Benchmarks that Model Enterprise Workloads

Using macrobenchmarks to measure and improve Linux scalability for real-world applications

Vaijayanthimala Anand, Hubertus Franke, Hanna Linder, Shailabh Nagar,
Partha Narayanan, Rajan Ravindran, Theodore Ts'o
IBM Corp.
{manand,frankeh,hannal,nagar,partha,rajancr,tytso}@us.ibm.com

Abstract

In this paper we demonstrate the use of macrobenchmarks in Linux® kernel development. We describe two macrobenchmarks, SPECjAppServer2002™ benchmark application and IBM®'s Trade, which are based on the Java™ platform and modeling enterprise applications typically found in large data centers. This paper shows how these macrobenchmarks were used to analyse potential improvements in the load balancing and yield behaviour of the 2.5 kernel's O(1) CPU scheduler. We also demonstrate how the macrobenchmarks helped debug the 2.5 kernels and compare their performance improvements over the 2.4 series.

1 Introduction

1.1 Microbenchmarks vs. Macrobenchmarks

Performance is a key driver for Linux kernel development. Several patches have been developed explicitly to improve Linux performance on various architectures. Most patches which seek to add a new kernel feature are expected to show that they minimize, if not eliminate, any negative performance impact on the system.

Over the years, various benchmarks have become popular in the kernel development community to assess the performance of patches. Most of these microbenchmarks measure specific aspects of system performance, such as tiobench for filesystem performance[Tiobench] and pipetest[Pipetest] for event delivery. Microbenchmarks have two advantages. First, they are typically both easy to set up and run and are free, making them accessible to all developers. This is particularly important for the widely dispersed open source kernel community. Second, microbenchmarks can be pivotal in determining the impact of a patch on a specific kernel subsystem.

The specificity of a microbenchmark limits its suitability for predicting overall system impact. Hence, developers often use microbenchmark *suites* such as lmbench[lmbench] and Contest[Contest]. By running a collection of microbenchmarks, each stressing a different aspect of the kernel, a more accurate picture of the overall system impact can be obtained.

Microbenchmarks (singly or in suites) suffer from two major disadvantages. First, they do not adequately capture the dynamic interactions between different kernel control paths which may be impacted by the same patch. Even if these control paths are tested individually, their interactions will not be appar-

ent. Even if the microbenchmarks could be made to run together, the interactions being tested would be ad-hoc. Second, microbenchmark suites are less representative of real world workloads. As such, while they can be used to gain a better understanding of the impact of a patch on a single subsystem, they are not ideal.

Macrobenchmarks such as Trade[Trade] and SPECjAppServer2002[SJAS] help fill this void. They exercise different parts of the kernel during runtime in ways that are representative of real world workloads that run on Linux. Such benchmarks are designed to compare hardware and software differences based on performance and cost-performance criteria. However, they can also be used to guide software development because they permit an orderly isolation and elimination of system-wide performance bottlenecks. Macrobenchmarks also allow non-kernel bottlenecks to be identified, further encouraging an evolutionary approach to kernel development.

Macrobenchmarks have their disadvantages as well. They are often expensive to purchase and are not open source. They are not easy to set up and often require multiple machines with above average physical resource requirements especially memory and disks. They may also need proprietary middleware, such as databases and Web Application Servers (hereafter referred to as AS), if freely available open-source alternatives are not performant enough or do not have the right feature set yet to allow the benchmark to be run.

One notable effort to provide free macrobenchmarks is being done by the Open Source Development Lab(OSDL). The OSDL's Database Test (DBT) benchmark suite[DBT] development effort is a welcome step in reducing the need to purchase specialized middleware in order to run macrobenchmarks. The Scalable Test Platform, also from OSDL, helps make enterprise class hardware available to all developers, further easing the hurdles in using macrobenchmarks.

1.2 J2EE-based Macrobenchmarks

The Java 2 Platform, Enterprise Edition, J2EE [J2EE] framework is a mechanism for creating distributed and Java-based enterprise class applications for various business domains such as manufacturing, supply-chain management, and on-line financial applications. Compared to the traditional transaction processing benchmarks such as TPC-H, TPC-C and TPC-W, the J2EE framework has not received much attention in the Linux benchmarking efforts. For this paper, two J2EE based macrobenchmarks, Trade and SPECjAppServer2002, are used to investigate Linux kernel performance.

