

# **Building a UML Sanbox for Embedded Linux Boot-time optimization**

Nicholas Mc Guire

Opentech EDV-Research GmbH

December 29, 2004

## Contents

|  |           |
|--|-----------|
| <b>1. Building a Linux Sand-box</b>          | <b>1</b>  |
| 1.1. What does this step add ?               | 1         |
| 1.2. Building UML                            | 2         |
| 1.3. Test filesystem                         | 4         |
| 1.3.1. Building a minimum FS for UML testing | 5         |
| 1.3.2. Test filesystem                       | 6         |
| 1.3.3. Putting it together                   | 6         |
| 1.3.4. uml Linux testing                     | 9         |
| 1.4. Kernel debugging with UML Linux         | 9         |
| 1.5. shutting down UML                       | 12        |
| 1.6. Kernel profiling                        | 12        |
| 1.7. Kernel code coverage                    | 12        |
| 1.7.1. Preparing for gcov                    | 13        |
| 1.7.2. Extracting code-coverage data         | 13        |
| 1.7.3. Limitations                           | 18        |
| 1.7.4. Cleaning up                           | 20        |
| <b>2. Conclusion</b>                         | <b>21</b> |
| <b>3. List of Acronyms</b>                   | <b>22</b> |

*Contents*

---

| Version | Author            | Date              | Comment     |
|---------|-------------------|-------------------|-------------|
| 1.0     | Nicholas<br>Guire | Mc<br>15 Dec 2004 | First shot  |
| 1.1     | Nicholas<br>Guire | Mc<br>27 Dec 2004 | Draft final |

## 1. Building a Linux Sand-box

Strictly speaking this is not real Linux at all, it is more a intermediate step if the problem can't be located at runtime easily, which is the case for the boot-process. But as the boot-time issues are located at a system scope, preventing simple isolation, UML (User Mode Linux), allows to analyze core mechanisms in a qualitative way, at this system scope, naturally with a strong impact on performance and with some constraints with respect to actual metrics and naturally with limitations concerning hardware specifics - never the less it has shown to be quite a usable tool for locating problems in embedded systems in general and is proposed as a at least instructive tool for analysis of boot-time issues.

The intention of the Linux sand-box is to analyze the target independent issues of boot-mechanisms and go to the target system, for the target specifics, as late as possible. Obviously the ability to single step the boot-process is helpful, and more so the profiling capabilities that allow fairly quick location of hot-spots in the boot-sequence.

### 1.1. What does this step add ?

Basically the change root test was importing the current environment so the testing in the chroot environment was limited to checking for binaries, libraries and system/application configuration files. What was missing is

- Kernel configuration
- System initialization phase
- Initial environment

By introducing this step we can validate the system initialization as far as it is hardware independent - that is the core mechanism can be validated - and the initial environment, which is always architecture independent.

The kernel configuration can only be validated in a limited way, core configuration settings like supported binary types and filesystem can be verified but anything that is hardware dependent naturally is missing. Also the UML kernel has some quite invasive changes so even driver validation can be somewhat limited, but basically if it fails in the UML sand-box it will most likely also fail in the real Linux kernel environment. More important it allows to located code hot-spots that may be promising for optimizations to reduce boot-times.

The clear advantage of the UML solution is that you need no root privileges for kernel reconfiguration and testing, and thus will most likely not destroy your desk-top system with some stupid typos, especially when operating on centralized resources this is relevant ! And the tests are quite a lot faster than actually rebooting a system all the time aside from the ability to monitor the system boot-process in much more detail that would be commonly allowed.

### 1.2. Building UML

Obviously you need to first download the patches and kernel source, unfortunately there are no kernel patches for all kernels, but the changes between minor release versions are generally not that invasive that findings obtained with the available systems are lost - though clearly a reassessment may be needed in some cases.

