USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux® Kernel

Ray Bryant
raybry@us.ibm.com
IBM® Linux Technology Center
Austin, Texas

John Hawkes
hawkes@engr.sgi.com
SGI™
Mountain View, California

Abstract

Lockmeter is a tool for instrumenting the spin locks in a multiprocessor Linux kernel. Lockmeter was designed to make minimal cache disruptions, taking care to both minimize cache misses and also to minimize interprocessor cache coherency operations. Lockmeter is "highly informative" in the sense that not only does it record overall statistics for each spin lock, it also reports (where possible) these statistics on a per caller basis. This allows one to readily identify which portions of the kernel code are responsible for causing lock contention. In this paper, we describe the capabilities and implementation of Lockmeter version 1.3.

Introduction

As Linux becomes more popular for use in Web server, E-commerce and other compute intensive application environments, performance requirements on the Linux kernel will be much more stringent than for the "traditional" Linux desktop environment. This is particularly the case when Linux is used as the operating system for a symmetric multiprocessor "server" machines.

Symmetric multiprocessor (SMP) system performance is typically determined by two factors: instruction path length and lock contention. Instruction path length (that is, the time required to execute a particular function in the kernel) can be measured and optimized on a uniprocessor system using profiling tools such as SGI kernprof [SGI kernprof] or the profiling facilities of the gcc compiler. Such tools have relatively low overhead and can be used to measure realistic workloads with relatively little perturbation to the system under test.

The second factor that can limit the performance of a multiprocessor kernel is lock contention. As a general rule, correct execution of a shared-memory multiprocessor kernel requires that a lock be acquired before accessing a shared resource (e. g. a shared data structure) and released after the access. Contention arises when more than one process in the system tries to acquire a lock at the same time. Correct execution requires that only one of the processes can succeed; the other processes must be delayed until the lock is released. A delay can be implemented either by "spinning" (i. e., executing a tight instruction loop that constantly tries to acquire the lock) or by suspending the task that is attempting to access the shared resource and dispatching that processor to run some other task in the system. Locks can thus be classified as either "spin" locks or "suspend" locks depending on how a conflicting-lock access is delayed.

Each class of lock has its advantages and disadvantages:

- Acquiring or releasing a spin lock can be very inexpensive but waiting for a lock in a spin loop wastes time that could be devoted to useful work.
- A task that is suspended while waiting for a lock does not consume processor time, but the cost of acquiring or releasing a suspend lock is much higher than it is for the spin lock case.

For these reasons, both spin locks and suspend locks are typically present in a multiprocessor operating system kernel. Spin locks, however are more primitive and are normally used to implement suspend locks. In either case, excessive contention for a lock can lead to poor system performance, either because too many tasks are suspended, or because too much time is wasted spinning waiting for a lock to become available.

While instruction path length is typically straightforward to instrument without significantly perturbing the workload being measured, it is difficult to instrument lock usage in the Linux kernel without significantly impacting system performance. In Linux the code to acquire and release a spin lock is highly optimized and coded as an in-line assembler

sequence. The result is that the Linux kernel can employ as few as two instructions to acquire a spin lock and one instruction to release it. Because spin lock operations are so inexpensive, they are used extensively throughout the kernel. Thus, increasing the number of instructions per lock, or, more importantly, causing an additional cache miss every time a lock is acquired or released, can significantly change system performance.

In this paper, we discuss *Lockmeter*, a tool for instrumenting the usage of spin locks in the Linux kernel. Lockmeter was developed and released as a kernel patch to the open source community by John Hawkes at SGI (Silicon Graphics) (based on a previous design by Jack Steiner at SGI). Ray Bryant of IBM subsequently enhanced this tool, and the enhanced version is now available at the SGI open-source website [SGILockmeter]. Lockmeter is supported for Intel® i386 (also called IA32 in this paper) and Alpha architectures, and will soon be supported for the Intel IA64 and MIPS architectures, although the only full implementation of Lockmeter at the moment is for IA32.

Lockmeter allows the following statistics (among others) to be measured for each spin lock:

- The fraction of the time that the lock is busy.
- The fraction of accesses that resulted in a conflict.
- The average and maximum amount of time that the lock is held.
- The average and maximum amount of time spent spinning for the lock.

This information can be used to determine which locks are causing the most contention and where in the kernel these locks are being held for excessive periods of time. Examining these statistics can help identify places in the kernel where the code may need to be restructured in order to achieve improved efficiency and performance of the Linux kernel on SMP systems.

Lockmeter is designed to minimize the number of additional cache misses and cache coherency traffic introduced by the instrumentation itself. Furthermore, Lockmeter is a raw data-gathering engine inside the kernel, leaving more computationally expensive data reduction operations to a usermode application called *Lockstat*. Lockstat retrieves the raw data from the kernel, merges it into a unified system view, and displays the results in a human-readable form.

