Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Keeping the Linux Kernel Honest

Testing Kernel.org kernels

Kamalesh Babulal
IBM
kamalesh@linux.vnet.ibm.com

Balbir Singh
IBM
balbir@linux.vnet.ibm.com

Abstract

The Linux™ Kernel release cycle has been short with various intermediate releases and with the lack of a separate kernel development tree. There have been many challenges with this rapid development such as early bug reporting, regression tracking, functional/performance testing, and test coverage by different individuals and projects. Many kernel developers/testers have been working to keep the quality of the kernel high, by testing as many possible subsystems as they can.

In this paper, we present our kernel testing methodology, infrastructure, and results used for the v2.6 kernels. We summarize the bug reporting statistics based on the different kernel subsystems, trends, and observations. We will also present code coverage analysis by subsystem for different test suites.

1 Introduction

The Linux kernel has been growing with every release as a result of the sheer number of new features being merged. These changes are being released at a very high rate, with each release containing a very large number of changes and new features. The time latency at which these changes are made is very short between every release cycle. All of this is occurring across many disparate hardware architectures.

This presents unique problems for a tester who must take all of the following into account:

- Lines of code added,
- Time interval between each release,
- Number of intermediate releases,

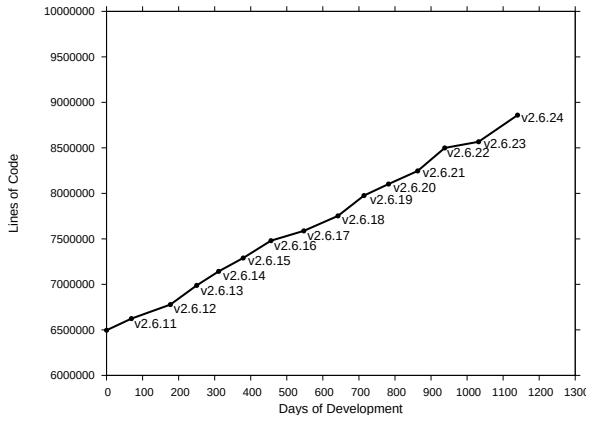

Figure 1: Size of the kernel with every release

- Testing on wide range of architectures and platforms,
- Regressions carried from previous release,
- Different development trees, and
- Various configurations and boot options to be tested.

In Section 2 we briefly explain the challenges. Section 3 outlines our methodology to meet the challenges. Section 4 presents the results of code coverage analysis, including the fault injection coverage results for the standard test cases and we discuss our future plans in Section 5.

2 Challenges

2.1 Different Trees

In the past, Linux kernel development had separate development and stable trees. The stable trees had an even number and the development trees had an odd number

as their second release number. As an example 2.5 was development release, while 2.4 was stable release. With the 2.6 kernel development, there are no separate *development* and *stable* trees. All the development is based upon the current stable tree and once the current development tree is marked as stable, it is released as the next mainline release.

There are intermediate development releases between major releases of a 2.6 kernels, each having its own significance. Figure 2 shows the different development trees and the flow of patches from development trees to the mainline kernel.

The `-stable` tree contains critical fixes for security problems or significant regressions identified for the mainline tree it is based upon. Once the mainline tree is released, the developers start contributing to the new features to be included in the next release. The new features are accepted for the inclusion during the first two weeks of development following the mainline release. After two weeks the merge window closes and the kernel is in feature freeze; no further features will be accepted into this release. This is released as `-rc1`. After `-rc1` the features are tested well and stabilized. During this period roughly weekly release candidates, `-rc`, are produced, as well as intermediate snapshots of Linus's git tree.

The `-rc` releases are a set of patches for the bug fixes and other important security fixes, based on the previous `-rc` release. For example, 2.6.25-rc3 will have the fixes for the bugs identified in 2.6.25-rc2.

In addition to the mainline releases, we have a number of testing trees for less stable features, as well as subsystem specific trees. The `-mm` tree has experimental patches and critical fixes that are planned to be pushed to the mainline kernel. For a new feature it is recommended that it be tested in `-mm`, since that tree undergoes rigorous testing, which in-turn helps in stabilizing the feature. `-mm` is rebased often to development releases to test the patch set against the development tree.

