Keeping Kernel Performance from Regressions

Tim Chen
Intel Corporation
tim.c.chen@intel.com

Leonid I. Ananiev
leoan@mail.ru

Alexander V. Tikhonov
Intel Corporation
alexander.v.tikhonov@intel.com

Abstract

The Linux* kernel is evolving rapidly with thousands of patches monthly going into the base kernel. With development at this pace, we need a way to ensure that the patches merged into the mainline do not cause performance regressions.

The Linux Kernel Performance project was started in July 2005 and is Intel's effort to ensure every dot release from Linus is evaluated with key workloads. In this paper, we present our test infrastructure, test methodology, and results collected over the 2.6 kernel development cycle. We also look at examples of historical performance regressions that occurred and how Intel and the Linux community worked together to address them to make Linux a world-class enterprise operating system.

1 Introduction

In recent years, Linux has been evolving very rapidly, with patches numbering up to the thousands going into the kernel for each major release (see Figure 1) in roughly a two- to three-month cycle. The performance and scalability of the Linux kernel have been key ingredients of its success. However, with this kind of rapid evolution, changes detrimental to performance could slip in without detection until the change is in the distributions' kernels and deployed in production systems. This underscores the need for a systematic and disciplined way to characterize, test, and track Linux kernel performance, to catch any performance issue of the kernel at the earliest time possible to get it corrected.

Intel's Open Source Technology Center (OTC) launched the Linux Kernel Performance Project (LKP) in the summer of 2005 (`http://kernel-perf.sourceforge.net`) to address the need to monitor kernel performance on a regular basis. A group of OTC engineers set up the test machines, infrastructure, and benchmarks; they started regular testing and analysis of Linux kernel's performance. We began to publish our test data on project website since July 2005.

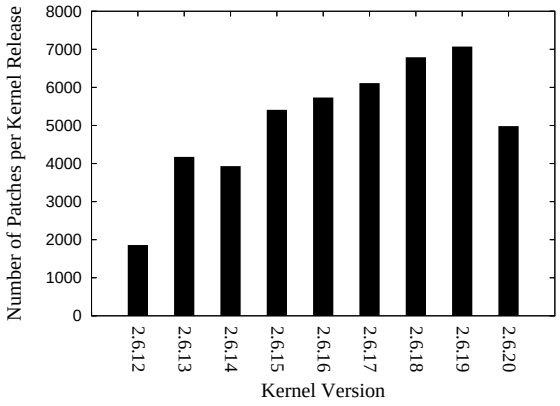

Figure 1: Rate of change in Linux kernel

2 Testing Process

Each release candidate of the Linux kernel triggers our test infrastructure, which starts running a benchmark test suite within an hour whenever a new kernel get published. Otherwise, if no new `-rc` version appears within a week, we pick the latest snapshot (`-git`) kernel for testing over the weekend. The test results are reviewed weekly. Anomalous results are double-checked, and rerun if needed. The results are uploaded to a database accessible by a web interface. If there were any significant performance changes, we would investigate the causes and discuss them on Linux kernel mailing list (see Figure 2).

We also make our data available on our website publicly for community members to review performance gains and losses with every version of the kernel. Ultimately, we hope that this data catches regressions before major kernel releases, and results in consistent performance improvement.

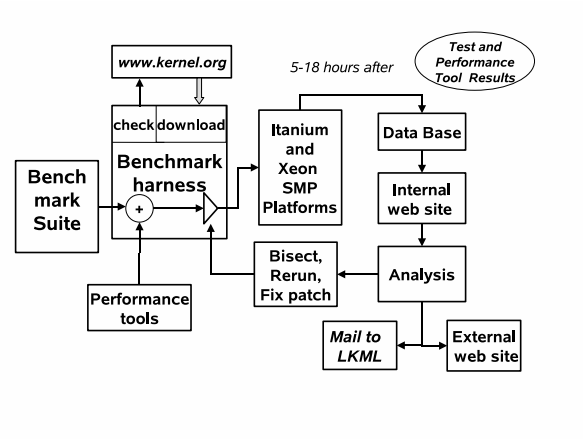

Figure 2: Performance testing process.

