Java, threads, and scheduling in Linux

Patching the kernel scheduler for better Java performance

Ray Bryant and Bill Hartner
IBM Linux Technology Center
January 2000

This paper examines the performance of the Linux kernel when the system is running a Java program using the IBM Developer Kit for Linux, Java Technology Edition, Version 1.1.8. The Java program is the VolanoMark benchmark, a communication-intensive benchmark that creates a large number of Java threads. This implementation of the Java technology uses a native-threading model; each Java thread is supported by an underlying Linux process. Our measurements show that 20% or more of total system time spent running this benchmark is spent running the Linux kernel scheduler. We examine some simple modifications to the scheduler and their impact on the time spent in the scheduler. Finally, we suggest changes to the Linux kernel that may more effectively support applications of this type. In particular, the Linux scheduler needs to be modified to more efficiently support large numbers of processes, for which we offer a patch here. Additionally, we conclude that Linux kernel needs ultimately to be able to support a many-to-many threading model.

One of the most common uses for Linux today is to provide an effective and load-tolerant platform for a Web server. When combined with Apache (or another Web server such as AOLServer), Linux can be the basis for a powerful and inexpensive Web server.

Just as Linux is important for hosting Web servers, Java is an important component for building scalable, Web-based application servers. By writing the server-side applications in Java, you can move the application from platform to platform with relative ease.

Given the importance of Java on Linux, it is important that we understand the scalability issues of both the IBM Developer Kit for Linux, Java Technology Edition, Version 1.1.8 (hereafter called IBM Java VM for Linux) and the underlying Linux system. In this paper we explore the performance impact of the threading model used by the IBM Java VM for Linux on the underlying Linux system. We use a benchmark based on a popular pure-Java application, VolanoChat (developed and sold by Volano LLC), as the workload to drive the system for this study.

In the following sections of this paper, we first discuss various threading models and how they have been implemented to support threading in a Java implementation. We then describe the VolanoChat benchmark, VolanoMark, and discuss our benchmark experiments. We then present performance measurements of VolanoMark and explore some minor modifications to the Linux kernel and how these modifications affect the performance of the Linux kernel while running the benchmark. In our concluding remarks, we suggest changes to the Linux kernel that may need to be made if applications of this type are to be effectively supported under Linux.

Threading alternatives for Java

We first need to establish some terminology. By *thread*, we mean a lightweight execution context (also known as a *lightweight process*). By *Java thread*, we mean a thread created via the Runnable interface in the Java language. By *user space thread*, we mean the execution context of a user program described by its registers, stack pointer, etc.; a user space thread is independent of the Java environment. By *kernel thread*, we mean a thread that is known to the operating system kernel; a kernel thread has additional execution context known only to the operating system kernel.

A user thread can be thought of as being attached (or *mapped*) to a kernel space thread when the user thread

is executing. A standard Linux process can be thought of as an address space containing precisely one user space thread mapped on top of one kernel thread. However, it is possible to multiplex a number of user space threads on top of a single kernel thread by manipulating the user thread state. Typically this involves the construction of a user-level scheduler to control execution of the various user level threads. This approach is known as *many-to-one threading* since a number of user space threads are mapped to a single kernel space thread. One advantage of many-to-one threading is that it normally can be implemented without requiring any special kernel support. Another threading model maps each user space thread to its own kernel thread. This is commonly called *one-to-one threading*. This is the threading model Linux currently supports.

Finally, we can have the full generality of both worlds: some number of user space threads is mapped to some other (typically smaller) number of kernel threads. This final approach is *called many-to-many threading*.

More information on many-to-one threading, one-to-one threading, and many-to-many threading is available.

Threading in the Java language

Threads are an essential part of programming in the Java language. Java threads are useful for many purposes, but because the Java language lacks an interface for non-blocking I/O (see the Resources section later in this article), threads are especially necessary in constructing communications intensive applications in Java. Typically one or more Java threads are constructed for each communications stream created by the Java program.

