A middleware performance characterization of Linux using IBM WebSphere Application Server

V. K. Anand
W. C. Jamison

As an open-source operating system, Linux™ made an impressive mark in the history of computing when interest from the corporate IT industry soared toward the end of 1990s. Linux, however, still faces some challenges in the areas of high performance, especially on symmetrical multiprocessor systems. This paper describes an initiative to investigate Linux performance using IBM WebSphere® Application Server software. The primary goal was to study how Linux performance and scalability could be improved by applying some of the new enhancements put into the kernel as well as by fine-tuning the middleware. We describe the results of this investigation and explain in detail how issues were resolved through our collaboration with the open-source community as well as within the IBM product teams.

Linux** has come a long way since it was first introduced in 1991. It is undoubtedly one of the most successful open-source programs in the market today. Two of its strong points are its cost-effectiveness and its availability on many hardware platforms, including powerful workstations and mainframes. At least four factors brought Linux to its current status. First, its open nature encouraged talented people from all over the world to collaborate in its development and maintenance with the goal of continually improving the software. Second, the emergence of vendors and distributors such as Red Hat, Caldera, Debian**, SUSE, and others provided the software support customers look for in a product. Third, Linux gained the confidence of big software companies such as IBM, Hewlett-Packard, and Sun Microsystems, which in turn promoted the operating system and encouraged its adoption. Finally, as more and more customers invested in Linux, software development companies and independent software vendors became more engaged in porting their products to this platform.

Linux now faces even greater challenges, as businesses begin to look at their Linux machines as the next generation of enterprise servers. This creates

high expectations for Linux to perform well, especially in the context of Internet-based services such as Web servers and application servers. Most companies today deploy networks of symmetrical multiprocessor (SMP) systems for their IT infrastructure. Originally, Linux was written for single processor systems; modern Linux kernels now support SMP systems, but traditionally have had scalability problems with them.

LINUX AND IBM MIDDLEWARE

IBM has a strong presence in the middleware market. For example, it holds the biggest market share for application servers today. As a Linux advocate, IBM views its middleware performance on Linux throughout all of its eServer* platforms as critical to the success of its Linux strategy. For this reason, a special work group within IBM was formed to specifically characterize the performance of Linux using IBM WebSphere* Application Server Version 5. The mission of the work group was to investigate the special characteristics of WebSphere Application Server and how the new enhancements to the Linux kernel could help improve its performance and scalability on SMP systems. Throughput and response times are the key metrics for performance.

WebSphere Application Server is a Java** 2 Enterprise Edition (J2EE**) server, and IBM's primary platform for e-business. Many other IBM products run on it, such as IBM WebSphere Commerce Suite, IBM WebSphere Portal Server and IBM Content Manager. Because it is implemented almost entirely in Java, it runs on many different hardware and operating-system platforms, including various Linux distributions.

OVERVIEW

This paper is not a formal performance report. Instead, our focus is on describing the effects of applying some of the major enhancements in the Linux kernel to the overall performance and scalability of WebSphere Application Server Version 5. We describe relative improvements from a known baseline in terms of percentages. When a negative percentage is obtained, we describe how we resolved the problem to gain a positive improvement. Although we performed the tests on several platforms, our discussions are based on the 32-bit Intel Architecture** (IA32**) unless mentioned otherwise.

In the following sections, we first introduce the work group that performed the study and provide information about the benchmark applications used. After that, we discuss the Linux enhancements that we applied. We also cover some issues and improvements that are not Linux-specific but helped boost performance, and which required collaboration with other IBM product teams. To get a sense of how Linux was performing relative to other operating systems, we ran the same tests on exactly the same machine with a different operating system. In the case of IA32, we used Microsoft Windows** 2000 and 2003 servers.

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

PERFORMANCE EVALUATION WORK GROUP

Our work group was composed of IBM teams from various organizations. Its primary goals were to make sure that we uncovered issues with Linux on WebSphere Application Server and to understand how we could achieve the best performance by applying enhancements and fixes to the Linux kernel and by fine-tuning parameters ("knobs") in the software stack. This necessitated collaborating with various groups, especially with the Linux open-source community, when problems and issues were discovered. We performed our benchmarking on the IA32, PowerPC*, and S/390* platforms.

We believe that the best test for Linux performance is an end-to-end "macro benchmarking" using applications that run on middleware, such as an application server. This is because the real performance landscape is only seen by customers when their applications are actually running on top of the underlying infrastructure.

The members of this work group included the IBM teams from WebSphere Application Server Performance, the Linux Technology Center (LTC), the Java Technology Center (JTC), DB2* Performance, and the various performance teams from the pSeries*, iSeries*, and zSeries* platforms.[1] LTC was our main liaison to the Linux open-source community. The work group remains active to this date and is continuing its studies on newer versions of WebSphere Application Server and Linux kernels.

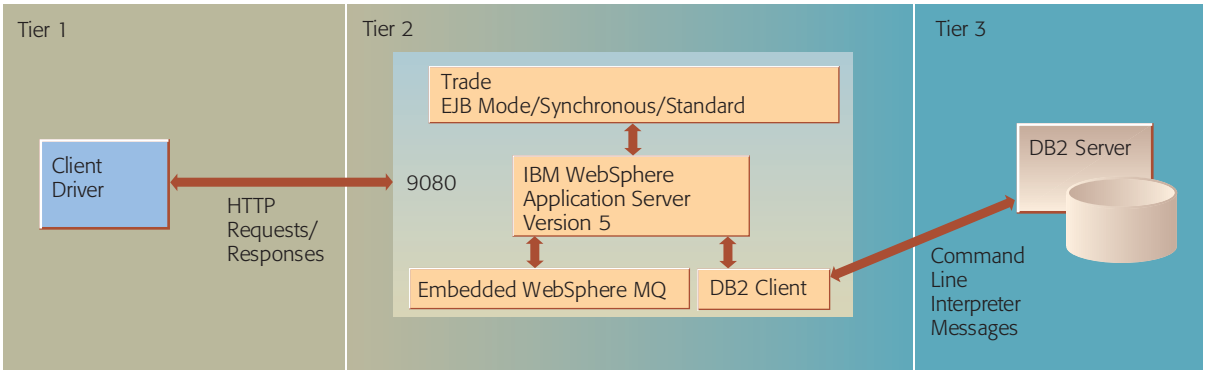

Figure 1
Three-tier configuration for Trade application

THE BENCHMARK APPLICATIONS

We chose two benchmark applications to test WebSphere Application Server Version 5 on different kernel levels of Linux. Each application stresses different parts of the application server.