J2EE applications consist of multiple layers. Performance analysis of such applications are involved and demanding as they depend on many factors. A typical J2EE stack is illustrated in Figure 1.

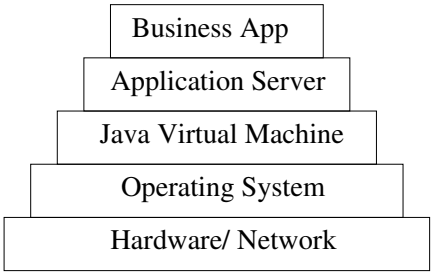

Figure 1: J2EE stack

The component which implements the actual application depends on the AS services. The AS in turn takes advantage of the underlying Java Virtual Machine (JVM) implementation. The Java applications call methods from the Java API libraries that provide access to the system resources through appropriate system calls.

The AS performance depends on many fac-

tors: caching support, transaction execution efficiency, JVM implementation, Enterprise Java Beans component pooling mechanisms, efficiency of persistent storage mechanisms, Java Database Connectivity, optimized driver support, etc. More information on J2EE best practices can be found in the literature such as Oracle9i and Java Performance [Oracle9i, EnterpriseJava, JavaPerf]. These studies focus on J2EE and its associated components rather than the operating system. By contrast, the focus in this paper is on the operating system; specifically, the Linux kernel. Two complex enterprise workloads are used to identify kernel performance issues and suggest possible kernel improvements.

1.3 Description of Trade

Trade [Trade] is a freely available benchmark developed by IBM. It is designed to measure AS performance. Trade is an end to end benchmark that models an online financial application. Specifically, an electronic stock brokerage providing web-based online securities trading.

Two versions of the Trade benchmark, Trade 2.7 and Trade 3.0, are used in this study. Trade 2.7 is a collection of Java classes, Java servlets, Java Server Pages (JSP), and Enterprise Java Beans integrated into a single application.

While Trade 2.7 is written based on J2EE 1.2, Trade 3.0 is the third generation of this benchmark making use of many features of J2EE 1.3 [J2EE] including local-interfaces, message driven beans, Container-Managed Relationship (CMR), etc. It also incorporates Web Services as one of its major enhancements. Many Application Servers in the industry implement these features.

This benchmark is used for performance research on a wide range of software components including the Linux operating system, AS, Java and more.

The Trade benchmark can be run in either a two tier or in a three tier configuration. In the two tier model, the client driver (which simulates clients of the online brokerage application) runs on one system, while the AS and the backend database runs on another. The AS executes J2EE applications which consist of two parts: the server side presentation logic and the server side business logic. In a three tier model, the AS and the backend databases run on separate systems, interconnected by a high speed network. We were more interested in the performance and scaling of the AS, so we chose to do our testing using a three tier configuration. Figure 2 shows such a configuration.

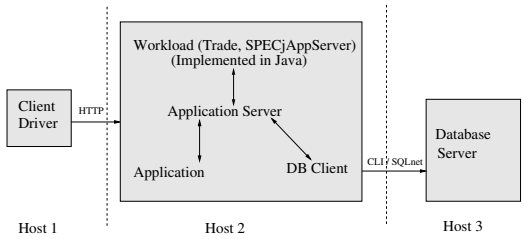

Figure 2: Three tier configuration for Trade 3.0 and SPECjAppServer2002

The client driver simulates requests of an online stock brokerage application, which makes a predefined mix of login, register, buy, sell, and quote requests of online securities. These requests come in as HTTP requests to the AS. Trade 3.0 has been configured to use the Enterprise Java Beans (EJB) mode meaning that all accesses to the back-end pass through the EJB container of the AS as opposed to the use of direct Java database connectivity (JDBC). All the orders are executed in a synchronous mode by the session and entity beans rather than being queued for asynchronus execution. The communication between the servlets and EJBs are done using the Remote Method Invocation (RMI) protocol. The backend database

stores 5000 users and 1000 securities applications. Database records are inserted, then modified as the benchmark progresses. To maintain reproducibility of the benchmark results, a database is initialized once and backed up. The database is restored before each test run.

The Trade application generates a large number of threads, of the order of 160, during its operation. The metric of interest in this benchmark is the number of web pages that are served by the AS.

