated in the recent gcov kernel patches. If you are using anything newer than 2.6.0, this can be skipped.

oops when recompiling kernel with gcov disabled

copying all .da files into the kernel tree, recompiling results in compile problem, and finally into a boot-time oops.

```
copied kernel/*.da to /usr/src/linux/kernel
touch all files in the kernel tree
reset compiler flags in top level makefile
turned off gcov in .config
make dep
make bzImage
```

```
(no module rebuild !)
compile error in sched.c (no compile time error)
touch all c h and bb files
```

The problem is not module related (all drivers static in the kernel).

compile time error when building with:

```
CFLAGS_KERNEL = -fbranch-probabilities

init/version.c:0: warning: file init/version.da not found, execution counts assumed to b
sched.c: In function ‘schedule’:
sched.c:703: error: corrupted profile info: prob for 53--2 thought to be 11434
sched.c:703: error: corrupted profile info: prob for 53-54 thought to be -1433
make[2]: *** [sched.o] Error 1
make[1]: *** [first_rule] Error 2
make: *** [_dir_kernel] Error 2
```

looking at the lines that are listed as causing the problems, gcov produced the branch info as expected and delivers no errors ?

`kernel/sched.c.gcov`:

```
        ...
        -:  698:same_process:
        -:  699: reacquire_kernel_lock(current);
    16892:  700: if (current->need_resched)
        -:  701: goto need_resched_back;
        -:  702: return;
        -:  703:}
        ...
```

removed data files that reported compiler errors (as seen above) - although all of them report no error when gcov is directly run on the respective c file.

```
kernel/sched.da
kernel/modules.da
drivers/char/vt.da
mm/swapfile.da
fs/ext3/super.da
ipc/util.da
```

the system boots and is operational with limitations, loopback working, filesystem read/write ok - consoles ok - heavy load does not bring down the box - modules cause a kernel oops though (in the module loading code the modules them selves are never touched).

for branch prediction we only need `-fprofile-arcs` -> removed test-coverage and rebuilt from scratch with gcov enabled in the kernel config.