2.1 Benchmarks Suite

We ran a set of benchmarks covering core components of the Linux kernel (memory management, I/O subsystem, process scheduler, file system, network stack, etc.).

Table 1 lists and describes the benchmarks. Most of the benchmarks are open source and can be duplicated easily by others.

2.2 Test Platforms

Currently we have a mix of Itanium® SMP machines and Xeon® SMP machines to serve as our test platforms, with the configurations as listed below:

4P Intel® Itanium™ 2 processor (1.6Ghz)
4P Dual-Core Intel® Itanium® processor (1.5Ghz)
2P Intel® Xeon® MP processor (3.4Ghz)
4P Dual-Core Intel® Xeon® MP processor (3.0Ghz)
2P Xeon® Core™2 Duo processor (2.66Ghz)
2P Xeon® Core™2 Quad processor (2.40Ghz)

2.3 Test Harness

We needed a test harness to automate the regular execution of benchmarks on test machines. Even though there were test harnesses from the Scalable Test Platform (`http://sourceforge.net/projects/stp`) and Linux Test Project (`http://ltp.sourceforge.net`), they did not fully meet all of our testing requirements. We elected to create a set of shell scripts for our test harness, which was easy to customize for adding the capabilities we need.

Short name	**Description**
Kbuild	Measures the speed of Linux kernel compilation.
Reaim7	Stresses the scheduler with up to thousands of threads each generating load on memory and I/O.
Volanomark	A chatroom benchmark to test java thread scheduling and network scalability.
Netperf	Measures the performance of TCP/IP network stack.
Tbench	Load testing of TCP and process scheduling.
Dbench	A stress test emulating Netbench load on file system.
Tiobench	Multithread IO subsystem performance test.
Fileio	Sysbench component for file I/O workloads.
Iozone	Tests the file system I/O under different ratio of file size to system memory.
Aiostress	Tests asynchronous I/O performance on file system and block device.
Mmbench	Memory management performance benchmark.
Httperf	Measures web server performance; also measures server power usage information under specific offered load levels.
Cpu-int/fp	An industry standard CPU intensive benchmark suite on integer and floating point operations.
Java-business	An industry standard benchmark to measure server-side Java*, tests scheduler scalability.

Table 1: Performance benchmark suite

Our test harness provides a list of services that are itemized below:

- It can detect and download new Linux kernels from `kernel.org` within 30 minutes after release, and then automatically install the kernel and initiate benchmark suites on multiple test platforms.

- It can test patches on any kernel and compare results with other kernel version.
- It can locate a patch that causes a performance change by automating the cycle of git-bisect, kernel install for filtering out the relevant patch.
- It uploads results from benchmark runs for different kernels and platforms into a database. The results and corresponding profile data can be accessed with a friendly web interface.
- It can queue tasks for a test machine so that different test runs can be executed in sequence without interference.
- It can invoke a different mix of benchmarks and profiling tools.

We use a web interface to allow easy comparison of results from multiple kernels and review of profiling data. The results may be rerun using the web interface to confirm a performance change, or automated git-bisect command be initiated to locate the patch responsible. The results are published in external site (`http://kernel-perf.sourceforge.net`) after they have been reviewed.

3 Performance Changes

During the course of the project, our systematic testing has revealed performance issues in the kernels. A partial list of the performance changes are listed in Table 2. We will go over some of those in details.

3.1 Disk I/O

3.1.1 MPT Fusion Driver Bug

There was a sharp drop in disk performance for the 2.6.13 kernel (see Figure 3). Initially we thought that it was related to the change in system tick from 1000Hz to 250Hz. After further investigation, it was found that the change in Hz actually revealed a race condition bug in the MPT fusion driver's initialization code.

Our colleague Ken Chen found that there were two threads during driver initialization interfering with each other: one for domain validation, and one for host controller initialization. When there were two host controllers, while the second host controller was brought up, the initialization thread temporarily disabled the channel for the first controller. However, domain validation was in progress on first channel in another thread (and possibly running on another CPU). The effect of disabling the first channel during in-progress domain validation was that it caused all subsequent domain validation commands to fail. This resulted in the lowest possible performance setting for almost all disks pending domain validation. Ken provided a patch and corrected the problem.