1.4 Description of the SPECjAppServer2002 Benchmark

SPECjAppServer2002 [SJAS] (hereafter referred to as SJAS) is a benchmark for measuring the performance and scalability of J2EE (Java 2 Enterprise Edition) application servers and containers, by emulating the heavyweight manufacturing, supply chain management, and order/inventory system representative of a Fortune 500 company. It is a derivative of the ECperf 1.1 benchmark [ECperf]. SJAS supports multiple configurations such as single, dual, multiple, and distributed nodes. We chose dual mode (3-tier configuration) for our setup: (i) a client driver emulator, (ii) an AS tier and (iii) a database backend tier. This paper always refers to the 2002 version of SPECjAppServer.

SJAS models four logical business entities (domains): customer, manufacturing, supplier and service provider, and corporate. In the customer domain, large and small orders are distinguished in that they trigger different transactions (e.g., credit checks, order change). The manufacturing domain processes the different orders and schedules parts with suppliers. The supplier domain decides which supplier to use and handles the transaction (e.g., order size, due date) with the supplier. The corporate domain handles the list of all customers, parts, suppliers, and credit information. SJAS can be implemented either in a centralized or distributed mode. In this paper we chose the centralized mode, which allows us to put all four business entities on a single AS.

The throughput of SJAS is driven by the number of order entries in the customer domain and the manufacturing domain and is measured in TOPS, which is the average number of successful total operations per second completed during the measurement interval. TOPS is linearly related to the injection rate (IR). The IR refers to the rate at which business transaction requests from the order entry application in the customer domain are injected into the AS. The goal is to drive the injection rate as high as possible. An injection rate is sustainable if at least 90% of each type of business transactions completes within a required response time.

Though a full SJAS benchmark run requires more with respect to reporting [SJAS], we are using the sustainable injection rate as a means to evaluate scalability and changes to the kernel.

Note: SPECjAppServer2002 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2002 results or findings in this publication have not been reviewed or approved by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2002 is located at `http://www.spec.org/osg/jAppServer2002`.

1.5 Hardware Configuration

The Trade and SJAS macrobenchmarks are complex and require a fair amount of tuning for getting useful results. Combined with the multiplicity of issues being investigated, it was difficult to ensure that all results presented in

this paper came from the same hardware setup. Four different environments were used to collect the experimental data shown in later sections. These environments will be denoted hereafter as Configurations A, B, C, and D.

Configuration A consists of a 4-way 700 Mhz Pentium(tm) III, 1MB L2 Cache, and 4GB RAM for the AS and a 4 way 700 Mhz Pentium III, 1MB L2 Cache and 4GB RAM for the backend. A 2-way Pentium III 1GHz system was used to drive the workload.

Configuration B consists of a 4-way Pentium III 900 Mhz, 2 MB L2 Cache, 2.5GB RAM for the AS and a 4-way Pentium III 500 Mhz, 512 KB L2 Cache, 3.2 GB RAM for the backend. The client was a 2-way Pentium III 850 Mhz, 256KB L2 Cache, 2GB RAM system.

Configuration C differed from Configuration B only in doubling the number of processors in the AS tier. Thus, it has an 8-way Pentium III 900 Mhz, 2 MB L2 Cache, 2.5GB RAM for the AS, and a 4-way Pentium III 500 Mhz, 512 KB L2 Cache, 3.2 GB RAM for the backend. The client remained a 2-way Pentium III 850 Mhz, 256KB L2 Cache, 2GB RAM system.

Configuration D includes a 8-way Pentium III 900 Mhz, 2MB L2 Cache, 24GB RAM for the AS, and a 8-way Pentium III 700 Mhz, 1MB L2 cache, 8 GB RAM for the backend.

2 Kernel Bug Detection Using Macrobenchmarks

One benefit of complex macrobenchmarks is their ability to find bugs in the kernel that otherwise might not be found until the kernel is run on a large real-world system. During initial experiments with the SJAS benchmark, one such bug was found, fixed, and included in the 2.5.63 kernel.

The sole symptom was a complete system hang of the middle tier, with no oops or diagnostic of any kind produced. The hang could be reproduced by stopping and restarting the application server between five and ten times. The problem was traced using the NMI (Non maskable interrupt) watchdog timer and taking stack traces of all CPUs in the system.

The problem turned out to be threads deadlocking in the kernel. On any multiprocessor system, one task (say A) acquired a spinlock with interrupts disabled. Thereafter A performed an operation requiring all other processors to flush their Translation Lookaside Buffers (TLBs). To flush remote TLBs, task A would send an inter-processor interrupt (IPI) and go into a busy wait for an acknowledgement. However, if the tasks on the other CPUs were busy waiting on the same spinlock and also had their interrupts disabled, they would never receive the IPI, thus leading to a deadlock.