compiles cleanly without test-coverage turned on

boots ok - still a module problem

1.16. Performance Impact

Although one generally will not run a production system with gcov enabled, it is relevant to know how much overhead it produces as this overhead is a distorting factor for any given load-profile that one is trying to optimize for.

```
                 L M B E N C H  3 . 0   S U M M A R Y
                 ------------------------------------
                  (Alpha software, do not distribute)

Basic system parameters
----------------------------------------------------------------------
              OS Description              Mhz  tlb  cache  mem   scal
                                              pages line   par   load
                                                    bytes
------------- ----------------------- ---- ----- ----- ------ ----
Linux 2.6.14G       i686-pc-linux-gnu 1599    32    64 2.7700    1
Linux 2.6.14G       i686-pc-linux-gnu 1599    32    64 2.6600    1
Linux 2.6.14G       i686-pc-linux-gnu 1599    32    64 2.7000    1
Linux 2.6.14S       i686-pc-linux-gnu 1599    32    64 2.7700    1
Linux 2.6.14S       i686-pc-linux-gnu 1599    32    64 2.7500    1
Linux 2.6.14S       i686-pc-linux-gnu 1599    32    64 2.7000    1

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------
              OS  Mhz null null      open slct sig  sig  fork exec sh
                      call  I/O stat clos TCP  inst hndl proc proc proc
------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
```

```
Linux 2.6.14G 1599 0.18 0.35 3.60 5.15 27.9 0.50 1.89 188. 1224 9577
Linux 2.6.14G 1599 0.18 0.37 3.93 5.43 29.1 0.50 1.92 201. 1213 9587
Linux 2.6.14G 1599 0.18 0.57 4.01 5.53 28.1 0.50 1.86 186. 1208 9547
Linux 2.6.14S 1599 0.19 0.55 3.24 4.84 8.80 0.49 1.98 183. 1200 9523
Linux 2.6.14S 1599 0.19 0.47 3.18 4.80 22.1 0.49 2.04 174. 1106 9070
Linux 2.6.14S 1599 0.19 0.38 3.12 4.50 18.4 0.49 1.97 163. 1064 8972

Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
           OS  intgr intgr  intgr  intgr  intgr
                bit   add    mul    div    mod
------------- ------ ------ ------ ------ ------
Linux 2.6.14G 0.6300 0.6300 2.5000   25.7   26.9
Linux 2.6.14G 0.6300 0.6300 2.5100   25.7   26.9
Linux 2.6.14G 0.6300 0.6300 2.5000   25.7   26.9
Linux 2.6.14S 0.6300 0.6300 2.5100   25.7   26.9
Linux 2.6.14S 0.6300 0.6300 2.5000   25.7   26.9
Linux 2.6.14S 0.6300 0.6300 2.5100   25.7   26.9

Basic float operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
           OS  float  float  float  float
               add    mul    div    bogo
------------- ------ ------ ------ ------
Linux 2.6.14G 2.5000 2.5100   11.0 6.2800
Linux 2.6.14G 2.5100 2.5000   10.9 6.2800
Linux 2.6.14G 2.5000 2.5100   11.0 6.2800
Linux 2.6.14S 2.5000 2.5000   10.9 6.2800
Linux 2.6.14S 2.5000 2.5100   10.9 6.2800
Linux 2.6.14S 2.5100 2.5000   10.9 6.2800

Basic double operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
           OS  double double double double
               add    mul    div    bogo
------------- ------  ------ ------ ------
Linux 2.6.14G 2.5000 2.5000   10.9 5.5700
Linux 2.6.14G 2.5100 2.5000   10.9 5.6100
Linux 2.6.14G 2.5000 2.5100   10.9 5.6000
Linux 2.6.14S 2.5000 2.5000   11.0 5.6600
Linux 2.6.14S 2.5000 2.5100   11.0 5.5900
Linux 2.6.14S 2.5100 2.5000   10.9 5.5700

Context switching - times in microseconds - smaller is better
---------------------------------------------------------------------
           OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
               ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
------------- ------ ------ ------ ------ ------ ------- -------
Linux 2.6.14G 0.8800 1.1700  102.2   32.9  123.0    33.4   112.1
Linux 2.6.14G 0.8300 1.1700  102.2   33.2  123.1    33.3   111.6
Linux 2.6.14G 0.8000 0.8100  102.2   32.7  123.4    33.3   111.0
Linux 2.6.14S 0.5600 1.2100   88.0   32.8  123.5    33.7   111.0
Linux 2.6.14S 1.2400 0.6500   82.7   32.2  123.8    34.0   110.7
Linux 2.6.14S 0.6100 0.9300   89.4   33.6  122.8    33.9   111.2

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
           OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
              ctxsw       UNIX         UDP         TCP conn
------------- ----- ----- ---- ----- ----- ----- ----- ----
Linux 2.6.14G 0.880 5.775 9.76  22.5  44.7  34.0  64.7 271.
Linux 2.6.14G 0.830 6.358 9.55  22.6  38.8  35.4  71.5 279.
Linux 2.6.14G 0.800 6.979 9.68  21.8  44.6  31.0  66.2 280.
Linux 2.6.14S 0.560 5.768 9.00  17.8  38.5  25.2  51.6 100.
Linux 2.6.14S 1.240 6.114 10.4  18.2  38.4  27.9  54.3 101.
Linux 2.6.14S 0.610 5.789 9.29  18.1  34.7  26.5  58.0 108.

File & VM system latencies in microseconds - smaller is better
---------------------------------------------------------------------
           OS   0K File      10K File     Mmap    Prot   Page   100fd
              Create Delete Create Delete Latency Fault  Fault  selct
------------- ------ ------ ------ ------ ------- ----- ------- -----
Linux 2.6.14G   43.0   15.5  129.9   33.7   784.0 0.231 2.02500 6.867
Linux 2.6.14G   44.4   16.6  137.9   33.2   785.0 0.409 2.12990 6.556
Linux 2.6.14G   65.9   16.4  153.2   36.0   819.0 0.387 2.11730  14.1
Linux 2.6.14S   35.0   16.5  115.2   32.7   931.0 0.531 2.58240  13.3
Linux 2.6.14S   34.2   16.1  115.4   31.5   943.0 0.605 2.62850 5.605
Linux 2.6.14S   34.9   17.0  106.3   34.0   715.0 0.313 2.20530 5.854

*Local* Communication bandwidths in MB/s - bigger is better
-------------------------------------------------------------------
          OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
```

```
                   UNIX      reread reread (libc) (hand) read write
------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
Linux 2.6.14G 150. 960. 95.3  389.2  696.1  287.4  282.9 608. 429.6
Linux 2.6.14G 147. 842. 94.5  391.4  696.0  286.6  286.8 608. 436.1
Linux 2.6.14G 156. 845. 94.6  390.3  696.0  284.9  285.8 608. 439.5
Linux 2.6.14S 160. 157. 102.  400.6  696.2  292.0  292.6 608. 425.4
Linux 2.6.14S 154. 161. 102.  398.6  696.3  290.4  289.2 608. 430.2
Linux 2.6.14S 158. 167. 101.  398.1  696.3  288.1  289.0 608. 432.7

Memory latencies in nanoseconds - smaller is better
    (WARNING - may not be correct, check graphs)
------------------------------------------------------------------
           OS   Mhz   L1 $   L2 $    Main mem    Guesses
-------------   ---   ----   ----    --------    -------
Linux 2.6.14G  1599 1.8770   12.7       157.6
Linux 2.6.14G  1599 1.8780   12.7       157.6
Linux 2.6.14G  1599 1.8780   12.7       157.6
Linux 2.6.14S  1599 1.8780   12.8       157.5
Linux 2.6.14S  1599 1.8770   12.8       157.5
Linux 2.6.14S  1599 1.8780   12.8       157.5
```