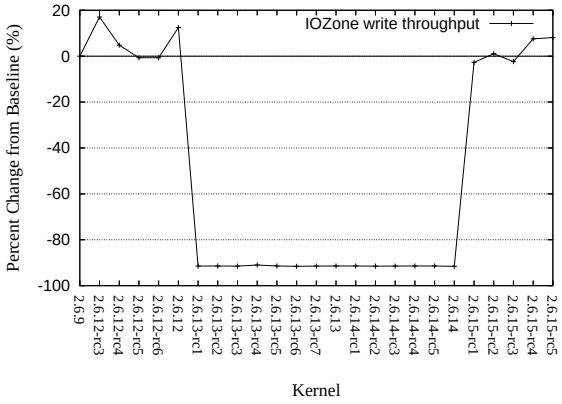

Figure 3: MPT Fusion driver bug

3.2 Scheduler

3.2.1 Missing Inter-Processor Interrupts

During the 2.6.15 time frame, there was a 60% decrease in Volanomark throughput on Itanium® test machines (see Figure 4). It was caused by a patch that caused rescheduled Inter Processor Interrupts (IPI) not to be sent from `resched_task()`, ending up delaying the rescheduling task until next timer tick, thus causing the performance regression. The problem was quickly reported and corrected by our team.

3.2.2 Scheduler Domain Corruption

During the testing of benchmark Httperf, we noticed unusual variation on the order of 80% in the response time of the web server under test for the 2.6.19 kernel. This was caused by a bug introduced when the

Kernel	Patch causing change	Effect
2.6.12-rc4	noop-iosched: kill O(N) merge scan.	Degraded IO throughput for noop IO scheduler by 32%.
2.6.13-rc2	Selectable Timer Interrupt Frequency of 100, 250, and 1000 HZ.	Degraded IO throughput by 43% due to MPT Fusion driver.
2.6.15-rc1	sched: resched and cpu_idle rework.	Degraded performance of Netperf (-98%) and Volanomark (-58%) on ia64 platforms.
2.6.15-rc2	ia64: cpu_idle performance bug fix	Fixed Volanomark and netperf degradations in 2.6.15-rc1.
2.6.15-rc5	[SCSI] mptfusion : driver performance fix.	Restored fileio throughput.
2.6.16-rc1	x86_64: Clean up copy_to/from_user. Remove optimization for old B stepping Opteron.	Degraded Netperf by 20% on Xeon® MP.
2.6.16-rc3	x86_64: Undid the earlier changes to remove unrolled copy/memset functions for Xeon® MP.	Reverted the memory copy regression in 2.6.16-rc1.
2.6.18-rc1	lockdep: irqtrace subsystem, move account_system_vtime() calls into softirq.c.	Netperf degraded by 3%.
2.6.18-rc4	Reducing local_bh_enable/disable overhead in irq trace.	Netperf performance degradation in 2.6.18-rc1 restored.
2.6.19-rc1	mm: balance dirty pages Now that we can detect writers of shared mappings, throttle them.	IOzone sequential write dropped by 55%.
2.6.19-rc1	Send acknowledge each 2nd received segment.	Volanomark benchmark throughput reduced by 10%.
2.6.19-rc1	Let WARN_ON return the condition.	Tbench degraded by 14%.
2.6.19-rc2	Fix WARN_ON regression.	Tbench performance restored.
2.6.19-rc2	elevator: move the back merging logic into the elevator core	Noop IO scheduler performance in 2.6.18-rc4 fixed and restored to 2.6.12-rc3 level

Table 2: Linux kernel performance changes seen by test suites

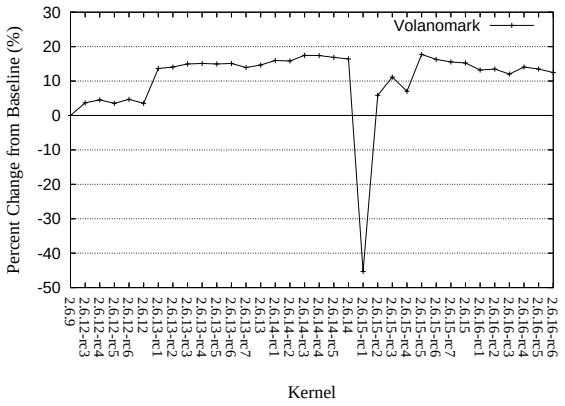

Figure 4: IPI scheduling bug

`cpu_isolated_map` structure was designated as init data. However, the structure could be accessed again after the kernel was initialized and booted when a rebuild of `sched_domain` was triggered by setting the `sched_mc_power_savings` policy. Subsequently, the corrupted `sched_domain` caused bad load-balancing behavior and caused erratic response time.