This issue was resolved by modifying the code to ensure that the spinlock was not held with interrupts disabled. The fix was included in the 2.5.63 kernel. Although the problem was easy to fix once the cause was determined, it took the right set of dynamic interactions, provided in this case by SJAS, to trigger the bug.

3 Comparing 2.4 and 2.5 Kernels

The project was initiated by using Trade 2.7 to test 2.4-based distribution kernels as well as then-current stock 2.5 kernels. Presented in Table 1 are the results of running Trade 2.7 in a three-tiered mode using configuration A.

The results obtained were unexpected. It was found that the 2.4 based distribution kernel (2.4-distro) performed better than the 2.5.59 stock kernel. To recheck the results, we ran the SJAS benchmark on a stock 2.4.20 kernel (2.4.20-stock) and compared results with

Kernel Version	#CPUs	TOPS	CPU Utilization (%)		
			user	system	idle
2.4.20-stock	4	Base1	68	14	17
2.5.59	4	Base1-3.8%	60	10	28
2.5.66	4	Base1+11.6%	70	12	16
2.4.20-stock	8	Base2	47	10	41
2.5.59	8	Base2+0%	36	6	56
2.5.66	8	Base2+24.0%	49	9	41

Table 2: Performance and middle tier CPU utilization of SJAS on 2.5.59 and 2.5.66 kernels using 2.4.20 as a baseline for 4-way (Base1) and 8-way (Base2) servers

Kernel	TOPS	%CPU Usage on AS
2.4-distro	Base1	87
2.5.59	Base1-14.4%	66
2.5.59+D7	Base1+2%	86

Table 1: Trade 2.7 results

processes runnable	context switches	CPU Utilization (%)		
		user	sys	idle
8	15689	74	18	8
12	18844	76	18	6
10	15778	71	21	8
11	16114	74	20	6
11	17629	74	17	8

Table 3: Output from vmstat for AS on a 2.4-x distro kernel using a 4-way server and Trade 2.7. Number of runnable processes are 2-3 times the number of processes.

the 2.5.59 kernel using Configurations C and D. The results, shown in Table 2, confirm that the 2.4.20-stock kernel exhibits better performance than 2.5.59 with the latter's TOPS decreasing by 3.8% on Configuration C and remaining unchanged in Configuration D.

At a later date, we also compared the performance on a 2.5.66 kernel and found that it performed significantly better than 2.4.20-stock with an *increase* in TOPS of 11.6% and 24.0% on Configurations C and D respectively. Table 2 shows that system time remained approximately the same for these two kernels though overall utilization was higher for 2.5.66. Isolating the performance changes between 2.5.59 and 2.5.66 is part of our future work. We felt our first task was to determine why the 2.5.59 kernel performed worse than the 2.4.20 and 2.4.20-distro, despite several scalability enhancements in 2.5.59.

Since distribution kernels have patches added on top of a 2.4 stock kernel, the profile data was analyzed in order to understand the observed differences. Comparing the vmstat outputs for a 2.4-x distro kernel (Table 3) to a 2.5.59 kernel (Table 4) we clearly see that the latter has fewer runnable processes in general and often has fewer runnable tasks than processors. Consequently, 2.5.59 shows higher idle times. The data initiated further investigation of the CPU scheduler behaviour.

In the next step, readprofile data was collected at a 60 second granularity during the steady run of Trade 2.7 on the same configuration as above. Comparing the data for the 2.4-distro kernel (Table 5) and 2.5.59 (Table 6), we see that `schedule()` is the costliest kernel function in both kernels.

The calls to `schedule()` drew our attention because they were still high on both lists even though 2.4-x uses the old scheduler and 2.5.59

runnable tasks	context switches	CPU Utilization (%)		
		user	sys	idle
3	12195	41	10	49
5	12079	41	9	50
7	17508	49	10	42
2	12087	41	9	50
3	11898	44	9	47

Table 4: Output from vmstat for AS on a 2.5.59 kernel running Trade 2.7 on a 4-way server. Number of runnable processes often dip below the number of processors and are low compared to the 2.4-x data shown earlier.