The execution environment for Java (the Java virtual machine) implements Java threads, through any of a number of alternative ways. One way is to implement Java threads as user space threads (a many-to-one model). This is how the green threads option of the Solaris Java 2 Classic VM is implemented (see Multithreading Options on the Solaris Operating system). Although this can be a highly efficient approach for certain applications, the green threads option has the disadvantages below. (For this reason the Java 2 Classic VM also supports a native threads option.) The disadvantages of the green threads approach are:

1. A Java implementation written using the green threads approach cannot exploit a symmetric multiple processor (SMP) system since only one kernel thread is being used to implement all of the Java threads; each kernel thread is executed by only one processor at a time.
2. A Java thread supported under green threads cannot call directly into a non-green-thread enabled library because the latter may itself call a system call that would cause the underlying kernel thread to wait, thus suspending execution of the entire Java virtual machine.
3. Efficiency of execution can be low because the green thread implementation must wrapper all calls to the operating system to avoid having the thread directly call a blocking system call.
4. It can be difficult to support multiple instances of per-Java thread system objects (such as connections to a database server) since multiple user-space threads share use of the single kernel thread executing in the Java virtual machine.

On the other hand, the Java implementation can choose to implement each Java thread using an underlying kernel thread (in the Java technology this is called a *native thread* implementation -- we would also call this a one-to-one model). This approach has several advantages:

1. A Java virtual machine written using the native threads approach can exploit an SMP system. Multiple processors can be executing Java threads at the same time because multiple kernel threads can be executing in the Java virtual machine at the same time.
2. A Java thread supported under native threads can use any of the operating system's thread-enabled library's directly, without the need for either modifying the library or wrappering calls into the library.
3. Efficiency of execution can be high, provided that primitives for dealing with threads in the operating system are efficient and scalable.
4. Each Java thread can have its own per-thread system objects since each Java thread appears to the operating system as a separate kernel thread.

Of course one disadvantage of the native threads implementation is that all thread operations become kernel operations; one neither cannot create, destroy, nor suspend a thread without making a system call. Green threads implementations, on the other hand, have the potential to make all such operations extremely cheap since they can happen entirely in user space and need deal only with the user level state of the thread. For the reasons given above, the IBM Java VM for Linux uses the native threading model. (This decision was also due to the fact that the IBM Java VM for Linux shares much common code with versions of the IBM Java VM for

other platforms (such as AIX), and only the native threads mode is supported by the IBM Java VM on these other platforms as well.) Other choices for Linux could have been made. For example, the Blackdown JDK (another Java implementation available for Linux) supports both green and native threads, selectable as a user option when the JDK is started. But this is a significant amount of work, since the default implementation of threads in Linux is itself a native-threading model, and the IBM Java VM for Linux uses this implementation.

Implementation of threads in Linux

Unlike some other implementations of Unix (such as AIX or Solaris), the Linux kernel itself does not provide an abstraction of allowing several kernel threads to be packaged inside a process. Instead, a special version of the fork() system call (also known as clone()) has been implemented that allows you to specify that the child process should execute in the same address space as its parent. This system call is then used by the Linux threading library to present a thread programming abstraction to the user. From the user interface standpoint, practically all of the POSIX thread interfaces are fully supported under Linux; however, the implementation is not fully POSIX-compliant due to some problems in signaling. The result, however, is that each Java thread in the IBM Java VM for Linux is implemented by a corresponding user process whose process id and status can be displayed using the Linux ps command.

Benchmark description

The benchmark used in this paper is VolanoMark (see Resources). VolanoMark was developed by John Neffenger (founder and Chief Technology Officer of Volano LLC) to measure the performance of Volano LLC's VolanoChat product on different Java implementations. VolanoMark consists of the VolanoChat server (herein called the server) and a second Java program that simulates chat room users (herein called the client). Neffenger periodically publishes the Volano Report, a summary of his results running VolanoMark using a number of different Java implementations on a number of different host operating systems (as of December 8, 1999, the report details the performance of 17 Java virtual machines on 7 operating systems).