The basic steps are no different that with any other kernel patch, a somewhat peculiar behavior is related to the ARCH variable passed to calls to make as it must be passed on all calls to make and if you forget it once and call make again things are messed up. To be explicit, doing `make menuconfig`, closing it and then doing, `make menuconfig ARCH=um`, will result in a garbled configuration.

The steps are:

- unpack kernel:

```
root@rtl20:/src # ls
linux-2.4.24.tar.bz2      uml-patch-2.4.24-1      uml-patch-2.4.24-3
root@rtl20:/src # tar -xjf linux-2.4.24.tar.bz2
root@rtl20:/src # ln -s linux-2.4.24 uml
root@rtl20:/src # cd uml
```

- patch kernel:

```
root@rtl20:/src/uml # patch -p1 --dry-run < ../uml-patch-2.4.24-1
```

If all went well in the dry run then do the actual patch

```
root@rtl20:/src/uml # patch -p1 < ../uml-patch-2.4.24-1
```

- configure kernel:  
For now we simply compile everything we need into the kernel - no modules (yet).

```
make menuconfig ARCH=um
```

```
General Setup --->
```

```
...
[*] Tracing thread support
...
<*> Host filesystem
<*> Host filesystem
< > Honeypot proc filesystem
[*] Management console
[*] Magic SysRq key
...
(0) Nesting level
(1) Kernel address space size (in .5G units)
[ ] Highmeme support
[*] /proc/mm (really only needed for skas support)
(2) kernel stack size order (8kB) is enough for a sandbox
[*] Real-time Clock
```

```
Character Device --->
```

```
[*] Virtual serial line
[*] file descriptor channel support
[*] null chanel support
[*] port chanel support
[*] pty chanel support
[*] tty chanel support
[*] xterm chanel support
    Default main console channel initialization: "fd:0,fd:1"
    Default console channel initialization: "xterm"
    Default serial line channel initialization: "pty"
...
```

```
Block Devices --->
```

```
[*] Virtual block devices
...
<*> RAM disk support
(32768) Default RAM disk size (32MB for a sandbox)
[*] initial RAM disk (initrd) support
```

```
Kernel hacking --->
...
[*] Enable kernel debugging symbols
[*] Enable ptrace proxy
[*] Enable gprof support
[*] Enable gcov support
```

On exit save your configuration - the default file is `.config`, as `make mrproper` (used to really clean the source tree before recompiling) removes all files that start with `.config`, you should copy your newly created `.config` to a safe name.

```
cp .config config_um1
```

This `config_um1` is what you should check into your source management system along with the sources and patches.

- compile kernel:  
Compiling the kernel is not really any different that for a standard kernel, just the last target is somewhat special as it is actually an executable and not a x86 boot sector like the normal compressed kernels.

```
make dep ARCH=um
make ARCH=um
make linux ARCH=um
```

We are deliberately skipping modules for now - we only want to use this as a development sand-box for boot-time optimizations of embedded Linux and not as a Honey-pot - there is a lot more fun one can have with UML though ;)

### 1.3. Test filesystem

For creation of a test-filesystem pleas refer to the embedded kickstart `filesystem` in a `file` section. We assume here that the steps described there to build a filesystem in a file and populate it with a minimum root are completed so you have a file at hand that would give you a minimum embedded system.

The test filesystem described here is too little to do any real work on embedded platforms but is a good starting point for working with UML. The filesystem we will build here is a uncompressed filesystem (not a ramdisk) and is the default method that UML will access its root filesystem.