Trade application

The Trade benchmark application was written by the WebSphere Application Server performance team for its own performance work. The application models an electronic stock brokerage firm that provides Web-based online securities trading. There are several versions of Trade, depending on the J2EE version supported by the application server. In our study, we used Trade Version 2.7 and Version 3, which are based on J2EE 1.2 and J2EE 1.3, respectively. For more information about J2EE, see Reference 2. More information about Trade is available at Reference 3.

Configuration

Figure 1 is a simple diagram of the 3-tier Trade configuration adopted by the work group. The client machine sends HTTP (HyperText Transport Protocol) requests directly to WebSphere Application Server through port 9080. The minimum requirement for the benchmark is to use Trade's EJB** (Enterprise JavaBeans**) runtime mode, where all access to the database uses the EJB technology, thereby exercising the container-managed persistence component of WebSphere Application Server more heavily. The order-processing mode is set to synchronous, which means that all buy and sell orders are completed immediately when the request is issued, removing the need for queuing messages. The access mode we used is standard, in which all communications between servers and EJBs are performed using the Java Remote Method Invocation (RMI) protocol. The scenario workload mix, which provides an equal distribution of Trade operations such as login, register, quotes, and buy, is also standard. For the Web interface, simple JSPs** (JavaServer Pages**) are used.

Run procedures

The Trade database on the DB2 server was populated initially with 5000 users and 1000 quotes. As the benchmark executed, the database was modified by updating and inserting records. In order to maintain consistency between runs, we kept a master copy of the original populated database and restored it for every new run. For every test run, the application server was restarted. The database was restored and the desired Trade configurations were reset. A warm-up run consisted of the following workload (expressed in terms of number of concurrent users and total number of requests submitted to the system) and executed in this sequence: one user, 1000 total requests; two users, 1000 total requests; five users, 1000 total requests; ten users, 1000 total requests; 25 users, 5000 total requests; 50 users, 5000 total requests; and 100 users, 5000 total requests. In a real environment, users spend some amount of "pause" time after requesting a page; for example, reading the contents or making decisions. In performance terminology, this is called "think time." In our experiments, there is no think time, which means that after a requested page is received, the next request is immediately sent. This is also equivalent to simulating more users than the actual number of users in the system. In effect, the

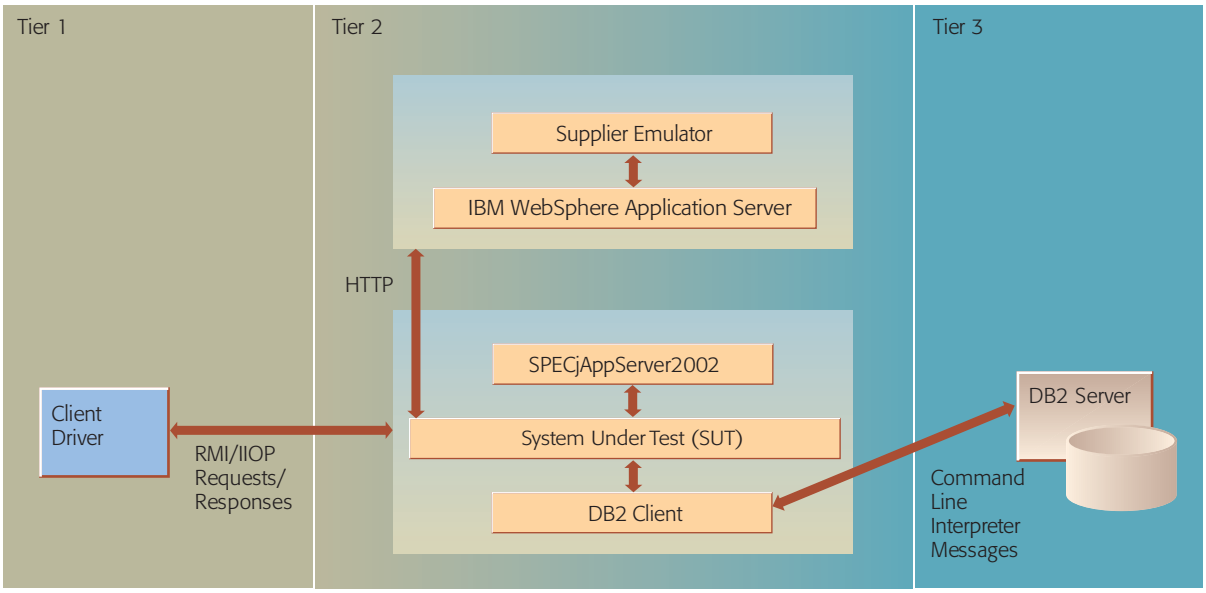

Figure 2
Three-tier configuration for SPECjAppServer2002

application server is stressed much more than in a real environment, given the same number of users.

The client driver used is WebSphere Performance Tool (WPT), which is available on the alphaWorks* Web site.[4] For a given test scenario, there are at least three test runs. The purpose is to ensure repeatability (defined by a tolerable margin of 5 percent variability). The final result is the average of these three runs. All test runs were executed with no think time.

SPECjAppServer2002 application

The other benchmark application is SPECjAppServer2002**, which is a public benchmark for enterprise Java applications based on J2EE Version 1.3. It is available from the Standard Performance Evaluation Corporation (SPEC).[5] The application emulates a heavyweight manufacturing, supply-chain management, and order/inventory system. Unlike Trade, the client does not use HTTP to send requests. Instead, Java clients communicate with the application through Remote Method Invocation/Internet Inter-Orb Protocol (RMI/IIOP), which consequently stresses the EJB container as well as the underlying Object Request Broker (ORB) layer of the application server. The EJB container is where the EJBs are executed; whereas, the ORB serves as the communication channel between the EJB container and the Java clients. Thus, SPECjAppServer2002 complements Trade's heavy Web container workload but light workload on the EJB container. For more information on SPECjAppServer2002, see Reference 6.

The SPECjAppServer2002 results or findings in this paper have not been reviewed or approved by SPEC; therefore, no comparisons or performance inferences can be made against any published SPEC results.

Configuration

The three-tier configuration for SPECjAppServer2002, also called the dual-node configuration by SPEC, is shown in ***Figure 2***. The client driver is a stand-alone Java program that sends order entry requests to the application server, which in turn interacts with the back-end DB2 database. The supplier emulator services the manufacturing orders sent by the application server (also called the system under test or SUT) when it needs inventory to service the order requests from the client.

In summary, the SPECjAppServer2002 benchmark consists of order and manufacturing applications. The throughput of this benchmark is directly related to the load (injection rate) used by these applications. Hence, the injection rate needs to be increased to scale up the throughput. The metric TOPS (total

operations per second) is the average number of successful operations completed during the measurement interval.