3.2.3 Rotating Staircase Dead Line Scheduler

The recently proposed RSDL (Rotating Staircase Dead Line) scheduler has generated a lot of interest due to its elegant handling of interactivity. We put RSDL 0.31 to test and found that for Volanomark, there is a big 30% to 80% slowdown. It turned out that the yield semantics in RSDL 0.31 were too quick to activate the yielding process again. RSDL 0.33 changed the yield semantics to allow other processes a chance to run, and the performance recovered.

3.3 Memory Access

3.3.1 Copy Between Kernel and User Space

During the 2.6.15-rc1 timeframe, we detected a drop up to 30% in Netperf's throughput on older Xeon® processor MP-based machines (see Figure 5). This was caused by a switch in the copy between user and kernel space to use repeat move string instructions which are slower than loop-based copy on Xeon® processor MP. This problem was corrected quickly. Later, when

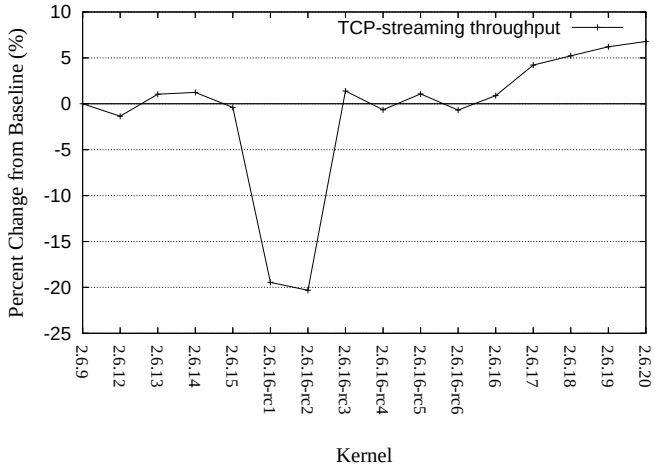

Figure 5: Xeon® processor MP's Netperf TCP-streaming performance made worse using string copy operations

the newer Core™2 Duo based Xeon®s became available with efficient repeat move string instructions, a switch to use string instructions in the 2.6.19 kernel actually greatly improved throughput. (see Figure 6).

3.4 Miscellaneous

3.4.1 Para-Virtualization

The para-virtualization option was introduced in the 2.6.20 time frame, and we detected a 3% drop in Netperf and Volanomark performance. We found that Para-virtualization has turned off VDSO, causing `int 0x80` rather than the more efficient `sysenter` to be used for system calls, causing the drop.

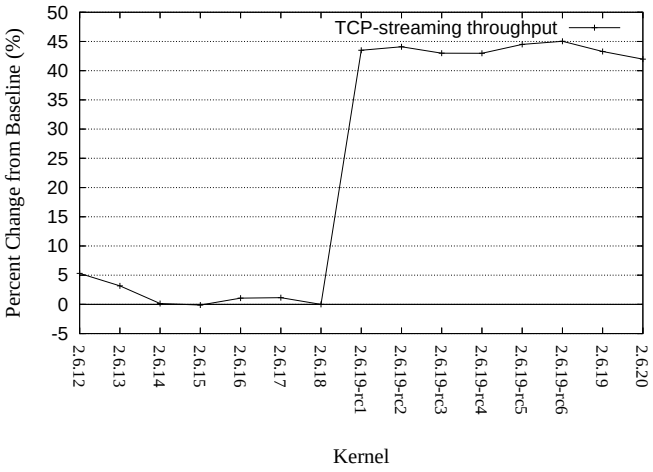

Figure 6: Xeon® Core™2 Duo's Netperf TCP-streaming performance improved with string copy operations

3.4.2 IRQ Trace Overhead

When the IRQ trace feature was introduced in 2.6.18-rc1, it unconditionally added `local_irq_save(flags)` and `local_irq_restore(flags)` when enabling/disabling bottom halves. This additional overhead caused a 3% regression in Netperf's UDP streaming tests, even when the IRQ tracing feature was unused. This problem was detected and corrected.