In addition to the per lock statistics, Lockmeter also provides statistics on a per function basis. One can readily determine from the Lockmeter output not only which locks have higher than acceptable contention levels, but also determine the functions from which the highly-contended calls originated. For this reason we describe Lockmeter as being "highly informative".

In the next sections of this paper, we first describe the Lockmeter implementation. Next, we present some sample Lockmeter output. We discuss the measurements contained in that output, as well as propose an interpretation of this data. We conclude with a discussion of areas of current investigation for improving Lockmeter.

Lockmeter Implementation

Multiprocessor Linux kernels of version 2.2.x and above use two types of spin locks to serialize access to shared data. (When a Linux kernel is compiled for a uniprocessor, conditional compilation flags are set so that neither the locks nor the locking instructions themselves are present in the kernel.) The two types of locks are a mutual exclusion lock, which inside the kernel is declared by the *spinlock_t* data type, and a multiple reader, single writer "read/write lock" which is declared using the *rwlock_t* data type. The latter locks do not support promotion from reader to writer; the lock must be released in order to make this transition.

In spite of the naming convention, both of these lock types are spin locks in the classical sense – that is, if the lock cannot be immediately acquired, then the requester enters a tight spin loop checking for the lock to be released by the current owner, at which point the requester again attempts to reacquire it. If the lock is never freed, then the requester will spin forever. No attempt is made to suspend or reschedule the requester. Semaphores and wait queues are used in those cases where the hold time of the lock is sufficiently long enough to make it worthwhile to suspend the requester and schedule another process. Spinlocks and read/write locks are reserved for those cases where the lock holding time is known to be short, or as primitives to build more robust locking primitives such as semaphores.

Implementation for *spinlock_t* Locks

The kernel mutual exclusion spin locks are declared as the data type *spinlock_t* and (for IA32) are implemented as a structure consisting of a single 32-

bit word. To acquire the lock one calls *spin_lock();* to release the lock one calls *spin_unlock().* There are variations of these calls that save and restore the interrupt state of the machine, but for the purposes of this paper we will ignore these variations (and will ignore them for the *rwlock_t* case below) since they consist of a wrapper that surrounds the *spin_lock()* or *spin_unlock()* call itself. Since the Lockmeter implementation modifies these basic macros, it modifies the variations as well.

In a non-lockmetered kernel, *spin_lock(), spin_unlock(),* and *spin_trylock()* are C language macros that resolve to gcc *inline* functions that contain assembler instructions that implement the lock operations. The Lockmeter patch renames these inline lock macros with a *nonmetered_* prefix (e.g., *nonmetered_spin_lock()*), then declares new *spin_lock()* (*et al*) macros that call external lockmetering routines *_spin_lock_()* and *_spin_unlock_().* Obviously, procedure calls are more expensive than a few inline assembler instructions, but the instrumented lock code is too large to be inserted inline.

The *nonmetered_*()* primitives are employed by *_spin_lock_()* and *_spin_unlock_()* to implement the instrumented locks. Thus the Lockmeter code is largely machine (and lock implementation) independent, since it uses the existing underlying lock primitives.

As an example of how this is done, consider the following pseudo-code implementation of *_spin_lock_():*

```
void __spin_lock_(spinlock_t *lock) {
    if (nonmetered_spin_trylock(lock)) {
            /* we acquired the lock without waiting */
            update statistics for this case
    } else {
            /* we must to spin to wait for the lock */
            record time we started spinning
            nonmetered_spin_lock(lock);
            record time we stopped spinning
            update statistics for this case
}
```

The key data structures used by the Lockmeter spin lock routines are:

- The *directory*, which is organized as a hash table keyed by the address that invokes the instrumented lock routine, and further indexed by lock type.
- The *count* array, which is a two-dimensional array indexed by hash-table index and by logical CPU number.

The hash table entries in the *directory* contain no lock statistics data; all that is present is the calling address for the instrumented lock routine, the address of the lock itself, and a lock type field. Separate lock types exist for *spinlock_t* locks, *rwlock_t*'s acquired for read mode, and *rwlock_t*'s acquired for write mode. The lock type information is used as part of the hash lookup and by the data reduction program, *Lockstat.*

When a hash lookup is performed, the *directory* entry information is used to resolve hash table synonyms. The result of the hash table search is an index that is used in combination with the current logical CPU number to select an entry in the *count* array where the actual statistics are stored. This makes the *directory* a performance-efficient read-mostly structure for all processors. The *directory* is only updated when each lock request is first encountered, making them relatively rare events once the system is running and the *directory* has been populated with the frequently used spin locks. A single nonmetered spin lock protects *directory* updates.