Run procedures

The run rules prescribed by the spec.org Web site are not strictly followed by the work group, as these results are for internal IBM use only and used under SPEC's research clause. However, the work group has certain run rules that were followed to collect these results. For a given injection rate, the SPECjAppServer2002 database is created, populated, and archived to be restored for each run using that injection rate. During execution, the database is scaled, but by a stepwise function, and does not grow linearly in size. Because the database is modified for each run, a new database needs to be loaded; for this purpose, the created database is saved.

For every run, the database is restored and the SUT and the supplier emulator are restarted before the benchmark is commenced. There is a warm-up time of 300 seconds, a steady runtime of 900 seconds, and a cool-down time of 150 seconds. This is half the time that was required by spec.org. We verified that reducing this time value does not affect the overall results of the run. The criteria for a successful run mandate that the response time of new order and manufacturing transactions are below a certain range and that there is a good mix of large and small order transactions.

LINUX ENHANCEMENTS, ISSUES, AND SOLUTIONS

Versions 2.4, 2.5 and 2.6 of the Linux kernel, which are referred to in this paper, are the development kernels available from kernel.org.[7] Linux distribution vendors usually choose a particular version of development kernel from kernel.org and apply additional patches that they consider important to derive the distribution kernels. Therefore, there is quite a bit of variation among the different distribution kernels and the kernel.org development kernels. This study uses the Red Hat and SUSE versions of the Linux distribution kernels.

The O(1) scheduler

Version 2.4 of the Linux development kernel has a scheduler that is not scalable on SMP systems.[8] This is because all runnable tasks are linked in a single run queue. Extreme contention results from all processors trying to access this one queue, and this introduces increased scheduling latency as the number of processors and/or tasks increase. Also, this single run queue is prioritized (sorted) based on the *goodness value* of a task. The goodness value is calculated from the task's priority, the amount of CPU it uses and other factors. It is computed for all tasks every time one needs to be dispatched to a CPU.

This scalability problem is solved by the introduction of the O(1) scheduler[9] in the Linux Version 2.5 development kernel. This new scheduler implements a run queue for each CPU promoting local CPU scheduling. The name O(1) ("order one") was given to indicate that this scheduler takes approximately the same amount of time irrespective of the current number of tasks and processors in the system. This is achieved by decomposing each CPU run queue into a number of "buckets" in priority order and then using a bitmap to identify the buckets that have runnable tasks. Because the number of supported priority levels is constant, the scheduler always takes an equal amount of time to select a task for scheduling. The O(1) scheduler was originally written and is being maintained by Ingo Molnar of Red Hat. This scheduler has also been accepted into the Linux Version 2.6 development "mainline" kernel.

Strong affinity is another feature that was introduced in the O(1) scheduler. This means that a task stays on the same CPU every time it is scheduled. Periodically, load balancing is performed on the run queues of the processors so that a CPU will not sit idle while another CPU has several runnable tasks to be scheduled.

Versions 2.5 and 2.6 of the kernel and some of the Linux 2.4-based distributions include the O(1) scheduler.[10] The LTC kernel performance team observed significant improvements when the O(1) scheduler was used on well-known benchmarks such as VolanoMark,[11] SPECWeb99,[12] netperf,[13] and so forth. Thus, we expected that using the O(1) scheduler would also boost performance in our middleware tests.

Load-balancing problem

When we tested Trade Version 2.7 on both Red Hat 7.2 and Red Hat Advanced Server 2.1 (RH AS 2.1), we were surprised to see a 20 percent throughput degradation on the latter. Red Hat 7.2 uses the old

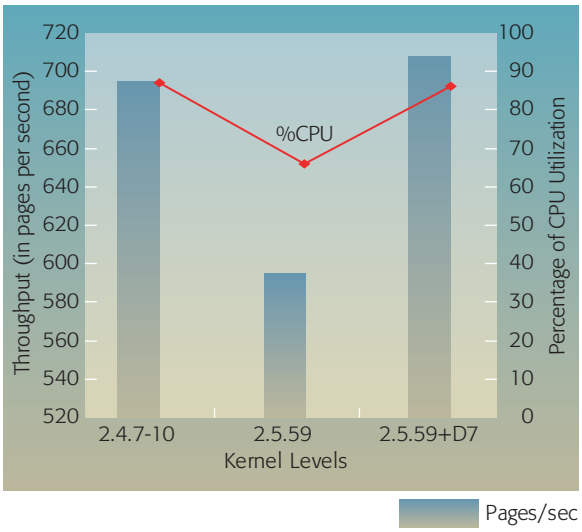

Figure 3
The load balance problem on Trade2.7

scheduler, whereas, RH AS 2.1 uses the O(1) scheduler. At first, we suspected that some new features of RH AS 2.1 might have caused the problem. However, two characteristics of the workload on RH AS 2.1 led us to believe that the problem was related to the scheduler; namely, (1) the CPU activity went down, making it 40 percent idle on the average compared to almost less than 5 percent idle in RH 7.2, and (2) there was a wide variation in the number of runnable tasks (from none to 25 tasks) throughout the run. In addition, the profiling data taken using *readprofile*, a Linux kernel tool that enables kernel timer-based profiling, revealed that the old scheduler was using seven times more CPU time than the O(1) scheduler to perform task scheduling.

To verify that the scheduler was indeed the source of problem, a quick test of Version 2.4.20 of the development kernel with and without the O(1) scheduler patch was performed. With the O(1) scheduler patch, the symptoms that were just described were observed again. A breakthrough analysis showed that not only did the number of runnable tasks vary widely, but they were also not evenly distributed among all the CPUs. Thus, we knew that the load-balancing algorithm of the O(1) scheduler was not working very well.

This problem was brought to the attention of the kernel developers. Some of the open-source community developers acknowledged seeing the load-balancing problem with the O(1) scheduler on some workloads. A few suggestions were given by the kernel open-source community, including: (1) trying out Andrea Arcangeli's version of the O(1) scheduler[14] because it has a better load balancing algorithm, (2) fixing the code by making load balancing more aggressive, and (3) modifying the code so that load balancing is activated more frequently. We followed the first suggestion. This version of the O(1) scheduler had added capabilities such as load balancing during a synchronous process wake-up. This means that an idle CPU is chosen to run an awakened process instead of scheduling it to a processor where it ran recently. This breaks the affinity to some extent. However, it is widely known that affinity and load balancing hardly complement each other. They are both important, but finding the right balance is a complicated job.