The `-next` release was introduced with the 2.6.24 development series. The `-next` tree has the patch changesets from different maintainers, intended to be merged into the next release. Changesets are rebased to the current development tree, which helps to resolve the merge conflicts and bugs before they get introduced in the next release. Resolving the conflicts and bugs on a regular basis would allow the development and stable releases to be based on the `-next` tree in the future (refer to [5] for more information on kernel trees).

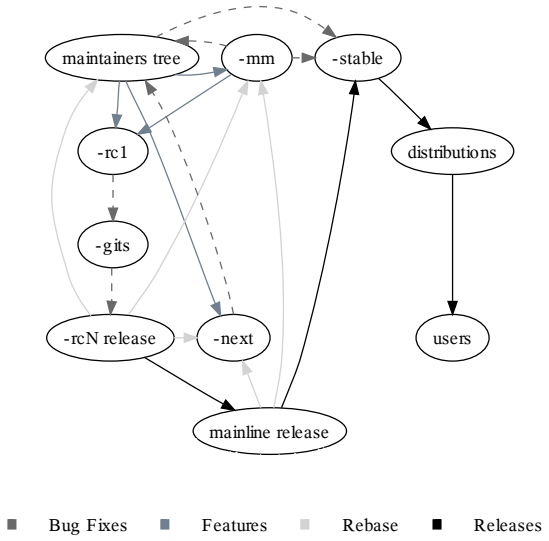

Figure 2: Linux Kernel Development Cycle

2.2 Release Frequency

Testing is essential to deliver high quality software and testing every phase of development is important if we are to catch bugs early. This is especially true for projects like Linux where the source is growing by 2.32% [1] with every release, as represented in Figure 1. These numbers are pretty high for any project. Starting from 2.6 kernel development series, there is no separate stable and development tree, which means that more code is being added and it needs to be tested thoroughly.

Bugs caught and fixed early in the cycle helps to maintain the kernel quality by:

- Providing a pleasant user experience,
- Avoiding building on top of buggy code, and
- Easier debugging, since the code is fresh in the authors mind.

The stable kernel releases are made approximately every 2-3 months, during which time an average of 3.18

v2.6 kernels						
versions	.20	.21	.22	.23	.24	Total
stable	22	8	20	18	5	73
stable-git	16	18	18	26	22	100
stable-mm	2	2	0	1	1	4
stable-mm +hotfixe(s)	0	0	0	3	6	9
rc	7	7	11	8	8	41
rc-git	59	52	57	70	47	285
rc-mm	12	6	10	7	4	41
rc-mm +hotfixe(s)	0	10	18	14	3	45
next	0	0	0	0	35	35
Total	118	103	134	147	131	633

Table 1: Summary of releases between kernel versions

changes are being accepted every hour. This has been the trend for the past $1\frac{1}{2}$ years.

Table 1 shows the data of various intermediate releases made for past $1\frac{1}{2}$ years. These incremental changes to the kernel source ensure that any code changes made to the kernel are well tested before they accepted into mainline.

With the average of 126 kernels (refer to Table 1) being released between two sequential major releases, we have at least one kernel per day being released. Developers end up limiting their testing to unit testing for the changes they make and start concentrating on new development. It is not practical for most of the developers to test their changes across all architectures supported by Linux. A significant amount of testing is being done by others from the community, which includes testers, developers, and vendors (including distribution companies).

2.3 Test Case Availability

There is a significant testing effort in the community by various individual testers, developers, distribution companies, and Independent Software Vendors. They contribute to the effort with combinations of testing methods such as:

- Compile and Boot test (including the cross compiler tests),
- Regression Testing,
- Stress Testing,
- Performance Testing, and
- Functional Testing.

These efforts are not being captured completely because most of these efforts are not visible. We will not be able to account for any testing done unless it is being published or shared. Many features are accepted into the mainline after stringent code reviews and ample testing in the development releases. But not many developers provide the test cases or guidelines to test their functionality even though it is in the interest of developers to keep the quality of their code high.

