Performance Testing the Linux Kernel

The re-aim workload

Cliff White
Open Source Development Labs
cliffw@osdl.org, http://www.osdl.org/archive/cliffw

Abstract

Good performance testing requires good tests and good procedures. This paper discusses experiences creating and using an automated test environment.

The paper also describes work done at Open Source Development Labs (OSDL™) in rewriting and modernizing the AIM7 and AIM9 benchmarks. The intent is to make the benchmarks relevant for modern hardware by making it flexible and extensible.

This paper talks about how to create a testing environment, how to automate it, and how to select and evaluate potential tests. The paper talks about the differences between low-level (micro) workloads and application-modeling (macro) workloads, using OSDL Scalable Test Platform tests as examples, and talk about the difference between tests that focus on specific areas and tests that exercise broad areas.

1 Introduction

Performance testing in kernel context is necessary to show that a projected improvement is in fact an improvement. Performance tests are used to measure large-scale application performance and small-scale system routine and system call performance.

There are two areas not specifically addressed by performance testing. One area is compliance which the Linux Standard Base and Linux Test Project test suites both address. The other area is reliability—demonstrating the ability to sustain proper operation over long time spans.

A goal of the OSDL's Scalable Test Platform is to measure performance, over and over again. To do this, we run publicly available workloads, and we create a few of our own. This paper describes work being done to revise an old workload suite, the AIM tests.

2 Creating a Proper Test Environment

A good test must be repeatable. It is very important that multiple runs of a test on the same hardware with the same kernel produce the same results. OSDL's STP creates this repeatable environment by re-loading the test machines with a new OS before every run. Thus, every test starts from an identical state. For non-STP testing, the system is set up for repeated runs by using a Makefile. Whenever a new test is created, the first thing created is a setup/tear-down Makefile. In the Makefile, careful track is kept of everything added to the system for the test.

When running the test, care should be taken to understand and control the test environment. There are a few areas to consider:

- Networking – This should be obvious, but any test of networking should be run on a private network, where no other traffic impacts the test measurement. This is especially important when a test is controlled or monitored via a network.

- Other shared resources – Might include shared storage arrays, or other devices. Again, it is best to use dedicated hardware or stop other users before testing.

- State of the system prior to test startup – This is especially important for repeating test results. Rebooting the system prior to every test run is one way to assure a known state. However, many tests are very influenced by cache effects and this must be considered. When testing database workloads, it is common to warm the database cache prior to taking any performance measurement.

- Repeat testing for repeatability – A single test result might be useful. A repeatable test result is much more usable. Statistically, three runs are about the minimum for good data, five or more runs are better.

2.1 Experiences from the STP

Here is some advice, culled from experiences adding tests to the OSDL's Scalable Test Platform.

- When scripting for automation, error recovery is everything. Error reporting is more important. Error discovery is most important. You will find that making things happen in a script is easy—knowing when things have not happened and doing the right thing thereafter is hard.

- Be very paranoid. Review and sanity check test results frequently. Hardware failures can be very sneaky; repeating known tests can be a good way to spot flaky hardware.

- When running a large or even a medium number of systems, administration tools are very important, especially tools that allow you to look at health over time.

- When testing kernels, sometimes the most interesting tests are the ones that do not run at all.

- Likewise, be aware of timeout conditions—the tests that never complete can also be interesting. You should have timeout conditions for each phase of an automated process.

- Build the tools to parse and present results when you build the test. If possible, build the tool so you can compare multiple runs.

- Likewise, instrument the test when you build the test. Add readprofile or oprofile if possible. However, be aware of the impact of your instrumentation; touching `/proc` too frequently can impact your results.

- Results presentation is very important. design the report so that the most important data is the easiest to see.

- Establish a baseline run you can use for comparison purposes.

- Compare frequently to that baseline. Test results in isolation are less interesting than comparisons to known conditions.