Unfortunately, using this O(1) scheduler did not help to bridge the performance gap. In fact, the throughput went down another 15 percent for Trade 2.7. However, it revealed to us a different issue, that of the queuing policy of the kernel yield function. We discuss this issue at length in the section "High rates of context switching."

While this work was ongoing, Ingo Molnar issued a patch called D7[9] for his O(1) scheduler. This patch removed the *affinity test* in his load-balancing algorithm. In the original algorithm, this test specified that a task which is already in a run queue cannot be migrated to another run queue if that task has run in its current CPU in the last n milliseconds. The value of n varies depending on the hardware architecture, cache size, memory bandwidth, and so forth. As a result, it is possible that the loads of the queues in a system will not be balanced properly if most of the tasks in the run queues still have affinity to their respective processors. By removing the affinity test, any idle CPU in the system can be given tasks in its run queue. We applied the D7 patch, and it fixed the performance problem with Trade 2.7, as shown in ***Figure 3***.

Because the original O(1) scheduler was made available for the Version 2.5.59 kernel, a comparison is made between this version and the Version 2.4.7–10 kernel that has the old scheduler. As seen in the figure, the throughput on Version 2.5.59 went

down by 14.3 percent. When the D7 patch was applied, the throughput lost was regained and came out approximately 2 percent ahead of the Version 2.4.7-10 kernel.

The same problem was observed using the SPECjAppServer2002 workload. Applying the D7 patch to the 2.5.59 kernel, however, improved performance by 16.4 percent. Comparing it to the old scheduler (Version 2.4.7-10), a 2 percent improvement was obtained, as seen in ***Figure 4***. Aside from the fact that the throughput on the 2.5.59 kernel went down, the CPU cycles also dropped significantly. Because the same injection rate is used in all of these tests, this indicates a serious bottleneck in Version 2.5.59 of the kernel. However, it also tells us that performance can be improved if all of the bottlenecks can be found. Assuming this could be done, we computed analytically the throughput "scaled to CPU" in each case; that is, the projected throughput was based on 100 percent CPU consumption. The resulting improvement, as seen in the figure, is better in the case of the 2.5.59 kernel.

Looking at Figure 4, it appears that the O(1) scheduler, even with the D7 patch, is not significantly better than the old scheduler for this particular workload. This is because the old scheduler had only one queue, and the scheduler was able to keep all the processors busy. Also, the number of runnable tasks in this workload was not in the order of thousands, so no scaling problem was evident with the old scheduler. The difference in performance between the two schedulers will be more significant when the number of runnable tasks is in the order of thousands.

With these tests, we found that both the Trade and SPECjAppServer2002 workloads do not seem to favor processor affinity of tasks. Both of these workloads are very stressful. Trade does not use any think time, and therefore, the work coming in is continuous and fills up the queues very quickly. Thus, in a highly stressful workload where requests come in continuously, load balancing results in better performance than strong processor affinity.

High rates of context switching

When Arcangeli's O(1) scheduler patch[15] was used for the Trade 2.7 benchmark, context switching was occurring at an alarming rate. The context switches on this kernel increased to approximately 30,000 per

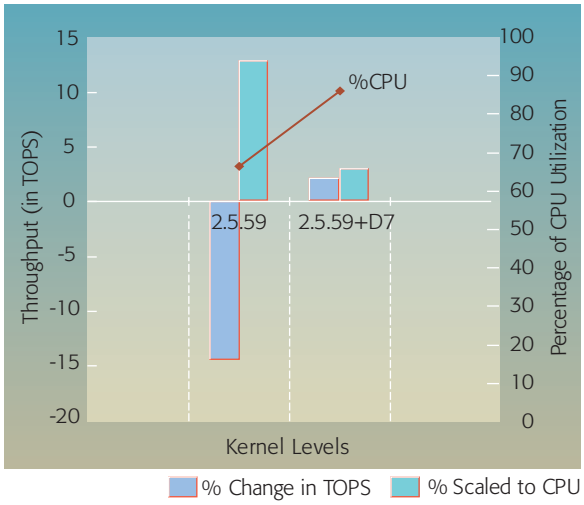

Figure 4
Fixing the load balance problem: Relative performance based on 2.4.7-10 Linux kernel on SPECjAppServer2002

second, or four times the usual rate of context switching observed for this workload.

In the kernel code, the *yield* function queues the yielding task to one of the scheduler's run queues. The scheduler then selects another task to run, resulting in a context switch. The *sys_yield* function is called by the application (in our case, the Java virtual machine is the application), but the kernel executes the function through the yield code. Where a yielding task is queued depends on the queuing policy used by the kernel. We believed that different queuing policies could have a very significant effect on context switches.

We performed detailed profiling of the queues in the yield code. ***Table 1*** summarizes the total calls to the yield function in our eight-way system and the distribution of the relative priority level of the yielding task and the task that was selected to run. For example, on CPU 0, there were a total of 263,711 yields, of which 145,103 had the yielding task and the selected task on the same priority level ("Same" row in the table.) "Only" in the table refers to a condition where there is no runnable task other than the yielding task, and this situation does not result in a context switch. As shown in the table, the values in the "Same" and "Only" rows are higher than the other rows, but the "Only" condition is an exceptional condition as the yielding task is the only

Table 1 Distribution of *yield* calls on an eight-way x440 system using the Arcangeli O(1) scheduler patch

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	112387	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	270413	276479	263134	265270	258827	277385	271337

runnable task in the system. When there are tasks with the same priority as the yielding task, the queuing policy used determines how long the yielding task is queued before it runs again.

An examination of the yield code in Arcangeli's patch and the 2.5.x kernels revealed that the queuing policies adopted by these two code sets are different. In Arcangeli's patch, the yielding task is queued right after the selected task, and thus makes the yielding task the head of the queue (see Policy P1 in ***Figure 5***). This implies that only one task will be able to run before the yielding task gets scheduled again. In the Red Hat AS 2.1 kernel, the yielding task is queued to the tail of the same priority queue, making it yield to all runnable tasks at the same priority level (Policy P2). In the Version 2.5.69 development mainline kernel, the yielding process is moved to the priority queue on the expired list, making it yield to all runnable tasks in the system (Policy P3, not shown in the figure) because the expired list becomes active only after all runnable tasks have exhausted their time slices.[16] It is clear that the yielding task yielded longer in the P2 case. This is desirable for the application, as no additional yield calls are issued. In contrast, in the