### 1.3.1. Building a minimum FS for UML testing

Brute force minimum fs build instructions - for more detailed description we refer you to the embedded kickstart session ??

```
root@rtl17: # dd if=/dev/zero of=/myfs bs=512 count=8192
root@rtl17: # mkfs.ext2 myfs
mke2fs 1.35 (28-Feb-2004)
myfs is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1024 inodes, 4096 blocks
204 blocks (4.98%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
1024 inodes per group

Writing inode tables: 0/1 done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 21 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
root@rtl17: # mkdir /myfs
root@rtl17: # mount -t ext2 -o loop myfs /myfs
root@rtl17: # tar -xjf busybox-1.0.tar.bz2
root@rtl17: # cd busybox-1.0
root@rtl17: # make menuconfig

Build Options --->
  [*] Build Busybox as a static binary (no shared libs)
  ...
Installation Options --->
  [ ] Don't use /usr
  (/myfs) Busybox install prefix

root@rtl17: # make
root@rtl17: # make install
```

### 1.3.2. Test filesystem

```
root@rtl17: # SHELL=/bin/ash chroot /myfs
```

```
BusyBox v1.00 (2004.12.27-13:56+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
#@rtl17:/# exit
```

```
root@rtl17: # sync
root@rtl17: # umount /myfs
root@rtl17: # gzip -9 myfs
root@rtl17: # cd linux-2.4.26
root@rtl17: # cp ../myfs.gz root_fs.gz
root@rtl17: # gunzip root_fs.gz
root@rtl17: # file rootfs
testfs: Linux rev 1.0 ext2 filesystem data
root@rtl17: # mount -t ext2 -o loop root_fs /myfs/
root@rtl17: # cd /myfs/
root@rtl17: # mkdir -p {proc,etc,dev}
root@rtl17: # cd dev/
root@rtl17: # mknod console c 1 5
root@rtl17: # mknod tty1 c 1 4
root@rtl17: # mknod tty2 c 2 4
root@rtl17: # mknod null c 1 3
root@rtl17: # mknod zero c 1 5
root@rtl17: # cd /root/uml/
root@rtl17: # cd linux-2.4.24
root@rtl17: # sync
root@rtl17: # umount /myfs/
```

### 1.3.3. Putting it together

Test UML with the filesystem image (uncompressed fs image - not a ramdisk !)

```
root@rtl17: # ./linux
Checking for the skas3 patch in the host...not found
Checking for /proc/mm...not found
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Checking for /dev/anon on the host...Not available (open failed with errno 2)
```

## 1. Building a Linux Sand-box

---

```
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Linux version 2.4.24-1um (root@rtl14) (gcc version 3.3.4) #1 Mon Dec 27 17:13:46 CET 2004
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/ubd0
Calibrating delay loop... 1998.84 BogoMIPS
Memory: 27160k available
Dentry cache hash table entries: 4096 (order: 3, 32768 bytes)
Inode cache hash table entries: 2048 (order: 2, 16384 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that ptrace can change system call numbers...OK
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...No, enabling workaround
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
VFS: Disk quotas vquot_6.5.1
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
JFFS version 1.0, (C) 1999, 2000 Axis Communications AB
JFFS2 version 2.1. (C) 2001 Red Hat, Inc., designed by Axis Communications AB.
pty: 256 Unix98 ptys configured
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
RAMDISK driver initialized: 16 RAM disks of 32768K size 1024 blocksize
loop: loaded (max 8 devices)
PPP generic driver version 2.4.2
Universal TUN/TAP device driver 1.5 (C)1999-2002 Maxim Krasnyansky
SCSI subsystem driver Revision: 1.00
scsi0 : scsi_debug, Version: 0.61 (20020815), num_devs=1, dev_size_mb=8, opts=0x0
  Vendor: Linux      Model: scsi_debug      Rev: 0004
  Type:   Direct-Access      ANSI SCSI revision: 03
Initializing software serial port version 1
```

## 1. Building a Linux Sand-box

---

Partition check:

```
ubda: unknown partition table
UML Audio Relay (host dsp = /dev/sound/dsp, host mixer = /dev/sound/mixer)
Initializing stdio console driver
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 4096)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (ext2 filesystem) readonly.
Mounted devfs on /dev
Bummer, could not run '/etc/init.d/rcS': No such file or directory
```