- Establish your hardware baselines in as much detail as you can. In a perfect world, what is the maximum rate your disk subsystem can deliver? Knowing these rates

can help you determine when a test is using real hardware and when a test is running from disk cache.

2.2 Macro and Micro Workloads

STP uses two very different types of tests when testing kernel performance. These tests are divided into macro and micro workloads. Micro workloads are tests that exercise a very small piece of the system, such as a single system call. These tests focus on the low-level performance details.

A macro workload is a simulation of a real-world task. Macro workloads are sometimes created from real customer workloads, or may be designed to a specification, such as the Transaction Processing Performance Council's TPC-N specification. These large-scale workloads might include OLTP applications, decision support systems or reservation and inventory systems. (OSDL's macro workloads include the Database test suite—the subject of another paper at this conference.)

It is important that we do not confuse the results of macro and micro workloads, or attempt to extrapolate too much real-world behavior from micro measurements. Micro benchmarks are usually developer-focused and not very useful for understanding customer needs.

When looking at macro benchmarks, avoid confusing simulation with reality, and extrapolating results beyond the specific configuration and problem tested. Many macro benchmarks are grounded in real customer needs and situations, but some are designed more for marketing price/performance comparisons.

2.2.1 The AIM Suite

The AIM suite was created by AIM Technology in the 1980's. The AIM company sponsored a yearly 'Hot Iron' [DEC] award for hardware manufactures, with prizes awarded in various price/performance categories. To quote from a press release[HP]:

> Since 1981, AIM Technology has provided vendors and end users comprehensive, unbiased performance testing to help users determine the best fit between their application needs and available systems and configurations.. . . AIM is an independent organization, as opposed to a vendor consortium, which allows AIM to bring an expert eye to performance measurement, not restrained by objectives of consortium members.

The company no longer exists and the awards are no longer being given out. SCO acquired rights to the AIM technology in 2000, and placed suites VII and IX of the test under the GPL. From the SCO web page[SCO]:

> The AIM benchmark technology has proved useful for more than a decade in measuring performance of hardware and versions of the Unix operating system. The benchmarks were licensed by nearly all of the vendors of Unix system hardware. More than 70 companies used these benchmarks to compare and tune products. In addition, because of the stressful multidimensional nature of the AIM workload many OS and hardware vendors have used the benchmarks as part of their quality assurance process.

The AIM suite combines features of both macro and micro tests. The suite consists of a list of sub-tests, also known as "jobs." Each job exercises a specific area of system functionality, such as file I/O, shared memory, process creation, and compute-intensive math tasks. Each job consists of a C function which is linked to the driver code. A job may loop repeatedly internal to the C function. (For example, each addition test does 1.5 million internal loops.) Lists of jobs are grouped into a "workload," contained in a workload file.

The test runs in two modes. In the single-user mode (AIM suite IX), each job in the workload file is executed serially by the test driver. An alarm is set, and the job is executed repeatedly until the alarm expires. The alarm time is referred to as the "test period." Performance is calculated by multiplying the number of job executions by the internal job loop count, and dividing by the test period. Results reported are iterations per test period and operations per second for each job.

In the multiuser mode, the driver forks a number of subprocesses, giving each an identical list of jobs. The length of the job list is variable, the default is 100 jobs per child. The jobs are identical to the single user case. Each job is given a weight (the 'hit' count). This weight is used to calculate the fraction of the total work performed by each subtest, the total work is the sum of all job weights.

A typical test executions consists of a series of passes wherein the number of subprocesses is increased on each pass. Each subprocess runs a randomly-ordered set of jobs until its list is exhausted. The driver waits for all the child processes to exit, and records the time between child start and child exit. From this data two numbers are calculated—the jobs per minute (JPM) and jobs per child process per minute (JPM_C)[SGI].

As the system load increases the jobs per minute increases until it reaches a peak. If the number of child processes continues to increase, the work per child per minute begins to decrease. Depending on the command line options, the test run terminates when child work decreases below a threshold or the number of child processes reaches the maximum desired. Results reported are parent time, total child time, jobs per minute, and jobs per minute per child.