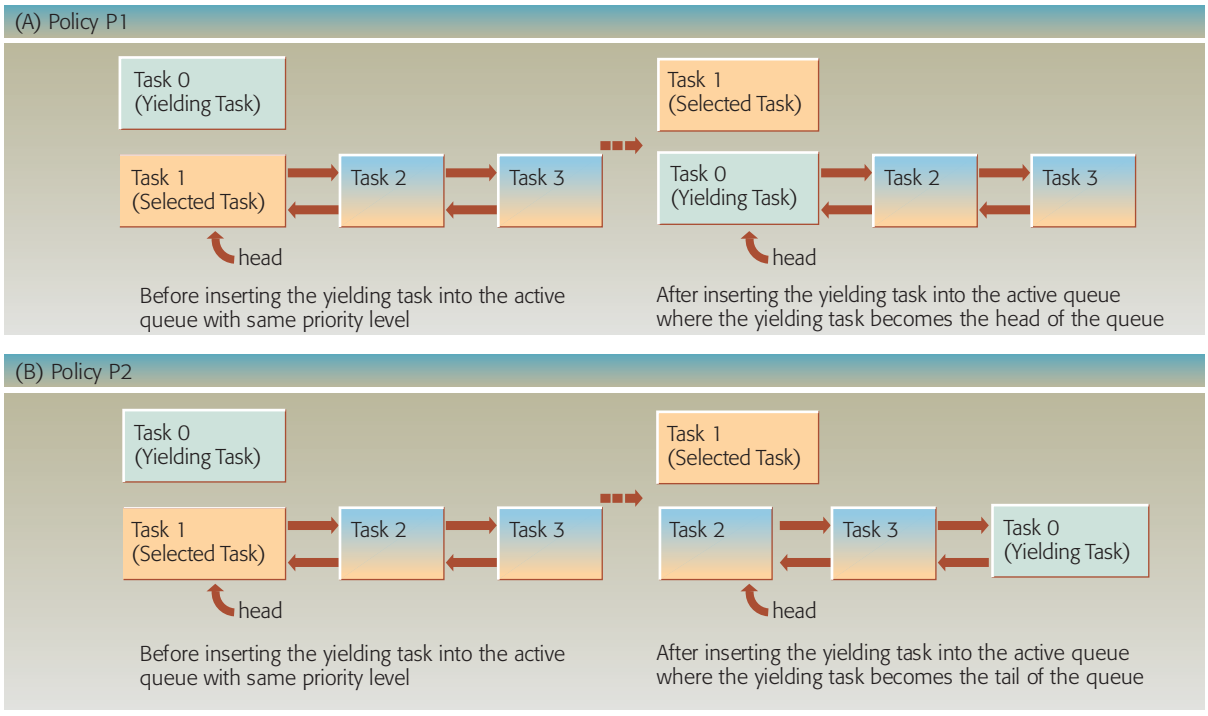

Figure 5
Queuing policies

P1 case, because the time interval during which the task yielded is not long enough, the application keeps issuing more yields, resulting in more context switches. Even though this is fixed in the 2.5.x development mainline kernels, SLES distribution kernels follow the P1 policy. The zLinux team also reported high rates of context switching as a result of their Trade benchmarking effort. This issue has two aspects: the queuing policy in the kernel and the number of yields that the application, in this case the Java virtual machine (middleware), is issuing.

As mentioned the queuing policy determines the length of time a yielding task waits until it is scheduled again. However, the kernel is unaware of the needs of the caller. An application typically calls the *sched_yield* function because the resource it needs is not yet available, but the application rarely knows how long it will need to wait for the resource to become available. The availability of the resource depends on external events and progress made by other competing tasks. Determining the optimum length of time the task needs to be yielded is a complex job. Moreover, the needs of applications vary and must be treated differently. For example, interactive applications require quick response time, and so policy P1 works very well. This is exactly the reason why the two kernels have two different behaviors when satisfying two different needs. Arcangeli adopted the P1 policy to support interactive applications. At IBM's request, he agreed to adopt policy P2 through a configuration option. For the SLES 8 SP3B1 kernel, he used P2 as the default yield policy. The improvements from using Policy P2 are shown in ***Table 2***. We discovered that using policy P3 yields results similar to using Policy P2 on both the Trade3 and SPECjAppServer2002 workloads.

Table 2 SPECjAppServer2002 benchmark results using queuing policies P1 and P2: (A) results; (B) system configuration

(A) Results		
Policy	**TOPS**	**WebSphere Application Server CPU Utilization**
P1	baseline	70%
P2	+32%	88%
(B) System Configuration		
Websphere Application Server		x440 P4, 4x2GHz, 2MB L3 cache, 4GB RAM, Intel Gigabit NIC
DB2 Server		x350 P3, 4x700MHz, 1MB L2 cache, 5GB RAM IBM ServRaid SCSI with 10 disks, Intel Gigabit NIC
Client		x330 P3, 2xlGHz, 4GB RAM, Intel Gigabit NIC
Kernel		SLES 8 SP2 (for P1), SLES 8 SP3B1 (for P2)

Using *futex* versus *sys_yield*

Even though the problem concerning the high rate of context switching has been resolved, there is a need for more automatic detection of what the optimum wait period for each application should be. Adopting a suitable dynamic policy in the kernel would be a preferable solution if applications continue to use the *sys_yield* mechanism. Arcangeli suggested that the application should use the new fast user-level *mutex* or *futex* mechanisms instead of *sys_yield*. The problem with yield is that it is nondeterministic and thus does not have clear semantics. The kernel does not know what the application is waiting for and therefore cannot estimate how long it should be made to wait. With *futex*, however, the application issues a futex *wait* call when there is contention. After the lock to the resource has been released, the application calls the kernel for a futex *wake*.[17] Thus, the kernel does not need to estimate a wait period. The Linux kernel open-source community in general is opposed to the idea of applications using *sys_yield* because every application seems to have different requirements as to how *sys_yield* should behave.

The *sys_yield* function is used basically by multithreaded applications to improve performance by giving up control of the processor to other tasks instead of waiting and consuming CPU cycles. The IBM Java virtual machine and its just-in-time (JIT) compiler both use *sys_yield* calls. A high number of yield calls are issued by the Java virtual machine as part of its three-tier locking scheme for Monitors (i.e., classes that control the access to resources by threads), which is its primary synchronization

mechanism. One study claims that over 19 percent of a Java application's time is spent on synchronization.[18] Given the importance of Monitors to performance, considerable effort has been expended throughout IBM in designing and implementing them for each platform on which the Java virtual machine runs.[19] Hence, the IBM team was initially not convinced that the three-tier monitor scheme needed any change.

