Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Regression Test Framework and Kernel Execution Coverage

Hiro Yoshioka
Miracle Linux Corporation
hyoshiok@miraclelinux.com

Abstract

We have developed a Linux kernel regression test framework ("crackerjack") and a branch coverage test tool (hereinafter "btrax") to capture kernel regression. Crackerjack is a harness program and a set of test programs. It runs test programs, records the results, and then compares the expected results. Therefore, if a particular system call failed in a release and was then fixed in a later release, crackerjack records this as non-favorable because of incompatibility. The btrax is an integrated component of crackerjack and is a tool to assess test programs' effectiveness. It uses Intel processor's branch trace capability and records how much code was traced by a test program. Crackerjack is initially designed for Linux kernel system call testing, but care has been taken to allow future expansion to other types of software.

1 Introduction

1.1 Test Early, Test Often

The Linux kernel development process does not explicitly define an automated mechanism for maintenance of compatibility. Unintended introduction of incompatibilities as the result of a bugfix and/or an upgrade do not get detected in an automated way. The basis of testing in the Linux kernel development community is predicated on frequent releases, with feedback and review by a large number of contributors. Therefore, end-users and middleware developers can find incompatibility problems only after the release of a kernel.

Developers may introduce new functionality which, intentionally or unintentionally, introduced incompatibility but there are few cases which describe the incompatibility explicitly. For example, you may write in the change log, "Function XXX is added" but you may not write "There is an incompatibility YYY because of introducing function XXX."

The cost of validating such incompatibility for middleware developers is increasing, therefore we need some mechanism to find such incompatibility.

If you find the incompatibility early, then analysing the issue and fixing it is very easy.

However, if you find the issue very late, some applications may already make use of this incompatible behavior and therefore you can not change or fix the behavior, even the fix is trivial.

Therefore, finding incompatibility very early has practical benefit not only for Linux kernel developers but also for middleware developers.

1.2 Regression Testing

1.2.1 Necessities of Regression Testing

The word *regression* has a negative connotation; *degrade* also has a bad image. Incompatibility is not always a bad thing. We have to have some incompatibility to introduce new features, bug fixes, and performance improvements.

Although the word *regression* has negative connotation, we will use the word in this paper, because the term "regression test" is commonly used in the testing community.

Regression testing is a mechanism to find different (diff) behavior of implementation. Maintaining compatibility is very important. But it is not always possible, we have to have some tradeoff.

1.2.2 Automatic Test

Testing is a boring process but we'd like to change so it is a fun and easy process.

We need a systematic way to find a regression of the Linux Kernel and OSS.

Regression testing is an automatic process which runs a test and compares the expected results, then validates the compatibility of the software. Regression testing is a common practice in commercial software developments but not well done in the open source software.

As such, we have developed a compatibility testing tool for the Linux kernel interface, the goal of which is to avoid unintended introduction of incompatibilities, and to effectively reduce application development cost by detecting such incompatibilities upfront. This tool is different from standard certification tests such as Linux Standard Base, in that the tool can detect incompatibilities between a particular version of a kernel and its revised versions.

The test tool includes following features.

- Automatically assess kernel behaviors and storing the results.
- Detect differences between stored results and point out incompatibilities.
- Manage (register, modify, remove) test results and expected test results.

We will promote the development of this test tool by communicating with the Linux kernel community, the test community, the North-east Asia OSS Promotion Forum, etc. from an early stage, through the so-called bazaar model.

1.2.3 Expected Usage

Linux kernel developer

Bugfix of an existing kernel: Verify that the bugfix does not destroy previous features in an incompatible way. Add a test program for the bug in question and verify that the existing kernel includes the bug and the bugfix version does not.

Development of new features: Verify that the new feature in question does not destroy previous features in an incompatible way. If the feature extension was an incompatible extension, verify the extent of such incompatibility. Add test program for the new feature to maintain compatibility in the future.

Middleware developer

These developers should execute a regression test against a new version of kernel, and if incompatibilities are found, specifically understand the extent of the incompatiblity. It will be easy to find kernel regression, if middleware tests are prepared and run.

1.2.4 Non Goals

The following are non goals.

- Certification Tests e.g. POSIX, LSB
- Performance Tests, e.g. Benchmarks

We don't build certification tests nor performance tests. They are non goals.

1.3 Summary of theme

Develop a regression test framework, the regression tests, and the test coverage measurement tool, as shown in figure 1. Regression test framework is a framework that executes each of the test sets, and is a basis for the framework of this Linux kernel compatibility testing tool.