The AIM suite provides a set of building blocks (the sub-tests) that can be combined to create a simulation of a real-world workload. The old test has several examples of these workloads, including simulations of databases, file servers, and compute servers. The workload can be adjusted by altering test weight or changing the test mix.

3 Re-aim – AIM rework

3.1 The driver

The AIM code has remained untouched since 1991. I re-wrote the driver portion of the code so that I could understand it, maintain it and enhance the list of sub-tests. After studying the old code for a time, I choose to write a new driver, preserving as much of the functionality of the old driver as was useful. No doubt a different coder could have continued to maintain the existing structure, I choose not to.

The old driver used global data structures and static defines to control the size of the test list, the number of test arguments, and other details. The static definitions were replaced with dynamicly linked lists for flexibility. The AIM7 and AIM9 tests use almost the same list of tests, so a common driver was desirable.

For convenience, the GNU autoconf tools were used for the build and install system. The fol-

lowing parts of the old AIM framework are essentially unchanged:

- The method of statically linking test modules to the driver engine code, and calling those modules through a function pointer.
- The method for calculating workload task distribution and weighting.
- The method for calculating disk file size and distribution.
- The majority of the test module code (not changed at this time).
- The method of calculating the results is unchanged, however the timing method and location of timestamps relative to driver sleep() has.
- The adaptive timer remains the default.

The current driver has the following options, shown in Table 1 which may help explain usage.

Most of the parameters can be specified in a configuration file, options in this file are ignored if the command line option is present. Disk directories and disk file sizes must be specified in a configuration file.

Several things were noted while re-doing the driver.

- The old multi-user test ran until the jobs/child/minute was less than 1.0. This is quite a load on modern systems, resulting runs greater than eight hours to attain convergence. This length of a run is generally not useful for such a performance test so the default crossover is jobs/child/minute less than 10.0, with a second switch to set this to a quick test

Options	Description
-d(x), –debug(x)	Turns on debugging output, 1 is default
-v, –verbose	Produces more output
-s(x), –startusers	Number of users at start
-e(x), –endusers	Number of users at end
-i(x), –increment	Number to increment by
-f(s), –file(s)	Workfile name, (default 'workfile')
-l(s), –list(s)	Config file name, (default 'reaim.config')
-c, –crossover	Run to crossover, (JPM/user less than 10.0)
-q, –quick	Run to quick crossover, (JPM/user less than 60.0)
-x	Runs until max JPM detected
-j(x),–jobs(x)	Number of jobs in tasklist, (minimum is workfile size)
-m, –multiuser	AIM7 style, default
-t, –timeroff	No adaptive timer
-o, –oneuser	Runs AIM9 style single thread
-p, –period	Length for single thread
-r, –repeat	Iterate entire test
-h, –help	This message

Table 1: Re-aim Options

value of 60.0. On a 2-CPU 800MHZ Pentium III system, the quick test converges at 15→50 users, depending on workload. The default crossover point is 50→200 users depending on the workload.

- The jobs per child is now adjustable, with a default value of 100. This can be used to cause a set number of child processes to do more or less work without changing the workload.

- In a perfect world, all children (doing equal work) should receive equal favor from the scheduler. In reality, as the number of children exceeds the number of CPUs, unfairness occurs and the child exit is serialized. In addition, the child exit timing is collected serially by the parent using `wait()`. The maximum and minimum child exit times are recorded to reflect this. This variance also appears in the standard deviation calculated by keeping a running total across all child exits.

- Timestamps are collected with the `times()` function. The parent time figure is effectively wall clock time for the test. This function also allows us to extract the system and user time as seen by the child process. This information is reported as a running total. The child time thus exceeds the parent time in the report.

- Filesize and poolsize (see below) are now set in the configuration file. If either is specified in the workfile, that setting overrides the configuration file, maintaining old behavior.