Ticks	Kernel function	Normalized Ticks
23969	**Total**	0.02
7071	default_idle	110.48
2388	schedule	1.52
822	csum_partial_copy...	3.31
799	send_sig_info	4.54
744	save_i387	1.29
511	tcp_v4_rcv	0.31

Table 5: Readprofile data for AS on a 2.4-distro kernel running Trade 2.7 on a 4-way server. Normalized ticks gives the number of ticks divided by the size of the function. `schedule()` figures are high though idle times are low compared to 2.5.59.

Ticks	Kernel function	Normalized Ticks
60332	**Total**	0.05
54048	default_idle	844.50
397	schedule	0.41
365	csum_partial_copy...	1.47
191	tcp_sendmsg	0.04
181	__kfree_skb	0.60
177	load_balance	0.19

Table 6: Readprofile data for AS on 2.5.59 running Trade 2.7 on a 4-way server. Normalized ticks gives the number of ticks divided by the size of the function. `schedule()` is costly despite the usage of the O(1) scheduler; also idle time is higher than in the 2.4-distro kernel.

uses the O(1) scheduler. To examine our hypothesis that the O(1) scheduler was causing the high idle times, we tested a 2.4.20 kernel with and without the O(1) scheduler using the same configuration as above. The results, not shown in this paper, were similar to the data shown earlier and confirmed the hypothesis. The 2.4.20 stock kernel produced 20% better throughput than the 2.4.20+O(1) scheduler. Further, 2.4.20+O(1) had fewer tasks in the run queue than the number of CPUs in the system and 40% idle time, similar to the results found in the 2.5.59 kernel.

Using snapshots of runqueue lengths in all CPUs at each timer tick, it was found that CPUs were going idle while there were runnable tasks on other runqueues. The imbalance in runqueue lengths across various CPU's while using O(1) led us to a careful examination of the load balancing logic of the O(1) scheduler. The analysis is discussed in the next section.

4 Load Balancing

Before discussing the results of various experiments, we revisit the load balancing differences between the old 2.4 scheduler and O(1). The 2.4 scheduler uses a single runqueue for all CPUs which leads to high lock contention and lock hold times when the number of tasks and CPUs start increasing. O(1) replaces the single runqueue with per-CPU runqueues. While choosing the next task to run on a CPU, it does not look at remote runqueues, maintaining the O(1) behaviour and preserving cache affinity. Consequently, it needs to explicitly balance the load on each runqueue by calling a `load_balance()` function. Workloads which are sensitive to load imbalance, such as Trade and SJAS, get affected by the effectiveness of `load_balance()`. In 2.5.59, `load_balance()` is called periodically on each CPU, with the frequency of invocation determined by the idleness of the CPU.

To improve the load balancing behaviour of the O(1) scheduler, we tried a series of patches from Ingo Molnar's D7 patch [D7-PATCH] to Andrea Arcangeli's try_to_wake_up patches (included within his O(1) patch [AA-O1-PATCH]) to a *find_busiest_queue* patch, created in-house [FBQ].

4.1 D7 patch

Ingo Molnar's D7 patch unconditionally migrates a task from a remote to the current runqueue if the current CPU is about to go idle. Table 1 shows that this patch helps 2.5.59 perform 2% better than 2.4-distro for Trade 2.7, more than making up for the 14.4% performance loss seen by stock 2.5.59. For an SJAS workload, the same patch helps 2.5.59 draw on a par with the 2.4.20 stock kernel, overcoming the 3.8% degradation seen by 2.5.59 versus 2.4.20 (Table 7). The 10% degradation of 2.4.20+O(1) compared to 2.4.20 in the same table reconfirm the earlier hypothesis that the O(1) scheduler is at least partially responsible for the performance loss of 2.5.59 compared to 2.4.20.

Kernel level	CPU Usage	% TOPS improved
2.4.20-stock	82%	baseline
2.4.20+O(1)	66%	-10.6%
2.5.59-stock	70%	-3.8%
2.5.59+D7	64%	no change

Table 7: SPECjAppServer2002 - v1.14, 4-way results on 2.5.59

Kernel level	CPU Usage	%TOPS improved
2.5.66-stock	82%	baseline
2.5.66+trytowakeup1	83%	+4.3%
2.5.66+trytowakeup2	89%	+0%
2.5.66+busiestqueue	82%	-4.3%

Table 8: SPECjAppServer2002 - v1.14 4-way results on 2.5.66

4.2 Load Balancing on Task Wakeup

The O(1) scheduler used in Andrea Arcangeli's 2.4 kernel tree contains two changes to do load balancing on task wakeup events in addition to the periodic invocations of `load_balance()` in the stock kernel's O(1). We implemented these changes as two separate patches for the 2.5.66 kernel.