A *directory* entry is either a "single-address" entry or a "multi-address" entry. A single-address entry contains the address of the *spin_lock()* caller and the address of the *spinlock_t* structure being locked. Such an entry results when a given caller always specifies the same lock address. Lock requests for statically allocated locks are typically of this type. A multi-address entry results when a particular lock request specifies different lock addresses on different invocations. Typically this happens when a lock routine is acquiring a *spinlock_t* that is part of a larger dynamically allocated structure.

Each new *directory* entry is initially allocated as a single-lock entry. It converts to a multi-lock entry if a search of the hash table finds a match on the caller's address, but the current lock address does not match the value previously associated with that caller. At that point, the entry converts to a multi-lock entry by setting the lock address in the entry to zero. The data reduction program, Lockstat, recognizes this as a special case and reports only a coalesced summary of all locks set by this caller's address. For the more common single-lock entries, Lockstat reports per caller statistics for each unique spin lock address.

Each occupied *directory* entry contains a unique index into the *count* array. Each *count* entry represents a spin lock address, and the entry contains

the count of lock requests for that specific spin lock, the cumulative sum of "hold" times and "wait" times for that lock, and the maximum observed "hold" and "wait" times. Thus, each time a spin lock is acquired and each time it is released, the lockmetering routines update a *count* entry.

An important implementation efficiency is to give each CPU its own private *count* array. One advantage this provides is that we do not need to protect concurrent *count* entry updates with a (nonmetered!) spin lock. Another advantage is that processor-private updates only perturb a single processor's cache, and thus colliding updates do not cause expensive inter-processor cache references.

Each entry of the *count* array maintains lock statistics for one *spin_lock()* call on a particular processor. (We refer to these as "per caller" statistics.) Lockstat calculates global statistics for a lock by aggregating all of the single address statistics entries that correspond to the same lock. Statistics for the multi-address entries are reported on a per-caller basis only.[1] Experience has shown that eliminating the distinction between single and multi-address entries (by hashing on the caller address and the lock address, for example) clutters the Lockmeter output with many temporary, single-use locks.

In the *_spin_lock_()* routine, the *count* array is updated as follows:

1. Look up the calling address in the *directory*.
2. Using the directory *index*, find the *count* entry and update acquire-time statistics such as number of times the lock has been requested and the spin time.
3. Store the acquisition timestamp in the *count* entry as the first step in calculating the lock "hold" time.

At unlock time we need to locate this same *count* entry in order to finish the "hold" time calculation, which means we need to determine the *directory* index of that entry. To compute the *directory* hash index, we would need the address of the previous *_spin_lock_()* call, but that is presumably unknown to us at *_spin_unlock_()* time. We could remember these *directory* indices in yet another per-processor data structure, but this would introduce more code to manage the data structure and potentially more costly cache misses.

We solve this problem by storing the hash index in the lock itself. In every implementation we have seen, a *spinlock_t* is allocated as a 32-bit location, but few of the 32 bits are actually used. We therefore can use some of the remaining bits to save the hash index that was computed in *_spin_lock_()* for later use at *_spin_unlock_()* time. Since the lock location is already in the cache of the local processor due to acquiring the lock, storing the hash index in the lock results in negligible additional overhead and no additional cache misses.

Implementation for Read Locks

As with *spin_lock()* and *spin_unlock(),* the non-instrumented multiprocessor kernel implements *read_lock()* and *read_unlock()* as inline functions that generate only a few assembly language instructions each. Lockmeter replaces these inline functions with calls to *_read_lock_()* and *_read_unlock_(),* respectively. The existing lock primitives are renamed to nonmetered versions, just as for the *spinlock_t* case, and the nonmetered versions are utilized by the instrumented lock routines to perform the actual lock operations.

In *_read_lock_(),* acquire time statistics such as the count of requests, the time spent spinning waiting for the lock, and the count of times that the lock was obtained without spinning are maintained in the *count* array at the appropriate hash index. The *directory* entry is set to lock type "*rwlock_t* acquired in read mode." As before, separate statistics entries are maintained for each processor. Thus we record per caller statistics for read locks for statistics that are completely known at read-lock acquire time.

Hold-time statistics for read locks are problematic, however, since more than one processor can concurrently hold the same read lock. Therefore, *_read_unlock_()* cannot easily determine which *_read_lock_()* it is releasing, and this makes direct calculation of the read lock hold time impossible. (Since a *spinlock_t* is a mutual exclusion lock, each *_spin_unlock_()* ends the hold time that began with the most recent *_spin_lock_()* for this lock, so determining the hold time is straightforward.) One could keep track of the correspondence between each *_read_lock_()* and its matching *_read_unlock_()* using a list associated with each processor, but maintaining this list would be expensive and of questionable analytic value.

[1] It is possible for a lock to be accessed both by static and dynamic lock requests; in this case the lock statistics will be split between global and per-caller statistics. We are not aware of any lock usage in the kernel that follows such a pattern.