- A method for detecting the maximum jobs per minute was added. When the `-x` option is used, the jobs per minute rate is tested by taking the standard deviation across the last five test iterations. If the standard deviation is less than 1.0 percent of the average, the test exits. In addition, if the the JPM rate drops more than 1.0 below the average, the test exits. Maximum jobs per minute are always reported.

3.2 Math tests

Time changes everything. Years ago, when computing was frequently referred to as "number-crunching," math performance was an exciting topic. Today, in the kernel context, when run single threaded, these math tests tell us very little. Fluctuations in the single-user (AIM9) integer math test times are undoubtedly due to non-math causes, and do not typically reflect a change in the kernel. The multiuser is a bit different—when we examine the multiuser case we see that all these test run entirely in user space. If we think of each subtest as a part of a larger workload, these user space functions are quite useful.

Table 2 shows typical parent times and child system and user times when running these tests on a 2-CPU system (Linux-2.4.18).

Test_Name	Parent	Child Sys	Child usr
add_short	5.96	0.00	1.18
add_double	15.71	0.00	3.13
add_float	10.58	0.00	2.09
add_int	16.90	0.00	3.37
add_long	16.93	0.00	3.38
mul_short	0.50	0.00	0.09
mul_long	0.42	0.00	0.07
mul_int	0.40	0.00	0.07
mul_float	17.43	0.00	3.48
mul_double	17.55	0.00	3.48
div_double	15.40	0.00	3.07
div_float	15.83	0.00	3.07
div_int	18.94	0.00	3.76
div_long	18.83	0.00	3.76
div_short	18.94	0.00	3.76

Table 2: Re-aim Math Tests

Number Forked	Parent Time	Jobs per Minute
Without math tests		
10	75.23	797.55
With math tests		
10	58.07	1064.23

Table 3: Database Load Comparisons

Number Forked	Parent Time	Jobs per Minute
Equal Weight		
10	39.17	1531.78
20	66.41	1806.96
Disk:math - 4:1		
10	57.38	1045.66
20	91.33	1313.92
Disk:math - 1:4		
10	26.53	2261.59
20	49.07	2445.49

Table 4: Effects of Test Weight

Adding these user space workloads to the multiuser test produces these results, shown in Table 3:

This appears a bit counter-intuitive—we have a longer test list, but it runs faster! Remember that the number of tests per child is constant (100 in this case). Adding the short user-space math tests to the workload actually decreases the amount of work per child. Here are some further examples of how changing a simple mix can change the run time. We'll start with four tests, equally weighted, then we will set the disk test weight to four times the weight for the math tests, then do the reverse. Results shown in Table 4:

There are fifteen of these math tests, all are tight loops. No changes in these tests are planned.

Num Forked	Parent Time	Child SysTime	Child UTime
10	23.70	7.64	4.10
20	26.29	15.30	8.27
30	29.02	23.02	12.30
40	31.55	31.48	16.38
50	35.91	39.54	20.33

Table 5: High System Time Load

3.3 Other Tests

The math tests are notable for consuming mostly user time. There is another list of tests that consume mostly system time. These tests include the various memory tests (brk, shared memory) and the various system call tests. (create/close, link, fork, exec.) Combining these tests into into a single workload does consume more system time, as seen in Table 5:

The current list of system-call focused test is a bit short. Repeated runs of various workloads have not yielded memory consumption at reasonable user levels.

Another current question involves the shell_rtn tests, which currently use the shortest possible shell script. In addition, the three functions calling the shell are identical. The reason for this duplication is unknown.

The intent is to examine other open sources of test routines for incorporation into this run framework.

3.4 Disk Tests

The disk tests in the old AIM test consist of three groups: basic block I/O tests, the same tests with an added sync, and the sync I/O tests. Each test determines file size from a global variable, `disk_iteration_count`. There are two configuration variables that control this, FILESIZE and POOLSIZE (speci-

fied in kilobytes or megabytes). If POOLSIZE is zero, each child will write or read a total of FILESIZE bytes. If POOLSIZE is non-zero, child file size is equal to FILESIZE + (POOLSIZE/number_of_children). Thus when POOLSIZE is non-zero, I/O per child will be reduced on each increase in child count.