VolanoMark can be run in two modes: loop back and network. In *loop back mode*, both the client and server components of VolanoMark are run on the same system. In *network mode*, the client runs on a system that is physically distinct from the system where the server runs; the client and server machines are connected by some type of local area network (typically Ethernet). All of the measurements reported in this paper run VolanoMark in loop back mode.

VolanoMark uses two parameters to control the size and duration of the benchmark run. The parameters specify the number of chat rooms and the number of messages each simulated user is to send to the server before quitting. For our measurements, the number of rooms in the benchmark was varied from 5 to 25 and the message count was 100. An additional configuration parameter controls the number of simulated users per room; this number was set to 20 for the measurements reported here.

In addition to the VolanoMark benchmark parameters, a number of parameters to the IBM Java VM for Linux can significantly affect the overall performance of a program being run by the JDK. The most important of these are the minimum and maximum sizes for the Java heap.

Java is a garbage-collected language (see Resources), and all objects allocated during the course of execution of a Java program are allocated from the Java heap. When space in the Java heap is exhausted (and at other times that are not relevant to the current discussion), execution of the Java program is suspended, and the garbage collector is invoked. The garbage collector examines all objects on the heap, determines which objects are no longer used, and converts the storage occupied by those objects into free space in the heap. Since objects are constantly being created by almost all Java code, attempting to run a Java program with a heap size that is too small can result in many garbage collection cycles, and overall program performance can significantly decrease.

Because we wished to concentrate in this paper on measuring performance of thread scalability, we used a constant heap size of 256 MB for all of the experiments described here. This heap size is sufficiently large that the overhead of garbage collection is small throughout our measurements (less than 5% of elapsed time was spent in garbage collection). However, these heap sizes are different than those used in the Volano Report. The throughput results of this paper are therefore not strictly comparable to those given in the Volano Report.

Each simulated user in the VolanoMark benchmark establishes a socket connection to the server. Because the Java API does not include non-blocking read and write APIs, VolanoMark is implemented by using a pair of

Java threads on each end (client and server) of each connection. For the loop back mode tests that we are measuring here, a single chat room creates 80 Java threads (2 per connection times 20 connections per room times 2 because both the client and the server run on the same machine). Thus as we vary the number of rooms in the benchmark from 5 to 25, the number of Java threads in the system varies from 400 to 2,000. Since the current implementation of the IBM Java VM for Linux supports each Java thread using an underlying kernel thread (native threads mode), the kernel must potentially deal with 400 to 2,000 threads in the run queue. (Of course, typically not all threads are waiting to run, so these numbers represent absolute upper bounds on the number of threads to schedule at any time.)

The overall flow of the VolanoMark benchmark is as follows. First, the requested number of rooms are constructed by establishing the required connections between the simulated users and the server. Once all connections have been established, each simulated user begins by sending a message to the server. As each message is received by the server, it is broadcast to all other members of that particular chat room by sending messages on each user's connections to the server. This process is repeated until each simulated client has sent the required number of messages. At this point the VolanoMark run is complete, and the benchmark statistics are calculated.

The key statistic that is reported both by ourselves and in the Volano Report is the message throughput, that is, the number of messages per second the server was able to handle during the benchmark run. In addition, in a second part of the report, the Volano Report also reports the maximum number of connections that the server was able to sustain (the top Java virtual machines are able to support up to 4,000 connections per virtual machine). Thus, in order to be well rated by the Volano Report, a Java virtual machine running VolanoMark must support a high level of throughput as well as scale to a large number of connections.

Why is VolanoMark an interesting benchmark?