Over the years there have been many test projects and suites with a primary focus to improve the quality of Linux. They are constantly undergoing lots of changes between every release. They have been adding new test cases to test the new features getting merged into the kernel, *but the updating is not fast enough to catch those early bugs and for some features we do not have test cases available*.

Any person who is interested in testing the kernel has no single, nor even a small number, of test projects which can cover most of the kernel subsystems. We do have a large number of test projects, but each is independent requiring installation and configuration; the test setup is a huge effort for a tester. Not all the testers have the harness infrastructure in place for testing due to the limitations of the hardware they own.

Most of the test projects act at best as regression, stress, performance test suites, or combinations thereof, but we do not have tests which can be used for functional testing of the new features being merged into the kernel. It is very critical for new code to be tested thoroughly, even before it gets merged in to the mainline.

In the existing scenario there are many valuable test cases/scripts available from individuals, which could expose bugs on other environments untested by them. *Sharing these test cases and scripts with the community* through one of the test projects will help in improving the testing much more by:

- Enabling the code to be tested on a variety of hardware,

- Improving the test coverage on executing all possible code paths,
- Avoiding duplication of test case development, and
- Making reproduction of bugs easier.

2.4 Kernel options

The Linux kernel is highly configurable, which allows specific features to be enabled as required. This means that to test the kernel fully we would have to test with each combination of these options. Testing kernels compiled only with the best known[1] configurations cannot expose any bugs hidden under the untried combinations. As an example the kernel can be configured to use any one of the following different memory models:

```
CONFIG_FLATMEM
CONFIG_DISCONTIGMEM
CONFIG_SPARSEMEM
CONFIG_SPARSEMEM_EXTREME
CONFIG_SPARSEMEM_VMEMMAP
```

They are mutually exclusive, which means the kernel can be compiled with only one of these options. Testing all combinations of memory models would require five different build and test cycles. Some of the combinations should be tested to improve the testing coverage of the kernel.

Many of the new features getting merged into the kernel are not tested by all individuals because their existence is not known to the tester. The example above shows how quickly permutations and combinations can grow. Usually what gets tested is the defaults on each architecture and the defaults that depend on the machine (testers do not deviate from a configuration that works for them).

It is important to test the responsiveness of the kernel with different boot options (refer to [10] for more available kernel parameters). For example, booting with less memory by passing `mem=<less memory>` as a boot parameter could test kernel behaviour when booted on a system with less memory. An extensive testing of this kind could take lots of kernel testing cycles, with total number of test combinations = number of boot parameter combinations x number of kernel configurations x number of releases.

[1] the configuration with which the kernel builds and boots without any issues

2.5 Existing Test Projects

There are many existing test projects used by the individual test contributors. We summarize some of them along with their key features:

LTP (Linux Test Project)[2] is a Regression/Functional test suite. It contains 3000+ test cases to test the basic functionality of the kernel. It is capable of testing/stressing filesystem, memory, scheduler, disk I/O, network, and system calls. It also provides some additional test suites such as pounder, kdump, open-hpi, open-posix, code coverage, and others.

It is ideal for running the basic functionality verification, with sufficient stress generated from the test cases. LTP does not support the kernel build test. LTP results can be formatted as HTML pages. It lacks the support for machine parseable logs. The test case results are either PASS or FAIL, which makes it complex for a tester to understand the reason behind test failure.

IBM autobench is a client harness project which supports setting up the test execution environment, execution of the test suites, and capturing the logs with environmental statics. It supports a kernel build test along with support for profiling. It is capable of executing test cases in parallel. The job control support is basic, allowing user to have minimal control over the way the tests are executed. The tool is written using bash/perl scripts.

Autotest[3] is an open source based client/server harness capable of running as a standalone client or is easily plugged into an existing server harness. Test cases included are capable of regression, functionality, stress, performance, kernel build tests, and they support various profilers. Autotest is written in python which allows the user to take more control of job execution by including python syntax in the job control file. Its object oriented and has a cleaner design.

Autotest has built-in error handling support. The logs are machine parseable with consistent exit status of the test executed as well as providing a descriptive message of the status. Parse[4] is built into the server harness. It summarizes the job execution results from different

[2] `http://ltp.sourceforge.net/`

[3] `http://test.kernel.org/autotest/`

[4] Parser used by the autotest to parse the test results `http://test.kernel.org/autotest/Parse`

testers[5] and formats them in query-able fashion for the tester to interpret them better.