2 Regression Test Framework - crackerjack

Crackerjack is a regression test framework which provides 1) execution of test sets, 2) reporting based on results of test set execution, and 3) management of test programs, expected results, test sets, and test results.

It is implemented using Ruby on Rails. Ruby makes it easy to modify, ensuring a low maintenance cost.

Regression testing is defined as testing two builds of software, with emphasis on detection of regression (incompatibility of functionality).

In Figure 1, the result (Rn) of a test executed on a certain version of software (Vn) is compared against an expected result (En). The initial expectation (E1) is typically identical to R1.

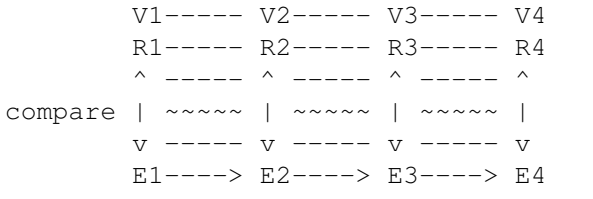

Figure 1: Comparison between test results (Rn) and expected results (En)

To detect a regression, the results from two separate builds have to be compared. If a result from a system call can be statically determined to be correct, such a comparison would return a valid result (OK) and or not (NG). We can add a comparison routine to determine the results.

```
pid = getpid ()
if (pid > 5000 && pid < 66536) {
       printf ("OK\n");
} else {
       printf ("NG\n");
}
return EXIT_SUCCESS
```

Figure 2: Example – Exact Match

```
pid = get_pid
printf ("%d\n", pid)
return EXIT_SUCCESS
```

Figure 3: Example - Range

In the first example above (see Figure 2), the test program statically determines whether the function call returned a valid result ('OK') or not ('NG'). In the second example (see Figure 3), the test program only outputs the function call result, for later comparison.

The comparison method may vary between test programs. Therefore, it is the test program developer's responsibility to define a comparison method for that test program.

The objective of the crackerjack system is to perform regression tests. crackerjack consists of (1) framework, (2) a collection of test programs, and (3) branch tracer (btrax). It is outside the scope of crackerjack project, to measure performance or certify compatibility.

2.1 Software Functionality

The framework provides the following functions.

- Run both GUI (Graphical User Interface) mode and CUI (Character User Interface) mode.
- Execute test programs.
- Compare test execution results and expected results.
- Report based on results of test set execution.
- Manage test programs, expected results, test sets, test target, and test results.
- Define compare programs.

The expected end-users of crackerjack are kernel developers, middleware developers, and test program developers.

For more information, please refer to the Appendix.

3 Branch Tracer for Linux – btrax

3.1 btrax

We have integrated a branch tracer for Linux (known as btrax hereinafter) in the crackerjack to find the execution coverage of regression tests.

This program (btrax) traces the branch executions of the target program, analyzes the trace log file, and displays coverage information and execution path. It's possible to trace it for an application program, a library, a kernel module, and the kernel.

The btrax consists of the following commands.

1. Collect the branch trace log via bt_collect_log,
2. Report the coverage information via bt_coverage,

3. Report the execution path via `bt_execpath`, and
4. Split the branch trace log by each process via `bt_split`.

The btrax collects a branch trace information using Intel's last branch record capability. `bt_collect_log` stores the branch trace log, `bt_coverage` and `bt_execpath` report the branch coverage and the execution path information respectively.

3.2 Coverage

The coverage information gathered includes function, branch, and state coverage. Usually, it would be displayed in address value order.

The branch tracer (`bt_collect_log`) collects the last branch information using Intel's last branch record capability.

`bt_coverage` analyzes the ELF files such as kernel, module, etc. and gets the branch and function-call information. Then, it analyzes the traced log(s) with this information, and displays the coverage information.

Function coverage displays how many functions were executed among total functions. Displayed functions are almost compatible with objdump's output. Note that it would not show the "function call coverage." (See Figure 12.)

Branch coverage displays how many conditional branches (i.e. both branch and fall-through) were executed among total ones. (See Figure 13.)

State (basic block) coverage shows how many states (basic block) were executed among total states. State means straight-line piece of code without any jumps or jump targets in the middle (a.k.a. basic block). In the previous "switch case" example, if the codes of the "case 0" and "case 2" were both executed three times, then each state and coverage would be as follows. (See Figure 15.)

A chain of function calls is displayed, for example, function FA calls function FB, and function FB calls function FC, and so on. If you would like to limit the coverage target to the functions which are included in function call tree, -I option can be used. In this case, the function call tree would be displayed with other coverage information (such as function/branch/state coverage). Example of the function call tree is shown below. (See Figure 16.)