3.4.3 Cache Line Bouncing

There was a 16% degradation of tbench in 2.6.18-rc14 (see Figure 7) We found that a change in the code triggered an inappropriate object code optimization in older gcc 3.4.5, which turned a rare write into a global variable into an always write event to avoid a conditional jump. As a result, cache line bouncing among the cpus increased by 70% from our profiling. A patch was later merged by Andrew into the mainline to sidestep this gcc problem.

4 Test Methodology

4.1 Test Configuration

We tried to tune our workloads so they could saturate the system as much as possible. For a pure

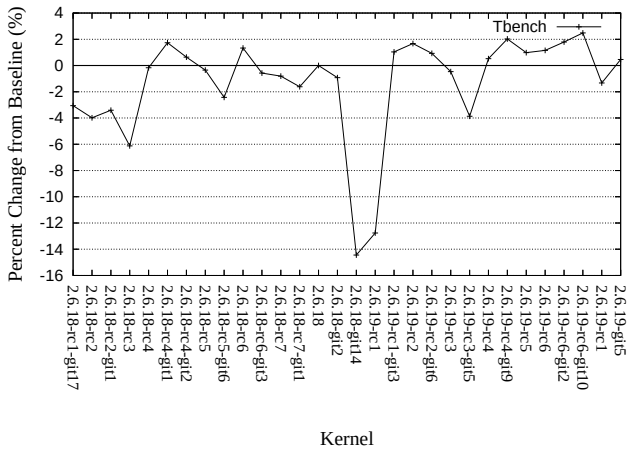

Figure 7: Tbench performance problem caused by cache line bouncing

CPU-bound workload, the CPU utilization was close to 100%. However for a workload involving I/O, the system utilization was limited by the time waiting for file and network I/O. The detailed benchmark options for each test are described on our website (`http://kernel-perf.sourceforge.net/about_tests.php`). Table 3 gives a sense of the average loading for different benchmarks. The loading profile is the standard one from vmstat.

For the disk-bound test workload, we reduced the amount of main memory booted to only 1GB (that's only a quarter to one-eighth of the memory of our system). The test file size was a few times of the size of memory booted. This made the actual effect of I/O dominant and reduced the effect of file I/O cache.

Name	% cpu	% io	% mem	% user	% sys
Reaim7	100	1	68	85	15
Aiostress	1	36	83	0	1
Dbench	37	28	95	1	36
Fileio	1	14	100	0	1
IOzone	1	23	99	0	1
Kbuild	79	9	90	74	5
Mmbench	2	66	99	0	2
Netperf	40	0	34	2	38
Cpu-int/fp	100	0	75	100	0
Java-business	39	0	89	39	0
tbench	97	0	41	5	92
Volanomark	99	0	96	45	54

Table 3: Sample system loading under benchmarks

4.2 Dealing with Test Variation

Variations in performance measurements are part of any experiment. To some extent the starting state of the system, like cpu cache, file I/O cache, TLB, and disk geometry, played a role. However, a large variation makes the detection of change in performance difficult.

To minimize variation, we do the following:

- Start tests with a known system state;
- Utilize a warm-up workload to bring the system to a steady state;
- Use a long run time and run the benchmark multiple times to get averages of performance where possible.

To get our system in known state, we rebooted our system before our test runs. We also reformatted the disk and installed the test files. This helped to ensure the layout of the test file and the location of journal on the disk to remain the same for each run.

We also ran warm-up workloads before the actual benchmark run. This helped bring the CPU caches, TLB, and file I/O cache into a steady state before the actual testing.

The third approach we took was to either run the benchmark for a long time or to repeat the benchmark run multiple times and measure the average performance. Short tests like Kbuild, when run repeatedly for 15 times in our test run, got a good average value with standard deviation below 1%. The average performance value has reduced variance and resembles more closely a well behaved Gaussian distribution [5]. Single run results for some benchmarks are far from a Gaussian distribution. One such example is Tbench.