3 How is the kernel being tested?

3.1 Methodology

Release early, release often[7] is the Linux kernel development philosophy. The development branches are released very frequently. As an example we have two `-git` releases per day, typically one `-next` release along with regular `-rc` and `-mm` releases (Different development releases have been explained in section 2.1). Testing development releases earlier and more often helps in identifying the patches that break the kernel. This allows fixing them earlier in the cycle. Before merging the patches, they should have been tested across all supported architectures, but it is not practical to expect all developers to test their patches that widely before merging.

Ideally we do the **Build test** on all releases on various hardware[6], with different configuration options. Build test is focused on build errors and warnings while building the kernel. This is followed by the **Regression test** suite. This helps uncover bugs introduced as side effects of new kernel changes. In order to ensure a regression free kernel, the suite is bundled with test suites from different test projects to test the filesystems, disk I/O, memory, scheduler, IPC, commands functional verification, and system calls with little stress.

The more thoroughly the development releases are tested, the better the Linux kernel quality is. Testing thoroughly requires two or more machine days based on the tests run on the releases. We selectively pick up development releases for a complete round of testing. Testing all releases with more than just build and regression tests would take too long and the important releases would be tested much later, after the release.

We do build and regression testing on all of the Linux kernel releases, which includes `-stable`, `-rc`, `-git`, `-mm`, `-next`. The focus is on certain development releases, which are tested more with stress tests and functionality tests along with some profile information extraction. As an example we focus more on `-majorrelease`, `-rc1`, and `-mm` releases. The debug options are tested on the major releases. `-mm` is one of the important development releases with bleeding edge features incorporated in it, so we test rounds of `-mm + hotfixes` if available. The Filesystem stress tests are executed over:

- Ext2/3 Filesystem
- Reiserfs Filesystem
- XFS Filesystem
- JFS Filesystem
- CIFS Filesystem
- NFS3/4 Filesystem

Comparing kernel performance under certain workloads on the machine to historical measures verifies the performance improvement or degradation. **Performance Testing** results are captured for almost all of the kernel releases. Results of workloads such as dbench, kernbench, and tbench are captured on the same machines, consistently validating the performance with every kernel release.

3.2 Infrastructure

Human hours are costlier in comparison to machine hours. Given the frequency of kernel releases, machine hours can be better used for setting up test environments and execution. Human hours can be best utilized in analyzing test results and debugging any bugs found. Figure 3 explains how the infrastructure works.

Mirror/Trigger: kernels are rsync'd to the local mirror, within a few minutes of the releases. Once the mirroring is complete, it acts as a trigger to test the newly downloaded kernel image. The trigger is initiated by any of the kernel releases mentioned in Section 2.1.

Test Selection/Job Queues: based on the kernel release, the predefined test cases are queued to the server for test execution. Section 3.1 explains the selection criteria of different predefined set of test cases to be queued, based upon the kernel release.

IBM's ABAT[7] server schedules the queued jobs from users based upon the availability of machine and does