Please press Enter to activate this console.

```
BusyBox v1.00 (2004.12.27-13:56+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
-sh: can't access tty; job control turned off
/ # mount -n -t proc proc /proc
/ # cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / ext2 ro 0 0
none /dev devfs rw 0 0
/proc /proc proc rw 0 0
/ # mount -n -o remount,rw rootfs /
/ # halt
/ # umount: none busy - remounted read-only
swapoff: /etc/fstab file missing
The system is going down NOW !!
Sending SIGTERM to all processes.
Sending SIGKILL to all processes.
The system is halted. Press Reset or turn off power
System halted.
tracing thread pid = 2470
root@rtl17: #
```

The issue of interest in UML boot-time testing is not the system post-init initialization phase but the steps of kernel boot proper, that is from Now booting kernel... to Freeing unused kernel memory: 68k freed. UML can off-course also speed up optimizations

of the init process and the script for system initialization without the need to actually reboot all the time - but for that step traditional code coverage and profiling tools are also sufficiently well suited.

#### 1.3.4. uml Linux testing

copy root filesystem to root.fs (which is the default name assumed by UML so we don't need any parameters to UML for the first test) in the top level Linux directory and run.

```
./linux
```

This will now boot linux up and you can login to the UML system.

#### 1.4. Kernel debugging with UML Linux

In this sample debug session we will not go into the details of gdb's commands we just present how to access UML from the debugger. We will start it up and set a breakpoint at start\_kernel, then inspect the kernel at that point with the list command, and set a further breakpoint. We then continue and after reaching that second breakpoint let it run free by typing continue again.

```
root@rtl10: # gdb
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux".
(gdb)
```

gdb is now started we next grab the PID of it on a second terminal with pidof gdb (or if /sbin is not in your path as is usually the case for non-root users, use /sbin/pidof gdb).

```
root@rtl10: # /sbin/pidof gdb
10407
```

Next we launch UML with the following command-line

## 1. Building a Linux Sand-box

---

```
root@rtl20: # ./linux ubd0=test debug gdb-pid=10407
```

```
Checking for the skas3 patch in the host...not found
Checking for /proc/mm...not found
tracing thread pid = 10537
```

This tracing thread is the process in the host Linux system that is used to communicate with the UML - our gdb needs to attach to this process so we can access linux - the UML command-line will halt at this point until we attached the debugger and continue the UML process

```
(gdb) attach 10537
Attaching to process 10537
Reading symbols from /home/hofrat/linux-2.4.24-target/linux...done.
0xa0168e11 in kill ()
(gdb) break start_kernel
Breakpoint 1 at 0xa0002447: file init/main.c, line 362.
(gdb) c
Continuing.
```

```
Breakpoint 1, start_kernel () at init/main.c:362
362     printk(linux_banner);
(gdb) l
357     /*
358     * Interrupts are still disabled. Do necessary setups, then
359     * enable them
360     */
361     lock_kernel();
362     printk(linux_banner);
363     setup_arch(&command_line);
364     printk("Kernel command line: %s\n", saved_command_line);
365     parse_options(command_line);
366     trap_init();
(gdb) break trap_init
Breakpoint 2 at 0xa00cf183: file trap_kern.c, line 193.
(gdb) c
Continuing.
```

```
Breakpoint 2, trap_init () at trap_kern.c:193
193     }
```

```
(gdb) step
start_kernel () at init/main.c:367
367      init_irq();
(gdb)
init_irq () at irq.c:820
820      enable_irq(TIMER_IRQ);
(gdb)
816      irq_desc[TIMER_IRQ].status = IRQ_DISABLED;
(gdb)
817      irq_desc[TIMER_IRQ].action = 0;
(gdb)
818      irq_desc[TIMER_IRQ].depth = 1;
(gdb)
819      irq_desc[TIMER_IRQ].handler = &SIGVTALRM_irq_type;
(gdb)
821      for(i=1;i<NR_IRQS;i++){
(gdb)
820      enable_irq(TIMER_IRQ);
(gdb) c
Continuing.
```