Three-tier locking

The three-tier lock implements a combination of three possible actions that can be taken when a process or thread faces contention: (1) block completely and wait to be notified, (2) yield the processor and try again, and (3) idle, consuming CPU cycles. A combination of these three actions seems to yield better performance when the workload has more threads than the number of processors. It has been found through empirical database studies done with DB2 that the optimal length of time to spin before blocking depends on the number of processors in the system.[20] As a result, the IBM Java virtual machine has adopted three loop counts to determine how many times each of these three actions is taken before blocking. These counts were given default values based on experimental studies.

As the SPECjAppServer2002 benchmark effort moved to an eight-processor system, the problem of context switching along with high Java lock contention surfaced again. To reduce the number of context switches, we suggested removing *sys_yield* calls completely from the three-tier locking scheme, making it a two-tier scheme. An experiment with this approach resulted in fewer context switches but higher lock contention and lower throughput, leading to negative eight-way scalability. That clearly showed that the time spent in these two-tier locks before blocking was not enough for the eight-way system. The JIT team continued the tuning experimentation and found that the default values set in the Java virtual machine for these three loop counts needed adjustments to increase the wait time before blocking. This was especially true because in addition to increasing the number of processors to eight, the speed of the processors also increased since the last empirical studies were done to set the default values. By increasing and fine-tuning these loop counts, the throughput for the eight-way system improved by 20 percent. The Java lock contention was reduced with these adjustments, but the number of context switches remained the same. These adjustments improved performance on other Java benchmark workloads as well, such as SPECjBB2000, not only on Linux but also on AIX*, Microsoft Windows, and so forth. The IBM team updated the default values of these loop counts for all Java virtual machine platforms in the follow-on release.

LTC continued working with the Java virtual machine team and created a prototype of the Java virtual machine that replaced *sys_yield* calls with *futex* calls in the three-tier locking for Linux.[19] This prototype study shows that using *futex* calls improved performance marginally, around 2 percent. However this study also shows that by running multiple SPECjBB[21] application instances on the same system, *futex* uses the CPU more efficiently than *sys_yield*. There is more experimentation to be done to understand the effects of *futex* in other Java workloads, and LTC will participate in that effort.

Large page support in Linux and the IBM Java virtual machine

Modern computer architectures support more than one page size. If an operating system supports multiple page sizes, applications have the option to specify which page size to use. Applications with large working sets (i.e., data and code needed for application execution) benefit from using larger page sizes, especially when they need to randomly access huge amounts of data. With smaller page sizes, this kind of random access usually results in page-table misses, which have significant performance impact. Large pages reduce the number of entries in the translation lookaside buffer (TLB) table; that is, each entry is backed by larger amount of memory. They further improve the process of memory prefetching by eliminating the necessity to restart prefetch operations on 4KB boundaries.

Linux supports two page sizes, 4KB and 4MB. The actual number of large pages available in the system is configurable through the *proc* interface. Also, the allocation of large pages depends on the availability of contiguous physical memory. Thus, it is recommended to allocate large pages during system bootup. However, because large pages are not available through anonymous memory allocation (e.g., *malloc*), a temporary file system called *hugetlbfs* has to be created, which can be deleted after establishing the mapping through *mmap*. Any

file used in *hugetlbfs* is backed by large pages. Another way to allocate large pages is by using the *shmget()* interface. In either case, the large pages should be made available in the system during the system startup time.

Version 1.4.2 of the IBM Java virtual machine for Linux supports huge TLB tables or large pages. However, the results shown here are derived from a prototype based on Version 1.4.1 with a Version 2.5.69 Linux development kernel and the SPECjAPPServer2002 benchmark on WebSphere Application Server Version 5.0.2. On a four-way system, the large page support improved throughput performace by 1.3 percent; on an eight-way system, it improved throughput performance by 3.3 percent.

WebSphere clustering

We discovered that scalability for up to eight processors remains a challenge with Linux. Even with the three-tier lock tuning leading to 20 percent more throughput on eight-way runs of SPECjAppServer2002, the scalability is still well below the expected range. Considering the limitation of 32-bit Java in taking advantage of the memory in this eight-way system, the remaining option is to use more instances of the application server on the machine. This is called *vertical clustering* or *cloning* > in the context of the WebSphere Application Server, and is a typical approach when a single application server cannot reach maximum performance; that is, it cannot maximally utilize the CPU, memory, and other resources.

In the clustering approach, we create multiple instances of the application server on the same machine and then install the benchmark applications on each application server. WebSphere Application Server has a built-in facility called workload management (WLM) in which requests are distributed evenly among the application servers (since both have the same application installed). By doing this, we are using more CPU cycles, as there are more processes that are actually running actively. This improved the scalability of moving from four to eight processors. The improvement is from a factor of 1.1 to 1.3 (a factor of 2.0 means perfect scaling).

NUMA characteristics of an eight-way system

Besides the middleware, the hardware system characteristics of the eight-way system were also evaluated to assess scalability. We used an eight-way x440 system for our IA32 test. The IBM x440 NUMA Server uses a modular hardware architecture design where each module or node consists of a set of processors and local memory. In this non-uniform memory access (NUMA) architecture, modules can be added to the system as desired. The eight-way system was formed using two NUMA nodes, each with four processors. The two nodes are connected by a high-speed interconnect bus.

The NUMA architecture was designed to surpass the scalability limits of the SMP architecture, where all memory accesses go through the same shared memory bus. The Linux kernel has to leverage this architecture to reach higher scalability. The scheduler has to maintain process affinity in such a way that a given process, along with its working set, does not have to migrate to a different node. The memory subsystem has to strive to allocate memory from the local node rather than from a remote node. The IBM Linux Technology Center, along with SGI and Fujitsu (through the SourceForge[22] open-source community project "NUMA"[23]) enhanced the memory subsystem and scheduler in the 2.5.x versions of the kernel to leverage the NUMA characteristics. Versions 2.5 and 2.6 of the Linux kernel have data structures and macros for determining the layout of the memory and processors on the system. The virtual memory subsystem uses these to make decisions on the optimal placement of memory processes.

The O(1) scheduler has been enhanced to include NUMA awareness in order to support locality of processes to memory by scheduling a process on the same node throughout the life of the process. Optionally, we can also force a process to stay on the same node by pinning the process to the processors on that node. To improve the scalability of SPECjAppServer2002 on the eight-way system, two application servers are pinned, one to each node in the x440 system, so that each Java virtual machine in which the Web application server is started is localized within the node for heap allocation and garbage collection. The NUMA enhancements in the memory subsystem of the Version 2.5 kernel ensure that process memory is allocated in the local node. By using this configuration, which utilizes two WebSphere Application Servers, the eight-way system scalability is improved from a factor of 1.3 to 1.6.