Although Neffenger originally developed VolanoMark as a benchmark for VolanoChat, VolanoMark has also become an important benchmark for comparing the performance of different implementations of Java. The Volano Report has been widely published in magazines such as JavaWorld (see resources>000, and the scores reported there have been used to provide a comparative evaluation of Java performance from different vendors. IBM, of course, would like the scores of its IBM Java VM for Linux to be as high as possible, and we have used this benchmark to assist in performance optimization of our virtual machine. Hence, VolanoMark is an important benchmark for IBM's Java VM for Linux (as well as other platforms). Also, since VolanoChat itself is a real Java application marketed and deployed in many environments, VolanoMark is an important benchmark because it allows us to evaluate the performance of a real application while running on the IBM Java VM for Linux.

Benchmark results

In this section we describe the measurements that we performed using VolanoMark. Each measurement was done using a Linux 2.2.12-20 kernel (also known as the Red Hat 6.1 kernel) as well as with a Linux 2.3.28 kernel obtained from www.kernel.org. Neffenger's suggestions (see his article on Linux) for modifying Linux to run with large numbers of processes as well as his suggestions for modifying LinuxThreads to support large numbers of threads were applied to the kernels and to libpthread.so. The patch to enable The IBM Kernel Trace Facility for Linux was applied to each kernel. The kernels were built as uniprocessor kernels with a standard SCSI and Ethernet device built into the kernel (not built as modules). The hardware used for all of these tests is described in the Appendix.

The first two measurements we will discuss in this section are the VolanoMark messages per second metric and profile data indicating the amount of time spent in the most heavily used ten kernel functions during the VolanoMark run. The profiling measurements were done using the profiling feature of the IBM Linux Kernel Trace Facility. Details on how the measurement data were collected are in the Appendix).

Figure 1 shows the VolanoMark messages per second achieved during a loop back run of VolanoMark 2.1.2 for the Linux kernels 2.2.12-20 and 2.3.28. Several aspects of this chart are interesting. The first interesting feature is that except for the 5 room (400 thread) case, VolanoMark when run under 2.2.12-20 produces a slightly higher throughput number than when the same experiment is performed with 2.3.28. The second interesting feature is that the throughput statistic decreases markedly as the number of threads increase. (The decrease is 19% for kernel 2.2.12-20 and 24% for kernel 2.3.28.) This decrease occurred in spite of the fact that the Java heap size was sufficiently large so that the percentage of time devoted to garbage collection averaged less than 5% in the worst (25 room / 2,000 thread) case.

To begin to understand why this throughput decrease was taking place, we next used the profiling feature of the IBM Kernel Trace Facility for Linux to measure the amount of time spent in each kernel routine during the VolanoMark run. These data were summarized down the ten busiest such routines; these data are shown in Figure 2 and Figure 3. Each vertical bar represents the fraction of time spent in the kernel that was spent executing the routine indicated by the legend.

The data presented in this chart are organized as follows. From the bottom bar to the top bar in each column, the routines are sorted by increasing total percent of kernel time associated with each routine across all 5 cases (400, 800, 1,200, 1,600 and 2,000 threads). Thus, the bottom bar may not be the largest bar in the column, but the routine corresponding to the bottom bar in each column had the largest sum of percentages across all of the cases shown.

The striking observation here is the amount of kernel time spent in the scheduler for the cases shown in Figure 2 and Figure 3. This number is between 30 and 50 percent for kernel 2.2.12-20 and between 37 and 55 percent for kernel 2.3.28.

Another way to look at this data is presented in Figure 4 where we have plotted the total percent of time spent in the kernel and the total percent of time spent in the scheduler. While the percentages of time spent in the kernel for the two cases are basically identical (the lines overlapped one another on the chart, so we show only a single line), the total percent of time spent in the scheduler is significantly higher for kernel 2.3.28 than it is for 2.2.12-20. This may reflect the effort that has been devoted to parallelization of the development line of kernels 2.3.x, which resulted in an overall reduction in the amount of time spent in kernel routines other than the scheduler.