[5] TKO (test.kernel.org) database is used to populate the results

[6] we cover the x86, Power™ and s390x architectures

[7] Automated Build And Test

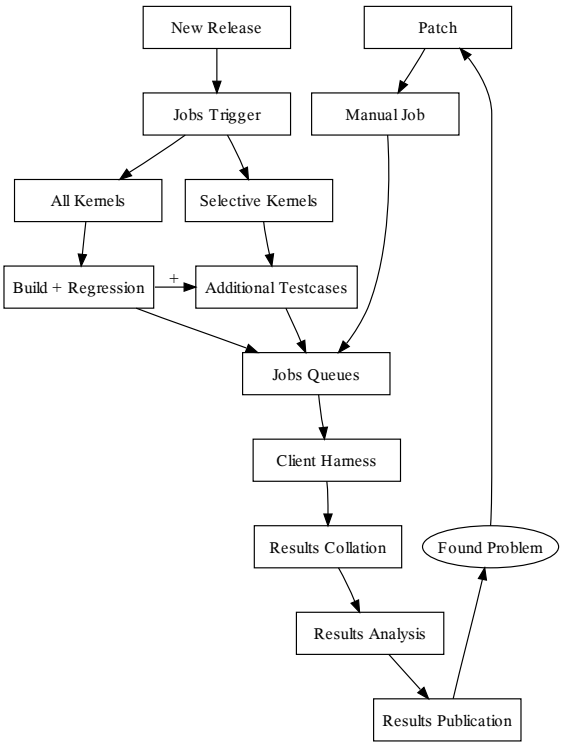

Figure 3: Infrastructure of kernel.org testing

more than just a simple queuing system. ABAT test framework is not open source, but the results are published to the community through `http://test.kernel.org`[8]

Client harness: both IBM autobench and autotest are supported as the client harness tools though their control file syntax is different. The client starts the test execution reading the job control file that is passed over when the job gets scheduled. It is responsible for building the appropriate kernel, running the tests, capturing the logs and other information, and making the results available. Section 2.5 explains more about these clients.

Results Collation: results are gathered asynchronously as the jobs complete and are pushed to `test.kernel.org`. They are grouped relevantly by TKO before publishing them. Kernel binaries and other system information dumps are stripped off the results. Results published at TKO are of a standard set of test cases that are used for testing.

[8]In collaboration between development and test team at IBM Linux Technology Center

Results Analysis: tests produce large amount of logs and other information. Analyzing the test information collected is time consuming. Relevant information is extracted and displayed as status using colour combinations each representing the percentage of test cases completed successfully. Results can be viewed in different layouts.[9] Performance data is analysed on selective benchmarks to provide historical performance graphs.

Results Publication: after automated analysis, the results are made available on the TKO website. Human monitoring is needed to take action on test failures or performance regressions. The problems are reported to the community (mostly via email) with the links to the test results.

Found Problem: when a test failure or performance regression is noticed, it is reported back to the community (mostly via email) by the person monitoring the results. Depending on the kernel release, another round of jobs are queued with the additional patches received for the problem reported. Currently only IBM engineers can manually submit the jobs on ABAT though the results and performance graphs are published as is done for regular jobs.

4 Results Analysis

The test procedure attempts to execute as many as possible code paths in the Linux kernel using different test projects. When combined together, the various tests tend to cover most of the kernel, but there has always been a gap between *tested and untested code*. Code coverage helps us to quantify this gap.

In this section we compare the code coverage results of `2.6.20, 2.6.21, 2.6.22, 2.6.23 and 2.6.24` for x86 and Power™ architecture. We also look at the results of executing some of these tests using the fault injection framework.

4.1 Code Coverage Setup

The gcov kernel patch[10] and lcov package[11] from LTP were used for the code coverage. Table 2 shows

[9]user selects the row and column heads. Condition based views are available.

[10]`http://ltp.sourceforge.net/coverage/gcov.php`

[11]`http://ltp.sourceforge.net/coverage/lcov.php`

Benchmarks	Description
runltp	A collection of tools for testing the Linux kernel and related features.
dbench	Filesystem benchmark that generates good filesystem load.
aio-stress	Filesystem benchmark that generates asynchronous I/O stress load.
aio-cp	Testing tool that copies files by using async I/O state machine.
hackbench	A benchmark for measuring the performance, overhead, and scalability of the Linux scheduler.
vmmstress	Performs general stress with memory race condtions between simultaneous read fault write fault, copy on write (COW) fault.
kernbench	A CPU throughput benchmark. It is designed to compare kernels on the same machine, or to compare hardware.
ltp-stress	Stresses the system using the LTP test suite.
hugepage-tests	Perform basic functional and stress tests for large pages
ramsnake	Allocate 1/8 of the system RAM and kick off threads to use them for 3600 seconds.
random_syscall	Pounds on syscall interface and does random syscalls
reaim	A multiuser benchmark that tests and measures the performance of open system multiuser computers.
sdet	Workload created by parallel execution of common UNIX commands.
libhugetlbfs	Interacts with the Linux hugetlbfs to make large pages available to applications in a transparent manner.
Others	Tested NFS, CIFS and the autofs filesystem.