At this point UML will complete booting up to the user space process (in our case a login shell from busybox). In the console where you launched UML you will see something like:

```
Linux version 2.4.24-1um (hofrat@rtl20) (gcc version 3.2.3) #1 Sun Oct 17 17:26:13 CEST 2
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
...<snip>...
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (ext2 filesystem) readonly.
Mounted devfs on /dev
init started: BusyBox v0.60.5 (2004.10.16-09:41+0000) multi-call binary
Bummer, could not run '/etc/init.d/rcS': No such file or directory
```

Please press Enter to activate this console.

At this like simply hit enter and log in - UML is now running under debugger control and you can stop UML by typing `!CNTRL-!C!` setting break points and typing `c` to continue (or any other gdb commands).

## 1.5. shutting down UML

In our sample session we are assuming a minimalistic setup so not even the proc filesystem was mounted at boot time. We need to mount proc manually and then we can call halt to terminate UML cleanly.

```
Busybox v0.60.5 (2004.10.16-19:41+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands
```

```
# mount -t proc proc /proc
# halt
```

```
The system is going down NOW !!
Sending SIGTERM to all processes.
Sending SIGKILL to all processes.
The system is halted. Press Reset or turn off power.
System halted.
```

```
root@rtl20: #
```

## 1.6. Kernel profiling

TODO: looks like this is broken in the later 2.4.X UML releases.

## 1.7. Kernel code coverage

What is code coverage good for ?

- to see which code in the source got executed
- to find out how often each line of code got executed
- and to locate computational hot-spots in conjunction with profiling data

### 1.7.1. Preparing for gcov

To enable code coverage of the Linux kernel it self - simply reconfigure UML with gcov support in the Kernel debug section

```
root@rtl17: # make menuconfig ARCH=um
```

```
Kernel hacking --->
[*] Enable kernel debugging symbols
[*] Enable ptrace proxy
[ ] Enable gprof support
[*] Enable gcov support
```

Note that this is not quite the same as simply compiling with `-fprofile-arcs -ftest-coverage` passed in the CFLAGS to the kernel make command. The problem with this would be that you end up with a lot of thing compiled with profiling that simply make a big mess - i.e. the files in scripts, that are used during the build process but not during kernel execution. The UML patch to the Makefiles takes care of this.

To recompile the kernel cleanly use the following procedure:

```
root@rtl17: # cp .config config_UML
root@rtl17: # make mrproper
root@rtl17: # cp config_UML .config
root@rtl17: # make oldconfig ARCH=um
root@rtl17: # make dep ARCH=um
root@rtl17: # make linux ARCH=um
```

### 1.7.2. Extracting code-coverage data

When ever you execute `./linux`, code coverage data is produced and dumped to files that carry the extension `.bb` (Basic Blocks) `.bbg` (Basic Block Graph), which are produced when including the `-ftest-coverage` in the CFLAGS to gcc, and `.da` (arc data), which are produced with include `-fprofile-arcs` in the CFLAGS.

A brief description of the gcov data formats follows in the next section - if you are not interested in the internals skip it.

The problem with the linux kernel code coverage is that you are compiling with optimization turned on - in fact you must compile with optimization or certain features of GCC (i.e. inline,

builtin\_expect, etc.) will not be available. The problem with this is that gcov is not able to follow possible reordering of execution or lines that gcc optimized away. The consequence of this is that the output of gcov sometimes seems wrong or at least strange. Before we look at the output format we will generate some code coverage data in human readable form for init/main.c the kernel main routine.

```
root@rtl17: # cd linux-2.4.26
root@rtl17: # ls init/main*
main.bb  main.bbg  main.c  main.da
root@rtl17: # gcov -o init init/main.c
root@rtl17: # ls -l main.c.gcov
-rw-r--r-- 1 root root 23500 Dec 27 20:07 main.c.gcov
```