The first patch, henceforth called try-

Kernel level	CPU Usage	%TOPS improved
2.5.66-stock	56%	baseline
2.5.66+trytowakeup1	60%	+4.4%
2.5.66+trytowakeup2	72%	+3.0%
2.5.66+busiestqueue	65%	+5.2%

Table 9: SPECjAppServer2002 - v1.14 8-way results on 2.5.66

towakeup1, modifies the `try_to_wake_up()` function to invoke load balancing whenever a task is being woken up. Using the task wakeup event as a load balancing trigger was also motivated by the high count for calls to `tcp_data_wait()`; the high count causes task wakeups in the profiling data similar to the one shown in Table 6 for 2.5.59. The trytowakeup1 patch improved SJAS throughput performance by 4.7% on Configuration C compared to the 2.5.66 stock kernel, as shown by Table 8. Configuration D showed a similar 4.3% improvement as seen in Table 9.

The second patch, henceforth called trytowakeup2, tries to explicitly address the problem of CPUs going idle by trying to place the task being woken up onto an idle CPU if possible. This is in contrast to trytowakeup1 which is only concerned with pulling tasks to the runqueue of the waker. While trytowakeup2 increases SJAS performance by 4.5% in Configuration D (Table 9), it has no effect in Configuration C(Table 8). The behaviour can be explained by the relative lack of idle CPUs in Configuration C (4-way AS) compared to Configuration D.

The final patch called busiestqueue [FBQ], aimed at improving the aggresiveness of the existing load balance function itself rather than changing the frequency or location of its invocation. In the normal O(1) scheduler, the `find_busiest_queue()` function is used by `load_balance()` to find the remote runqueue with the maximum number of runnable tasks from which tasks can be pulled to the current runqueue. The `load_balance()` code checks whether tasks on the remote runqueue are suitable for migration but if none are found suitable, it does not try to find another runqueue and try again. The busiestqueue patch remedies this behaviour by modifying `find_busiest_queue` and its invocation by `load_balance()` to ensure that all remote runqueues are examined for tasks to migrate. The results from using the patch are mixed. Configuration C shows a performance degradation of 5.1%(Table 8) whereas Configuration D shows a 2.4% improvement(Table 9). Evidently, the patch is too aggressive and the extra cycles spent in trying to find another remote runqueue prove too costly. We are in the process of finetuning the patch by limiting the number of iterations in the search for a busy queue.

The trytowakeup1 and busiestqueue patches increased performance by around 5% on the 8-way Configuration D when applied individually and in combination (data not shown). This suggests that one or the other approach is sufficient in achieving better load balancing and leads to the question of which one should be used. The answer will lie in the effect of the patches on other workloads and is part of our future work.

5 Yielding to Other Tasks

The system call `sys_sched_yield()` causes the calling task to yield execution to another task that is ready to run on the current processor. Multi-threaded applications often use `sys_sched_yield()` to improve interactive response or to improve the performance of the system by letting the scheduler use the processor resources more effectively. This is particularly true if the applications use traditional userspace locks (not based on futexes).

However, the benefits realized from the use of `sys_sched_yield()` are heavily dependent on the implementation of the system call. The CPU scheduler selects the next task to run and determines how long the yielding task will remain on the runqueue before getting a chance to run again. The following implementations

of `sys_sched_yield()` are feasible:

PA The yielding process is queued right after the next task on the same priority queue. Effectively, it yields only to the next task at the same priority level.

PB The yielding process is queued at the tail of its priority queue making it yield to all runnable tasks at the same priority level.

PC The yielding process is moved to the priority queue on the expired list effectively making it yield to all runnable tasks in the system (as the expired list becomes the active list only after all runnable tasks have exhausted their timeslices).

The yielding task rarely knows how long it needs to yield before it can attempt to acquire a shared resource again as the availability of the shared resource depends on external events and progress made by competing tasks. For instance, an interactive application might see reduced response time if policy PA were used. But a task polling for a shared resource such as a userspace spinlock, might benefit from PB or PC which allows the task holding the resource to run and potentially release it for use by the yielding task. As the CPU scheduler is unaware of the task's rationale for using `sys_sched_yield()`, it cannot decide the best yielding interval either. Hence different Linux distributions have tried all three policies.