4 Linux Kernel System Call Test Coverage

We made test programs which use Linux system calls and used the crackerjack with the btrax. We measured the function coverage, the branch coverage, and the state coverage of the each test program execution (Table 1).

We selected 50 system call test programs of LTP and measured the execution coverages of them as our benchmark. (Table 2)

We know the LTP is comprehensive test suite but the execution coverage is not large enough.

The average function coverage, branch coverage, and state coverage are 41.39%, 23.1%, and 30.94% respectively. (Note: 38%, 10%, and 12% of the test programs exceeded 50% of function, branch, and state coverage.)

It is very difficult to increase the execution coverage. The following are the reasons.

- exception/error condition: Making an exception program is not easy. Making an Error condition is not easy. Sometimes it is not feasible.
- asynchronous processing: For example, making spin/lock, spin/wait condition is not easy.
- excluding functions from coverage measures: For example, a common routine like printk has a lot of execution path. If a given system call uses printk, it contains a lot of execution path which are not covered the test programs.

We need to exclude some functions from the coverage measurement but it is hard to find such functions. Pruning unnecessary code path is difficult. If a given system call has more than three digits of function calls, it implies insufficient pruning.

It is almost impossible to run codepaths for low memory situations. It is even questionable that such test could even run.

System call	Func coverage %	branch coverage %	state coverage %
13 time	100.0	70.0	88.24
20 getpid	0.00	100.00	0.00
20 getpid (wb)	100.00	100.00	100.00
25 stime	69.23	41.67	50.68
27 alarm	14.19	2.50	5.65
30 utim (wb)	30.77	10.79	16.73
30 utime	23.56	8.81	13.38
42 pipe	67.74	31.91	49.79
42 pipe (wb)	82.26	47.04	65.60
45 brk	23.88	8.96	12.74
45 brk (wb)	23.13	9.12	12.53
66 setsid	29.41	12.08	18.10
78 gettimeofday	87.50	52.00	70.31
79 settimeofday	77.78	45.19	57.04
82 old_select	3.96	1.78	2.67
87 swapon	19.12	9.05	12.21
90 old_mmap	7.14	2.91	3.54
91 munmap	13.74	5.78	7.66
92 truncate	4.48	1.06	1.77
93 ftruncate	0.88	0.11	0.20
101 ioperm	19.78	8.84	10.91
103 syslog	5.45	1.67	2.44
104 setitimer	3.79	1.44	2.02
105 getitimer	1.35	0.45	0.61
110 iopl	100.00	50.00	81.82
115 swapoff	12.28	4.12	6.18
116 sysinfo	15.11	3.31	5.79
120 clone	19.30	6.79	9.92
121 setdomainname	7.46	1.88	3.15
124 adjtimex	5.67	1.05	1.79
125 mprotect	7.82	3.90	5.28
142 select	3.96	1.78	2.67
144 msync	1.52	0.42	0.62
147 getsid	72.73	46.67	61.70
150 mlock	2.08	0.37	0.72
151 munlock	1.66	0.31	0.56
152 mlockall	24.12	9.29	12.62
153 munlockall	1.66	0.21	0.44
162 nanosleep	31.76	13.41	18.67
163 mremap	13.83	6.21	8.71
168 poll	2.30	0.95	1.47
187 sendfile	1.01	0.11	0.19
193 truncate64	4.48	1.06	1.77
194 ftruncate64	0.88	0.11	0.2
203 setreuid	1.27	0.23	0.38
204 setregid	100.00	20.00	46.43
205 getgroups	1.54	0.25	0.41
206 setgroups	1.26	0.10	0.23
208 setresuid	1.27	0.26	0.41
209 getresuid	66.67	33.33	60.00
210 setresgid	100.00	18.42	45.16
211 getresgid	66.67	33.33	60.00
218 mincore	0.62	0.06	0.10
219 madvise	1.60	0.50	0.73