Lockmeter solves these problems as follows. We first require that the *rwlock_t* be declared as a pair of 32 bit words. (The Linux 2.3.99-pre6 kernel uses the 2nd word as a debug flag; Lockmeter merely requires that this debug word be present. We cannot store any data in the first word of the lock structure since in the current implementation of read/write spin locks there are no unused bits in that word.)

The first time a read or write lock is acquired on a variable of type *rwlock_t*, a read lock index is allocated for the lock. This is done by incrementing a global variable and assigning the next available index to this lock variable. The read lock index is stored in part of the second word of the lock for use on subsequent lock operations. The read lock index specifies the location in the *read_lock_count* array, where hold time statistics for this lock are stored.

Like the *count* array for *spinlock_t* metering, the *read_lock_count* array is a two-dimensional array; the first index being the read lock index, the second index being the current logical CPU number. The *read_lock_count* array contains running sum and count information necessary to calculate average read lock hold times as well as the overall read lock utilization time. Once again, statistics for each lock are maintained in storage private to each processor, and the data reduction program, Lockstat, is responsible for merging these statistics across processors.

However, for read locks, the data is not kept on a per caller basis, since the read lock index is the same for all users of the lock. Per caller statistics for read lock hold times would require that we match read lock and unlock operations using a per processor lock list, and we regard that approach as too expensive to employ.

The running sum for lock hold times in the *read_lock_count* structure are updated as follows based on an idea from [IBM1, IBM1A]:

At lock acquire: *running_sum -= get_cycles64();*
At lock release: *running_sum +=get_cycles64();*

(Here *get_cycles64()* returns the current 64 bit Time Stamp Counter (TSC register) value.) This approach keeps us from having to maintain the lock acquire time for each processor for each read lock (and it keeps us from having to deal with recursion problems related to a read lock that is set more than once by a particular processor).

The above works because the above is equivalent to the more straightforward algorithm, where at acquire time we record a timestamp:

lock_acquire_time = get_cycles64();

And at release time we decrement the current timestamp from the saved acquisition timestamp:

running_sum += get_cycles64()
– lock_acquire_time;

(provided that *lock_acquire_time* is kept on a per processor basis.)

This calculation can alternatively be done as:

running_sum += get_cycles64(); / (1) */*
running_sum -= lock_acquire_time; / (2) */*

And since addition is commutative, we will get the same result if we do *(2)* before *(1)*. If we do (2) first, we might as well do it at lock request time and avoid the temporary variable:

running_sum -= get_cycles64(); / (2) */*

then at lock release time, we do (1):

running_sum += get_cycles64(); / (1) */*

which is how we described the calculation originally.

However, we do have the problem that the *running_sum* is only correct when there are no read lock holders. Otherwise, the *running_sum* is a negative number (provided we assume that the maximum read lock hold time is very small compared to the current *get_cycles64()* value). Complete details of how this is done can be found in the source code [SGILockmeter] and [IBM2]. For this paper it will suffice to say that read lock hold time statistics are enabled and disabled on a per lock basis, and a transition from enabled to disabled state is only allowed when there are no read lock holders of the lock.

The last part of the read lock statistics to be discussed here is read lock utilization; that is, the fraction of time that a particular *rwlock_t* is owned in read mode by one or more readers. This is done by maintaining in the *read_lock_count* array a running sum of the busy period lengths that ended on the current processor for this lock. (A busy period is defined as the time starting when the number of read lock holders for a lock transitions from zero to one, until the next time that the number of readers transitions from one to zero.) It is easy enough for the instrumented read lock routines to recognize the start and end of a busy period; the hard problem is how to update the running sum of busy period lengths without using a global variable. Using a global variable would cause too much interprocessor cache coherency traffic as well as requiring a locked update of some kind to deal with concurrent accesses.

The solution here [IBM3] is to maintain in the *read_lock_count* entry for this processor (and this lock) the last time (measured using *get_cycles64()*) that a busy period started due to a *_read_lock_()* call executed on this processor for this lock. Also, when a busy period starts, the current logical CPU number is stored in the *rwlock_t* structure. (Since this structure is pulled into the local cache when the lock is acquired, this operation causes minimal additional overhead.)

When a processor detects the end of a busy period in *_read_unlock_()*, that processor can determine the busy period's start time by looking in the appropriate *read_lock_count* array entry for the processor that started the busy period; the logical CPU number of that processor is obtained from the *rwlock_t* structure. From this and the current time, we know the busy period length. The busy period length is then added to a running sum of busy period lengths, and the number of busy periods is incremented. (Both of these variables are stored in per processor storage in the *read_lock_counts* array.) This approach results in at most one remote cache reference at the end of each busy period.