To understand the impact of these policies on macrobenchmarks such as Trade and SPECjAppServer2002, we collected profile data to see the number of `sys_sched_yield()` calls issued. Table 10 shows that `sys_sched_yield()` accounts for almost one third of all calls to `schedule()` when Trade 2.7 is run on Configuration A.

Table 11 shows the data collected by instrumenting the `sys_sched_yield()` for a 1 minute run of Trade 2.7 on 2.4.20 using Configuration A. In the table, *higher, lower* and *same* refer to the number of `sys_sched_yield()` calls in which there was at least one task on a higher, lower and same priority level as the yielding task. The row labelled *only* counts the number of `sys_sched_yield()` calls in which the yielding task was the only one on its runqueue. We see that most tasks in the system are on the same priority queue as the yielding task. Hence, the policy adopted by `sys_sched_yield()` is likely to have a significant impact on performance.

Ticks	Kernel Functions
6826403	**Total**
2523245	sys_sched_yield+11d
2236660	cpu_idle+3e
1312369	schedule_timeout+9d
327747	schedule_timeout+184

Table 10: Functions calling `schedule()` for a 2 minute run of Trade 2.7 on Configuration A

The 2.5.65 stock kernel uses the PC policy. We implemented the other two policies, PA and PB and compared their performance with PA using Trade 3 in Configuration D. PB and PC turned out to have the same results for the benchmark which follows from Table 11. As there are very few tasks on lower priority levels when `sys_sched_yield()` is called, PB and PC are effectively the same policy. Hence only PA and PC results are shown in Table 14. We see that the pages per second (pg/s) drops by 32.6% if PA is used instead of the default PC policy. CPU usage (usg) and efficiency (effncy) also see a corresponding drop. Similar results are seen for SJAS (not shown). Using PA instead of PC decreases TOPS by 10% on a 4-way.

The reasons become clear from the vmstat outputs of PA and PC shown in Tables 12 and 13 respectively. The number of context switches

Relative Priority	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
Same	145103	157055	163064	156379	162783	161733	167366	177876
Only	117653	112196	112387	105653	101420	96053	108830	92293
Higher	26	34	28	29	31	25	33	36
Lower	929	937	1000	1073	1036	1016	1156	1132
Total	263711	270222	276479	263134	265270	258827	277385	271337

Table 11: sys_sched_yield call count and the distribution of tasks on priority queues relative to the yielding task, using Trade 2.7 on 2.4.20 in Configuration A. The data indicates that most tasks in the system were on the same priority queue as the yielding task.

increases almost fourfold (from approximately 6700 to 27000) when PA is used. As PA causes the yielding process to get scheduled much sooner, the shared resource it waits on is generally not available, thus leading to frequent context switches as it yields again and again. For such an application, the default policy of letting all other runnable tasks run once is a good choice.

The kernel development community has been discussing alternative policies for `sys_sched_yield()` in order to improve response time for interactive applications. The results shown here indicate that such changes might adversely affect macrobenchmarks like Trade. However, this is only true until application servers start using the new fast user-level mutex (futex) feature provided by the 2.5 kernels.

6 Conclusions and Future Work

In this paper, we have examined two macrobenchmarks, Trade and SPECjAppServer2002. Both benchmarks model complex workloads utilizing the J2EE framework, which are popular in many enterprise data centers. We have shown a case study of a kernel bug that was triggered by these benchmarks and which would have been hard to find otherwise.

procs		system		cpu	
r	in	**cs**	us	sy	id
14	6067	27204	63	10	27
9	5868	29230	60	9	31
12	5337	24765	61	8	30
10	6021	27753	61	9	30
5	5947	27496	64	10	25

Table 12: vmstat output collected while using Policy A showing high context switches and high idle times. *r*, *in*, and *cs* refer to the number of runnable tasks, interrupts, context switches respectively, while *us*, *sy*, and *id* refer to the percentage of time spent by CPUs in user mode, system mode, and idling respectively.

procs		system		cpu	
r	in	**cs**	us	sy	id
18	7788	6903	85	14	0
26	7168	6686	84	11	6
24	8010	6798	87	12	1
23	8083	6727	87	13	0
22	7934	6212	87	13	1

Table 13: vmstat output collected for Trade 3 running on Configuration D, while policy PC is used to implement `sys_sched_yield()`. The other labels are explained in the caption for Table 12. Context switches and idle time are significantly lower for PC compared to PA.