Table 1: Summary of coverage of tests %

System call	Func coverage %	branch coverage %	state coverage %
13 time	75.00	60.00	82.35
20 getpid	100.00	100.00	100.00
25 stime	61.54	35.42	47.95
27 alarm	53.45	15.58	25.03
30 utime	28.85	9.83	15.27
42 pipe	77.42	44.08	61.32
45 brk	13.43	6.76	8.88
66 setsid	23.53	7.92	12.76
78 gettimeofday	87.50	44.00	67.19
79 settimeofday	66.67	32.69	45.77
82 old_select	No exist	No exist	No exist
87 swapon	29.08	11.40	16.80
90 old_mmap	17.41	11.25	13.11
91 munmap	21.15	9.45	13.24
92 truncate	19.77	6.21	9.73
93 ftruncate	15.48	4.38	7.22
101 ioperm	4.27	1.56	2.02
103 syslog	26.32	13.64	14.13
104 setitimer	48.44	11.01	21.05
105 getitimer	67.86	24.65	40.64
110 iopl	100.00	100.00	84.62
115 swapoff	20.61	5.90	9.75
116 sysinfo	86.96	40.35	59.06
120 clone	27.52	11.48	16.80
121 setdomainname	100.00	50.00	71.43
124 adjtimex	57.14	16.18	26.49
125 mprotect	11.53	5.70	7.57
142 select	10.88	4.72	6.61
144 msync	5.50	2.45	3.25
147 getsid	72.73	26.67	48.94
150 mlock	20.60	8.39	12.23
151 munlock	5.32	3.12	3.76
152 mlockall	17.82	7.08	9.88
153 munlockall	2.31	0.39	0.69
162 nanosleep	58.82	15.16	25.32
163 mremap	17.57	7.68	11.04
168 poll	6.12	1.86	3.13
187 sendfile	12.77	3.72	6.05
193 truncate64	No exist	No exist	No exist
194 ftruncate64	No exist	No exist	No exist
203 setreuid	11.76	4.08	5.69
204 setregid	100.00	96.43	96.43
205 getgroups	83.33	40.00	60.78
206 setgroups	9.43	2.95	4.55
208 setresuid	10.92	4.14	5.68
209 getresuid	66.67	50.00	80.00
210 setresgid	100.00	68.42	96.88
211 getresgid	66.67	50.00	80.00
218 mincore	13.57	4.85	7.04
219 madvise	11.43	4.03	6.15

Table 2: Summary of coverage of tests (LTP)%

In spin lock situations, program code normally runs through locked side path. It is hard to prepare conditions for competing locks.

Making race condition is also hard to test.

There is a common function called from gettimeofday / settimeofday. In this common function, paths are uniquely determined by get / set. So other execution paths have never been executed.

5 Discussion

5.1 Kernel Tests – Writing a Good Test is Very Hard

5.1.1 Test Coverage

Writing good kernel tests is very difficult. Our data shows that the execution coverage is low. As discussed above, it is very hard to increase the execution coverage.

5.1.2 man Pages are Not Enough

Some man pages do not have enough detail information. For example, utime does not have a description of error return values.

If we don't have clear definition, we can not determine if the test result is OK or NG.

5.1.3 Behavior of 2.4 vs. 2.6

2.6 introduced more strict parameter checking. We discovered a condition where 2.4 did not report errors but 2.6 did. (We discovered implementation-dependent incompatibility.)

For example, settimeofday second had loose error checking, but 2.6 introduced more strict error checking.

We think fixing a bug is important but changing behavior may introduce other incompatibility. So we need to know as early as possible to assess it.

5.2 Finding Incompatibility

For portability, middleware developers should preferably not be using implementation-dependent and undefined functions. However there are cases when such functions are used without intention.

There is a large hidden cost to avoid unintended introduction of incompatibility. It is said that the most costly activity in development of commercial software is the maintenance of backward compatibility.

Crackerjack detects not only unintended incompatibilities, but also intended incompatibilities. Both are important for notification to middleware developers. It is important to notify the changes in behavior, in a timely manner. We can detect behavior diffs across versions.

For example, the memory range acquired from `brk()` is static but the memory range is randomized by turning on the exec shield. Crackerjack detects such a specification change.

We can write user-defined compare program not only simple OK/NG but allows some statistical allowance.

6 Related Work

6.1 LTP

LTP (Linux Test Project) is a set of Linux test programs which includes the Linux kernel test, stress tests, benchmark tests, and so on. LTP is a comprehensive test suite but it is not intended to be a regression test suite.

The Linux kernel test validates the POSIX definition of the kernel therefore it does not cover features like implementation defined, implementation dependent, and or undefined functionality.

On the other hand, crackerjack is a test harness and tries to capture all execution behavior and difference between each version. Such behavior includes not only standard features but also implementation defined, implementation dependent, and undefined by the standard.

Crackerjack finds the diff of implementation behavior.

We can integrate crackerjack with the LTP. For example, adding some regression tests to LTP and invoke them from crackerjack.

6.2 gcov/lcob

The gcov/lcob is a coverage tool. There is a kernel patch to measure the test coverage of the Linux kernel. The gcov uses gcc to get the coverage data but you need to patch the kernel.

The btrax uses Intel's last branch record capability and you don't need any patch nor rebuild kernel. So you can measure your standard kernel without rebuilding kernel.