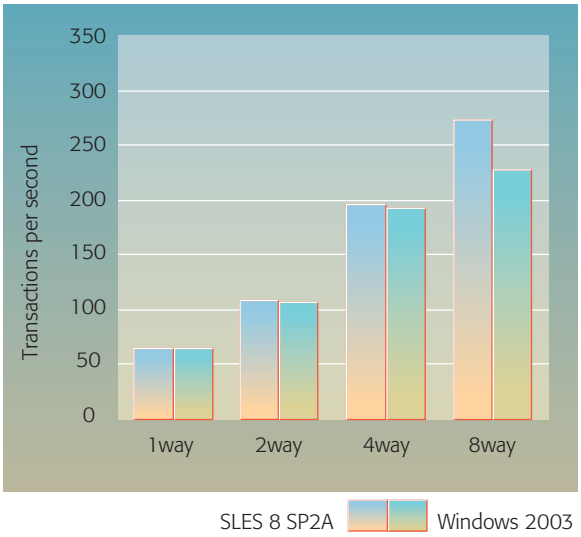

Figure 6
SMP scaling comparison using Trade3: Linux vs Microsoft Windows

OTHER ISSUES: DEADLOCK IN SPECJAPPSERVER2002

During the performance evaluation effort of the 2.5.x versions of the kernel on WebSphere using the SPECjAppServer2002 application, a deadlock in the Linux Version 2.5.59 development kernel was uncovered. This problem was reported to the Linux kernel mailing list, and the LTC kernel development team helped to resolve the bug. Because most of the kernel developers use "microbenchmarks"[24] for evaluating performance, they may or may not encounter some of the issues that might surface using multitier enterprise benchmarks such as SPECjAppServer2002 or Trade.

When a task acquires a "spinlock"[25] with interrupts disabled and then performs an operation that requires a flush of the TLBs in other processors, it sends an interprocessor interrupt (IPI) to other processors. It then enters a busy wait state, awaiting an acknowledgment. The deadlock occurs if another CPU is waiting for the same spinlock that the task is currently holding.

A patch for the deadlock was submitted to the kernel mailing list, which facilitates the open-source community verification of patches through their code review and testing. After the patch was tested in our laboratory and after making sure there was no objection to the posted patch from the community, the patch was sent to the kernel maintainers for acceptance into the development mainline kernel. The problem was found in Version 2.5.59 of the kernel, and the patch was accepted into the Version 2.5.63 of the kernel.

Table 3 System configuration for SMP scaling comparison using Trade3

Client machine	SunFire V880 8-way, 900 MHz, 16GB RAM OS: Solaris 9 Driver: WPT 1.9.4.1
System under test	IBM xSeries x440 8-way, 2 GHz, 6GB RAM OS: SLES 8 SP2A (for Linux) Microsoft Windows 2003 Enterprise Edition (for Windows) Application server: WebSphere Application Server V5.0.2
Database system	Hewlett-Packard RP7410 8-way, 875 MHz, 16GB RAM OS: HP-UX 11i Database: DB2 8.1 FP 2

Comparing Linux with Windows on IA32

Finally, this section describes the comparison we made between Linux and Windows using Trade3 and SPECjAppServer2002. To ensure an "apples-to-apples" comparison, we used exactly the same physical machines in our configuration and maintained the tuning values for all middleware and user applications, using exactly the same run procedures. In the other words, only the operating system changed. ***Figure 6*** shows the summary of the Trade3 comparison. The system configuration and results for this comparison are shown in ***Table 3*** and ***Table 4***. We used the SLES 8 SP2A release of the distribution kernel from SUSE for Linux and Microsoft Windows 2003 Enterprise Server for the other operating system.

At first glance, Figure 6 and Table 4 clearly show that the performance of Linux is very comparable with that of Windows. Although Windows utilized the CPU better than Linux, as more processors were added (except in the case of eight processors), Linux produced more throughput, indicating better efficiency. The scaling factor for each operating system was exactly the same for up to four processors. The

Table 4 Results for SMP scaling comparison

Scaling	One way		Two way		Four way		Eight way	
Operating system	Linux	Windows	Linux	Windows	Linux	Windows	Linux	Windows
Users	15	15	30	30	60	60	120	120
Response time (secs)	0.23	0.23	0.27	0.28	0.31	0.30	0.44	0.53
WebSphere Application Server percentage of CPU utilization	100	100	97	100	96	100	92	90
Database percentage of CPU utilization	3	3	6	6	13	13	17	15
Scaling	1.0	1.0	1.7	1.7	3.0	3.0	4.2	3.6

eight-way scaling results were not good for both operating systems, with a very low factor of 4.2 for Linux (8.0 being the perfect scaling). The scaling results for Windows are not as high as the Linux results, but it must be remembered that the amount of tuning applied to the Linux system was not applied to the Windows system. Nevertheless, it is our expectation that the Windows scaling results for an 8-way system will not improve drastically even when the best tuning parameters possible are used.

Table 5 shows the results of the SPECjAppServer2002 comparison between Windows 2000 and two Linux kernels. The first Linux kernel, SLES 8 SP3B2, is a distribution kernel from SUSE and the other kernel is an early version of the new Version

Table 5 Linux and Microsoft Windows comparison using SPECjAppServer2002: (A) comparison; (B) system configuration

(A) Comparison						
	Four Way		Eight Way (One Java virtual machine)		Eight Way (Two Java virtual machines)	
	TOPS	Scaling	TOPS	Scaling	TOPS	Scaling
Windows2000	Baseline	1.0	Baseline	1.28	Baseline	1.48
SLES 8 SP3B2	+3.4%	1.0	−6.1%	1.16	+5%	1.50
2.6-test2 Kernel	+3.4%	1.0	−0.002%	1.24	+7.9%	1.55

(B) System Configuration	
Client Machine:	x330 2-Way (Pentium 3) 1GHz, 4GB RAM 256 KB L2 Cache, Intel Gigabit Ethernet card Operating System: SLES 8 SP3 Beta 1
SUT:	x440 8-way (Pentium 4) 2 GHzm, 8 GB RAM 2MB L3 Cache Operating System: Windows 2000 Advanced Server SLES 8 SP3 Beta 2 + patches Linux Kernel 2.6-test2 Application Server: IBM WebSphere Application Server v5.0.2
Database: (for Windows 2000)	x400 4-way 2GHz, 8MB RAM with exp300 disk array Operating System: Windows 2000 Advanced Server Database: DB2 V8.1 FP 3
Database: (for Linux)	x350 4-way 1MB L2 Cache 5GB RAM 512 KB L2 Cache Operating System: SLES 8 SP3 Beta 1 Database: DB2 V8.1 FP 3

2.6 development kernel from kernel.org. In this comparison, we use the results of Windows 2000 as the baseline.