The file main.c.gcov is where all the code coverage data is compiled together. Viewing it with an editor we can see each line of code preceded by two columns, the first column contains either a number indicating how often a line was executed, a dash - if the line in the source file contained no code (i.e. a closing braces only or the like), or ##### if the line was never executed at all. The second column contains the line number in the source file and the third column is the source line it self.

```
-: 0:Source:init/main.c
-: 0:Object:init/main.bb
-: 1:/*
-: 2: * linux/init/main.c
-: 3: *
...
-: 14:#include <linux/config.h>
-: 15:#include <linux/proc_fs.h>
-: 16:#include <linux/devfs_fs_kernel.h>
-: 17:#include <linux/unistd.h>
```

The files start with line 1 - the lines marked with 0: were added in by gcov. All the comment files obviously were not executed, this is also true for the #include lines. In case the included files them selves contain code that was executed a file is created with the header filename as base-name and the .gcov extensions.

```
root@rtl17: # ls -l *.gcov
-rw-r--r-- 1 root root 23500 Dec 27 20:07 main.c.gcov
-rw-r--r-- 1 root root 20749 Dec 27 20:07 string.h.gcov
-rw-r--r-- 1 root root 5212 Dec 27 20:07 unistd.h.gcov
```

## 1. Building a Linux Sand-box

---

These files now only contain the code from string.h and unistd.h that were called within main.c - so you can have multiple string.h.gcov files if looking at different files - and you naturally will overwrite them if executing gcov on a different source file from the same trace - so one must be quite careful when picking apart a source tree.

A function that was never executed would appear like below:

```
 -: 131:static int __init profile_setup(char *str)
#####: 132:{
#####: 133:     int par;
#####: 134:     if (get_option(&str,&par)) prof_shift = par;
#####: 135:     return 1;
 -: 136:}
```

This is not the same thing as lines marked as empty - these lines actually occupy main memory and reduce the alignment quality of the kernel. Lines marked as containing no code can be due to them simply containing no code (like the closing parenthesis in the above example) or because it was compile time conditional code that was not included for this configuration (configured for X86 in our case):

```
 -: 508:#if defined(CONFIG_PPC)
 -: 509:     ppc_init();
 -: 510:#endif
```

A further case of "unused" code showing up in gcov output is when inline assembler code is used - in that case the asm sequence is counted as a single line and the actual assembler code appears as unused:

```
 -: 127:static inline int strcmp(const char * cs,const char * ct,size_t count)
18: 128:{
branch 0 taken 75%
branch 1 taken 100%
 9: 129:register int __res;
 9: 130:int d0, d1, d2;
 9: 131:__asm__ __volatile__(
 -: 132:     "1:\tdecl %3\n\t"
 -: 133:     "js 2f\n\t"
<snip>
 -: 141:     "3:\tsbbl %%eax,%%eax\n\t"
 -: 142:     "orb $1,%%al\n\t"
```

## 1. Building a Linux Sand-box

---

```
-: 143:     "4:"
-: 144:         : "=a" (__res), "=&S" (d0), "=&D" (d1), "=&c" (d2)
-: 145:         : "1" (cs), "2" (ct), "3" (count));
-: 146: return __res;
-: 147: }
```

The next snippet is the type of small scale optimization that we will be looking for - in this specific configuration the do loop returned from the body of the loop and did not terminate - so we can omit any code that is after the while. Naturally going through the linux kernel and plucking out code will not only make it non-portable but also non-reconfigurable so this is something you most likely only want to do for very specific systems.

```
1: 144:     p = &__setup_start;
1: 145:     do {
8: 146:         int n = strlen(p->str);
8: 147:         if (!strncmp(line,p->str,n)) {
2: 148:             if (p->setup_func(line+n))
1: 149:                 return 1;
-: 150:         }
7: 151:         p++;
7: 152:     } while (p < &__setup_end);
#####: 153:     return 0;
-: 154: }
```