Figure 8 superimposes the distribution of throughput from a single run of Tbench versus a normal distribution with the same standard deviation. It can be seen that the distribution from a single benchmark run is bimodal and asymmetric. Therefore using a single measurement for comparison is problematic with the issues raised in [1-4]. For these cases, a more accurate performance comparison is made by using average values, which resemble much more closely a normal distribution and have smaller deviations.

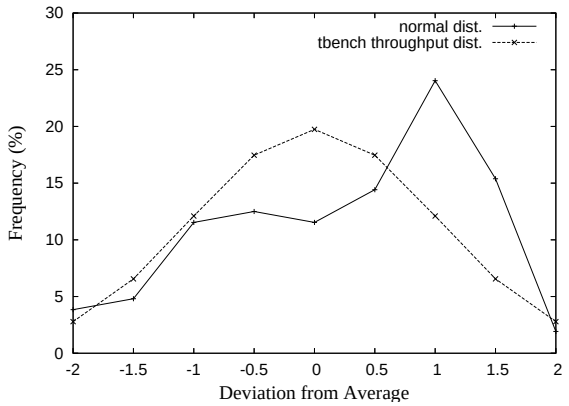

Figure 8: Tbench throughput distribution

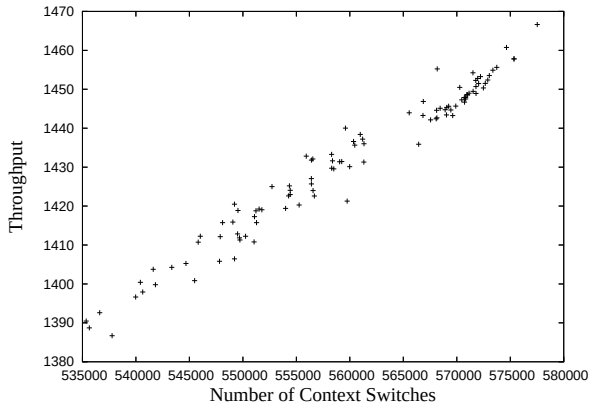

Figure 9: Tbench throughput vs. context switches

Sometimes we can better understand the underlying reason for performance variation by correlating the performance variation with changes in other profiling data. For example, with Tbench, context switch has a 0.98 correlation coefficient with the throughput (see Figure 9). This gives an indication that the variation in context switch rate is highly correlated with the variation in throughput. Another example is Kbuild (see Figure 10), where we find the number of merged IO blocks had a –0.96 correlation coefficient with the kernel compile time, showing that the efficiency of disk I/O operations in merging IO blocks is critical to throughput.

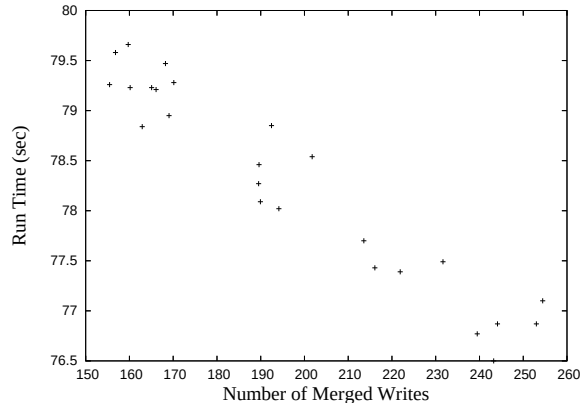

Figure 10: Kbuild runtime vs. number of merged IO blocks

This kind of almost one-to-one correlation between throughput and profiling data can be a big help to check whether there is a real change in system behavior. Even though there are variations in throughput from each run, the ratio between the throughput and profile data should be stable. So when comparing two kernels, if there is a significant change in this ratio, we will know that there are significant changes in the system behavior.

We have also performed a large number of runs of benchmarks on a baseline kernel to establish the benchmark variation value. Both max and min values are saved in a database to establish a confidence interval for a benchmark. This value is used for results comparison: if the difference in measured performance values is more than the confidence interval, then there is a significant change in the kernel performance that should be looked into. In general, disk-I/O-bound benchmarks have much higher variation, making it much harder to detect small changes in performance in them.