Table 2: Benchmarks used for Code Coverage

the benchmarks used. Coverage was run on x86 and Power™ architectures, with following configurations:

8P Intel® XEON™, 10GB Memory
2P IBM® POWER5+™, 7GB Memory

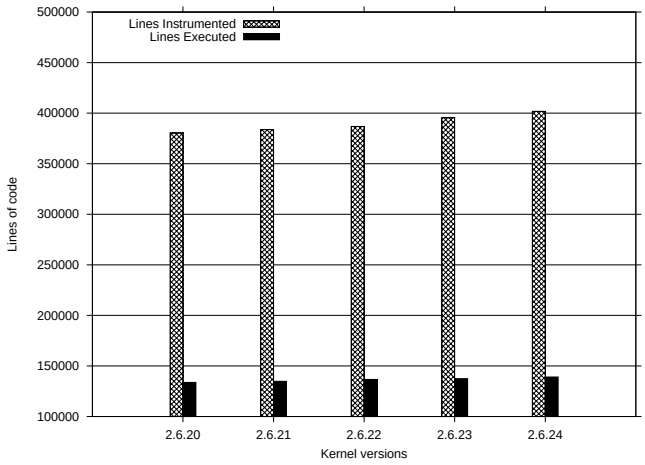

Figure 4: Code coverage on x86 architecture

Kernels	Lines Instrumented	Lines Executed
2.6.20	380,333	133,718
2.6.21	383,800	134,800
2.6.22	386,868	136,653
2.6.23	395,520	137,472
2.6.24	401,802	139,052

Table 3: Lines Instrumented Vs Executed on x86 architecture

Figure 4 and Table 3 show number of lines instrumented and the code covered of the various kernels mentioned for the x86 architecture. The figure shows the following trends:

- The number of lines instrumented shows a modest increase of 5.64%. When compared to the rate of change of the kernel, it does not seem significant. The code coverage data above fails to show that changed lines are also covered as the kernel version changes.
- The code coverage shows a modest increase of 3.98%. It is very encouraging to see code coverage increase as the number of lines in the kernel increase.
- Versions 2.6.23 and 2.6.24 show a trend of decline in code coverage percentage. The coverage

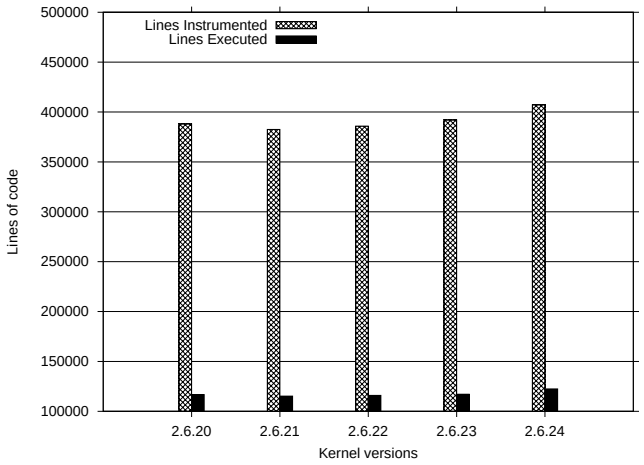

Figure 5: Code coverage on Power™ architecture

Kernels	Lines Instrumented	Lines Executed
2.6.20	388,019	116,736
2.6.21	382,552	115,242
2.6.22	385,853	116,010
2.6.23	391,993	117,168
2.6.24	407,194	122,431

Table 4: Lines Instrumented Vs Executed on Power™ architecture

of 2.6.20 kernel was 35.15%, where as for 2.6.24 it is 34.6%.

Figure 5 and Table 4 show the coverage of the various kernels mentioned for the Power™ architecture. The figure shows the following trends:

- The number of lines of instrumented code show a trend similar to the ones for the x86 architecture.
- The code coverage has increased with increasing kernel versions.
- The code coverage percentage however is less than that of the x86 architecture, close to 30%.
- There is no decline in the code coverage percentage as seen on the x86 architecture, indicating that x86 is growing rapidly.[12]

Figures 6,7 and Tables 5,6 show the component-wise break up for the code coverage obtained for the 2.6.24

[12] which might be true, due to the x86 and x86-64 merge.