6.3 autotest

The autotest is a harness program of invoking several tests program including LTP, benchmark programs and so on.

We can plug crackerjack into autotest.

7 Future Work

7.1 Kernel Development Process

We believe finding incompatibility in an early stage is very important and all of us can get much benefit from it.

It is good thing to run regression tests on every kernel release. Adding this practice into the kernel development process is our big challenge. We need to show our benefit to the kernel community and convince them to use it.

7.2 Expanding the Area

The concept of regression testing is very simple, just find the diff of the implementation between releases. Crackerjack can be used on other types of software interfaces, for example, the `/proc` file system, glibc, and so on.

The system calls are very stable and we don't see much incompatibility in them. However, /proc file system is much more flexible, so we may easily find some incompatibility.

7.3 Crackerjack

A richer set of default compare programs for crackerjack is needed. Today we have to add compare programs if the default does not match your needs.

We need methodology to relieve the test program developers. It might be a test pattern, convention or environment.

Crackerjack has to record system information, environment, and reproducible information.

Current development focuses on regression test framework and test coverage tool. Extending the test to all of Linux kernel functions, and sustained execution of regression tests, would wait until the next project.

7.4 Community Activity

Our development is supported by a working group of Japan OSS promotion forum which consists of private sector and public sector. There is a collaboration with China and Korea groups too.

We'd like to expand this activity with the Linux and test community.

8 Acknowledgements

This project is supported by the Information Technology Promotion Agency (IPA), Japan.

We would like to thank my colleagues and their contributions. Satoshi Fujiwara (Hitachi) implements the btrax. Takahiro Yasui (Hitachi) and Masato Taruishi (Red Hat KK) wrote kernel tests and measured the Linux kernel execution and the branch coverage. Kazuo Yagi (Miracle Linux) implements the crackerjack.

The project team: Hitachi team; Satoshi Fujiwara (btrax), Takahiro Yasui (kernel test programs), Hisashi Hashimoto, Yumiko Sugita, and Tomomi Suzuki.

Miracle Linux team; Kazuo Yagi (crackerjack and kernel test programs), Ryo Yanagiya, and Hiro Yoshioka.

Red Hat KK team; Masato Taruishi (kernel test programs) and Toshiyuki Takamiya.

References

[1] Linux Test Project
http://ltp.sourceforge.net/

[2] autotest
http://test.kernel.org/autotest/

[3] ABAT http://test.kernel.org

[4] Test Tools Wiki(OSS Testing Summit)
http://developer.osdl.org/dev/test_tools/index.php/Main_Page

[Eric Raymond] The Cathedral and the Bazaar
http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar

[5] Ruby on Rails
http://www.rubyonrails.org/

Appendix:

A crackerjack – Getting Started

A.1 Get the Code and make

The latest source code is always available at https://crackerjack.svn.sourceforge.net/svnroot/crackerjack/

You can get it by Subversion. For example, see the Figure 4 "Getting Source code and make."

```
$ svn co \
  https://crackerjack.svn.sourceforge.net/svnroot/crackerjack/
A    crackerjack/trunk
A    crackerjack/trunk/crackerjack
(...snip...)
Checked out revision 477.

$ make
(...snip...)
```

Figure 4: Getting Source Code and make

If a test program has compiler errors, then try `make -k`. Some system calls are not available in old Linux kernels, for example, 2.6.9 does not have mkdirat, mknodat etc.

```
$ su -
Password:
# cd /usr/src/crackerjack/trunk/crackerjack/
# ./crackerjack
crackerjack>h
 number push test program to stack
 d      Delete(pop) stack
 e      register test program result
        on stack as Expected result
 h      help
 l      List the test programs
 p      Print stack
 x      eXecute test program on stack
```

Figure 5: Invoke crackerjack

```
crackerjack>l
0000) access
0001) adjtimex
0002) adjtimex/whitebox
0003) alarm
...
```

Figure 6: List the Tests Command

A.2 Invoke crackerjack

Become the root user and run crackerjack.

Read help with 'h' command. (See Figure 5)

List the test programs with 'l' command. (See Figure 6)

Push the test programs to the stack with "number" which corresponds to the test program you want. You can look the contents of stack with `p` command, and pop the test programs from stack with `d` command.

Execute the test program on stack with the 'x' command. (See Figure 7.)

Quit the crackerjack with the 'q' command. (See Figure 8.)

A.3 Run in Non-interactive way

You can execute test programs from a file which indicates the order of tests (order file). (See Figure 9.)