4.3 Profiling

Our test harness collected profiling data during benchmark runs with a set of profiling tools: `vmstat`, `iostat`, `sar`, `mpstat`, `ps`, and `readprofile`. The profiling data provided information about the load on the CPU from user applications and the activities of the kernel's subsystems: scheduler, memory, file, network, and I/O. Information about I/O queue length and hot kernel functions had been useful for us in locating bottlenecks in the kernel and to investigate the cause of performance changes. The waiting time from `vmstat` can be combined with `wchan` information from `ps` to gain insight to time spent by processes waiting for events. Table 4 provides a profile of waited events for a run snapshot of Aiostress and Reaim7 benchmarks as an example.

Aiostress		Reaim7	
1209	pause	18075	pause
353	io_getevents	8120	wait
13	get_write_access	1218	exit
12	sync_buffer	102	pipe_wait
6	stext	62	start_this_handle
3	sync_page	1	sync_page
1	congestion_wait	2	sync_page
1	get_request_wait	2	cond_resched

Table 4: The events waited by Aiostress and Reaim7

4.4 Automated Git-Bisect

The git bisect utility is a powerful tool to locate the patch responsible for a change in behavior of the kernel from a set of patches. However, manually running it to bisect a large patch-set repeatedly to find a patch is tedious. One has to perform the steps of bisecting the patch set into two, rebuild, install, and reboot the kernel for one patch set, run the benchmark to determine if the patch set causes an improvement or degradation to the performance, and determine which subset of the two bisected patch sets contains the patch responsible for the change. Then the process repeats again on a smaller patch set containing the culprit. The number of patches between two rc releases are in the hundreds, and often 8 to 10 repetitions are needed. We added capability in our test harness to automate the bisect process and benchmark run. It is a very useful tool to automatically locate the patch responsible for any performance change in O(log n) iterations.

4.5 Results Presentation

After our benchmark runs have been completed, a wrapper script collects the output from each benchmark and puts it into a 'comma separated value' format file that is parsed into a MySQL* database. The results are accessible through an external web site `http://kernel-perf.sourceforge.net` as a table and chart of percentage change of the performance compared to a baseline kernel (selected to be 2.6.9 for older machines, and 2.6.18 for newer ones). Our internal web site shows additional runtime data, kernel config file, profile results, and a control to trigger a re-run or to perform a git bisect.

4.6 Performance Index

It is useful to have a single performance index that summarizes the large set of results from all the benchmarks being run. This approach has been advocated in the literature (see [1]-[4]). This is analogous to a stock market index, which gives a sense of the overall market trend from the perspective of individual stock, each weighted according to a pre-determined criterion.

Benchmark	Number of subtests	Deviation %	Weight per metric
Reaim7	1	0.46	2
Aiostress	8	12.8	0.01
Cpu-int/fp	2	0.6	1
Dbench	1	11.3	0.1
fileio	1	11.8	0.1
Iozone	21	14.7	0.01
Kbuild	1	1.4	1
Mmbench	1	4.9	0.2
Netperf	7	1.6	0.15
Java-Business	1	0.6	1
tbench	1	12.7	0.5
Tiobench	9	11.4	0.01
Volanomark	1	0.8	1

Table 5: Number of subtests, variations weights on subtests for each benchmark

We use the geometric mean of ratios of performance metric to its baseline value of each benchmark as a performance index, as suggested in [2]. We weigh each benchmark according to its reliability (i.e., benchmarks with less variance are weighed more heavily). If a benchmark has a large number of subtests producing multiple metrics, we put less weight on each metric so the benchmark will not be over-represented in the performance index. Table 5 shows the weights being chosen for some of our benchmarks.

We use a weighted version of the geometric mean to aid us in summarizing the performance of the kernel. This weighted geometric index, though somewhat subjective, is a very useful tool to monitor any change in overall kernel performance at a glance and help guide us to the specific benchmarked component causing the change. Figure 11 shows the performance index produced over our benchmark suite. It is interesting to note that from the limited perspective of the benchmark suite