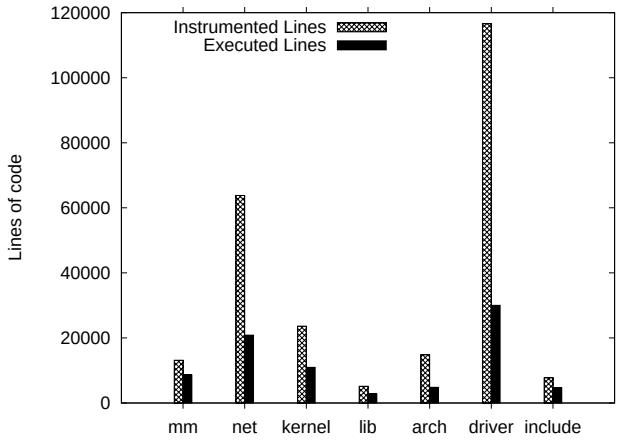

Figure 6: Component-wise code coverage on 2.6.24 kernel with x86 architecture

Directories	Lines Instrumented	Lines Executed
mm	1,3147	8,597
net	63,802	20,855
kernel	23,633	10,962
lib	5,152	2,785
arch	14,836	4,810
driver	116,623	30,031
include	7,751	4,707

Table 5: Lines Instrumented Vs Executed on 2.6.24 kernel with x86 architecture

Directories	Lines Instrumented	Lines Executed
mm	14,585	9,133
net	65,441	20,749
kernel	23,407	10,916
lib	5,062	2,799
arch	24,954	5,700
driver	88,195	8,930
include	7,638	4,434

Table 6: Lines Instrumented Vs Executed on 2.6.24 kernel with Power™ architecture

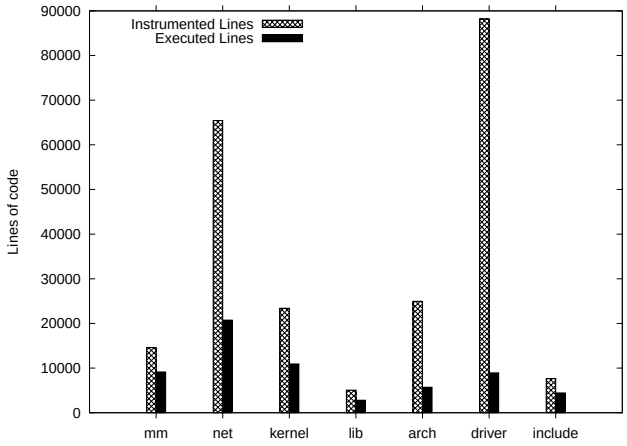

Figure 7: Component-wise code coverage on 2.6.24 kernel with Power™ architecture

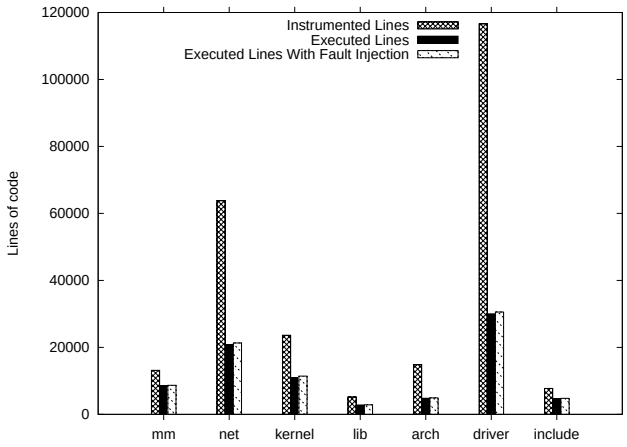

Figure 8: Fault injection code coverage on 2.6.24 kernel with x86 architecture

kernel on the x86 and Power™ architectures respectively. The component-wise break up shows some interesting trends as well:

- The `mm`, `kernel`, `lib`, and `include` subdirectories are among those that have the highest code coverage.
- The subsystem with highest coverage is `mm`, with close to 65% coverage
- `drivers` and `arch` subdirectories are among those that have the least code coverage, The main focus of our testing is not to test architecture or platform-specific code.