This next sequence from `init/main.c` is sort off the dream result of profiling - one line of code executed 30 million times - so if you manage to optimize it only a bit or optimize it away, as is possible with the `loops-per-jiffies` patch ??, then the overall execution time reduction can be significant even if the code line itself is not really doing much work.

```
12: 175:         ticks = jiffies;
30118758: 176:         while (ticks == jiffies)
-: 177:             /* nothing */;
-: 178:             /* Go .. */
```

Aside from this code coverage based on `gcov` output file inspection, `gcov` has a few command-line switches that can help to locate optimization potentials:

```
root@rtl17: # gcov -o init -b init/main.c
100.00% of 6 lines executed in file include/asm/unistd.h
```

## 1. Building a Linux Sand-box

---

```
83.33% of 6 branches executed in file include/asm/unistd.h
66.67% of 6 branches taken at least once in file include/asm/unistd.h
100.00% of 4 calls executed in file include/asm/unistd.h
Creating unistd.h.gcov.
100.00% of 12 lines executed in file include/asm/arch/string.h
33.33% of 6 branches executed in file include/asm/arch/string.h
33.33% of 6 branches taken at least once in file include/asm/arch/string.h
No calls in file include/asm/arch/string.h
Creating string.h.gcov.
70.91% of 165 lines executed in file init/main.c
60.00% of 40 branches executed in file init/main.c
45.00% of 40 branches taken at least once in file init/main.c
80.70% of 57 calls executed in file init/main.c
Creating main.c.gcov.
```

This shows us a summary of branch execution and of calls made from main.c, along with the summary information the main.c.gcov file was also modified, it now has summary information on branches embedded in the form:

```
 -: 131:static int __init profile_setup(char *str)
#####: 132:{
#####: 133:     int par;
#####: 134:     if (get_option(&str,&par)) prof_shift = par;
call    0 never executed
branch  1 never executed
#####: 135:     return 1;
 -: 136:}
```

for a never taken code sequence - and for a actually used code sequence:

```
    1: 449:     call = &__initcall_start;
   77: 450:     do {
   77: 451:         (*call)();
call   0 returns 100%
   77: 452:         call++;
   77: 453:     } while (call < &__initcall_end);
branch 0 taken 99%
 -: 454:
```

In this case we can't eliminate any lines of code but we can improve branch prediction: it would make sense to improve the while condition by using gcc's builtin\_expect - which the linux developers alias to likely/unlikely.

That should do as an introduction to gcov - the rest is best learned by playing with the few other available options - see the man page ;)

### 1.7.3. Limitations

UML provides code coverage for the architecture independant code parts of the kernel - the real low level stuff, like the boot code, is not covered, as also the assembler files. Asside from this structural limitation the most severe problem is optimization as we will show in the following paragraph.

Its a bit of an artificial example that we created - but it demonstrates the point quite nicely - code coverage and optimization don't mix - so be carfull when interpreting gcov output files from the kernel !

In this example gcc happily optimized the entire for loops body away - recognizing that none of the variables are ever used later - never the less gcov lists them as being executed - so if you would then try to optimize these lines you would obviously not find any performance increase (see the assembler output below) !

```
 -:      0:Source:file.c
 -:      0:Object:file.bb
 -:      1:#define likely(x)      __builtin_expect((x),1)
 -:      2:#define unlikely(x)    __builtin_expect((x),0)
 -:      3:
 1:      4:main(){
 1:      5:    int i;
 1:      6:    long long loop = 1000;
 1:      7:    int j,k,l;
 1:      8:    j=k=l=0;
 1:      9:    while(loop--){
500500: 10:        for(i=0;i<loop;i++){
499500: 11:            if(unlikely(i!=0)){j++;}
499500: 12:            if(unlikely(i!=1)){k++;}
499500: 13:            if(unlikely(i!=2)){l++;}
499500: 14:            if(unlikely(i!=3)){l--;}
499500: 15:            if(unlikely(i!=4)){k--;}
499500: 16:            if(unlikely(i!=5)){j--;}