For example, specifying a FILESIZE of 10K and a POOLSIZE of 100K will result in a single child creating a 110K byte file on each disk device listed. Two children will create a 60K file, etc. 24 children will create a 14K file, consuming 328KB per disk device.

The old AIM tests follow this sequence:

- `creat()` file
- write file
- `close()` file descriptor
- `open()` file descriptor
- do test

This results in the disk test running entirely from cache. I added a second set of disk tests using this method:

- `creat()` file
- write file
- `close()` file descriptor
- `sync()`
- `open()` file descriptor
- do test

This simple change noticeably impacted performance:

		Random Disk Writes
disk_rw	without sync()	21922 (1K) per second
disk_rw	with sync()	1218.78 (1K) per second

The first number is more indicative of real-world hardware performance, but the cache-only version of the tests may be of greater interest to kernel developers.

The third category of disk tests performs the same operations, but descriptors are opened with the O_SYNC flag. (The read-only test is not performed, of course.) This test is of lesser interest, due to the relative slowness of O_SYNC.

The current disk tests do all IO at 1K block sizes. Future improvements to the disk test suite include:

- Tests that use O_DIRECT and raw IO.
- Tests that use a common file created during the test setup or prior to the test run, requiring noticeable non-cached IO.
- Tests that produce measurable read activity, period. This is a weakness of the cache-intensive design of the current tests. Many test runs show little or no real read IO—files are created, read and destroyed too quickly.
- Tests that attempt to consume a noticeable percentage of the cache.
- Temporary file creation is currently serialized, multiple devices should work in parallel.

The final test is disk_src, which does a series of directory searches. This test is of interest due to its use of dcache. Future enhancements include creating a script which will allow other trees to be searched by disk_src, in place of the current fakeh.tar.

Run Time	Change
2 seconds	2.39%
4 seconds	2.17%
8 seconds	2.02%
15 seconds	1.52%
30 seconds	1.46%
45 seconds	1.62%

Table 6: Single user variation—3 runs each

3.5 Comparison of AIM9 Duration

This comparison attempts to show the useful duration for the single user (AIM9) test run. A proper duration should produce stable results from run to run. To test this, a single user test was run three times using a list of fifty-four tests. Average change between tests was compared across the three runs, as shown in Table 6. (Note: Each test must complete one full loop.) While the run-to-run performance does stabilize slightly when the test duration exceeds fifteen seconds, run-to-run stability does not improve noticeably beyond that point. This has been reflected in the choice of default settings for the single user run duration (10 seconds).

4 Run results

4.1 List of the workloads

Appendix A has a list of the various workloads with run times on several sample configurations.

4.2 Comparisons – 2.5

Table 7 is a quick comparison of a 2.5 patch set, which is a subset of one of Martin Bligh's trees. We can see by this quick comparison that the patch does improve performance. The test

Forks	JPM-mjb	JPM	delta
10	1167.50	1074.15	8.3%
20	1240.24	1219.13	1.7%
22	1252.72	1219.14	2.7%
100	1247.36	1203.68	3.6%

Table 7: Comparison of 2.5.68 and 2.5.68-mjb0.5

was run on a small 2-CPU system, with 1GB of physical memory and IDE disks.

5 Conclusions

I have described the work that has been done to change from AIM to Re-aim. I intend to spend a great deal more time adding to the list of test cases and otherwise improving the usefulness of the tests.

6 Availability

The Re-aim code is available from Sourceforge:

```
http://sourceforge.net/
   projects/re-aim-7
```

Or via BitKeeper:

```
bk://bk.osdl.org/aimrework
```

7 Trademarks

8 Acknowledgements

Thanks to Ruth Forester and John Hawkes for advice, and OSDL for support.

References

[SCO] Web page announcing AIM suite release, `http://www.caldera.com/developers/community/contrib/aim.html`, 2000.