Directories	Lines Instrumented	Lines Executed	
		Disabled	Forced
mm	13,147	8,597	8,688
net	63,802	20,855	21,306
kernel	23,633	10,962	11,431
lib	5,152	2,785	2,853
arch	14,836	4,810	4,918
driver	116,623	30,031	30,552
include	7,751	4,707	4,777

Table 7: Fault Injection code coverage on 2.6.24 kernel with x86 architecture

We used the **Fault Injection framework**[13] to get more coverage of the error handling path. The kernel was configured with:

```
N > /debug/fail_page_alloc/task-filter
20 > /debug/fail_page_alloc/probability
2000 > /debug/fail_page_alloc/interval
-1 > /debug/fail_page_alloc/times
0 > /debug/fail_page_alloc/space
1 > /debug/fail_page_alloc/verbose
N > /debug/fail_page_alloc/ignore-gfp-wait

N > /debug/fail_make_request/task-filter
20 > /debug/fail_make_request/probability
2000 > /debug/fail_make_request/interval
-1 > /debug/fail_make_request/times
0 > /debug/fail_make_request/space
1 > /debug/fail_make_request/verbose
N > /debug/fail_make_request/ignore-gfp-wait
```

(refer to [9] for more configuration details) The results of the coverage are shown in Figure 8. Getting coverage results with fault injection enabled turned out to be very challenging, since most applications are not ready to deal with failures. The test applications we saw would fail and abort their operation on error. We had to manually make changes to get coverage data with fault injection and we were forced to keep the failure rate very low. Due to these factors, we did not see a significant improvement in code coverage with the fault injection framework enabled, it was just 0.6% improvement. The test cases and infrastructure need to be enhanced to deal with the failures forced from the fault injection framework.

[13] injects errors at various kernel layers, helping to test error handling

5 Future Plans

We plan to extend our testing by adopting and updating test cases from various test projects and test scripts published on the mailing list to improve the coverage of untested code. We intend to test the practically possible permutations of kernel configurations and boot options with selective releases. We intend to build tests with the a cross-compiler setup will help us in finding the build errors over various platforms. Testing the kernel error path is very critical to avoiding surprises under certain situations. We could perform error handling path testing on selective kernels using the fault injection framework available in the kernel.

6 Conclusion

Testing kernel releases *earlier and often*, helps in fixing the bugs earlier in the cycle. The earlier the problem gets fixed, the lower the costs involved in fixing it. Our infrastructure tests various kernels across different hardware, using many benchmarks and test suites performing build, regression, stress, functional, and performance testing. Benchmarks are run selectively depending upon the kernel release. Testing results are contributed back to the community through `test.kernel.org`. We discussed some of the harnesses commonly used by the projects.

Code coverage results explain the gap between the lines of code being added and tested. For better and more complete testing of the kernel, we need test cases that can help us better test existing and new features. For these we require developers to share their tests and testing methodology. The fault injection framework helps testing the error handling part of the kernel, so improvements made to the framework could result in better kernel coverage.

7 Acknowledgments

We would like to thank Andy Whitcroft for his input to and review of drafts of this paper.

We also owe lot of thanks to Sudarshan Rao, Premalatha Nair, and our teammates for their active support and enthusiasm.

8 Legal Statement

References

[1] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson, *How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*, `http://www.linux-foundation.org/publications/linuxkerneldevelopment.php`, April 2008.

[2] Fully Automated Testing of the Linux Kernel, Martin Bligh and Andy P. Whitcroft. In *Proceedings of the Linux Symposium 2006.*

[3] Linux Test Project, `http://ltp.sourceforge.net/`

[4] Autotest, http://test.kernel.org/autotest

[5] Kernelsource/Documentation/HOWTO.

[6] Linux Kernel Mailing List. linux-kernel@vger.kernel.org, http://lkml.org/

[7] Linux Test Project - Test Tools Matrix. http://ltp.sourceforge.net/tooltable.php

[8] The Cathedral and the Bazaar, Eric Steven Raymond.

[9] kernelsource/Documentation/fault-injection/fault-injection.txt

[10] kernelsource/Documentation/kernel-parameters.txt