You can create an order-file using the 'l' command similar to the following then execute the test. (See Figure 10.)

```
crackerjack>1
1
crackerjack>2
1 2
crackerjack>3
1 2 3
crackerjack>4
1 2 3 4
crackerjack>5
1 2 3 4 5
crackerjack>x
 Action   SystemCallName      Id
 X        adjtimex            20070412133111
 X        adjtimex/whitebox   20070412133111
 X        alarm               20070412133111
 X        alarm/whitebox      20070412133111
 X        brk/basic           20070412133111
```

Figure 7: Execute Tests

```
crackerjack>q
#
```

Figure 8: Quit with the 'q' Command

A.4 GUI mode

Invoke the GUI server similar to the following then use a web browser to access localhost:3000. (See Figure 11.)

You may need to install the Ruby on Rails.

B btrax – Getting Started

B.1 Basic Concept

This program (btrax) traces the branch executions of the target program, analyzes the trace log file, and displays coverage information and execution path. It's possible to trace it for an application program, a library, a kernel module, and the kernel.

The btrax consists of the following commands.

```
# ./crackerjack -h
USGE: crackerjack [option] [FILE]
-l        list the test programs
-x FILE   execute the test program on reading order from file
-e FILE   register the current result as the expected result
-c result_kerv,result_id,expected_kerv,expected_id
          compare the current result to the expected result
-b        execute with btrax, avaiable with -e option
-h        show this help
-v        show version
```

Figure 9: crackerjack Non-interactive Mode

```
# ./crackerjack -l > m.order
# ./crackrjack -x m.order
```

Figure 10: crackerjack Non-interactive with 'l' and 'x' Commands

```
# ./crackerjack-gui-server
```

Figure 11: crackerjack GUI Mode

1. Collect the branch trace log via `bt_collect_log`,
2. Report the coverage information via `bt_coverage`,
3. Report the execution path via `bt_execpath`,
4. Split the branch trace log by each process via `bt_split`.

The btrax collects a branch trace information using Intel's last branch record capability. `bt_collect_log` stores the branch trace log, `bt_coverage` and `bt_execpath` report the branch coverage and the execution path information respectively.

B.2 Coverage

The coverage information includes function coverage, branch coverage and state coverage. Usually, it would be displayed in address value order.

B.2.1 bt_coverage

`bt_coverage` analyzes the ELF files such as kernel, module, etc. and get the branch and function-call information. Then, it analyzes the traced log(s) with this information, and displays the coverage information.

`bt_coverage` tries to show the source information such as source file name and line number, but if there is no debug information in the ELF file, it shows only the address value.

You can check whether the source code was executed or not by using html output (for this, it needs debug information in the ELF file and the source code files). Note that inline functions and macros would not be colorized

correctly in the html files. It is because they were expanded to the other functions and could not be found in the ELF file.

You can also compare the two log files' kernel coverage by generating html files. Even if the log files were generated on the different kernels, `bt_coverage` can still compare them.

B.2.2 Function Coverage

It displays how many functions were executed among total functions. Displayed functions are almost compatible with objdump's output. Note that it would not show the "function call coverage." For example, refer to the below chart ('//' means comment). (See Figure 12.)

```
(example source code)
funcA(); // executed once
funcB(); // executed once
...
funcB();
...
funcB();
funcC();

(then, coverage output would be...)
 ---- function coverage (2/3=66.67%) ----
 (OK) <funcA> (1) // funcA executed once
 (OK) <funcB> (1) // funcB executed once
 (NT) <funcC> (0) // funcC not executed
```

Figure 12: Function Coverage

B.2.3 Branch Coverage

It displays how many conditional branches (i.e. both branch and fall-through) were executed among total ones. For example, branch coverage counting for each coverage case is as follows. (See Figure 13.)

There is a case that the branch address would be determined by indirect addressing such as "switch case" code. In this case, it is impossible to know the branch addresses and number of these by analyzing the ELF file. (See Figure 14.)

We call this kind of branch as "unknown branch." Unknown branches are counted whether the branch was executed or not, and are counted separate from normal branches. In this example, if all of the switch case codes were not executed, branch coverage would be as follows.

```
(example source code)
  Address    C              Assembler
        1:   if (xxx)              jxx LABEL
        2:      aaa;               aaa
        3:   bbb;          LABEL:  bbb
(coverage counting for each case)
```

branch (1->3)	fall-through (1->2)	coverage counting	symbols
not executed	not executed	0/2=0.00%	NT
not executed	executed	1/2=50.00%	HT
executed	not executed	1/2=50.00%	HT
executed	executed	2/2=100.00%	OK