[HP] Press Release with AIM description, `http://www.compaq.com.hk/press/release/99press/990623.html`, 1995.

[DEC] Press Release with AIM description, `http://wint.decsy.ru/alphaservers/digital/v0000022.htm`, 1995.

[SGI] Ruth Forester, et al. "Filesystem Performance and Scalability in Linux 2.4.17," *Proceedings of the 2002 USENIX Annual Technical Conference*, Berkeley, CA 2002 `http://oss.sgi.com/projects/xfs/papers/filesystem-perf-tm.pdf`.

A Re-aim Results

A.1 Example runs

This appendix shows how various workloads perform on some sample systems. Workloads were run until max sustainable jobs were reached. The results shown below are the maximum users obtained by each workload. Several iterations are shown in some cases to demonstrate typical run termination—the adaptive timer was used for these runs. See the source package for a listing of each workfile. Some of the workload have arbitrary names reflecting time. This is not intended as a hardware comparison.

We notice that for several of the workloads, scaling is roughly linear across the three configurations. For other workloads, most noticeable the fserver and Dbase, performance on the quad system jumps markedly. However, the adaptive timer skews the increment such that comparisons may not be relied upon—any true comparison should be made without the adaptive timer. (The adaptive timer was used in this case to reduce total run time.) The additional disks on the Quad system appear to impact run times. The other systems under test use disks which are shared by the system. (`/tmp` or `/usr/tmp`) The quad system has 5 spindles of disk devoted to the tests. The actual test report includes data on standard deviation and confidence levels. These columns have been removed, due to text formatting requirements.

The systems:

1. Single CPU
   PIII - 600MHZ
   384KB RAM
   single IDE disk
   Linux-2.5.68 -stock
   FILESIZE 10k
   POOLSIZE 100k

2. Dual CPU
   PIII - 868MHZ
   1GB RAM
   Dual IDE disk
   Linux-2.5.68 - stock
   FILESIZE 10k
   POOLSIZE 1m

3. Quad CPU
   PIII - 700MHZ

4GB RAM
5 SCSI disks
Linux-2.4.20 - stock
FILESIZE 10k
POOLSIZE 1m

A.2 The Workloads

workfile.all_utime Table 8. All these tests run entirely in user space.

workfile.alltests Table 9. The full test list.

workfile.compute Table 10. From the old test. Simulation of a compute-intensive server. 31.7% of this workload are tests from the all_utime list.

workfile.dbase Table 11 Simulation of a database load. 21.8% percent of this workload are tests from the all_utime list.

workfile.disk Table 12. The disk tests with no other work. All tests in this list are weighted equally. Notice the difference between this workload and the fserver workload, which includes other subtests.

workfile.fivesec Table 13 A completely artificial grouping of tests, based on their run duration when tested on a UP system.

workfile.fserver Table 14 Simulation of a file server. 21.8% of this mix is 100% user time tests, which matches the dbase workfile.

workfile.fivesec Table 15 A completely artificial grouping of tests, based on their run duration when tested on a UP system.

workfile.shared Table 16. Simulation of a multi-user shared server, assumed to be supporting telnet clients. 39.7% of the work mix are 100% user time tests.

workfile.short Table 17 A completely artificial

Max Jobs per minute				
Single - 1044.37 (1 user)				
Dual - 2938.27 (7 users)				
Quad - 4896.00 (12 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
5	29.32	0.00	29.32	1043.66
Dual				
14	29.27	0.01	56.72	2927.23
Quad				
20	25.04	0.03	100.04	4888.18

Table 8: All User Time Workload

Max Jobs per minute				
Single - 1839.22 (118 users)				
Dual - 4233.31 (345 users)				
Quad - 7207.78 (281 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
222	674.92	79.09	583.36	1835.42
Dual				
545	727.78	288.98	973.43	4223.53
Quad				
281	217.54	219.01	611.41	7207.78
343	317.68	494.41	750.89	6024.74

Table 9: All Tests Workload

grouping of tests, based on their run duration when tested on a UP system.