Implementation for Write Locks

Since write locks are mutual-exclusion locks, the implementation of metering write mode locks on a *rwlock_t* is basically the same as it is for *spin_lock()*. The only difference is that the hash table index for the lock statistics entry is saved in the *read_lock_counts* array instead of in the lock itself. This allows us to keep per-caller statistics for write mode locks on a *rwlock_t*.

When the *directory* entry is being searched, the hash function is based upon the return address of the *_write_lock_()*. The *directory* entry is set to lock type of "*rwlock_t* acquired in write mode".

Each *rwlock_t* in the kernel thus can have three statistics structures associated with it:

1. The global read lock counts structure in the *read_lock_count* array.
2. A statistics entry containing read lock spin wait times on a per caller basis in the *count* array. This entry is updated only at *read_lock()* time with statistics that are known at the time that the read lock is acquired.
3. A statistics entry containing write lock spin and hold times on a per caller basis in the *count* array. This entry is updated at *write_lock()* and *write_unlock()* times.

The Lockstat Program

Lockstat is the user interface to the Lockmeter facility. It implements the following categories of operations:

- Instrumentation control: Lockmeter statistics can be enabled, disabled, queried, reset, or released under control of the Lockstat program. ("release" frees the kernel storage occupied by the *count*, *directory*, and *read_lock_count* data structures.)
- Data reduction: Lockmeter statistics can be read from the kernel and either saved for later data reduction or immediately reduced and printed. Either function can be performed on-demand or automatically using a periodic timer. During data reduction, Lockstat uses the kernel's *System.map* file to translate lock names and function addresses into symbolic names.

A typical measurement scenario would look something like the following:

```
# begin measurement
lockstat on
# run experiment
. . .
# end measurement
lockstat off
# reduce data
lockstat –m System.map print > lock.report
# clear statistics to prepare for next measurement
lockstat reset
```

Alternatively, one could use the "lockstat get" command to fetch the raw lockmeter statistics for printing at a later time.

Lockstat supports the notion of multiple measurement "intervals". Each execution of "*lockstat on*" begins a new measurement interval; executing "*lockstat off*" ends the interval. Statistics from each interval are merged together during data reduction. The "reset" command is used to clear data from previous measurement intervals and begin a new set of measurements. This facility allows one to run several repetitions of an experiment and have Lockstat merge the statistics from the repetitions into a single Lockstat report.

Additional documentation on Lockstat is provided by the "*--help*" option to Lockstat; of course the source code itself is available at [SGILockmeter].

Lockstat Output

The Lockstat report consists of three major sections: "SPINLOCKS", "RWLOCK READERS" and "RWLOCK WRITERS." Each of these sections is subdivided into per-lock statistics entries and per-caller statistics entries. A condensed version of a Lockstat report is provided in the Appendix and is discussed in the following. Due to space constraints, this condensed report provides statistics for only a small fraction of the locks present in a real Lockstat report. Also, in order to make the report fit onto a printed page, we have removed some of the columns of Lockstat output.

The top of the report shows information about the system being measured. (For this report the system was an IBM Netfinity® 7000 M10 Intel® Pentium® II Xeon™ 4-way SMP system.) This particular Lockstat report was generated for all 4 processors. However, since Lockmeter statistics are kept on a per processor basis, one could also generate a report based on a single processor's lock usage.

The data presented here summarizes Lockmeter statistics recorded during an experiment of approximately 140 seconds in length. The workload being run for this experiment is Volanomark™ [VMark], a benchmark written in the Java™ language. For these measurements, Volanomark was run using the IBM® Developer Kit for Linux®, Java™ Technology Edition, Version 1.1.8. For further details about this benchmark environment, see [JTThreads, SMPPerf]. For discussion purposes here, it is sufficient to know that the benchmark is CPU bound and causes a large number of Linux processes (threads) to be created and scheduled.

"SPINLOCKS" Section of the Report

The SPINLOCKS section of the Lockstat report shows statistics information for the *runqueue_lock* and the *timerlist_lock* (see *Note 1* and *Note 2* in the report). These locks protect the scheduler queue and the timer list data structures, respectively.

The first line of the report for each lock shows overall statistics, while subsequent lines provide per-caller statistics for that lock. At the end of the SPINLOCKS section (see *Note 2*) are some multilock statistics entries. (These entries can be recognized because there is no lock name associated with this part of the report.) These entries report on *spin_lock()* calls that request more than one lock address during the measurement run; typically this means that the locks being set are dynamically allocated. Rather than report on each individual lock, Lockstat aggregates all such requests and reports them only by lock caller. As previously discussed, this avoids cluttering the Lockstat output (and the kernel Lockmeter statistics) with information about many temporary, single-use locks.