On the four-way system, we see that both Linux kernels yield better throughput than Windows. However, when moving to the eight-way system (for a single instance of the application server), Linux falls behind Windows. The scaling of Linux is also behind, especially with SLES 8 SP3B2. The good news, however, is that this gap has been narrowed by the new Version 2.6 kernel, with only a difference of 0.04 in the scaling factor. We tried the clustered approach where two application servers were used, each one pinned to a four-processor node in our NUMA-based machine. As we can see from the scaling, both Windows and Linux benefited from this approach quite significantly. It is also interesting to note that Linux has benefited a lot more in this approach, as it surpassed the scaling and throughput of Windows. This may be a manifestation that Linux supports NUMA better than Windows. Thus we achieved a scaling of 1.55 out of a perfect scaling of 2.0 for the Linux 2.6 kernel.

CONCLUSION

We have provided a detailed description of the work we have done to improve the performance and scalability of WebSphere Application Server Version 5 on the Linux platform. The overall performance of the middleware, and hence the user applications, depends heavily on Linux because it is at the bottom of the software stack. We have demonstrated performance gains by applying some of the key enhancements in the Linux kernel and have analyzed some of the characteristics of the workload with each patch that we have tried. Overall, the performance of WebSphere Application Server on Linux is comparable to that of other operating systems on the same hardware. Scalability on SMP systems has improved for up to four processors. Beyond that, however, major work and research in this area are still needed. Thus, it is fair to say that Linux is ready to be an enterprise server provided processors are limited to a maximum of four when SMP systems are used.

The work group continues to work on its mission with the additional goal of ensuring that performance does not regress with newer releases of WebSphere Application Server and Linux distributions. Some of the major areas that need to be investigated further include NPTL (Native POSIX** Thread Library for Linux) measurements; differentiating the performance of *futex* and *sys_yield*; trying out a distributed configuration of WebSphere Application Server, that is, a clustered configuration of application server nodes; extending scalability improvements to eight processors; and conducting additional competitive evaluations with other operating systems.

ACKNOWLEDGMENTS

We would like to thank our management teams and Andrea Arcangeli from SUSE and recognize all of the hard work of the WPLP (WebSphere Application Server Performance on Linux Platforms) team members, Christopher Blythe, Kenichiro Ueno, Rajan Ravindran, Brian Twichell, the JIT/Java virtual machine team, and the DB2 team from Toronto.

CITED REFERENCES AND NOTES

1. The LTC itself performed the tests on the xSeries platform.
2. Java 2 Platform, Enterprise Edition (J2EE), http://java.sun.com/j2ee.
3. Pulsar—eBusiness Benchmarks for WebSphere Application Server, http://pulsar.raleigh.ibm.com.
4. IBM alphaWorks: Emerging Technologies, http://www.alphaworks.ibm.com.
5. SPEC—Standard Performance Evaluation Corporation, http://www.spec.org.
6. SPECjAppServer2002, http://www.spec.org/jAppServer2002/.
7. The Linux Kernel Archives, http://www.kernel.org.
8. M. Kravetz and H. Franke, "Linux Multi-queue Scheduler," (2001), http://lse.sourceforge.net/scheduling/mq1.html.
9. I. Molnar, "O(1) Scheduler Version 2.5.59," http://people.redhat.com/mingo/O(1)-scheduler/sched-2.5.59-D7.
10. The O(1) scheduler was "backported" to some of the later releases of the Version 2.4 distribution kernels.
11. Volano: The Volano Report and Benchmark Tests, http://www.volano.com/benchmarks.html.
12. SPECWeb99 User's Guide, http://www.spec.org/Web99/docs/users_guide.html.

13. The Public Netperf Homepage, http://www.netperf.org/netperf/NetperfPage.html.
14. A. Arcangeli, "Scheduler for Linux Kernel Version 2.4.20," The Public Linux Archive, http://www.kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/2.4.20rclaal/.
15. This patch is based on the 2.4.20 kernel.
16. V. Anand, H. Franke, H. Linder, S. Nagar, P. Narayan, R. Ravindran, and T. Ts'o, "Benchmarks that Model Enterprise Workloads," *Proceedings of the Ottawa Linux Symposium* (2003), pp. 434–446, http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Tso-OLS2003.pdf.
17. R. Russell, M. Kirkwood, and H. Franke, "Fuss, Futexes and Furwocks: Fast User-Level Locking in Linux," *Proceedings of the Ottawa Linux Symposium* (2002), pp. 479–495, http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz.
18. E. Armstrong, "HotSpot: A New Breed of Virtual Machine", Java World, March 1998.
19. R. Dimpsey, R. Arora, and K. Kuiper, "Java Server Performance: A Case Study of Building Efficient, Scalable Jvms," *IBM Systems Journal* **39**, No. 1, 151–174 (November 2000).
20. D. Guniguntala, "2 Tier IBM JVM Monitor Implementation with Fastlocks," http://bvrgsa.ibm.com/projects/l/ltcisl/public/jvm/fastlock/fastlock.html.
21. SPEC JBB2000, http://www.spec.org/jbb2000.
22. SourceForge.net is the world's largest open-source software development Web site, with the largest repository of open-source code and applications available on the Internet. SourceForge.net provides free services to open-source developers.
23. Open Source NUMA Project, http://lse.sourceforge.net/numa.
24. "Microbenchmarks" are small programs written to measure the performance of a single subsystem such as a network, SCSI layer, file system, or memory. They are usually easy to set up and run.
25. "Spinlock" is a busy-wait method of ensuring mutually exclusive use of a resource.

Accepted for publication November 12, 2004.
Published online April 12, 2005.

Vaijayanthimala K. Anand
*IBM Systems Group, LTC, 11501 Burnet Road, Austin, TX, 7878 (*manand@us.ibm.com*).* Ms. Anand is a member of the Linux Kernel Performance team in the Linux Technology Center. Her interests include networks, Java technology, and kernel performance. She has a Master's degree in computer science from the University of Houston.

Wilfred C. Jamison
*IBM Software Group, AIM Division, 3039 Cornwallis Rd., Research Triangle Park, North Carolina 27709 (*wjamison@us.ibm.com*).* Dr. Jamison is currently a member of the on demand software strategy team. He was a member of the WebSphere Performance team when this study was conducted and led the study at that time. His interests include software performance, Java technology, distributed systems, and programming methodologies. He received a Ph.D. degree in computer science in 1998 from Syracuse University. ■