(if branch was executed 3 times, and fall-through was executed 1 time, then coverage output would be...)

```
---- branch coverage (OK:1,HT:0,NT:0/2=100.00% UK:0/0=100.00%)

(OK) 1 [3/1] 3:2 // 1->3 3 times, and 1->2 1 time executed
```

Figure 13: Branch Coverage

```
(example source code)
     Address  C                    Assembler
     -------  -------------------  -------------------
           1: switch (xxx) {               jmp *(%eax)
           2: case 0: aaa;                 aaa
           3:         break;               jmp LABEL
           4: case 1: bbb;                 bbb
           5:         break;               jmp LABEL
           6: case 2: ccc;                 ccc
           7:         break;       LABEL:
           8: }
```

Figure 14: Branch Coverage

```
---- branch coverage (OK:0,HT:0,NT:0/0=100.00% UK:0/1=0.00%)
(UN) 1 [0/x] ----------:xxxxxxxxxxx     // 1 jumps nowhere
```

Or, if the codes of the "case 0" and "case 2" were both executed 3 times, then branch coverage would be as follows.

```
---- branch coverage (OK:0,HT:0,NT:0/0=100.00% UK:1/1=100.00%)
(UT) 1 [3/x] 2:xxxxxxxxxxx         // 1->2 executed 3 times
(UT) 1 [3/x] 6:xxxxxxxxxxx         // 1->6 executed 3 times
```

In the coverage output, you cannot check that how many branches were executed for the unknown branches. But each case block's execution information would be showed in state coverage. Although you can check that in html output.

B.2.4 State (Basic Block) Coverage

It shows how many states (basic block) were executed among total states. State means straight-line piece of code without any jumps or jump targets in the middle *a.k.a. basicblock*. In the previous 'switch case' example, if the codes of the "case 0" and "case 2" were both executed three times, then each state and coverage would be as follows. (See Figure 15.)

```
(example source code)
Address  C                  Assembler
-------  -----------------  --------------------
    1: switch (xxx) {               jmp *(%eax)
       -  -  -  -  -  -  -  -  -  -  -  -  -// state border
    2: case 0: aaa;                 aaa
    3:         break;               jmp LABEL
       -  -  -  -  -  -  -  -  -  -  -  -  -  -
    4: case 1: bbb;                 bbb
    5:         break;               jmp LABEL
       -  -  -  -  -  -  -  -  -  -  -  -  -  -
    6: case 2: ccc;                 ccc
       -  -  -  -  -  -  -  -  -  -  -  -  -  -
    7:         break;       LABEL:
    8: }

(coverage output would be...)
------ state coverage (4/5=80.00%) ------
(OK) 1    // state from address 1 was executed
(OK) 2
(NT) 4    // state from address 4 was not executed
(OK) 6
(OK) 7
```

Figure 15: State (basic block) Coverage

B.2.5 Function Call Tree

It means the chain of function call, for example, function FA calls function FB, and function FB calls function FC, and so on. If you would like to limit the coverage target to the functions which are included in function call tree, -I option can be used. In this case, function call tree would be displayed with other coverage information (such as function/branch/state coverage). Example of the function call tree is shown below. (See Figure 16.)

```
==== includes: sys_open ======
==== excludes: schedule,printk,dump_stack,panic,show_mem ====
==== function tree (59/498=11.85%) ======
(OK) <sys_open>:fs/open.c,1101     (3, F:498)
(OK) +-<do_sys_open>:fs/open.c,1079     (3, F:497)
(OK) +-+-<do_filp_open>:fs/open.c,874   (3, F:196)
(OK) +-+-+-<nameidata_to_filp>:fs/open.c,942 (3, F:4)
(OK) +-+-+-+-<__dentry_open>:fs/open.c,799   (3, F:3)
(NT) +-+-+-+-+-<wake_up_process>:kernel/sched.c,1521 (0, F:1)
//... snip ...
(UT) <generic_file_open>:fs/open.c,1216 (2)
(UT) <dummy_inode_follow_link>:security/dummy.c,324   (1)
//... snip ...
```

Figure 16: Function Call Tree

In this example, we can see that the `sys_open` was executed (it is seen as 'OK') 3 times and it contains 498 functions including itself (it is seen as `(3, F:498)`). If the function was already displayed, then it is displayed with `(--)` symbol instead of `(OK)` or `(NT)`.

A function call tree may contain unnecessary functions for the user. In this case, it could be excluded by using -E option. You can also check that if it would be excluded, then how many functions would be decreased by using "F:N" information (we call it "function excluding guidance") in the function call tree.

Function excluding guidance is also displayed in the "function coverage," and each function's information is sorted by this value.