The first column of the SPINLOCKS report (labeled UTIL) is the lock utilization.[2] This is defined as the fraction of time that the lock was held during the report interval. The second column (CON) is the fraction of lock requests that found the lock was busy when it was requested. The third and fourth columns (HOLD MEAN and MAX) show the mean (average) and maximum hold times for the lock. The next two columns (WAIT MEAN and MAX) give the mean and maximum times that a lock requester had to wait to obtain a lock.[3] The next column of the report (TOTAL) gives the total number of requests that occurred for the lock during the measurement interval. In a full Lockstat report, subsequent columns display the number of requests that had to spin for the lock; this and similar columns have been removed from the report here in order to conserve space. The last column gives either the lock name (*runqueue_lock* for the case at *Note 1*) or the calling location, as appropriate.

"RWLOCK READERS" Section of the Report

The RWLOCK READERS section of the Lockstat report provides statistics about *read_lock()* requests for *rwlock_t* locks. Like the SPINLOCKS section, this section is divided into a lock statistics section and a multilock section; to conserve space the latter has been omitted from this condensed report. We show here entries for the *tasklist_lock* and the *xtime_lock*. The *task_list* lock is held in read mode to scan over all tasks in the system, while the *xtime_lock* is held in read mode to read system time information.

Just as the previous report, the UTIL column gives the lock utilization. For read-mode locks, this is defined as the fraction of time that there is at least one reader for the lock. Total utilization for this lock is the sum of the utilizations from the "RWLOCK READERS" and "RWLOCK WRITERS" sections of the Lockstat report. The "HOLD MEAN" column gives the average time that the lock was held in read

[2] The UTIL field can optionally be replaced by a rate request field via a Lockstat command-line option.

[3] The mean wait time is defined as the average over all requesters that had to wait, rather than over all requesters.

mode, averaged over all readers of the lock. Given the current implementation Lockmeter, this statistic is only available on a global and not a per-caller basis.

The "MAX READERS" column gives the maximum number of readers that simultaneously held the lock at any one time. The fact that this number is 5 on a 4-way SMP system indicates that some processor holds this lock more than once. One possible explanation for this is that the timer-interrupt code obtains a read lock on *task_list* lock; if all four processors in the system already hold the *task_list* lock and then a timer-interrupt occurs, this will cause one of the processors to re-lock the *tasklist_lock*.

The "RDR BUSY PERIOD" columns provide mean and maximum times of busy periods for the lock. A busy period for a read lock is defined as the period of time starting when the number of read-lock holders goes from zero to one and ending when there are no read-lock holders of the lock. Busy period information tells us how long a write-requester might have to wait in order to obtain the lock. The sum of the lengths of busy periods is used to calculate the read-lock utilization.

"RWLOCK WRITERS" Section of the Report

The "RWLOCK WRITERS" section of the Lockstat report provides statistics about *write_lock()* requests for *rwlock_t* locks. The same format and locks discussed in the "RWLOCK READERS" section above are used here.

The utilization column (UTIL) in this section of the report gives the fraction of time that the lock was held in write mode. The wait-time statistics for a write-mode lock are given as the mean and maximum wait time over all requests as well as the mean and maximum wait times for write-lock requesters who had to wait due to other write-lock holders of the lock (as opposed to write-lock requesters who had to wait due to other read-lock holders of the lock). These statistics are given in the "WAIT (ALL)" and "WAIT (WW)" columns of the report, respectively. Similarly, the "SPIN ALL" and "SPIN (WW)" columns of the report give the count of requesters who had to spin for the lock and the count of requesters who had to spin for the lock due to another write-lock holder.

Analysis of the Lockstat Data

As an example of the kind of analysis one might do with the Lockstat data, we now discuss the results reported by Lockstat for this benchmark.

Inside the Linux kernel, the *runqueue_lock* protects access to the scheduler queue. Since the Volanomark benchmark is a scheduler intensive benchmark [JTThreads, SMPPerf] we expect to see contention for the *runqueue_lock* in this report. At *Note 1* we see that utilization of this lock is nearly 22% and that 25% of the requests for this lock had to spin-wait for the lock. One can see that most of the time spent holding this lock was due to requests that occurred at *schedule+0xd0*. From the numbers reported here, one can see that this caller was responsible for 40% of the requests to this lock and 67% of the utilization. This caller also held the lock for longest time (average and maximum). Examination of the kernel/sched.c source code shows that this lock is held from entry of *schedule()* until after the goodness calculation has completed. Previous work [JTThreads, SMPPerf] has shown that for this benchmark, the goodness calculation can consume a significant amount of CPU time. These lock statistics for the *runqueue_lock* validate that observation.

Note also how the hold times due to *schedule+0xd0* impact the other lock requests. See, for example, the request at *schedule+0x444*. While the utilization of the lock due to this request is only 1.7%, 56% of all such lock requests had to spin for the lock. This indicates that although the lock holding time for this request is low, usage of the lock elsewhere causes contention and results in an average 14 microsecond delay and a maximum delay of 285 microseconds. Examination of the source code shows that this lock request is from the inlined function *__schedule_tail()* just before the scheduler returns, running a new process. Hence the 14 microsecond mean wait time here directly affects the latency of the system in starting to run a new process.