Max Jobs per minute				
Single - 803.29 (5 users)				
Dual - 1429.25 (7 users)				
Quad - 4708.68 (753 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
6	45.57	1.90	43.63	797.89
Dual				
10	43.35	3.73	78.56	1397.92
Quad				
753	969.10	246.37	3620.94	4708.68
1007	1300.27	348.33	4838.78	4693.19

Table 10: Compute Workload

Max Jobs per minute				
Single - 806.63				
Dual - 1186.52				
Quad - 1124.23				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
10	73.64	3.92	68.51	806.63
Dual				
53	265.33	38.30	457.17	1186.52
Quad				
383	2626.73	7089.40	2706.01	866.10

Table 11: Dbase Workload

Max Jobs per minute				
Single - 1059.20				
Dual - 2753.83				
Quad - 9723.69				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
52	309.29	19.06	11.38	1059.20
65	396.09	24.01	14.32	1033.86
Dual				
259	592.52	272.62	45.07	2753.83
349	816.90	370.08	61.38	2691.52
386	927.06	423.03	67.48	2623.13
464	1131.55	518.55	81.22	2583.36
Quad				
352	374.70	766.36	53.39	5918.33
510	330.43	899.31	76.65	9723.69
807	867.55	3087.03	119.11	5860.30

Table 12: Disk Workload

Max Jobs per minute				
Single - 2014.13				
Dual - 3872.86				
Quad - 10995.93				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
24	70.78	16.60	12.21	2014.13
32	101.06	22.16	16.34	1880.86
35	110.06	24.29	17.73	1888.97
41	130.73	28.48	20.89	1862.92
Dual				
136	208.59	158.24	51.78	3872.86
180	287.05	210.50	68.99	3724.79
198	319.77	231.98	75.75	3678.02
Quad				
432	265.98	698.35	170.78	9647.64
550	297.11	875.30	214.95	10995.93
799	1023.44	3577.29	314.54	4637.36

Table 13: FiveSec Workload

Max Jobs per minute				
Single - 1617.78				
Dual - 4267.88				
Quad - 149.06				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
17	63.68	9.74	37.24	1617.78
19	72.01	11.17	41.57	1598.94
23	89.66	13.41	50.25	1554.54
Dual				
328	465.73	241.80	483.01	4267.88
367	525.04	272.13	540.58	4235.91
449	639.48	339.93	661.62	4254.93
531	756.40	402.28	782.39	4254.18
Quad				
141	5732.32	4832.17	283.59	149.06
145	5968.47	4803.54	290.69	147.22
146	6112.96	5288.60	292.91	144.74

Table 14: Fserver Workload

Max Jobs per minute				
Single - 952.59				
Dual - 2945.31				
Quad - 5007.89				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
12	81.63	3.42	69.14	952.59
Dual				
187	411.42	57.24	730.43	2945.31
Quad				
48	62.11	12.72	231.40	5007.89

Table 15: Long Workload

Max Jobs per minute				
Single - 1177.14				
Dual - 2232.94				
Quad - 2153.06				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
12	59.33	5.09	51.69	1177.14
16	80.37	6.84	68.84	1158.64
Dual				
28	72.98	23.81	95.95	2232.94
34	103.61	26.71	116.53	1909.85
Quad				
132	520.91	386.16	436.88	1474.80
182	624.65	585.43	628.17	1695.73
291	786.61	1135.61	1002.55	2153.06
400	1409.01	3649.24	1380.71	1652.22

Table 16: Shared Workload

Max Jobs per minute				
Single - 45333.33				
Dual - 166909.09				
Quad - 222545.45				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
6	0.82	0.38	0.42	44780.49
Dual				
9	0.33	0.27	0.39	166909.09
Quad				
4	0.11	0.24	0.23	222545.45
8	0.26	0.47	0.45	188307.69

Table 17: Short Workload

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*