Lines listing Linux 2.6.14S are standard (unpatched Linux), lines listing 2.6.14G are gcov patched kernel runs. Note that there is very little overhead with the exception of TCP connection times - which we believe is a result of the stack complexity and of TCP. So for some classes of systems it might be realistic to actually use gcov in production systems. With the 2.4.X patches we found larger overheads (we are not going to list them here as they are no longer current), so for 2.4.X based systems gcov is most likely not usable in production code.

Results are from three consecutive runs of lmbench-3.0-3a [4] with gcov enabled/disabled.

1.17. RT-performance impact

The overhead that is visible in scheduling jitter with gcov enabled in rtlinux patched kernel (RTLinux/GPL Version 3.2-rc1 on 2.4.29 was used) is negligible. The worst case was de-facto un-changed, the average slightly increased.

Note though that the rtlinux modules were not instrumented for gcov, so this is not too surprising. The interesting part is that the jitter was increased after rebuilding the kernel with profiling data feed back (using -fbranch-probabilities) - though again only the average case went up not the worst case (see note on BTB design of x86).

1.18. General Issues

1.18.1. Authors

Hubertus Franke <frankeh@us.ibm.como> Rajan Ravindran <rajancr@us.ibm.com>
Peter Oberparleiter <Peter.Oberparleiter@de.ibm.com>

(Pleas corect me if this information is incorrect or out dated)

1.18.2. License

```
Copyright (c) International Business Machines Corp., 2002-2003
Licensed under GPL V2
```

1.18.3. Patch status

Looks well maintained within the Linux Test Project (ltp on sourceforge) current gcov patches are at 2.6.14. The problems noted for the 2.4.25 don't seem to be around any more.

1.18.4. Related work

- KFI:
  Kernel function instrumentation - instrumentation at the call level - with precise timings but very large overhead and huge data volumes.
- Oprofile: Kernel level profiling in 2.6 (patches for 2.4.X available) statistic data of low-level events (only for x86 ?)
- UML:
  Older versions of UML supported gcov and gprof (as of 2.4.24 kernel gprof is broken - in 2.6.X gcov is supported but it looks like gprof is out for good).

1.19. Conclusion

Obviously a must for any validation effort. Aside from that it is a good starting point for any system level optimization effort - a fairly quick way to get a good picture of where hot-spots are in the kernel (estimated effort ¡= one week for a 2.6.X kernel with default configuration).

Due to the overhead and the gcov inherent distortion one needs to run this in iterative stages

- full kernel code coverage -> locate hot-spots
- gcov only turned on for the identified functions/subsystems -> reevaluate hot-spots.

Very usable, no special documentation needed, if you know how to use gcov then this will work naturally - the only "ugliness" is the need to copy the .da files to run gcov properly (this could be nicely wrapped up in a make target in the top level kernel Makefile). A further wish-list entry would be a way to start/stop data collection in a way that allows smaller load windows (though this would require a massive change in the way the instrumentation is implemented).

2. List of Acronyms

`BTB` - Branch Trace Buffer
`CVS` - Concurent Version Control
`GNU` - GNU Not UNIX (recursive acronym)
`LKML` - Linux Kernel Mailing List
`KFI` - Kernel Function Instrumentation
`GCC` - GNU C Compiler
`MBR` - Master Boot Record
`LTT` - Linux Trace Toolkit
`TSC` - Time Stamp Counter (x86)
`X86` - Intel 80X86 Processor family

References

[1] Free Software Foundation, *Free Documentation License*, as published by the Free Software Foundation, `http://www.gnu.org/copyleft/fdl.html`,FSF,2004

[2] Free Software Foundation, *General Public License*, as published by the Free Software Foundation, `http://www.gnu.org/copyleft/gpl.html`,FSF,2004

[3] Slackware Linux, *Slackware 10.1*, `http://www.slackware.org/`,Slackware Linux Inc.,2005

[4] Larry McVoy and Carl Staelin, *LMBench - Tools for Performance Analysis*, `http://lmbench.sourceforge.net/`,2005

[5] CE Linux Forum, *Kernel Function Instrumentation*, `http://tree.celinuxforum.org`, (C) CE Linux Forum Member Companies, 2005.

[6] Nicholas Mc guire, *Runtime Debuging in Embedded Systems - available tools and usage*`http://DSLabs.lzu.edu.cn/Publications.html`DSLabs, SISE, University of Lanzhou,2006

[7] OpenTech EDV Research GmbH - OpenTech documents, `http://www.opentech.at/documents.html`,OpenTech,2005