Similarly, if we examine the *tasklist_lock* (see *Note 5* in the Lockstat report) we see that *do_fork()* (the worker routine that implements the fork system call) was delayed an average of 7.3 microseconds and a maximum of over 1.5 milliseconds waiting for the *tasklist_lock*. Note that it could have been worse, since the maximum read-lock busy period for this lock was over 8 milliseconds (*Note 4*).

Of course, these observations are correct only for this particular benchmark and are not necessarily indicative of actual bottlenecks in the Linux kernel.

Rather they are provided as examples of where potential bottlenecks might occur and how the Lockstat report can be used to find such problems.

Concluding Remarks

As the Linux kernel continues to be deployed on large SMP servers, additional performance optimization of the kernel will be required in order to achieve performance comparable to the more mature operating systems that have traditionally been used on such hardware. As systems become larger and more complex, we will need increasingly powerful tools to analyze and diagnose system performance problems. Lockmeter endeavors to be one such measurement tool in a Linux performance toolbox that can be used to diagnose and optimize Linux system performance.

We are continuing to improve Lockmeter and Lockstat. Alternative implementations of the read/write lock statistics recording mechanism are currently under investigation, as are benchmark studies to estimate the Lockmeter measurement overhead. Such overhead measurements should be available at the time of the 2000 Atlanta Linux Showcase and Conference. An updated version of this paper should also be available at that time on the SGI lockmeter website [SGILockmeter] or the IBM Linux Technology Center website [IBMLTC].

References

[SGI Kernprof]: "Kernel Profiling," http://oss.sgi.com/projects/kernprof

[SGI Lockmeter]: "Kernel Spinlock Metering for Linux." http://oss.sgi.com/projects/lockmeter

[IBM1] United States Patent 5,872, 913, "System and Method for Low Overhead, High Precision Measurements using State Transitions," Robert F. Berry et al.

[IBM1A] United States Patent 5,920,689, "System and Method for Low Overhead, High Precision Measurements using State Transitions," Robert F. Berry et al.

[IBM2] "Efficient Update of Shared Resource Usage Statistics," IBM Technical Disclosure Bulletin, *to appear.*

[IBM3] "Careful Update and Control of Hold Time Statistics for Shared Multiuser Resources," IBM Technical Disclosure Bulletin, *to appear.*

[Volano]: "Volano Java Chat Room," Volano LLC. http: //www.volano.com.

[JTThreads]: "Java technology, threads, and scheduling in Linux--Patching the kernel scheduler for better Java performance," Ray Bryant, Bill Hartner, IBM. http://www4.ibm.com/ software/developer/library/java2 /index.html.

[SMPPerf]: "SMP Scalability Comparisons of Linux® Kernels 2.2.14 and 2.3.99," Ray Bryant, Bill Hartner, Qi He, and Ganesh Venkitachalam, Proceedings of the 2000 Atlanta Linux Showcase and Conference, Atlanta, Ga., October, 2000.

[IBMLTC] Linux Technology Center, http: //oss.software.ibm.com/developerworks/ opensource/linux/.