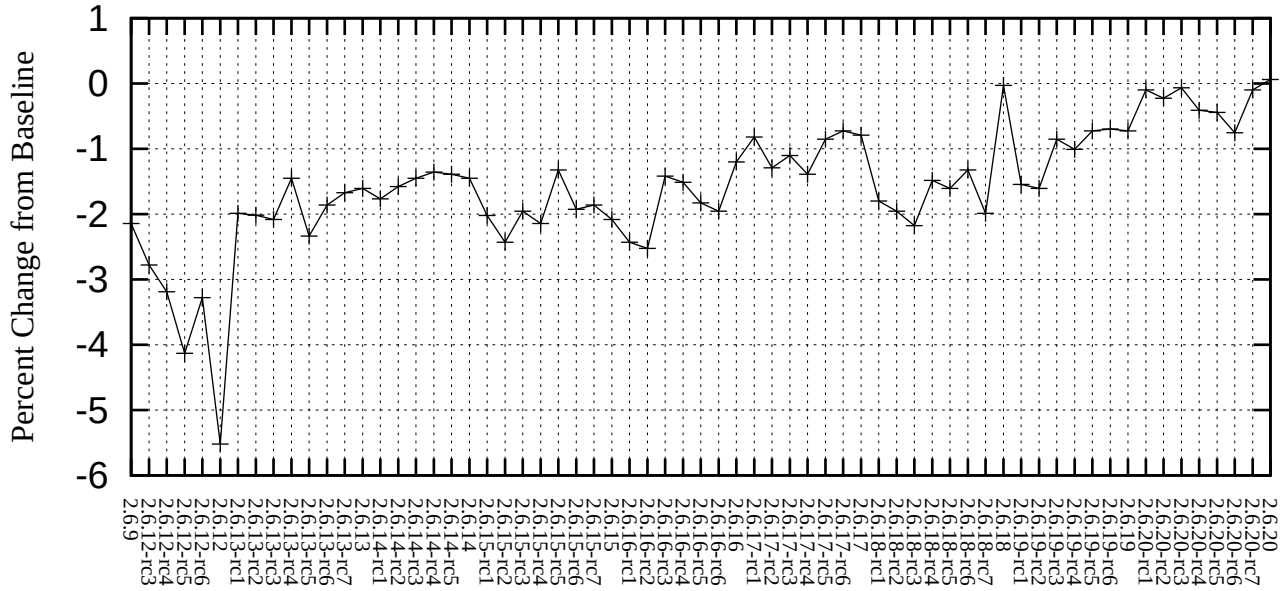

Figure 11: Weighted geometric mean performance for all benchmarks

we run regularly, the index for the 2.6 kernel series has been trending upwards.

5 Conclusion

Our project set up the infrastructure to systematically test every kernel release candidate across multiple platforms and benchmarks, and also made the test data available to the community on the project website, `http://kernel-perf.sourceforge.net`. As a result, we have been able to catch some kernel regressions quickly, and worked with the community to fix them. However, with rapid changes in the kernel, the limited coverage from our regular benchmark runs could uncover only a portion of performance regressions. We hope this work will encourage more people to do regular and systematic testing of the Linux kernel, and help prevent performance problems from propagating downstream into distribution kernels. This will help to solidify Linux's position as a world-class enterprise system.

Acknowledgments

The authors are grateful to our previous colleagues Ken Chen, Rohit Seth, Vladimir Sheviakov, Davis Hart, and Ben LaHaise, who were instrumental in the creation of the project and its execution. A special thanks to Ken, who was a primary motivator and contributor to the project.

Legal

References

[1] J.E. Smith, "Characterizing Computer Performance with a Single Number," Communications of ACM, 31(10):1202–1206, October 1988.

[2] J.R. Mashey, "War of the Benchmark Means: Time for a Truce" ACM SIGARCH Computer Architecture News. Volume 32, Issue 4 (September 2004), pp. 1–14. ACM Press, New York, NY, USA.

[3] J.R. Mashey, "Summarizing Performance is No Mean Feat" Workload Characterization Symposium, 2005. Proceedings of the IEEE International. 6–8 Oct. 2005 Page(s): 1. Digital Object Identifier 10.1109/IISWC.2005.1525995

[4] L. John, "More on finding a Single Number to indicate Overall Performance of a Benchmark Suite," Computer Architecture News, Vol. 32, No 1, pp. 3–8, March 2004.

[5] D.J. Lilja, "Measuring computer performance: a practitioner's guide," Cambridge University Press, 2005.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
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*
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.*