To understand why so much time was being spent in the scheduler, we next examined the scheduler code itself. From this examination, it became apparent that a significant amount of time could be spent calculating the scheduler's goodness measure. Each time the scheduler is entered, the goodness value for each process in the run queue is calculated. The process with the highest goodness value, or in the case of ties, the first such process encountered with that highest goodness value, is selected as the next process to run. For large numbers of threads, examining the goodness function for every thread in the run queue can become a time-consuming process.

To quantify this statement, we added some measurement code to the scheduler itself (see Scheduler measurement code in the Appendix). These additions allowed us to record the distribution of the run queue length (Figure 5) and the number of cycles required to calculate the goodness function itself (Figure 6). Figure 5 is based on Linux kernel 2.3.28; Figure 6 shows results for both 2.2.12-20 and 2.3.28. Figure 6 also gives the VolanoMark messages per second observed for the trials measured.

First, note the long tail of the distribution in Figure 5; even though the average size is 32.2 with a standard deviation of 19.8, the maximum queue size observed was 414. This indicates that while the average run queue length was only 32 or so, there were times when the run queue was much longer. Second, note that the curves shown in Figure 6 are basically linear except for the initial part of the curve. If we fit a linear model to this curve, it indicates that each additional element of the run queue takes approximately 167 cycles for the 2.2.12-20 kernel and 179 cycles for the 2.3.28 kernel.

IBM kernel performance patch

After looking at these data, it was observed that the data used in the goodness function calculation were spread across several cache lines in the task structure. We wondered what would happen if the fields required by the goodness function were placed together in the task structure. Figure 7 shows the result of re-running the scheduler profiling experiment using a kernel where the fields required for the goodness function were reorganized to be adjacent to each other in the task structure. A patch for 2.3.28, applying the reorganization, is available. (Of course, this modification assumes there are not other places in the kernel where alternative reference patterns would suggest that the task structure be reorganized in a different way.) If we fit linear models on top of the new curves, we find out that each additional member of the run queue now costs approximately 108 cycles for the 2.2.12-20 kernel and 115 cycles for the 2.3.28 kernel -- a savings of approximately 35%!

The savings are so large because:

1. It takes fewer cache line misses to access the goodness function data for each run queue element (see

Reduced cache line misses for reorganized task structure in the Appendix).
2. It takes fewer instructions to access the data itself (see Reduced instruction count for reorganized task structure in the Appendix).

If we use the modified kernel to run the VolanoMark benchmark and plot the resulting throughput data, we get the chart shown in Figure 8. And if we repeat the kernel profiling measurements and plot the data, we get the chart shown in Figure 9. Figure 10 shows the plot of percent of total time in kernel and total time in scheduler. Thus with this simple kernel modification, we were able to increase VolanoMark throughput an average of 7% (the differences between the runs ranged from 4 to 11%, with an average of 7% across the runs) as well as making a reduction in the amount of kernel time spent in the scheduler.

We next asked the question, "What would happen if *somehow* the goodness search was made very fast?". To simulate this, we introduced a simple hack into the scheduler that would select the process at the head of the run queue to run next. (Obviously this is a poor choice for a scheduling algorithm!) Figure 11, Figure 12, and Figure 13 show the results of this experiment. In these figures, the data labeled "2.3.28" is for the unmodified kernel, the data labeled "2.3.28+task" is for the kernel with the task structure reorganization patch applied, and the data labeled "2.3.28+task+sched" includes both the task structure path and the scheduling hack.

While Figure 12 shows that the scheduler is no longer the heaviest user of CPU time in the kernel, the throughput results of Figure 11 for this case are disappointing. Obviously there are some negative effects of this crude scheduling technique that do not allow the 14% reduction in overall time spent in the scheduler (the difference in Figure 13 between the bottom blue line for 2.3.28+task+sched and the green line for 2.3.28 fourth from the bottom ranges between 11 and 17% with an average of 14%) from being converted into a corresponding increase in VolanoMark message throughput. However, this measurement does suggest what could happen to the fraction of time spent in the scheduler if the amount of time spent calculating the goodness function were drastically reduced.