Trademark and Copyright Information

Appendix: (Condensed) Lockstat Output

```
System: Linux testlinux.austin.ibm.com 2.3.99-pre6 #147 SMP Tue Aug 22 15:18:05 CDT 2000 i686

All (4) CPUs

Start time: Fri Aug 25 16:09:05 2000
End   time: Fri Aug 25 16:11:26 2000
Delta Time: 141.33 sec.
Hash table slots in use:      356.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
SPINLOCKS        HOLD              WAIT
   UTIL   CON    MEAN( MAX ) MEAN ( MAX  ) TOTAL  NAME
. . .
  21.86% 25.12% 3.3us( 87us) 4.4us(311us) 9480279 runqueue_lock                          ←Note 1
   1.62% 25.14% 3.3us( 13us) 1.1us(136us)  696844  __wake_up+0x110
   0.00%  0.00% 0.2us(0.2us)   0us             1  __wake_up_sync+0xfc
   0.00% 23.60% 5.1us( 19us) 4.3us(112us)     322  process_timeout+0x1c
   0.00%  8.71% 0.5us(1.3us) 0.2us(7.1us)     551  release+0x28
   0.00% 61.11% 2.0us(5.1us) 1.2us(5.9us)      36  schedule_tail+0x48
  14.69% 19.63% 5.5us( 87us) 1.7us(311us) 3806754  schedule+0xd0
   1.71% 56.33% 2.1us( 12us)  14us(295us) 1150208  schedule+0x444
   0.51% 14.35% 4.3us( 28us) 0.5us( 51us)  165306  schedule+0x710
   0.68% 12.89% 0.8us(4.5us) 0.7us(178us) 1224206  send_sig_info+0x2a0
   0.00% 40.57% 4.8us( 10us) 0.9us( 11us)     801  setscheduler+0x68
   0.78% 46.40% 0.9us(6.1us)  15us(265us) 1210679  sys_sched_yield+0xc
   1.88%  5.56% 2.2us( 14us) 0.4us(164us) 1224571  wake_up_process+0x18
. . .
   0.95%  3.87% 0.6us( 17us) 0.1us(9.4us) 2380970 timerlist_lock                          ←Note 2
   0.13%  4.66% 0.4us(4.1us) 0.1us(9.4us)  405952   add_timer+0x14
   0.33%  4.50% 0.5us(4.3us) 0.1us(9.2us) 1004500   del_timer+0x14
   0.00%  0.18% 0.5us(1.5us) 0.0us(2.1us)     565   del_timer_sync+0x20
   0.31%  3.35% 0.6us(4.2us) 0.0us(8.2us)  777490   mod_timer+0x18
   0.03%  1.86% 3.0us( 17us) 0.0us(4.8us)   14134   timer_bh+0x12c
   0.16%  0.99% 1.2us(5.1us) 0.0us(6.3us)  178329   timer_bh+0x2b4
. . .
   3.47%  0.00% 7.0us(142us) 0.0us(7.1us) 702015 __wake_up+0x24                           ←Note 3
   0.22%  0.56% 0.4us( 36us) 0.0us(5.9us) 811287 dev_queue_xmit+0x30
   0.01%  0.00% 1.1us(5.9us)   0us         14558 do_IRQ+0x40
   0.01%  0.00% 4.7us( 31us)   0us          1521 do_brk+0x108
   0.00%  0.00% 0.2us(0.9us)   0us           429 do_exit+0x240
   0.00%  0.00% 0.2us(0.6us)   0us           338 do_fork+0x6fc
. . .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
RWLOCK READERS HOLD    MAX   RDR BUSY PERIOD    WAIT
  UTIL  CON    MEAN  READERS MEAN    (   MAX  ) MEAN ( MAX )   TOTAL NAME
. . .
  52.91% 0.00% 105.8us   5   114.8us (8274.8us) 0.0us(3.3us) 1402747 tasklist_lock       ←Note 4
        0.00%                                     0us             28  count_active_tasks+0x10
        0.23%                                   0.0us(2.3us)     429  exit_notify+0x1c
        0.00%                                     0us             5  exit_notify+0xb8
        0.00%                                     0us           576  get_pid_list+0x18
        0.00%                                   0.0us(3.0us) 1224079  kill_something_info+0xb8
        0.00%                                     0us           11002  proc_pid_lookup+0x4c
        0.00%                                     0us               7  proc_root_lookup+0x30
        0.00%                                     0us          165306  schedule+0x6d0
        0.00%                                     0us              25  session_of_pgrp+0x14
        1.12%                                   0.0us(3.3us)     801  setscheduler+0x78
        0.00%                                     0us              18  sys_setpgid+0x38
        0.00%                                     0us               1  sys_setsid+0x10
        0.00%                                     0us             461  sys_wait4+0x158
        0.00%                                     0us               9  will_become_orphaned_
                                                                        pgrp+0x14
. . .
 0.33%  0.06%   0.5us   2     0.5us  (   6.5us) 0.0us(528us)  856780 xtime_lock
        0.06%                                   0.0us(528us)  856780  do_gettimeofday+0x10
. . .
```

Appendix: (Condensed) Lockstat Output (continued)

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
RWLOCK WRITERS   HOLD        WAIT (ALL)    WAIT (WW)       SPIN SPIN
UTIL    CON  MEAN ( MAX ) MEAN ( MAX  ) MEAN (  MAX ) TOTAL ALL  WW  NAME

0.00% 10.53% 0.8us(2.7us) 9.8us(1515us) 0.8us( 1.7us)  1691 173   5 tasklist_lock
0.00% 10.68% 1.2us(2.6us) 7.3us(1515us) 0.8us( 1.7us)   833  84   5  do_fork+0x8a4       ←Note 5
0.00%  2.80% 0.2us(0.8us) 0.2us(  13us)   0us           429  12   0  exit_notify+0x284
0.00% 17.95% 0.7us(2.7us)  25us( 486us)   0us           429  77   0  release+0x78
. . .
0.12%  1.29% 6.2us(802us) 0.0us(7.7us)  0.9us( 4.3us) 28352 281  84 xtime_lock
0.03%  1.68% 2.8us(802us) 0.0us(7.7us)  1.0us( 3.1us) 14193 180  59  timer_bh+0x14
0.10%  0.89% 9.6us( 24us) 0.0us(6.4us)  0.8us( 4.3us) 14159 101  25  timer_interrupt+0x14
```