There is the function call whose address would be determined by indirect addressing such as "function call by using function pointer." In this case, it is impossible to know the function address by analyzing the ELF file. So, this kind of functions would not be displayed in the function tree. Example of this kind of function call is shown below.

```
(example source code)
Address  C                                   Assembler
-------  ----------------------------------  ----------------
      1: int call_function(void (*func),,, )
      2: {
      3:         func();                     call *0x84(%edx)
      4: }
```

If you would like to include this kind of function to the coverage target, you must check the source code if there is no information. To improve this situation, (`bt_coverage`) also analyzes the trace log(s) and shows the functions which were executed by indirect addressing. This kind of function is displayed with `(UT)` mark (we call it "function including guidance"). Read the man page for more information.

B.3 Get the Code, Build and Install

The project home page is the following, Download the tarball from the project page.

`http://sourceforge.net/projects/btrax/`

Read the README and follow the instruction. (See Figure 17.)

B.4 Checking System Call Coverage

In order to determine the correctness of our regression test, we measure the execution branch coverage of tests. The following subsections show the steps to take to measure the branch coverage using the btrax.

B.4.1 Create a program calling system call

Create a regression test which uses a system call(s). (You can use LTP as an example.)

```
Build and Install.

 $ cd $(WHERE_YOUR_WORK_DIRECTORY)
 $ tar jxvf btrax-XXX.tar.bz2
 $ cd btrax-XXX

Install the command.

 $ make
 $ su (input super-user password here)
 # make install

Create relayfs mount point.

 # mkdir /mnt/relay
```

Figure 17: Build and Install

B.4.2 Start Branch Trace

Start branch trace. (See Figure 18.) Note that `syscall_name` must be defined in the kernel's `sys_call_table`.

```
# bt_collect_log --syscall \
  $(syscall_name) -d $(ODIR)\
  -c $(program)
```

Figure 18: Start branch trace

B.4.3 Checking Coverage

To check the system call coverage summary, do the next command. (See Figure 19.)

```
# cd $(ODIR)
# bt_coverage --ker -f \
  `echo $(ls cpu*)|sed 's/\s\+/,/g'` \
  -I $(syscall_name) -s
```

Figure 19: Checking coverage

To check the system call coverage detail, do the next command. (See Figure 20.)

If you want to exclude the function(s) from coverage result, use the -E option. (See Figure 21.) To check whether the code in that system call was executed or not, do the next. (See Figure 22.) Note that html files are using the Javascript. If you have some trouble browsing, check the Javascript setting.

```
# bt_coverage --ker -f \
  `echo $(ls cpu*)|sed 's/\s\+/,/g'` \
  -I $(syscall_name)
```

Figure 20: Checking Call coverage

```
# bt_coverage --ker -f \
  `echo $(ls cpu*)|sed 's/\s\+/,/g'` \
  -I $(syscall_name) \
  -E schedule,dump_stack,printk
```

Figure 21: Excluding functions

```
# bt_coverage --ker -f \
  `echo $(ls cpu*)|sed 's/\s\+/,/g'` \
  -I $(syscall_name) \
  -E schedule,dump_stack,printk \
  -o $(HTML_OUT_DIR) -S $(KERN_SRC_DIR)
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 22: Excluding functions

B.4.4 Compare System Call Coverage

```
# bt_coverage --ker \
  -I $(syscall_name)\
  -E schedule,dump_stack,printk \
  -o $(HTML_OUT_DIR)\
  -S $(KERN_SRC_DIR)\
  -f `echo $(log1/cpu*)|sed 's/\s\+/,/g'` \
  --f2 `echo $(log2/cpu*)|sed 's/\s\+/,/g'`
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 23: Comparing the system call coverage

To compare same kernel's system call coverage, do the next command line which uses the -S, -f and –f2 options. In this example, each log directory is log1 and log2. (See Figure 23.)

```
# bt_coverage --ker\
 -I $(syscall_name)\
 -E schedule,dump_stack,printk \
 -o $(HTML_OUT_DIR)\
 -u uname_r1 --u2 uname_r2\
 -S src1 --S2 src2 \
 -f `echo $(log1/cpu*)|sed 's/\s\+/,/g'`\
 --f2 `echo $(log2/cpu*)|sed 's/\s\+/,/g'`
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 24: Comparing the system call coverage

If you had traced different kernel's system calls, you can also compare these logs. To compare the different kernel's system call coverage, use the next command line which also employs the `-u` option. In this example, each log directory is `log1` and `log2`, each kernel version is `uname_r1` and `uname_r2`, and each kernel source directory is `src1` and `src2`. (See Figure 24.)