Kernel	Policy	Pg/s	Usg	Effncy
2.5.65	PC	Baseline	95%	100%
2.5.65	PA	-32.6%	75%	85%

Table 14: Comparison of `sys_sched_yield()` implementations using Trade 3 on Configuration D.

The macrobenchmarks were also used to reveal deficiencies in the load balancing logic of the 2.5 kernel's O(1) CPU scheduler. Various patches were used to increase the aggressiveness of load balancing and reduce the probability of CPUs going idle despite the presence of runnable tasks in the system. Based on our observations, we suggest the following four load balancing policies might be of help for workloads sensitive to load imbalance such as Trade and SJAS:

- Load balance during initial placement of tasks by choosing the idle processor
- Load balance during wakeup by choosing the idle processor
- Load balance the queues aggressively (similar to patches described above) when a processor goes idle
- Consider providing aggressive load balancing through a configuration option

More patches will be produced to implement the above catagory of improvement and the investigation will continue to find a fair load balancer to improve these workloads for SMP (Symmetrical Multi Processor) and NUMA (Non Uniform Memory Access) systems. Any load balancing patches proposed will need to be tested using different workloads to make sure that they do not degrade performance by unnecessary balancing.

The final part of this paper examined different implementations of the `sys_sched_yield()` call and concluded that the existing 2.5.65 implementation performed well for macrobenchmarks such as Trade and SJAS.

There is still much work to be done in exploring how the kernel can more efficiently support J2EE-based workloads. As we have seen, these workloads tend to be very sensitive to scheduler issues, and changes which benefit one workload may actually cause harm to other workloads.

Further tuning of the application and improvements in the Linux kernel has improved the CPU utilization of these benchmarks. Hence, while initial attempts to use spinlock metering to find lock contention was not fruitful, we anticipate that future work in improving the benchmark score of these workloads will include finding and fixing lock contention problems.

We have used, and are continuing to use, macrobenchmarks as a method for finding potential areas for improvement in the Linux 2.5 kernel, especially as it relates to the Linux scheduler. We hope we have demonstrated to the reader that more complex benchmarks are a useful tool for the kernel developer interested in improving the performance and scalability of the Linux kernel.

7 Acknowledgments

The authors would like to thank the following people for their help: Jianwen Alex Feng, Bill Hartner, Wilfred Jamison, Sandra Johnson, and Rick Lindsley.

8 Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

References

[AA-O1-PATCH] Andrea Arcangeli, `http://www.kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/2.4.20rcaa1`

[FBQ] Rick Lindsley, `http://www.ibm.com/linux/ltc/patches/?patch_id=865`

[Contest] Con Kolivas, `http://members.optusnet.com.au/ckolivas/contest/`

[D7-PATCH] Ingo Molnar, `http://people.redhat.com/mingo/O(1)-scheduler/sched-2.5.59-D7`

[DBT] Open Source Development Lab, `http://sourceforge.net/projects/osdldbt`

[J2EE] Java 2 Platform, Enterprise Edition `http://java.sun.com/j2ee/overview.html`

[ECperf] `http://java.sun.com/j2ee/ecperf/`

[EnterpriseJava] Kingsum Chow, Ricardo Morin, and Kumar Shiv, *Enterprise Java Performance: Best Practices*, Intel Technology Journal, Vol 07 (Feb 2003) p. 32–48

[JavaPerf] Java Performance Tuning, `http://www.javaperformancetuning.com`

[lmbench] Carl Staelin and Larry McVoy, `http://sourceforge.net/projects/lmbench`

[Oracle9i] Oracle9i Application Server, *Architecting for J2EE performance*, `http://otn.oracle.com/products/ias/pdf/ArchitectingForJ2EEPerformance.pdf`

[Pipetest] Ben LaHaise, `http://www.kvack.org/~blah/ols2002.tar.gz`

[PMQS] Hubertus Franke, Shailabh Nagar, Mike Kravetz, Rajan Ravindran, *PMQS:Scalable Linux Scheduling for High End Servers, ALS'01*, Annual Linux Symposium, Oakland, 11/2001

[SJAS] Standard Performance Evaluation Corp., `http://www.specbench.org/jAppServer2002/`

[Tiobench] Mika Kuoppala, `http://sourceforge.net/projects/tiobench`

[Trade] Application Server Benchmark, `http://www-3.ibm.com/software/webservers/appserv/benchmark3.html`

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.*