Concluding remarks

We have observed that communications applications written in Java that support large numbers of connections typically create a correspondingly large number of Java threads. We have discussed the alternatives for supporting these threads under a Java virtual machine running on Linux, and we have explained that the IBM Java VM for Linux uses a "native threads" implementation of Java threads. Under this implementation of the IBM Java VM for Linux and of threading in current versions of Linux, the result is that each Java thread is executed by a separate Linux process. Communication applications written in Java that are run by the IBM Java VM for Linux can therefore create a large number of Linux processes when they are run.

We then described VolanoMark, a benchmark of the VolanoChat server from Volano LLC. We pointed out that one of the metrics used to evaluate a Java virtual machine in the Volano Report was the total number of connections supported per VolanoChat instance. We then measured the performance of VolanoMark as the number of connections increased, and we examined the amount of time spent in the system scheduler using a kernel profiling tool. We observed cases where over 25% of total system time is spent in the scheduler in the system. We then demonstrated a simple change to the task structure of the kernel that would reduce this cost somewhat and we also demonstrated that the reason that so much time is spent in the scheduler is due to the cost of the goodness function calculation of scheduler.

We conclude that if Linux is to be used to run Java applications like VolanoChat, then the measurements described in this paper suggest that the following changes may be needed.

- The Linux scheduler needs to be modified to more efficiently support large numbers of processes. This is a large system scalability feature that many people believe is required in order for Linux to become enterprise ready (see "Making Linux a World-Class Enterprise Server OS" on The Linux Scalability Project home page). This is not just a Java issue. Other applications also wish to make use of threads to improve performance.
- The Linux kernel needs to be able to support a many-to-many threading model. At the moment, the Linux kernel cannot support a many-to-many threading model (see, for example, a FAQ on kernel threads). A many-to-one threading model solves the scheduler bottleneck, but has other problems including lack of SMP scalability for the application, or insufficient thread state to support some application requirements. The Benchmark results section of this paper shows the potential problems of the one-to-one threading model. A many-to-many threading model would allow a large number of

user-level threads to be created without requiring these threads to be scheduled by the Linux kernel scheduler; those threads that require thread state that is not supportable totally in user mode could be supported by binding such a user thread to a particular kernel-level thread.

While the VolanoMark benchmark studied here is an example of a specific workload that places high demands on the scheduler of the underlying operating system kernel, we believe that the effects described here can be present to some extent in other important workloads as well. We look forward to working with the members of Linux community to design, develop, and measure prototypes of Linux code to support the changes described above. Whether or not these changes make their way into the base kernel is another matter, and only by experimenting with these changes and studying their behavior under more general workloads will the Linux community be able to determine whether such changes should be incorporated into the base kernel or not.

Another alternative to consider would be to modify the IBM Java VM for Linux to support many-to-one threading (green threads). This is a possible stop-gap solution, due to the problems with green threads (outlined in this paper as well as in Multithreading Options on Solaris Operating Systems -- see Resources). We observe, for example, that only the native threads option is currently supported for the Sun Java 2 Client VM, an advanced Java virtual machine that uses the HotSpot technology. We do not regard this as a general solution for the threading problems discussed in this paper.

Appendix

This appendix includes the following sections:

The IBM Kernel Trace Facility for Linux

The IBM Kernel Trace Facility for Linux consists of a kernel patch that implements the trace facility itself as well as a set of user-level programs to read the trace data and produce data analysis reports. The Trace Facility provides a general-purpose, low-overhead facility for recording data generated by trace hooks inserted in the kernel. Each trace hook is identified by a major and minor code pair; typically hooks related to an entire kernel subsystem are grouped together using the same major code and a minor code is used to uniquely identify each hook of the group. The trace facility time stamps the data recorded by each hook and handles storage allocation of the hook data in the trace buffer. In its present form, the Trace Facility uses a fixed size buffer (allocated at boot time) to record the trace data. User-level interfaces are provided to reset, start and stop the trace, to enable or disable hooks on a major code basis, and to write the data that has been collected out to a user-specified file. This trace facility forms the basis for a number of different performance measurement tools that we have developed for internal IBM use on Linux. For example, the data collected by our scheduler measurement code was written to the kernel trace and analyzed using a special trace analysis program (however, for brevity, we did not show this portion of the code here).