```

## 1. Building a Linux Sand-box

---

```
499500: 17:          if(unlikely(i!=0)){j++;}
499500: 18:          if(unlikely(i!=1)){k++;}
499500: 19:          if(unlikely(i!=2)){l++;}
499500: 20:          if(unlikely(i!=3)){l--;}
499500: 21:          if(unlikely(i!=4)){k--;}
499500: 22:          if(unlikely(i!=5)){j--;}
-: 23:          }
-: 24:      }
-: 25: }
```

The for loops body was all optimized away. Code coverage and `-O#` don't mix well ! Here is the assembler code of the above example (generated by `gcc -O2 file.c -S` ).

```
.file "file.c"
.text
.p2align 4,,15
.globl main
.type main, @function
main:
    pushl   %ebp
    movl   %esp, %ebp
    pushl   %esi
    pushl   %ebx
    andl   $-16, %esp
    movl   $999, %ecx
    xorl   %ebx, %ebx
.L23:
    xorl   %esi, %esi
    cmpl   %ebx, %esi
    jge    .L28
    .p2align 4,,15
.L27:
    incl   %esi
    movl   %esi, %eax
    cld
    cmpl   %ebx, %edx
    jl     .L27
    jg     .L6
    cmpl   %ecx, %esi
    jb     .L27
```

```
.L6:
    addl    $-1, %ecx
    adcl    $-1, %ebx
    movl    %ecx, %eax
    andl    %ebx, %eax
    incl    %eax
    jne     .L23
    leal   -8(%ebp), %esp
    popl    %ebx
    popl    %esi
    popl    %ebp
    ret
.L28:
    jg      .L6
    cmpl    %ecx, %esi
    jb      .L27
    jmp     .L6
    .size   main, .-main
    .section .note.GNU-stack,"",@progbits
    .ident  "GCC: (GNU) 3.3.4"
```

no trace of the ++ and – operands that were in the for loop.

#### 1.7.4. Cleaning up

The UML Makefile has no provisions to clean away the profiling/code-coverage data files so here is a shell command to clean things:

```
root@rtl17: # cd linux-2.4.26
root@rtl17: # find . -name "*.bbg" -exec rm {} \;
root@rtl17: # find . -name "*.bb" -exec rm {} \;
root@rtl17: # find . -name "*.da" -exec rm {} \;
root@rtl17: # make mrproper
```

After this your linux tree should be clean again.

## 2. Conclusion

User Mode Linux is a tool that allows you to inspect the boot-process in detail and allows to optimize parts based on code-coverage output, whereby one must be careful with the output generated by gcov as it does not mix well with optimization which is mandatory for kernel compilation (there are patches to allow unoptimized compilation - but that buys you little as the optimizations you would find likely are only valid for un-optimized code).

UML is a well maintained and active project on sourceforge and thus the risk of building on this technology for system optimization and problem understanding purposes is low.

The boot-process can be monitored from the entry into the kernel proper - what happens before the kernel proper (that is `_start_kernel`) can only be viewed in a limited manner.

Nevertheless the information gained from the code coverage output is valuable to not only locate bottle necks but also to allow stripping of code for a specific platform which can significantly reduce the kernel size and also improve performance of the boot-process.

### 3. List of Acronyms

CVS - Concurrent Version Control

UML - User Mode Linux

tty - TeleType (terminal)

pty - Pseudo Terminal

SysRq - System Request

mm - Memory Map

GNU - GNU Not UNIX (recursive acronym)

## **References**

- [1] - Embedded Linux Kickstart Session, <http://www.opentech.at/documents.html>, 2004.
- [2] - PresetLPJ: Loops-per-jiffies boot-time optimization patch, <http://tree.celinuxforum.org/pubwiki/moin.cgi/PresetLPJ>, 2004