The kernel profiling feature of the IBM Kernel Trace Facility uses the capabilities of the trace facility to implement a kernel profiling utility. This is implemented by inserting a hook into the kernel timer routine that records information about the currently running process on every timer tick. This information includes the instruction counter and the mode (user or kernel) of the process. When the trace records for the profiling feature are analyzed, this information allows us to estimate the amount of time spent in each kernel routine (excluding those portions of code run with interrupts disabled) as well as the amount of time consumed by each user process.

While this feature, in its present form, is very similar to the profiling facility included by default in the Linux kernel, the generality of the trace facility allows us to support the following measurement techniques that would be difficult or impossible to support under the existing Linux kernel facility:

1. Recording how much time is consumed by each user level process
2. Recording of exact kernel instruction addresses found when the timer interrupt occurred

3. Recording of and accounting for interrupt time separately from the kernel profile recorded time
4. Tracing of dispatch and undispatch activities of the kernel scheduler
5. Tracing of sequential behavior, such as lock acquires and releases

Additionally, the kernel profiling feature can be extended to support profiling of user space programs as well. We are in the process of developing this extension.

We are currently examining the feasibility of releasing the IBM Kernel Trace Facility for Linux as open source code. The release would likely also include the kernel profiling feature and the data reduction routines associated with that feature as well as a general purpose trace reader. The latter would allow users to add customized trace hooks to the kernel and to use the trace facility as a mechanism to collect and export the data recorded by those customized hooks. If the decision is approved to release the IBM Kernel Trace Facility for Linux, further details will be published on the developerWorks Linux zone.

Scheduler measurement code

See the kernel scheduler code. Additions to measure the time spent calculating the goodness function are shown in red. The original code from 2.3.28 is shown in black. Similar changes were made to measure goodness function performance in 2.2.12-20. The *getPerfClock()* function referred to here uses the RDTSC instruction to read and return the current value of the Pentium time stamp counter register. These and other additions (not relevant to the discussion here) allowed us to measure the data shown in Figure 5, Figure 6, and Figure 7.

Reduced cache line misses for reorganized task structure

To measure the number of cache lines fetched during the goodness calculation, we inserted code to count the number of DCU_LINES_IN events recorded by the Pentium II performance monitor, as described in the Intel Architecture Software Developer's Manual Volume 3: System Programming, Appendix A "Performance Monitoring Events, "Total Lines Allocated in the DCU". A decrease in this number indicates a reduced number of cache misses.

For the standard 2.2.12-20 kernel, we found that DCU_LINES_IN was typically 4 for each goodness calculation. For the 2.2.12-20 kernel with task structure cache alignment patch, the DCU_LINES_IN counter was typically 1 per each calculation. Since the code we are executing in each case is the same to within 4 instructions (see for details), we infer from a reduction here that the number of data cache line misses during the goodness calculation has decreased.

Reduced instruction count for reorganized task structure

See the the code that gcc generates for the run queue search loop with and without the reorganized task structure (version: egcs-2.91.66, called with options -O2 -march=i686 and the usual Linux compilation options: -fomit-frame-pointer -fno-strict-aliasing -fno-strength-reduce). As you can see from the instructions marked in red, the compiler generates 4 fewer instructions for the second case, and this may result in a corresponding speed improvement for that version of the code.

Summary of machine configuration and experimental setup

In all cases for the data presented here, the machine used for the test was an IBM Netfinity 7000 M10 with 1.5 GB of real storage; in all cases the Linux kernel was booted with a memory size of 1 GB. 4 MB of this storage was dedicated to the trace buffer of The IBM Kernel Trace Facility for Linux. Although this machine is a 4 processor SMP, the kernels for the data presented here were built as UP kernels so that all measurements were done as if this machine was only configured with a single processor. The Netfinity 7000 M10 used has 4 Pentium II processors and uses the Xeon chip set. L2 cache size on this machine is 1024 KB per processor; L1 cache size is 32 KB.

For each of the experiments described, the VolanoChat server was started on one IBM Java virtual machine while the VolanoMark client was started using a second instance of the Java virtual machine. At the end of the test, the client virtual machine terminates, while the server virtual machine continues to run VolanoChat. Because there are potentially some startup overheads in the server virtual machine, the first VolanoMark run of each set was discarded and the data reported here were the average of the data observed in the rest of the trials (either 4 or 5 data points depending on the exact measurement; 5 data points were averaged for the Volano throughput and kernel profiling experiments; 4 data points were averaged for the scheduler measurement experiments). The kernel was normally not rebooted between experiments.

The IBM Java virtual machine used in these tests is an internal build of IBM Java VM for Linux; this internal build fixes a problem we encountered when running VolanoMark in network mode with 50 rooms in the client. This version is referred to as "GA + TQLOCK Fix" on the charts. That is, the fix we made was applied to the GA version of the IBM Developer Kit for Linux, Java Technology Edition, Version 1.1.8. An updated version of this virtual machine that includes this fix may be released by IBM in the future.

All of the data described in this paper was measured using Volanomark 2.1.2. All tests reported here were performed without independent verification by Volano LLC. Since run rules different from those specified by the Volano Report were used in this paper, none of the throughput results of this paper should be compared with those given in the Volano Report.

Figures

The figures referenced in this paper are grouped for your convenience.

Resources

- VolanoMark home page
- Volano Report benchmark results
- Neffenger's paper on Java on Linux
- Neffenger's paper on scalability of platforms in *JavaWorld*
- "The need for speed, stability" in *JavaWorld*
- *Java Thread Programming*
  , Paul Hyde; Sams, ISBN 0672315858 (September 1999)
- *Multithreaded Programming with Pthreads*
  , Bil Lewis, Daniel J. Berg; Prentice Hall, ISBN 0-13-680729-1 (1998)
- Java 1.1.8 IBM Developer Kit for Linux
- IBM kernel performance patch
- Kernel scheduler code
- Code that gcc generates for the run queue search loop

Linux threading links

- Many-to-one threading
- One-to-one threading
- Many-to-many threading
- Linux threading library
- Signaling and threading in Linux

Sun and Java-related threading links

- A Sun site on threading in Java
- Multithreading Options on the Solaris Operating system
- Details on Java's garbage collection

About the authors

Ray Bryant is a Senior Software Engineer with IBM in Austin, Texas. He is a performance specialist in the IBM Linux Technology Center. His primary interests are solving hardware and software scalability issues of Linux systems running large server workloads. Before moving to Austin, he was a Research Staff Member with the IBM T. J. Watson Research Center, where he worked with the Mach operating system on a 64-way shared memory multiprocessor. He has been working with Unix for far longer than he wishes to admit, but after dealing with the alternative for a while, he is happy to be back working with Linux. He can be reached at raybry@us.ibm.com.

Bill Hartner works in the Network Computer Software Division of IBM as a member of the IBM Linux Technology Center. Bill has worked on operating system performance in the areas of file system, scheduling and dispatching, SMP scalability, file serving, and network protocol stack performance. Bill is currently working on Linux performance analysis. He can be reached at bhartner@us.ibm.com.

Acknowledgments
The authors would like to acknowledge the assistance of Rajiv Arora, Robert Berry, and Jerry Burke in reviewing this document for publication. We would also like to acknowledge the interest and assistance of Ingo Molnar and Andrea Arcangeli. Ingo, in particular, has provided many useful insights into the design of current Linux scheduler and made several suggestions related to the experiments performed in this paper.

Copyright and trademark information